Microsoft

Programming Windows Store Apps with HTML, CSS, and JavaScript

Second Edition

Professional

Kraig Brockschmidt

PUBLISHED BY
Microsoft Press
A Division of Microsoft Corporation
One Microsoft Way
Redmond, Washington 98052-6399

Microsoft Press books are available through booksellers and distributors worldwide. If you need support related to this book, email Microsoft Press Book Support at mspinput@microsoft.com. Please tell us what you think of this book at http://aka.ms/tellpress.

Unless otherwise noted, the companies, organizations, products, domain names, e-mail addresses, logos, people, places, and events depicted in examples herein are fictitious. No association with any real company, organization, product, domain name, e-mail address, logo, person, place, or event is intended or should be inferred.

This book expresses the author's views and opinions. The information contained in this book is provided without any express, statutory, or implied warranties. Neither the authors, Microsoft Corporation, nor its resellers, or distributors will be held liable for any damages caused or alleged to be caused either directly or indirectly by this book.

Acquisitions, Developmental, and Project Editor: Devon Musgrave
Cover: Twist Creative • Seattle and Joel Panchot

Table of Contents

Introduction

Welcome, my friends, to Windows 8.1! On behalf of the thousands of designers, program managers, developers, test engineers, and writers who have brought the product to life, I'm delighted to welcome you into a world of **Windows Reimagined**.

This theme is no mere sentimental marketing ploy, intended to bestow an aura of newness to something that is essentially unchanged, like those household products that make a big splash on the idea of "New and Improved *Packaging*!" No, starting with version 8, Microsoft Windows truly has been reborn—after more than a quarter-century, something genuinely new has emerged.

I suspect—indeed expect—that you're already somewhat familiar with the reimagined user experience of Windows 8 and Windows 8.1. You're probably reading this book, in fact, because you know that the ability of Windows to reach across desktop, laptop, and tablet devices, along with the global reach of the Windows Store, will provide you with many business opportunities, whether you're in business, as I like to say, for fame, fortune, fun, or philanthropy.

We'll certainly see many facets of this new user experience throughout the course of this book. Our primary focus, however, will be on the reimagined *developer* experience.

I don't say this lightly. When I first began giving presentations within Microsoft about building Windows Store apps, I liked to show a slide of what the world was like in the year 1985. It was the time of Ronald Reagan, Margaret Thatcher, and Cold War tensions. It was the time of VCRs and the discovery of AIDS. It was when *Back to the Future* was first released, Michael Jackson topped the charts with *Thriller*, and Steve Jobs was kicked out of Apple. And it was when software developers got their first taste of the original Windows API and the programming model for desktop applications.

The longevity of that programming model has been impressive. It's been in place for nearly three decades now and has grown to become the heart of the largest business ecosystem on the planet. The API itself, known today as Win32, has also grown to become the largest on the planet! What started out on the order of about 300 callable methods has expanded three orders of magnitude, well beyond the point that any one individual could even hope to understand a fraction of it. I'd certainly given up such futile efforts myself.

So when I bumped into my old friend Kyle Marsh in the fall of 2009, just after Windows 7 had been released, and heard from him that Microsoft was planning to reinvigorate native app development for Windows 8, my ears were keen to listen. In the months that followed I learned that Microsoft was introducing a completely new API called the Windows Runtime (or WinRT). This wasn't meant to replace Win32, mind you; desktop applications would still be supported. No, this was a programming model built from the ground up for a new breed of touch-centric, immersive apps that could compete with those emerging on various mobile platforms. It would be designed from the app developer's point of view, rather than the system's, so that key features would take only a few lines of code to implement rather than hundreds or thousands. It would also enable direct native app development in multiple programming languages. This meant that new operating system capabilities would surface to those developers without having to wait for an update to some intermediate framework. It also meant that developers who had experience in any one of those language choices would find a natural home when writing apps for Windows 8 and Windows 8.1.

This was very exciting news to me because the last time that Microsoft did anything significant to the Windows programming model was in the early 1990s with a technology called the Component Object Model (COM), which is exactly what allowed the Win32 API to explode as it did. Ironically, it was my role at that time to introduce COM to the developer community, which I did through two editions of *Inside OLE* (Microsoft Press, 1993 and 1995) and seemingly endless travel to speak at conferences and visit partner companies. History, indeed, does tend to repeat itself, for here I am again, with another second edition!

In December 2010, I was part of the small team who set out to write the very first Windows Store apps using what parts of the new WinRT API had become available. Notepad was the text editor of choice, we built and ran apps on the command line by using abstruse Powershell scripts that required us to manually type out ungodly hash strings, we had no documentation other than oft-incomplete functional specifications, and we basically had no debugger to speak of other than the tried and true `window.alert` and `document.writeln`. Indeed, we generally worked out as much HTML, CSS, and JavaScript as we could inside a browser with F12 debugging tools, adding WinRT-specific code only at the end because browsers couldn't resolve those APIs. You can imagine how we celebrated when we got anything to work at all!

Fortunately, it wasn't long before tools like Visual Studio Express and Blend for Visual Studio became available. By the spring of 2011, when I was giving many training sessions to people inside Microsoft on building apps for Windows 8, the process was becoming far more enjoyable and exceedingly more productive. Indeed, while it took us four to six weeks in late 2010 to get even Hello World to show up on the screen, by the fall of 2011 we were working with partner companies who pulled together complete Store-ready apps in roughly the same amount of time.

As we've seen—thankfully fulfilling our expectations—it's possible to build a great app in a matter of weeks. I'm hoping that this ebook, along with the extensive resources on http://dev.windows.com, will help you to accomplish exactly that and to reimagine your own designs.

Work on this second edition began almost as soon as the first edition was released. (I'd make a quip about the ink not being dry, but that analogy doesn't work for an ebook!) When Windows 8 became generally available in the fall of 2012, work on Windows 8.1 was already well underway: the engineering team had a long list of improvements they wanted to make along with features that they weren't able to complete for Windows 8. And in the very short span of one year, Windows 8.1 was itself ready to ship.

At first I thought writing this second edition would be primarily a matter of making small updates to each chapter and perhaps adding some pages here and there on a handful of new features. But as I got deeper into the updated platform, I was amazed at just how much the API surface area had expanded!

Windows 8.1 introduces a number of additional controls, an HTML webview element, a stronger HTTP API, content indexing, deeper OneDrive support, better media capabilities, more tiles sizes (small and large), more flexible secondary tile, access to many kinds of peripheral devices, and more options for working with the Windows Store, like consumable in-app purchases. And clearly, this is a very short list of distinct Windows 8.1 features that doesn't include the many smaller changes to the API. (A fuller list can be found on Windows 8.1: New APIs and features for developers).

Furthermore, even as I was wrapping up the first edition of this book, I already had a long list of topics I wanted to explore in more depth. I wrote a number of those pieces for my blog, with the intention of including them in this second edition. A prime example is Appendix A, "Demystifying Promises."

All in all, then, what was already a very comprehensive book in the first edition has become even more so in the second! Fortunately, with this being an ebook, neither you nor I need feel guilty about matters of deforestation. We can simply enjoy the process of learning about and writing Windows Store Apps with HTML, CSS, and JavaScript.

And what about Windows Phone 8.1? I'm glad you asked, because much of this book is completely applicable to that platform. Yes, that's right: Windows Phone 8.1 supports writing apps in HTML, CSS, and JavaScript, just like Windows 8.1, meaning that you have the same flexibility of implementation languages on both. However, the decision to support JavaScript apps on Windows Phone 8.1 came very late in the production of this book, so I'm only able to make a few notes here and there for Phone-specific concerns. I encourage you to follow the Building Apps for Windows blog, where we'll be posting more about the increasingly unified experience of Windows and Windows Phone.

Who This Book Is For

This book is about writing Windows Store apps using HTML, CSS, and JavaScript. Our primary focus will be on applying these web technologies within the Windows platform, where there are unique considerations, and not on exploring the details of those web technologies themselves. For the most part, I'm assuming that you're already at least somewhat conversant with these standards. We will cover some of the more salient areas like the CSS grid, which is central to app layout, but otherwise I trust that you're capable of finding appropriate references for most everything else. For Java Script specifically, I can recommend Rey Bango's Required JavaScript Reading list, though I hope you'll spend more time reading *this* book than others!

I'm also assuming that your interest in Windows has at least two basic motivations. One, you probably want to come up to speed as quickly as you can, perhaps to carve out a foothold in the Windows Store sooner rather than later. Toward that end, Chapter 2, "Quickstart," gives you an immediate experience with the tools, APIs, and some core aspects of app development and the platform. On the other hand, you probably also want to make the best app you can, one that performs really well and that takes advantage of the full extent of the platform. Toward this end, I've also endeavored to make this book comprehensive, helping you at least be aware of what's possible and where optimizations can be made.

Let me make it clear, though, that my focus in this book is the Windows platform. I won't talk much about third-party libraries, architectural considerations for app design, and development strategies and best practices. Some of these will come up from time to time, but mostly in passing.

Nevertheless, many insights have come from working directly with real-world developers on their real-world apps. As part of the Windows Ecosystem team, myself and my teammates have been on the front lines bringing those first apps to the Windows Store. This has involved writing bits of code for those apps and investigating bugs, along with conducting design, code, and performance reviews with members of the Windows engineering team. As such, one of my goals with this book is to make that deep understanding available to many more developers, including you!

What You'll Need (Can You Say "Samples"?)

To work through this book, you should have Windows 8.1 (or a later update) installed on your development machine, along with the Windows SDK and tools. All the tools, along with a number of other resources, are listed on Developer Downloads for Windows Store Apps. You'll specifically need Microsoft Visual Studio Express 2013 for Windows. (Note that for all the screenshots in this book, I switched Visual Studio from its default "dark" color theme to the "light" theme, as the latter works better against a white page.)

We'll also acquire other tools along the way as we need them in this ebook, specifically to run some of the examples in the companion content. Here's the short list:

- Live SDK (for Chapter 4)
- Bing Maps SDK for Windows Store Apps (for Chapters 10 and beyond)
- Visual Studio Express 2013 for Web (for Chapter 16)
- Multilingual App Toolkit (for Chapter 19)

Also be sure to visit the Windows 8.1 Samples Pack page and download at least the JavaScript samples. We'll be drawing from many—if not most—of these samples in the chapters ahead, pulling in bits of their source code to illustrate how many different tasks are accomplished.

One of my secondary goals in this book, in fact, is to help you understand where and when to use the tremendous resources in what is clearly the best set of samples I've ever seen for any release of Windows. You'll often be able to find a piece of code in one of the samples that does exactly what you need in your app or that is easily modified to suit your purpose. For this reason I've made it a point to personally look through every one of the JavaScript samples, understand what they demonstrate, and then refer to them in their proper context. This, I hope, will save you the trouble of having to do that level of research yourself and thus make you more productive in your development efforts.

In some cases I've taken one of the SDK samples and made certain modifications, typically to demonstrate an additional feature but sometimes to fix certain bugs or demonstrate a better understanding that came about after the sample had to be finalized. I've included these modifications in the companion content for this book, which you can download at

http://aka.ms/BrockschmidtBook2/CompContent

The companion content also contains a few additional examples of my own, which I always refer to as "examples" to make it clear that they aren't official SDK content. (I've also rebranded the modified samples to make it clear that they're part of this book.) I've written these examples to fill gaps that the SDK samples don't address or to provide a simpler demonstration of a feature that a related sample shows in a more complex manner. You'll also find many revisions of an app called "Here My Am!" that we'll start building in Chapter 2 and we'll refine throughout the course of this book. This includes localizing it into a number of different languages by the time we reach the end.

There are also a number of videos that I've made for this book, which more readily show dynamic effects like animations and user interaction. You can find all of them at

http://aka.ms/BrockschmidtBook2/Videos

Beyond all this, you'll find that the Windows Store app samples gallery as well as the Visual Studio sample gallery let you search and browse projects that have been contributed by other developers—perhaps also you! (On the Visual Studio site, by the way, be sure to filter on Windows Store apps because the gallery covers all Microsoft platforms.) And of course, there will be many more developers who share projects on their own.

In this book I occasionally refer to posts on a number of blogs. First are a few older blogs, namely the Windows 8 App Developer blog, the Windows Store for Developers blog, and—for the Windows 8 backstory of how Microsoft approached this whole process of reimagining the operating system—the Building Windows 8 blog. As of the release of this book, the two developer blogs have merged into the Building Apps for Windows blog that I mentioned earlier.

A Formatting Note

Throughout this book, identifiers that appear in code, such as variable names, property names, and API functions and namespaces, are formatted with a color and a fixed-point font. Here's an example: `Windows.Storage.ApplicationData.current`. At times, certain fully qualified names—those that that include the entire namespace—can become quite long, so it's necessary to occasionally hyphenate them across line breaks, as in `Windows.Security.Cryptography.CryptographicBuffer.-convertStringToBinary`. Generally speaking, I've tried to hyphenate after a dot or between whole words but not within a word. In any case, these hyphens are never part of the identifier except in CSS where hyphens are allowed (as in `-ms-high-contrast-adjust`) and with HTML attributes like `aria-label` or `data-win-options`.

Occasionally, you'll also see identifiers that have a different color, as in `datarequested`. These specifically point out events that originate from Windows Runtime objects, for which there are a few special considerations for adding and removing event listeners in JavaScript, as discussed toward the end of Chapter 3. I make a few reminders about this point throughout the chapters, but the purpose of this special color is to give you a quick reminder that doesn't break the flow of the discussion otherwise.

Acknowledgements

In many ways, this isn't *my* book—that is, it's not an account of my own experiences and opinions about writing apps for Windows. I'm serving more as a storyteller, where the story itself has been written by the thousands of people in the Windows team whose passion and dedication have been a constant source of inspiration. Writing a book like this wouldn't be possible without all the work that's gone into customer research; writing specs; implementing, testing, and documenting all the details; managing daily builds and public releases; and writing again the best set of samples I've ever seen for a platform. Indeed, the words in some sections come directly from conversations I've had with the people who designed and developed a particular feature. I'm grateful for their time, and I'm delighted to give them a voice through which they can share their passion for excellence with you.

A number of individuals deserve special mention for their long-standing support of this project. To Mahesh Prakriya, Ian LeGrow, Anantha Kancherla, Keith Boyd and their respective teams, with whom I've worked closely, and to Kathy Carper, Roger Gulrajani, Keith Rowe, Dennis Flanagan, and Adam Denning, under whom I've had the pleasure of serving.

Thanks also to Devon Musgrave at Microsoft Press, who put in many long hours editing my many long chapters, many times over. My teammates, Kyle Marsh, Todd Landstad, Shai Hinitz, Patrick Dengler, Lora Heiny, Leon Braginski, and Joseph Ngari have also been invaluable in sharing what they've learned in working with real-world partners. A special thanks goes to Kenichiro Tanaka of Microsoft Japan, for always being the first one to return a reviewed chapter to me and for joyfully researching different areas of the platform whenever I asked. Many bows to you, my friend! Nods also to others in our international Windows Ecosystem teams who helped with localizing the Here My Am! app for Chapter 19: Gilles Peingné, Sam Chang, Celia Pipó Garcia, Juergen Schwertl, Maarten van de Bospoort, Li-Qun Jia, and Shai Hinitz.

The following individuals all contributed to this book as well, with chapter reviews, answers to my questions, deep discussions of the details, and much more. I'm grateful to all of you for your time and support:

Shakil Ahmed	Ryan Demopoulos	Jakub Kotynia	Jason Olson	Adam Stritzel
Arvind Aiyar	Scott Dickens	Jared Krinke	Elliot H Omiya	Shijun Sun
Jessica Alspaugh	Tyler Donahue	Victoria Kruse	Lisa Ong	Ellick Sung
Gaurav Anand	Brendan Elliott	Nathan Kuchta	Larry Osterman	Sou Suzuki

Chris Anderson	Matt Esquivel	Elmar Langholz	Rohit Pagariya	Simon Tao
Erik Anderson	David Fields	Bonny Lau	Ankur Patel	Henry Tappen
Axel Andrejs	Sean Flynn	Wonhee Lee	Harry Pierson	Chris Tavares
Tarek Ayna	Erik Fortune	Travis Leithead	Steve Proteau	David Tepper
Art Baker	Jim Galasyn	Dale Lemieux	Hari Pulapaka	Lillian Tseng
Adam Barrus	Gavin Gear	Chantal Leonard	Arun Rabinar	Sara Thomas
Megan Bates	Derek Gephard	Cameron Lerum*	Matt Rakow	Ryan Thompson
Tyler Beam	Marcelo Garcia Gonzalez	Brian LeVee	Ramu Ramanathan	Bill Ticehurst
Matthew Beaver	Sean Gilmour	Jianfeng Lin	Sangeeta Ranjit	Peter Torr
Kyle Beck	Sunil Gottumukkala	Tian Luo	Ravi Rao	Stephen Toub
Ben Betz	Scott Graham	Sean Lyndersay	Brent Rector	Tonu Vanatalu
Johnny Bregar	Ben Grover	David Machaj	Ruben Rios	Jeremy Viegas
John Brezak	Paul Gusmorino	Mike Mastrangelo	Dale Rogerson	Alwin Vyhmeister
John Bronskill	Chris Guzak	Jordan Matthiesen	Nick Rotondo	Nick Waggoner
Jed Brown	Zainab Hakim	Ian McBurnie	David Rousset	David Washington
Kathy Carper	Rylan Hawkins	Sarah McDevitt	George Roussos	Sarah Waskom
Vincent Celie	John Hazen	Isaac McGarvey	Jake Sabulsky	Marc Wautier
Raymond Chen	Jerome Holman	Jesse McGatha	Gus Salloum	Josh Williams
Rian Chung	Scott Hoogerwerf	Matt Merry	Michael Sciacqua	Lucian Wischik
Arik Cohen	Stephen Hufnagel	Markus Mielke	Perumaal Shanmugam	Dave Wood
Justin Cooperman	Sean Hume	Pavel Minaev	Edgar Ruiz Silva	Kevin Michael Woley
Michael Crider	Mathias Jourdain	John Morrow	Poorva Singal	Charing Wong
Monica Czarny	Damian Kedzierski	Feras Moussa	Karanbir Singh	Bernardo Zamora
Nigel D'Souza	Suhail Khalid	John Mullaly	Peter Smith	Michael Ziller
Priya Dandawate	Deen King-Smith	Jan Nelson*	Sam Spencer	
Darren Davis	Daniel Kitchener	Marius Niculescu	Edward Sproull	
Jack Davis	Kishore Kotteri	Daniel Oliver	Ben Srour	

* For Jan and Cameron, a special acknowledgement for riding down from Redmond, Washington, to visit me in Portland, Oregon (where I was living at the time), and sharing an appropriately international Thai lunch while we discussed localization and multilingual apps.

Let me add that during the production of this second edition, I did manage to lose the extra weight that I'd gained during the first edition. All things must balance out, I suppose!

Finally, special hugs to my wife Kristi and our son Liam (now seven and a half), who have lovingly been there the whole time and who don't mind my traipsing through the house to my office either late at night or early in the morning.

Free Ebooks from Microsoft Press

From technical overviews to drilldowns on special topics, these free ebooks are available in PDF, EPUB, and/or Mobi for Kindle formats, ready for you to download:

http://aka.ms/mspressfree

The “Microsoft Press Guided Tours” App

Check the Windows Store soon for the Microsoft Press Guided Tours app, which provides insightful tours of new and evolving technologies created by Microsoft. While you’re exploring each tour’s original content, the app lets you manipulate and mark that content in ways to make it more useful to you. You can, of course, do the usual things—such as highlight, add notes, mark as favorite, and mark to read later—but you can also

- view all links to external documentation and samples in one place via a Resources view;
- sort the Resources view by Favorites, Read Later, and Noted;
- view a list of *all* your notes and highlights via the app bar;
- share text, code, or links to resources with friends via email; and
- create your own list of resources, as you navigate online resources, beyond those pointed to in the Guided Tour.

Our first Guided Tour is based on this ebook. Kraig acts as a guide in two senses: he leads experienced web developers through the processes and best practices for building Windows Store apps, and he guides you through Microsoft’s extensive developer documentation, pointing you to the appropriate resources at each step in your app development process so that you can build your apps as effectively as possible.

Enjoy the app, and we look forward to providing more Guided Tours soon!

Errata & Book Support

We’ve made every effort to ensure the accuracy of this ebook and its companion content. Any errors that are reported after the book’s publication will be listed on http://aka.ms/BrockschmidtBook2/Errata. If you find an error that is not already listed, you can report it to us through the comments area of the same page.

If you need additional support, email Microsoft Press Book Support at mspinput@microsoft.com.

Please note that product support for Microsoft software and hardware is not offered through the previous addresses. For help with Microsoft software or hardware, go to http://support.microsoft.com. Support for developers can be found on the Windows Developer Center's support section, especially in the Building Windows Store apps with HTML5/JavaScript forum. There is also an active community on Stack Overflow for the winjs, windows-8 , windows-8.1, windows-store-apps, and winrt tags.

We Want to Hear from You

At Microsoft Press, your satisfaction is our top priority, and your feedback our most valuable asset. Please tell us what you think of this book at

http://aka.ms/tellpress

We know you're busy, so we've kept it short with just a few questions. Your answers go directly to the editors at Microsoft Press. (No personal information will be requested.) Thanks for your input!

Stay in Touch

Let's keep the conversation going! We're on Twitter: http://twitter.com/MicrosoftPress. And you can keep up with Kraig here: http://www.kraigbrockschmidt.com/blog.

Chapter 1

The Life Story of a Windows Store App: Characteristics of the Windows Platform

Paper or plastic? Fish or cut bait? To be or not to be? Standards-based or native? These are the questions of our time….

Well, OK, maybe most of these aren't the grist for university-level philosophy courses, but certainly the last one has been increasingly important for app developers. Standards-based apps are great because they run on multiple platforms; your knowledge and experience with standards like HTML5 and CSS3 are likewise portable. Unfortunately, because standards generally take a long time to produce, they always lag behind the capabilities of the platforms themselves. After all, competing platform vendors will, by definition, always be trying to differentiate! For example, while HTML5 has a standard for geolocation/GPS sensors and has started on working drafts for other forms of sensor input (like accelerometers, compasses, near-field proximity, and so on), native platforms already make these available. And by the time HTML's standards are in place and widely supported, the native platforms will certainly have added another set of new capabilities.

As a result, developers wanting to build apps around cutting-edge features—to differentiate from their own competitors!—must adopt the programming language and presentation technology imposed by each native platform or take a dependency on a third-party framework that tries to bridge the differences.

Bottom line: it's a hard choice.

Fortunately, Windows 8 and Windows 8.1 provide what I personally think is a brilliant solution for apps. Early on, the Windows team set out to solve the problem of making native capabilities—the system API, in other words—*directly* available to *any* number of programming languages, including JavaScript. This is what's known as the Windows Runtime API, or just *WinRT* for short (an API that's making its way onto the Windows Phone platform as well).

WinRT APIs are implemented according to a certain low-level structure and then "projected" into different languages—namely C++, C#, Visual Basic, and JavaScript—in a way that looks and feels natural to developers familiar with those languages. This includes how objects are created, configured, and managed; how events, errors, and exceptions are handled; how asynchronous operations work (to keep the user experience fast and fluid); and even the casing of method, property, and event names.

The Windows team also made it possible to write native apps that employ a variety of presentation technologies, including DirectX, XAML, and, in the case of apps written in JavaScript, HTML5 and CSS3.

This means that Windows gives you—a developer already versed in HTML, CSS, and JavaScript standards—the ability to *use what you know* to write fully native Windows Store apps using the WinRT API and still utilize web content! And I do mean *fully* native apps that both offer great content in themselves and integrate deeply with the surrounding system and other apps (unlike "hybrids" where one simply hosts web content within a thin, nearly featureless native shell). These apps will, of course, be specific to the Windows platform, but the fact that you don't have to learn a completely new programming paradigm is worthy of taking a week off to celebrate—especially because you won't have to spend that week (or more) learning a complete new programming paradigm!

It also means that you'll be able to leverage existing investments in JavaScript libraries and CSS template repositories: writing a native app doesn't force you to switch frameworks or engage in expensive porting work. That said, it is also possible to use multiple languages to write an app, leveraging the dynamic nature of JavaScript for app logic while leveraging languages like C# and C++ for more computationally intensive tasks. (See "Sidebar: Mixed Language Apps" later in this chapter, and if you're curious about language choice for apps more generally, see My take on HTML/JS vs. C/XAML vs. C++/DirectX on my blog.)

A third benefit is that as new web standards develop and provide APIs for features of the native platform, the fact that your app is written in the same language as the web will make it easier to port features from your native app to cross-platform web applications, if so desired.

Throughout this book we'll explore how to leverage what you know of standards-based web technologies—HTML, CSS, and JavaScript—to build great Windows Store apps for Windows 8.1. In the next chapter we'll focus on the basics of a working app and the tools used to build it. Then we'll look at fundamentals like the fuller anatomy of an app, incorporating web content, using controls and collections, layout, commanding, state management, and input (including sensors), followed by chapters on media, animations, contracts through which apps work together, live tiles and toast notifications, accessing peripheral devices, WinRT components (through which you can use other programming languages and the additional APIs they can access), expanding your reach through localization and accessibility, and working with the Windows Store. There is much to learn—it's a rich platform!

For starters, let's talk about the environment in which apps run and the characteristics of the platform on which they are built—especially the terminology that we'll depend on in the rest of the book (highlighted in *italics*). We'll do this by following an app's journey from the point when it first leaves your hands, through its various experiences with your customers, to where it comes back home for renewal and rebirth (that is, updates). For in many ways your app is like a child: you nurture it through all its formative stages, doing everything you can to prepare it for life in the great wide world. So it helps to understand the nature of that world!

Terminology note What we refer to as *Windows Store apps*, or sometimes just *Store apps,* are those that are acquired from the Windows Store and for which all the platform characteristics in this chapter (and book) apply. These are distinctly different from traditional *desktop applications* that are acquired through regular retail channels and installed through their own setup programs. Unless noted, then, an "app" in this book refers to a Windows Store app.

What about Windows Phone? The answer is yes! Windows Phone 8.1 supports writing apps with HTML, CSS, and JavaScript using much of what we'll be learning about in this book for Windows Store apps. However, this capability on Windows Phone happened very late in the production of this book, so I'm able to provide only a few details. I've included a brief overview later in this chapter under "Sidebar: Writing Windows Phone Apps with HTML, CSS, and JavaScript," with pointers to where you'll be able to find more information.

Leaving Home: Onboarding to the Windows Store

For Windows Store apps, there's really one port of entry into the world: customers always acquire, install, and update apps through the Windows Store. Developers and enterprise users can side-load apps, but for the vast majority of the people you care about, they go to the Windows Store and nowhere else.

This obviously means that an app—the culmination of your development work—has to get into the Store in the first place. This happens when you take your pride and joy, package it up, and upload it to the Store by using the Store/Upload App Packages command in Visual Studio. To do this, you'll need to create a developer account with the Store by using the Store > Open Developer Account command in Visual Studio Express (and this account works for both Windows and Windows Phone). Visual Studio Express and Expression Blend, which we'll also be using, are free tools you can obtain from http://dev.windows.com. This also works in Visual Studio Ultimate, the fuller, paid version of Microsoft's development environment.

The *package* itself is an *appx* file (.appx)—see Figure 1-1—that contains your app's code, resources, libraries, and a *manifest*, up to a combined limit of 8GB. The manifest describes the app (names, logos, etc.), the *capabilities* it wants to access (such as media libraries or specific devices like cameras), and everything else that's needed to make the app work (such as file associations, declaration of background tasks, and so on). Trust me, we'll become great friends with the manifest!

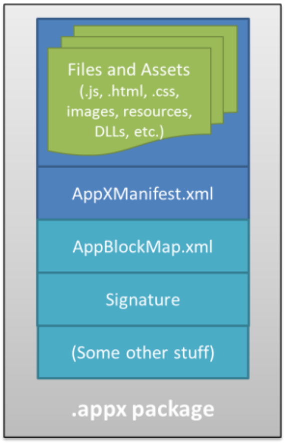


FIGURE 1-1 An appx package is simply a zip file that contains the app's files and assets, the app manifest, a signature, and a sort of table-of-contents called the blockmap. When uploading an app, the initial signature is provided by Visual Studio; the Windows Store will re-sign the app once it's certified.

> **Blockmaps make updates easy** The blockmap is hugely important for the customer experience of app updates (which are automatically installed by default in Windows 8.1) and, as a consequence, for your confidence in issuing updates. It describes how the app's files are broken up into 64K blocks. In addition to providing certain performance optimizations and security functions (like detecting whether a package has been tampered with), the blockmap describes exactly what parts of an app have been updated between versions so that the Windows Store need download *only those specific blocks* rather than the whole app anew. This greatly reduces the time and overhead that a user experiences when acquiring and installing updates. That is, even if your whole app package is 300MB, an update that affects a total of four blocks would mean your customers are downloading only 256 kilobytes.

The upload process will walk you through setting your app's name (which you do ahead of time using the Store > Reserve App Name and Store > Associate App with the Store commands in Visual Studio), choosing selling details (including price tier, in-app purchases, and trial periods), providing a description and graphics, and also providing notes to manual testers. After that, your app goes through a series of job interviews, if you will: background checks (malware scans and GeoTrust certification) and manual testing by a human being who will read the notes you provide (so be courteous and kind!). Along the way you can check your app's progress through the Windows Store Dashboard.[1]

The overarching goal with these job interviews (or maybe it's more like getting through airport security!) is to help users feel confident and secure in trying new apps, a level of confidence that isn't generally found with apps acquired from the open web. Because all apps in the Store are certified, signed, and subject to ratings and reviews, customers can trust all apps from the Store as they would

[1] All of the automated tests except the malware scans are incorporated into the Windows App Certification Kit, affectionately known as the WACK. This is part of the Windows SDK that is itself included with the Visual Studio Express/Expression Blend download. If you can successfully run the WACK during your development process, you shouldn't have any problem passing the first stage of onboarding. We'll learn about the WACK in Chapter 20.

trust those recommended by a reliable friend. Truly, this is wonderful news for most developers, especially those just getting started—it gives you the same access to the worldwide Windows market that has been previously enjoyed only by those companies with an established brand or reputation.

It's worth noting that because you set up pricing, trial versions, and in-app purchases during the on-boarding process, you'll have already thought about your app's relationship to the Store quite a bit! After all, the Store is where you'll be doing business with your app, whether you're in business for fame, fortune, fun, or philanthropy.

Indeed, this relationship spans the entire lifecycle of an app—from planning and development to distribution, support, and servicing. This is, in fact, why I've started this life story of an app with the Windows Store, because you really want to understand that whole lifecycle from the very beginning of planning and design. If, for example, you're looking to turn a profit from a paid app or in-app purchases, perhaps also offering a time-limited or feature-limited trial, you'll want to engineer your app accordingly. If you want to have a free, ad-supported app, or want to use a third-party commerce solution for in-app purchases (bypassing revenue sharing with the Store), these choices also affect your design from the get-go. And even if you're just going to give the app away to promote a cause or to just share your joy, understanding the relationship between the Store and your app is still important. For all these reasons, you might want to skip ahead and read the "Your App, Your Business" section of Chapter 20, "Apps for Everyone, Part 2," before you start writing your app in earnest. Also, take a look at the Certify your app topic on the Windows Developer Center.

Anyway, if your app hits any bumps along the road to certification, you'll get a report back with all the details, such as any violations of the App certification requirements for the Windows Store (part of the Windows Store agreements section). Otherwise, congratulations—your app is ready for customers!

Sidebar: The Store API and Product Simulator

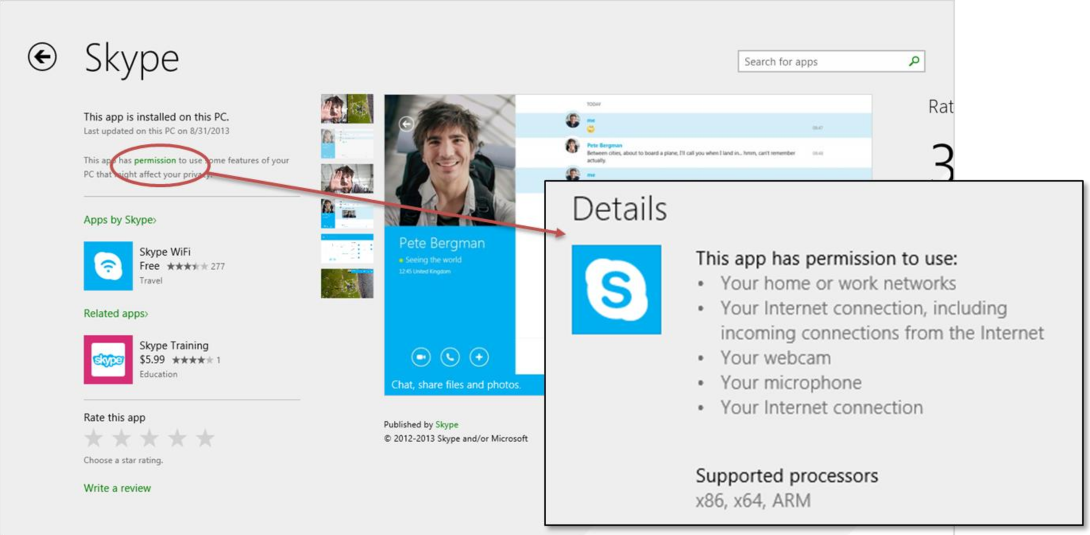
At run time, apps use the `Windows.ApplicationModel.Store.CurrentApp` class in WinRT to retrieve their product information from the Store (including in-app purchase listings), check license status, and prompt the user to make purchases (such as upgrading a trial or making an in-app purchase).

This begs a question: how can an app test such features before it's even in the Store? The answer is that during development, you use these APIs through the `CurrentApp`**`Simulator`** class instead. This is entirely identical to `CurrentApp` (and in the same namespace) except that it works against local data in an XML file rather than live Store data in the cloud. This allows you to simulate the various conditions that your app might encounter so that you can exercise all your code paths appropriately. Just before packaging your app and sending it to the Store, just change `CurrentAppSimulator` to `CurrentApp` and you're good to go. (If you forget, the simulator will simply fail on a non-developer machine, like those used by the Store testers, meaning that you'll fail certification.)

Discovery, Acquisition, and Installation

Now that your app is out in the world, its next job is to make itself known and attractive to potential customers. What's vital to understand here is what the Windows Store does and does not do for you. Its primary purpose is to provide a secure and trustworthy marketplace for distributing apps, updating apps, transparently handling financial transactions across global markets, and collecting customer reviews and basic telemetry (crash dumps)—which taken together is a fabulous service! That said, the mere act of onboarding an app to the Windows Store does not guarantee anyone will find it. That's one reality of publishing software that certainly hasn't changed. You still need to write great apps and you still need to market them to your potential customers, using advertising, social media, and everything else you'd do when trying to get a business off the ground.

That said, even when your app is found in the Store it needs to present itself well to its suitors. Each app in the Store has a *product description page* where people see your app description, promotional graphics, ratings and reviews, and the capabilities your app has declared in its manifest, as shown in Figure 1-2. That last bit means you want to be judicious in declaring your capabilities. A music player app, for instance, will obviously declare its intent to access the user's music library but usually doesn't need to declare access to the pictures library unless it has a good justification. Similarly, a communications app would generally ask for access to the camera and microphone, but a news reader app probably wouldn't. On the other hand, an ebook reader might declare access to the microphone *if* it had a feature to attach audio notes to specific bookmarks.



This app is installed on this PC.
Last updated on this PC on 8/31/2013

This app has permission to use some features of your PC that might affect your privacy.

Between cities, about to board a plane, I'll call you when I land in... hmm, can't remember actually.

Chat, share files and photos.

Choose a star rating.

Published by Skype
© 2012-2013 Skype and/or Microsoft

This app has permission to use:

- Your home or work networks
- Your Internet connection, including incoming connections from the Internet
- Your webcam
- Your microphone
- Your Internet connection

FIGURE 1-2 A typical app page in the Windows Store; by tapping the Permissions link at the upper left, the page pans to the Details section, which lists all the capabilities that are declared in the manifest (overlay). You can see here that Skype declares five different capabilities, all of which are appropriate for the app's functionality.

The point here is that what you declare must make sense to the user, and if there are any doubts you should clearly indicate the features related to those declarations in your app's description. Otherwise the user might really wonder just what your news reader app is going to do with the microphone and might opt for another app that seems less intrusive.[2]

The user will also see your app pricing, of course, and whether you offer a trial period. Whatever the case, if they choose to install the app (getting it for free, paying for it, or accepting a trial), your app now becomes fully incarnate on a real user's device. The appx package is downloaded to the device and installed automatically along with any dependencies, such as the *Windows Library for JavaScript* (see "Sidebar: What is the Windows Library for JavaScript?"). As shown in Figure 1-3, the Windows deployment manager creates a folder for the app, extracts the package contents to that location, creates *appdata* folders (local, roaming, and temp, which the app can freely access, along with settings files for key-value pairs), and does any necessary fiddling with the registry to install the app's tile on the *Start screen*, create file associations, install libraries, and do all those other things that are again described in the manifest. It can also start live tile updates before your app is even run the first time if you provide an appropriate URI in your manifest. There are no user prompts during this process—especially not those annoying dialogs about reading a licensing agreement!

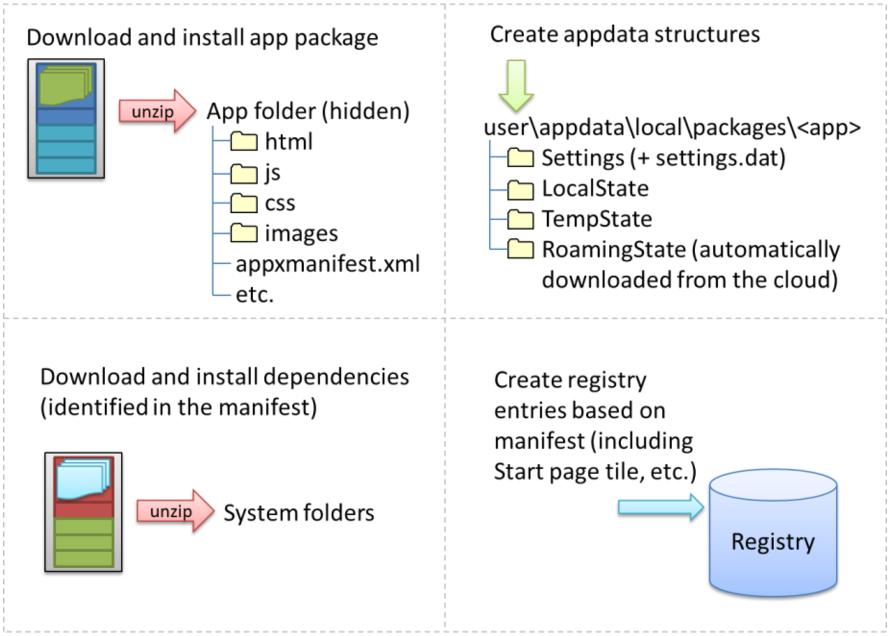

Download and install app package
unzip
App folder (hidden)
html
js
css
images
appxmanifest.xml
etc.
Create appdata structures
user\appdata\local\packages\<app>
Settings (+ settings.dat)
LocalState
TempState
RoamingState (automatically downloaded from the cloud)
Download and install dependencies (identified in the manifest)
unzip
System folders
Create registry entries based on manifest (including Start page tile, etc.)
Registry

FIGURE 1-3 The installation process for Windows Store apps; the exact sequence is unimportant. Any roaming state that exists in the cloud from the app on other devices is automatically downloaded as part of installation.

[2] The user always has the ability to disallow access to sensitive resources and devices at run time for those apps that have declared the intent. They can do this for a specific app through the system-provided Settings > Permissions command or generally through the various section under PC Settings > Privacy.

In fact, licensing terms are integrated into the Store; acquisition of an app implies acceptance of those terms. (However, it is perfectly allowable for apps to show their own license acceptance page on startup, as well as require an initial login to a service if applicable.) But here's an interesting point: do you remember the real purpose of all those lengthy, annoyingly all-caps licensing agreements that we pretend to read? Almost all of them basically say that you can install the software on only one machine. Well, that changes with Windows Store apps: instead of being licensed to a machine, they are licensed *to the user*, giving that user the right to install the app on up to eighty-one different devices.

In this way Store apps are a much more *personal* thing than desktop apps have traditionally been. They are less general-purpose tools that multiple users share and more like music tracks or other media that really personalize the overall Windows experience. So it makes sense that users can replicate their customized experiences across multiple devices, something that Windows supports through automatic roaming of app data and settings between those devices. (More on that later.)

In any case, the end result of all this is that the app and its necessary structures are wholly ready to awaken on a device as soon as the user taps a tile on the Start screen or launches it through features like Search and Share. And because the system knows about everything that happened during installation, it can also completely reverse the process for a 100% clean uninstall—completely blowing away the appdata folders, for example, and cleaning up anything and everything that was put in the registry. This keeps the rest of the system entirely clean over time, even though the user may be installing and uninstalling hundreds or thousands of apps. This is like the difference between having guests in your house and guests in a hotel. In your house, guests might eat your food, rearrange the furniture, break a vase or two, feed leftovers to the pets, stash odds and ends in the backs of drawers, and otherwise leave any number of irreversible changes in their wake (and you know desktop apps that do this, I'm sure!). In a hotel, on the other hand, guests have access only to a very small part of the whole structure, and even if they trash their room, the hotel can clean it out and reset everything as if the guest was never there.

Sidebar: What Is the Windows Library for JavaScript?

The HTML, CSS, and JavaScript code in a Windows Store app is only parsed, compiled, and rendered at run time. (See the "Playing in Your Own Room: The App Container" section below.) As a result, a number of system-level features for apps written in JavaScript, like controls, resource management, and default styling are supplied through the Windows Library for JavaScript, or *WinJS*, rather than through the Windows Runtime API. This way, JavaScript developers see a natural integration of those features into the environment they already understand, rather than being forced to use different kinds of constructs.

WinJS, for example, provides HTML implementations of a number of controls, meaning that instances of those controls appear as part of the DOM and can be styled with CSS like other intrinsic HTML elements. This is much more natural for developers than having to create an instance of some WinRT class, bind it to a separate HTML element, and style it through code or some other proprietary markup scheme. Similarly, WinJS provides an animations library built on

CSS that embodies the Windows user experience so that apps don't have to figure out how to re-create that experience themselves.

Generally speaking, WinJS is a *toolkit* that contains a number of independent capabilities that can be used together or separately. WinJS thus also provides helpers for common JavaScript coding patterns, simplifying the definition of namespaces and object classes, handling of asynchronous operations (that are all over WinRT) through *promises*, and providing structural models for apps, data binding, and page navigation. At the same time, it doesn't attempt to wrap WinRT unless there is a compelling scenario where WinJS can provide real value. After all, the mechanism through which WinRT is projected into JavaScript already translates WinRT structures into forms that are familiar to JavaScript developers.

Truth be told, you can write a Windows Store app in JavaScript without WinJS or just pick and choose what parts of the library are to your liking! But I think you'll find that it saves you all kinds of tedious work. In addition, WinJS is shared between every Store app written in JavaScript, and it's automatically downloaded and updated as needed when dependent apps are installed. We'll see nearly all of its features throughout this book, and you can always explore what's available through the Windows API reference (just scroll down to where you see WinJS and its subsidiary namespaces in the left-hand table of contents).

Sidebar: Third-Party Libraries

Apps can freely use third-party libraries by bundling them into their own app package, provided of course that the libraries use only the APIs available to Windows Store apps and follow necessary security practices that protect against script injection and other attacks. Many apps use jQuery 2.0 (see jQuery and WinJS working together in Windows Store apps); others use Angular.js, Box2D, Prototype, and so forth. Apps can also use third-party binaries, such as WinRT components, by bundling them with their app package. See this chapter's "Sidebar: Mixed Language Apps."

For an index of the ever-growing number of third-party solutions that are available for Windows Store apps, visit the Windows Partner Directory at http://services.windowsstore.com/.

Of course, bundling libraries and frameworks into your app package will certainly make that package larger, raising natural concerns about longer download times for apps and increased disk footprint. This has prompted requests for the ability to create shared framework packages in the Store (which Microsoft supports only for a few of its own libraries like WinJS). However, the Windows team devised a different approach. When you upload an app package to the Store, you will still bundle all your dependencies. On the consumer side, however, the Windows Store automatically detects when multiple apps share identical files. It then downloads and maintains a single copy of those files, making subsequent app installations faster and reducing overall disk footprint.

This way the user sees all the benefits of shared frameworks in a way that's almost entirely transparent to developers. The one requirement is that you should avoid, if possible, recompiling or otherwise modifying third-party libraries, especially larger ones like game engines, because you'll then produce different variations that must be managed separately.

Playing in Your Own Room: The App Container

Now, just as the needs of each day may be different when we wake up from our night's rest, Store apps can wake up—be activated—for any number of reasons. The user can, of course, tap or click the app's tile on the Start screen. An app can also be launched in response to charms like Search and Share, through file or protocol associations, and a number of other mechanisms. We'll explore these variants as we progress through this book. But whatever the case, there's a little more to this part of the story for apps written in JavaScript.

In the app's package folder are the same kind of source files that you see on the web: .html files, .css files, .js files, and so forth. These are not directly executable like .exe files for apps written in C#, Visual Basic, or C++, so something has to take those source files and produce a running app with them. When your app is activated, then, what actually gets launched is that *something*: a special *app host* process called wwahost.exe[3], as shown in Figure 1-4.

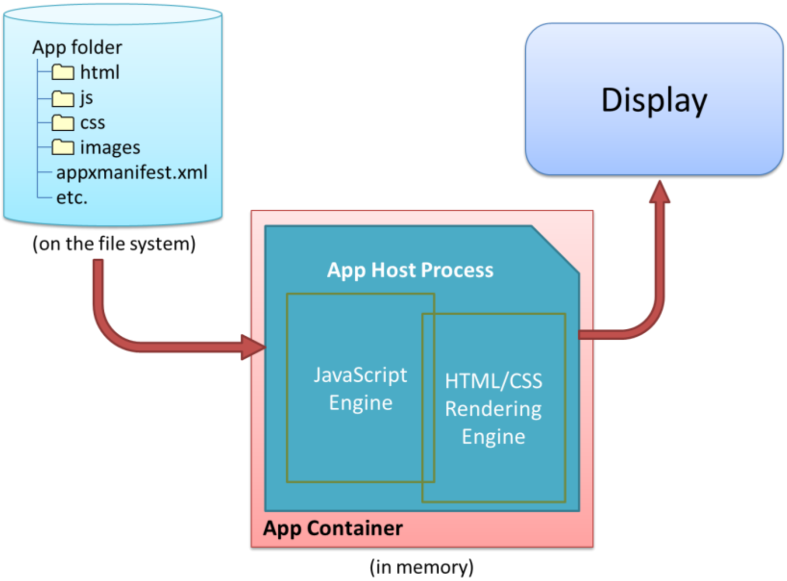


FIGURE 1-4 The app host is an executable (wwahost.exe) that loads, renders, and executes HTML, CSS, and JavaScript, in much the same way that a browser runs a web application.

[3] "wwa" is an old acronym for Windows Store apps written in JavaScript; some things just stick....

The app host is more or less Internet Explorer without the browser chrome—more in that your app runs on top of the same HTML/CSS/JavaScript engines as Internet Explorer, less in that a number of things behave differently in the two environments. For example:

- A number of methods in the DOM API are either modified or not available, depending on their design and system impact. For example, functions that display modal UI and block the UI thread are not available, like `window.alert`, `window.open`, and `window.prompt`. (Try `Windows.UI.Popups.MessageDialog` instead for some of these needs.)
- The `MSApp` object, which represents the app host's capabilities, provides many of the same methods like `requestAnimationFrame` as it does within Internet Explorer, and it also provides additional features for Store apps.
- The engines support additional methods and properties on a variety of elements—such as `audio`, `video`, and `canvas`—that are specific to being an app as opposed to a website.
- The default page of an app written in JavaScript runs in what's called the *local context* wherein JavaScript code has access to WinRT, can make cross-domain HTTP requests, and can access remote media (videos, images, etc.). However, you cannot load remote script (from `http[s]` sources, for example), and script is automatically filtered out of anything that might affect the DOM and open the app to injection attacks (e.g., `document.write` and `innerHTML` properties).
- Other pages in the app, as well as *webview* and `iframe` elements within a local context page, can run in the *web context* wherein you get web-like behavior (such as remote script) but don't get WinRT access nor cross-domain HTTP requests (though you can still use much of WinJS). Web context elements are generally used to host web content on a locally packaged page (like a map control, as we'll see in Chapter 2, "Quickstart"), or to load pages that are directly hosted on the web, while not allowing web pages to drive the app (a violation of Store policy).

For full details on all these behaviors, see HTML and DOM API changes list and HTML, CSS, and JavaScript features and differences on the Windows Developer Center, http://dev.windows.com. As with the app manifest, you should become good friends with the Developer Center.

All Store apps, whether hosted or not, run inside an environment called the *app container*. This is an insulation layer, if you will, that generally blocks local interprocess communication and either blocks or *brokers* access to system resources. The key characteristics of the app container are described as follows and illustrated in Figure 1-5:

- All Store apps run within a dedicated environment that cannot interfere with or be interfered by other apps, nor can apps interfere with the system.
- Store apps, by default, get unrestricted read/write access only to their specific appdata folders on the hard drive (local, roaming, and temp). Access to everything else in the file system (including removable storage) has to go through a broker. This gatekeeper provides access only if the app has declared the necessary capabilities in its manifest and/or the user has specifically allowed it. We'll see the specific list of capabilities shortly.

- Access to sensitive devices (like the camera, microphone, and GPS) and certain classes of peripherals is similarly controlled—the WinRT APIs that work with those devices will fail if the broker blocks those calls because the app hasn't declared the appropriate capability in its manifest or the user has denied permission at run time. And access to critical system resources, such as the registry, simply isn't allowed at all.
- Store apps cannot programmatically launch other apps by name or file path but can do so through file or URI scheme associations. Because these associations are ultimately under the user's control, there's no guarantee that such an operation will start a specific app. However, we do encourage app developers to use app-specific URI schemes that will effectively identify your specific app as a target. Technically speaking, another app could come along and register the same URI scheme (thereby giving the user a choice), but this is unlikely with a URI scheme that's closely related to the app's identity.
- Store apps are isolated from one another to protect from various forms of attack. This also means that some legitimate uses (like a snipping tool to copy a region of the screen to the clipboard) cannot be written as a Windows Store app; they must be a desktop application.
- Direct interprocess communication is blocked between Store apps, between Store apps and desktop applications, and between Store apps and local services. (Exceptions are made for side-loaded apps in enterprise environments, and in some debugging conditions.) Apps can still communicate through the cloud (web services, sockets, etc.), and many common tasks that require cooperation between apps—such as Search and Share—are handled through *contracts* in which those apps need not know any details about each other.

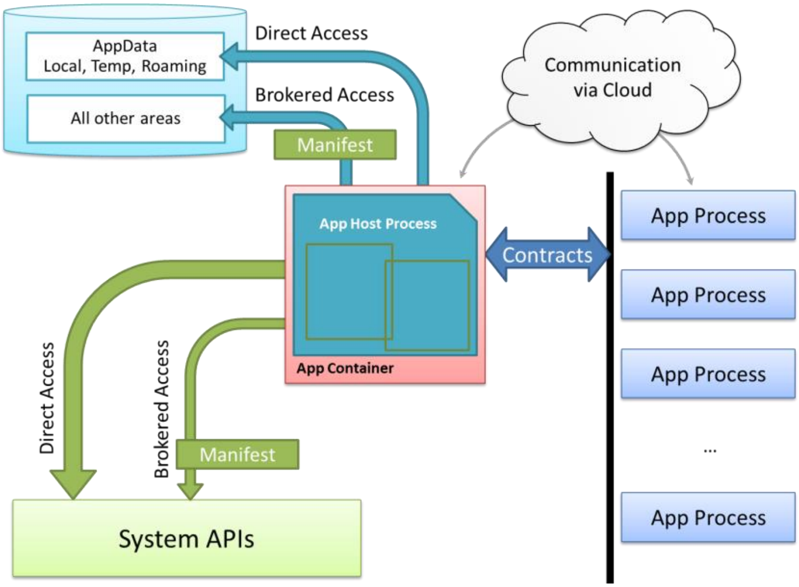


FIGURE 1-5 Process isolation for Windows Store apps.

Sidebar: Mixed Language Apps

Windows Store apps written in JavaScript can access only WinRT APIs directly. Apps or libraries written in C#, Visual Basic, and C++ also have access to a subset of Win32 and .NET APIs, as documented on Win32 and COM for Windows Store apps. Unfair? Not entirely, because you can write a *WinRT component* in those other languages that make functionality built with those other APIs available in the JavaScript environment (through the same *projection* mechanism that WinRT itself uses). Because these components are written in compiled languages, they can execute faster than the equivalent code written in JavaScript and also offer some degree of intellectual property protection (e.g., hiding algorithms either through obfuscation or by using a fully compiled language like C++).

Such *mixed language apps* thus use HTML/CSS for their presentation layer and JavaScript for some app logic while placing the most performance critical or sensitive code in compiled components. The dynamic nature of JavaScript, in fact, makes it a great language for gluing together multiple components. We'll see more in Chapter 18, "WinRT Components."

Note that when your main app is written in JavaScript, we recommend using only WinRT components written in C++ to avoid having two managed environments loaded into the same process. Using WinRT components written in C# or Visual Basic does work but incurs added memory overhead and risks memory leaks across multiple garbage collectors.

Sidebar: Assigned Access (Kiosk Mode)

In a variety of scenarios, such as a public kiosk, it's desirable to allow only a single app to run on a device and to prevent users from accessing most system capabilities like the Start screen. This feature, called *assigned access*, is configured through PC Settings > Accounts > Other Accounts > Set Up An Account For Assigned Access, where you associate a specific account with the one app that's allowed for that account. This can also be done through PowerShell scripts. For more information, see Assigned access: FAQ and Assigned Access Cmdlets.

Different Views of Life: Views and Resolution Scaling

So, the user has tapped on an app tile, the app host has been loaded into memory, and it's ready to get everything up and running. What does the user see?

The first thing that becomes immediately visible is the app's *splash screen*, which is described in its manifest with an image and background color. This system-supplied screen guarantees that at least *something* shows up for the app when it's activated, even if the app completely gags on its first line of code or never gets there at all. In fact, the app has 15 seconds to get its act together and display its main window, or Windows automatically gives it the boot (terminates it, that is) if the user switches away. This avoids having apps that hang during startup and just sit there like a zombie, where often the user can only kill it off by using that most consumer-friendly tool, Task Manager. (Yes, I'm being sarcastic—Task Manager is today much more user-friendly than it used to be.) Of course, we highly recommend that you get your app to an interactive state as quickly as possible; we'll look at some strategies for this in Chapter 3, "App Anatomy and Performance Fundamentals." That said, some apps will need more time to load, in which case you can create an *extended splash screen* by making the initial view of your main window look the same as the splash screen. This satisfies the 15-second time limit and lets you display other UI while the app is getting ready. Details are in Appendix B, "WinJS Extras."

Now, when a normally launched app comes up, it might share space with other apps, but often it has full command of the entire screen—well, not entirely. Windows reserves a one pixel space along every edge of the display through which it detects edge gestures, but the user doesn't see that detail. Your app still gets to draw in those areas, mind you, but it will not be able to detect pointer events therein. A small sacrifice for full-screen glory!

The purpose of those *edge gestures*—swipes from the edge of the screen toward the center—is to keep both system and app commands (like menus and other commanding UI) out of the way until needed—an aspect of the design principle called "content before chrome." This helps the user stay fully immersed in the app experience. The left and right edge gestures are reserved for the system, whereas the top and bottom are for the app. Swiping up from the top or bottom edges, as you've probably seen, brings up the *app bar* on the bottom of the screen where an app places most of its commands, and possibly also a *navigation bar* on the top. We'll see these in Chapter 9, "Commanding UI."

When running full-screen, the user's device can be oriented in either portrait or landscape, and apps can process various events to handle those changes. An app can also *lock* the orientation as needed, as well as specify its *supported orientations* in the manifest, which prevent Windows from switching to an unsupported orientation when the app is in the foreground. For example, a movie player will generally want to lock into landscape mode during playback such that rotating the device doesn't change the display. We'll see these details in Chapter 8, "Layout and Views."

What's also true is that your app might not always be running full-screen, even from first launch. In landscape mode, you app can share the screen real estate with perhaps as many as four other apps, depending on the screen size.[4] (See Figure 1-6.) In these cases it's helpful to refer to the app's display area as a *view*. By default, Windows allows the user to resize a view down to 500 pixels wide, and you can indicate in your manifest that your app supports going down to 320 pixels wide, increasing the likelihood that the user will keep it visible alongside other apps.

[4] For developers familiar with Windows 8, the distinct view states of filled, snapped, fullscreen-portrait, and fullscreen-landscape are replaced in Windows 8.1 with variable sizing.

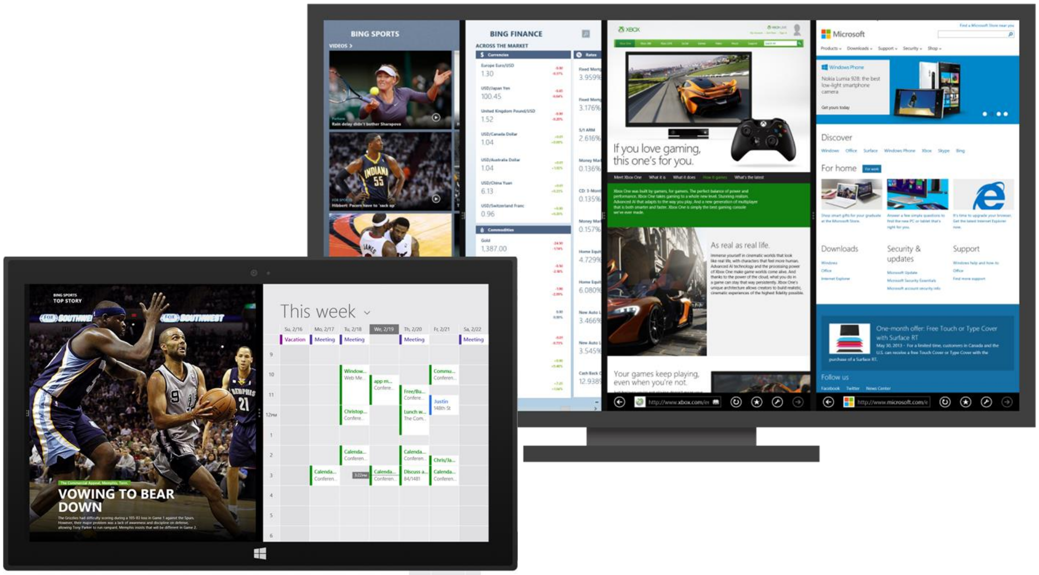

BING SPORTS
BING FINANCE
XBOX
Microsoft
If you love gaming, this one's for you.
As real as real life.
Your games keep playing, even when you're not.
This week
VOWING TO BEAR DOWN

FIGURE 1-6 Various arrangements of Windows Store apps—a 50/50 split view on the smaller screen (in front), and four apps sharing the screen on a large monitor (behind). Depending on the minimum size indicated in their manifests, apps must be prepared to show properly in any width and orientation, a process that generally just involves visibility of elements and layout and that can often be handled entirely within CSS media queries.

In practical terms, variable view sizing means that your layout must be *responsive*, as it's called with web design, accommodating different aspect ratios and different widths and heights. Generally speaking, most if not all of this can be handled through CSS media queries using the `orientation` feature (to detect portrait or landscape aspect ratio) along with `min-width` and `max-width`. We'll see distinct examples in Chapter 2. It's also worth noting that when one app launches another through associations or other contracts, it can specify whether and how it wants its view to remain visible. This makes it possible to really have two apps working together side by side for a shared purpose. Indeed, the default behavior when the user activates a hyperlink in an app is that the browser will open in a 50/50 split view alongside the app.

Apps can also programmatically spawn *multiple views*, which the user can size and position independently of one another, even across multiple monitors. (For this reason, views cannot depend on their relative placement and should represent separate functions of the app.) An app can even *project* a secondary view such that it always shows full-screen on a second monitor, as is appropriate for an app that shows speaker notes in one view and a presentation in the other.

In narrow views, especially the optional 320px minimum, apps will often change the presentation of content or its level of detail. For instance, in portrait aspect ratios (height > width), horizontally oriented lists are typically switched to a condensed vertical orientation. But don't be nonchalant about this: consciously design views for every page in your app and design them well. After all, users like to look at things that are useful and beautiful, and the more an app does this with all its views, the more likely it is that users will again keep that app visible even while they're working in another.

Another key point for all views is that they aren't mode changes. When a view is resized but still visible, or when orientation changes, the user is essentially saying, "Please stand over here in this doorway, or please lean sideways." So the app should never change what it's doing (like switching from a game board to a high score list) when updating a view's layout; it should just present a view appropriately for that width and orientation.

With all views, an app should make good use of all its available screen real estate. Across your customer base, your app will be run on many different displays, anywhere from 1024x768 (the minimum hardware requirement) to resolutions like 2560x1440 and beyond that are becoming more common. The guidance here is that views with fixed content (like a game board) will generally scale in size to fill the available space, whereas views with variable content (like a news reader) will generally show more content. For more details, refer to Guidelines for window sizes and scaling to screens and the Windows Design Center.

It might also be true that you're running on a high-resolution device that has a very small screen (high *pixel density*), such as the 10.6" Surface Pro that has a 1920x1200 resolution, a 13.3" QHD device, or an 8" device with an even sharper screen. Fortunately, Windows does automatic *scaling* such that the app still sees a 1366x768 display (more or less) through CSS, JavaScript, and the WinRT API. In other words, you almost don't have to care. The only concern is bitmap (raster) graphics, with which you'll ideally provide scale-specific variants as we'll see in Chapters 3 and 8. Fortunately, the Windows Store automatically manages resources across scale factors and languages (for localization) such that it downloads only those resources that a user needs for their configuration.

As a final note, when an app is activated in response to a contract like Search or Share, its initial view might not be a typical app view at all but rather its specific landing page for that contract that overlays the current foreground app. We'll see these details in Chapter 15, "Contracts."

Sidebar: Single-Page vs. Multipage Navigation

When you write a web application with HTML, CSS, and JavaScript, you typically end up with a number of different HTML pages and navigate between them using `<a href>` tags or by setting `document.location`.

This is all well and good and works in a Windows Store app, but it has several drawbacks. One is that navigation between pages means reloading script, parsing a new HTML document, and parsing and applying CSS again. Besides obvious performance implications, this makes it difficult to share variables and other data between pages, as you need to either save that data in persistent storage or stringify the data and pass it on the URI.

Furthermore, switching between pages is visually abrupt: the user sees a blank screen while the new page is being loaded. This makes it difficult to provide a smooth, animated transition between pages as generally seen within the Windows personality—it's the antithesis of "fast and fluid" and guaranteed to make designers cringe.

To avoid these concerns, apps written in JavaScript are typically structured as a single HTML page (basically a container `div`) into which different bits of HTML content, called *page controls* in WinJS, are loaded into the DOM at run time, similar to how AJAX works. This *DOM replacement* scheme has the benefit of preserving the script context and allows for transition animations through CSS and/or the WinJS animations library. We'll see the details in Chapter 3.

Those Capabilities Again: Getting to Data and Devices

At run time, now, even inside the app container, your app has plenty of room to play and to delight your customers. It can employ web content and connectivity to its heart's content, either directly hosting content in its layout with the webview control or obtaining data through HTTP requests or background transfers (Chapter 4). An app has many different controls at its disposal, as we'll see in Chapters 5, 6, and 7, and can style them however it likes from the prosaic to the outrageous. Similarly, designers have the whole gamut of HTML and CSS to work with for their most fanciful page layout ideas, along with a Hub control that simplifies a common home page experience (Chapter 8). An app can work with commanding UI like the app bar (Chapter 9), manage state and user data (Chapters 10 and 11), and receive and process *pointer events*, which unify touch, mouse, and stylus (Chapter 12—with these input methods being unified, you can design for touch and get the others for free; input from the physical and on-screen keyboards are likewise unified). Apps can also work with *sensors* (Chapter 12), rich media (Chapter 13), animations (Chapter 14), contracts (Chapter 15), *tiles and notifications* (Chapter 16), and various devices and printers (Chapter 17). They can optimize performance and extend their capabilities through WinRT components (Chapter 18), and they can adapt themselves to different markets (Chapter 19), provide accessibility (Chapter 19), and work with various monetization options like advertising, trial versions, and in-app purchases (Chapter 20).

Note For a more complete mapping between different app features and the chapters of this book, see the "Feature Roadmap and Cross-Reference" section at the end of this chapter.

Many of these features and their associated APIs have no implications where user privacy is concerned, so apps have open access to them. These include controls, touch/mouse/stylus input, keyboard input, and sensors (like the accelerometer, inclinometer, and light sensor). The appdata folders (local, roaming, and temp) that were created for the app at installation are also openly accessible. Other features, however, are again under more strict control. As a person who works remotely from home, for example, I really don't want my webcam turning on unless I specifically tell it to—I may be calling into a meeting before I've had a chance to wash up! Such devices and other protected system features, then, are again controlled by a broker layer that will deny access if (a) the capability is not declared in the manifest, *or* (b) the user specifically disallows that access at run time. Those capabilities are listed in the following table:

Capability	**Description**	**Prompts for user consent at run time**
Internet (Client)	Outbound access to the Internet and public networks (which includes making requests to servers and receiving information in response).[5]	No
Internet (Client & Server) (superset of *Internet (Client)*; only one needs to be declared)	Outbound and inbound access to the Internet and public networks (inbound access to critical ports is always blocked).	No
Private Networks (Client & Server)	Outbound and inbound access to home or work intranets (inbound access to critical ports is always blocked).	No
Music Library *Pictures Library* *Video Library*[6]	Read/write access to the user's Music/Pictures/Videos area on the file system (all files).	No
Removable Storage	Read/write access to files on removable storage devices for specifically declared file types.	No
Microphone	Access to microphone audio feeds (includes microphones on cameras).	Yes
Webcam	Access to camera audio/video/image feeds.	Yes
Location	Access to the user's location via GPS.	Yes
Proximity	The ability to connect to other devices through near-field communication (NFC).	No
Enterprise Authentication	Access to intranet resources that require domain credentials; not typically needed for most apps. Requires a corporate account in the Windows Store.	No
Shared User Certificates	Access to software and hardware (smart card) certificates. Requires a corporate account in the Windows Store.	Yes, in that the user must take action to select a certificate, insert a smart card, etc.
[other arbitrary devices and peripherals]	Access to specific devices via USB, HID, Bluetooth, WiFi Direct, and NFC communication transports.	Yes

When user consent is involved, calling an API to access the resource in question will prompt for user consent, as shown in Figure 1-7. If the user accepts, the API call will proceed; if the user declines, the API call will return an error. Apps must accordingly be prepared for such APIs to fail, and they must then behave accordingly.

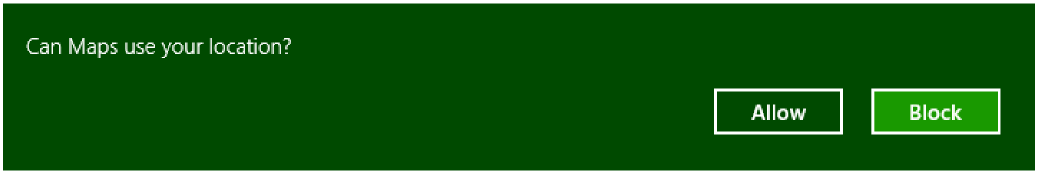

Can Maps use your location?

FIGURE 1-7 A typical user consent dialog that's automatically shown when an app first attempts to use a brokered capability. This will happen only once within an app, but the user can control their choice through the Settings charm's Permissions command for that app or through PC Settings > Privacy.

[5] Note that network capabilities are not necessary to receive push notifications because those are received by the system and not the app.

[6] The *Documents Library* capability that was present in Windows 8 no longer exists in Windows 8.1 because the scenarios that actually needed it can be handled through file pickers.

When you first start writing apps, really keep the manifest and these capabilities in mind—if you forget one, you'll see APIs failing even though all your code is written perfectly (or was copied from a working sample). In the early days of building the first Windows Store apps at Microsoft, we routinely forgot to declare the *Internet (Client)* capability, so even things like getting to remote media with an `img` element or making a simple call to a web service would fail. Today the tools do a better job of alerting you if you've forgotten a capability, but if you hit some mysterious problem with code that you're sure should work, especially in the wee hours of the night, check the manifest!

We'll encounter many other sections of the manifest besides capabilities in this book. For example, you can provide a URI through which Windows can request tile updates so that your app has a live tile experience even before it is run the first time. The removable storage capability requires you to declare the specific file types for your app (otherwise access will generally be denied). The manifest also contains *content URIs*: specific rules that govern which URIs are known and trusted by your app and can thus act to some degree on the app's behalf. Furthermore, the manifest is where you declare things like your supported orientations, *background tasks* (like playing audio, tracking geofences, or handling real-time communication), contract behaviors (such as which page in your app should be brought up in response to being invoked via a contract), custom protocols, and the appearance of tiles and notifications. You and your app will become bosom buddies with the manifest.

The last note to make about capabilities is that while programmatic access to the file system is controlled by certain capabilities, the user can always point your app to other noncritical areas of the file system—and any type of file—through the file picker UI. (See Figure 1-8.) This explicit user action is taken as consent for your app to access that particular file or folder (depending on what you're asking for). Once you're app is given this access, you can use certain APIs to record that permission so that you can get to those files and folders the next time your app is launched.

In summary, the design of the manifest and the brokering layer is to ensure that the user is always in control where anything sensitive is concerned, and as your declared capabilities are listed on your app's description page in the Windows Store, the user should never be surprised by your app's behavior.

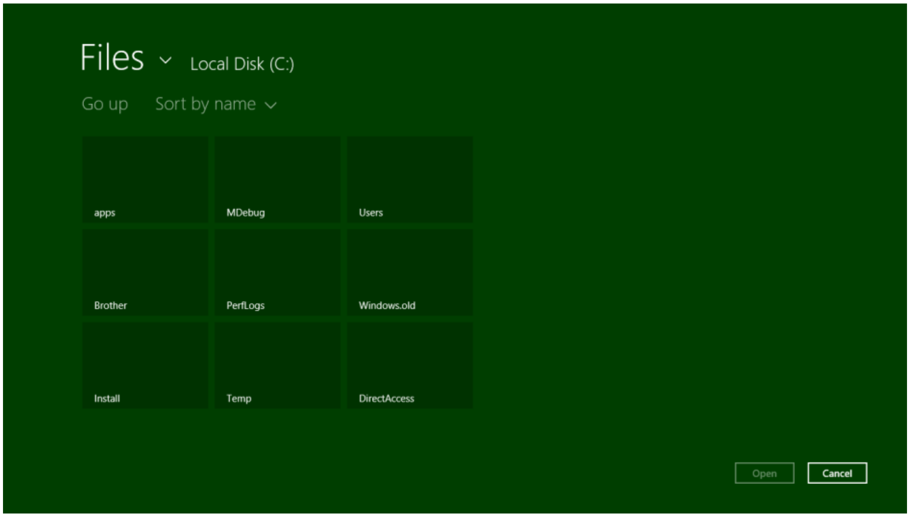


FIGURE 1-8 Using the file picker UI to access other parts of the file system from within a Store app, such as folders on a drive root (but not protected system folders). This is done by tapping the down arrow next to "Files." Typically, the file picker will look much more interesting when it's pointing to a media library!

Taking a Break, Getting Some Rest: Process Lifecycle Management

Whew! We've covered a lot of ground already in this first chapter—our apps have been busy, busy, busy, and we haven't even started writing any code yet! In fact, apps can become really busy when they implement certain sides of contracts. If an app declares itself as a Share *target*, a Contact or Appointments *provider*, or a File Picker *provider* in its manifest (among other things), Windows will activate the app in response to the appropriate user actions. For example, if the user invokes the Share charm and picks your app as a Share target, Windows will activate the app with an indication of that purpose. In response, the app displays its specific share UI—not the whole app—and when that task is complete, Windows will shut your app down again (or send it to the background if it was already running) without the need for additional user input.

This automatic shutdown or sending the app to the background are examples of built-in *lifecycle management* for Windows Store apps that helps conserve power and optimize battery life. One reality of traditional multitasking operating systems is that users typically leave a bunch of apps running, all of which consume power. This makes sense with desktop apps because many of them can be at least partially visible simultaneously. But for Store apps, the system is boldly taking on the job itself and using the full-screen nature of those apps (or the limited ability to share the screen) to its advantage.

Apps typically need to be busy and active only when the user can see them (in whatever view). When most apps are no longer visible, there is really little need to keep them idling. It's better to just turn them off, give them some rest, and let the visible apps utilize the system's resources.

So when an app goes to the background, Windows will automatically *suspend* it after about 5 seconds (according to the wall clock). The app is notified of this event so that it can save whatever state it needs to (which I'll describe more in the next section). At this point the app is still in memory, with all its in-memory structures intact, but it will simply not be scheduled for any CPU time. (See Figure 1-9.) This is very helpful for battery life because most desktop apps idle like a gasoline-powered car, still consuming a little CPU in case there's a need, for instance, to repaint a portion of a window. Because a Windows Store app in the background is completely obscured, it doesn't need to do such small bits of work and can be effectively frozen. In this sense it is much more like a modern electric vehicle that can be turned on and off as often as necessary to minimize power consumption.

If the user then switches back to the app (in whatever view, through whatever gesture), it will be scheduled for CPU time again and *resume* where it left off (adjusting its layout for the view, of course). The app is also notified of this event in case it needs to re-sync with online services, update its layout, refresh a view of a file system library, or take a new sensor reading because any amount of time might have passed since it was suspended. Typically, though, an app will not need to reload any of its own state because it was in memory the whole time.

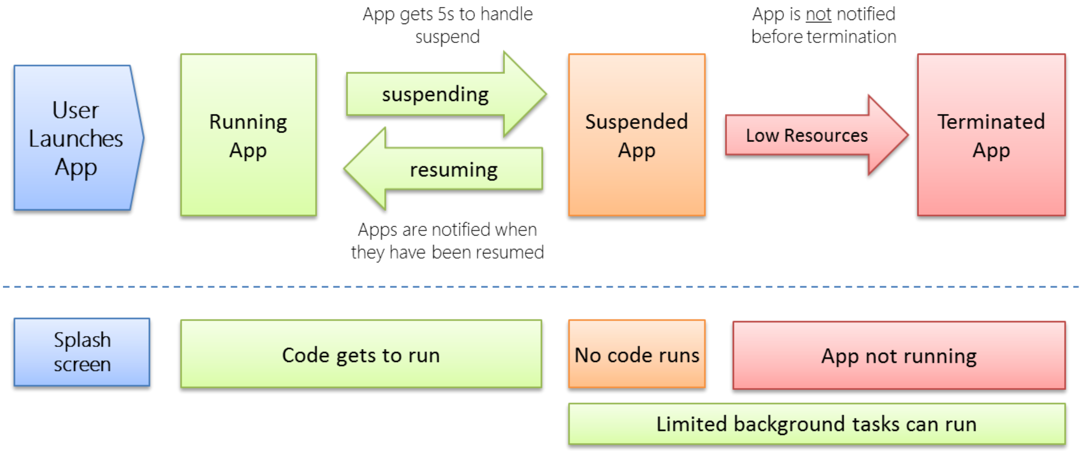

App gets 5s to handle suspend
App is not notified before termination
User Launches App
Running App
suspending
resuming
Suspended App
Low Resources
Terminated App
Apps are notified when they have been resumed
Splash screen
Code gets to run
No code runs
App not running
Limited background tasks can run

FIGURE 1-9 Process lifetime states for Windows Store apps.

There are a couple of exceptions to this. First, Windows provides a *background transfer* API—see Chapter 4, "Web Content and Services"—to offload downloads and uploads from app code, which means apps don't have to be running for such purposes. Apps can also ask the system to periodically update *live tiles* on the Start screen with data obtained from a service, or they can employ *push notifications* (through the Windows Push Notification Service, WNS) that Windows can handle directly—see Chapter 16, "Alive with Activity." Second, certain kinds of apps do useful things when they're not visible, such as audio players, communications apps, or those that need to take action when specific system events occur (like a network change, user login, etc.). With audio, as we'll see in Chapter 13, "Media," an app specifies background audio in its manifest (where else!) and sets certain properties on the appropriate audio elements. This allows it to continue running in the background. With system events, as we'll also see in Chapter 16, an app declares background tasks in its manifest that are tied to specific functions in their code. In this case, Windows will run that task (while the app is suspended) when an appropriate trigger occurs. This is shown at the bottom of Figure 1-9.

Over time, of course, the user might have many apps in memory, and most of them will be suspended and consume very little power. Eventually there will come a time when the foreground app—especially one that's just been launched—needs more memory than is available. In this case, Windows will automatically *terminate* one or more apps, dumping them from memory. (See Figure 1-9 again.)

But here's the rub: unless a user explicitly closes an app—by using Alt+F4 or a top-to-bottom swipe-and-hold, because Windows Store policy specifically disallows apps with their own close commands or gestures—she still rightly thinks that the app is running. If she activates it again (as from its tile), she will expect to return to the same place she left off. For example, a game should be in the same place it was before (though automatically paused), a reader should be on the same page, and a video should be paused at the same time. Otherwise, imagine the kinds of ratings and reviews your app will be getting in the Windows Store!

So you might say, "Well, I should just save my app's state when I get terminated, right?" Actually, no: your app will *not* be notified when it's terminated. Why? For one, it's already suspended at that time, so no code will run. In addition, if apps need to be terminated in a low memory condition, the last thing you want is for apps to wake up and try to save state which might require even more memory! It's imperative, as hinted before, that apps save their state when being suspended and ideally even at other checkpoints during normal execution. So let's see how all that works.

Remembering Yourself: App State and Roaming

To step back for a moment, one of the key differences between traditional desktop applications and Windows Store apps is that the latter are inherently stateful. That is, once they've run the first time, they remember their state across invocations (unless explicitly closed by the user or unless they provide an affordance to reset the state explicitly). Some desktop applications work like this, but most suffer from a kind of identity crisis when they're launched. Like Gilderoy Lockhart in *Harry Potter and the Chamber of Secrets*, they often start up asking themselves, "Who am I?"[7] with no sense of where they've been or what they were doing before.

Clearly this isn't a good idea with Store apps whose lifetime is being managed automatically. From the user's point of view, apps are always running (even if they're not). It's therefore critical that apps first manage settings that are always in effect and then also save their session state when being

[7] For those readers who have not watched this movie all the way through the credits, there's a short vignette at the very end. During the movie, Lockhart—a prolific, narcissistic, and generally untruthful autobiographer—loses his memory from a backfiring spell. In the vignette he's shown in a straitjacket on the cover of his newest book, *Who am I?*

suspended. This way, if the app is terminated and restarted, it can reload that session state to return to the exact place it was before. (An app receives a flag on startup to indicate its previous execution state, which determines what it should do with saved session state. Details are in Chapter 3.)

There's another dimension to statefulness: remember from earlier in this chapter that a user can install the same Windows Store app on up to eighty-one devices? Well, that means that an app, depending on its design, of course, can also be stateful *between* those devices. That is, if a user pauses a video or a game on one device or has made annotations to a book or magazine on one device, the user will naturally want to be able to go to another device and pick up at exactly the same place.

Fortunately, Windows makes this easy—really easy, in fact—by automatically roaming app settings and state, along with Windows settings, between trusted devices on which the user is logged in with the same Microsoft account, as shown in Figure 1-10. When roaming state exists, it's automatically downloaded as part of app installation, so it's there when the app is first launched on a new device.

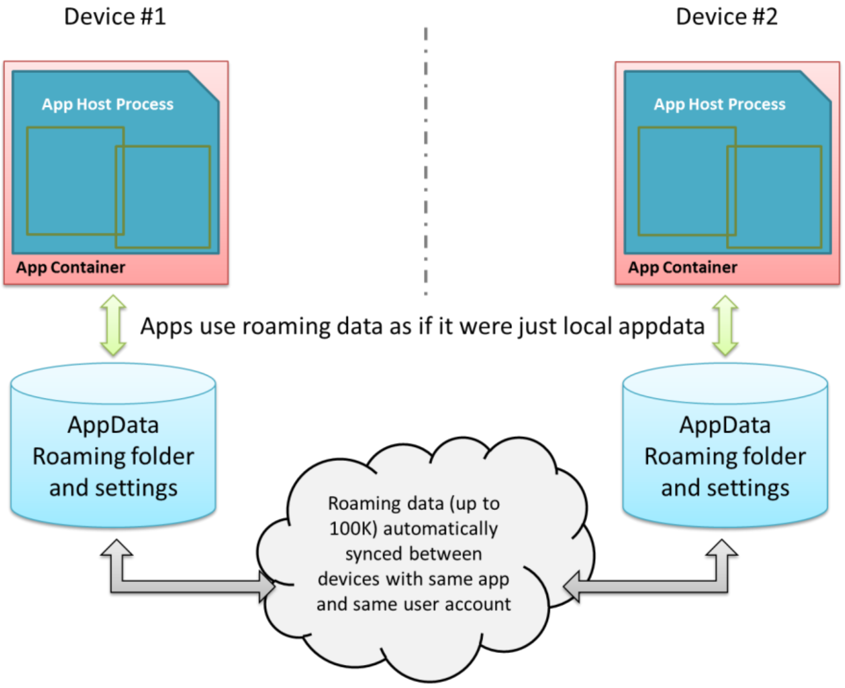

Device #1
Device #2
App Host Process
App Host Process
App Container
App Container
Apps use roaming data as if it were just local appdata
AppData Roaming folder and settings
AppData Roaming folder and settings
Roaming data (up to 100K) automatically synced between devices with same app and same user account

FIGURE 1-10 Automatic roaming of app roaming data (folder contents and settings) between devices.

They key here is understanding how and where an app saves its state. (We already know when.) If you recall, there's one place on the file system where an app has unrestricted access: its appdata folder. Within that folder, Windows automatically creates subfolders named LocalState, RoamingState, and TempState when the app is installed. (I typically refer to them without the "State" suffix.) The app can programmatically get to any of these folders at any time and can create in them all the files and subfolders that will fulfill its heart's desire. There are also APIs for managing individual Local and

Roaming settings (key-value pairs), along with groups of settings called *composites* that are always written to, read from, and roamed as a unit. (These are useful when implementing the app's Settings features for the Settings charm, as covered in Chapter 10, "The Story of State, Part 1.")

Now, although the app can write as much as it wants to the appdata areas (up to the capacity of the file system), Windows will automatically roam the data in your Roaming sections only if you stay below an allowed quota (~100K, but there's an API for that). If you exceed the limit, the data will still be there locally but none of it will be roamed. Also be aware that cloud storage has different limits on the length of filenames and file paths as well as the complexity of the folder structure. So keep your roaming state small and simple. If the app needs to roam larger amounts of data, use a secondary web service like OneDrive or Windows Azure Mobile Services (which we'll see more of in Chapter 16).

The app really needs to decide what kind of state is local to a device and what should be roamed. Generally speaking, any kind of settings, data, or cached resources that are device-specific should always be local (and Temp is also local), whereas settings and data that represent the user's interaction with the app are potential roaming candidates. For example, an email app that maintains a local cache of messages would keep those local but would roam account settings (sans passwords; see Tip below) so that the user can configure the app on one device and have that configuration apply on every other device. The app would probably also maintain a per-device setting for how it downloads or updates emails so that the user can minimize network/radio traffic on a mobile device. A media player, similarly, would keep local caches that are dependent on the specific device's display characteristics, and it would roam playlists, playback positions, favorites, and other such settings (should the user want that behavior, of course).

> **Tip** For passwords in particular, always store them in the Credential Locker (see Chapter 4). If the user allows password roaming (PC Settings > OneDrive > Sync Settings > Other Settings > Passwords), the locker's contents will be roamed automatically.

When state is roamed, know that there's a simple "last writer wins" policy where collisions are concerned. If you run the same app on two devices at the same time, don't expect there to be any fancy merging or swapping of state. After all kinds of tests and analysis, Microsoft's engineers finally decided that simplicity was best!

Along these same lines, if a user installs an app, roams some settings, uninstalls the app, and then within "a reasonable time" reinstalls the app, she will find that those settings are still in place. This makes sense, because it would be too draconian to blow away roaming state in the cloud the moment she just happened to uninstall an app on all her devices. There's no guarantee of this behavior, mind you, but Windows will apparently retain roaming state for an app for some time.

Sidebar: Local vs. Temp Data

For local caching purposes, an app can use either local or temp storage. The difference is that local data is always under the app's control. Temp data, on the other hand, can be deleted if the user runs the Disk Cleanup utility. Local data is thus best used to support an app's functionality, and temp data is used to support run-time optimization at the expense of disk space.

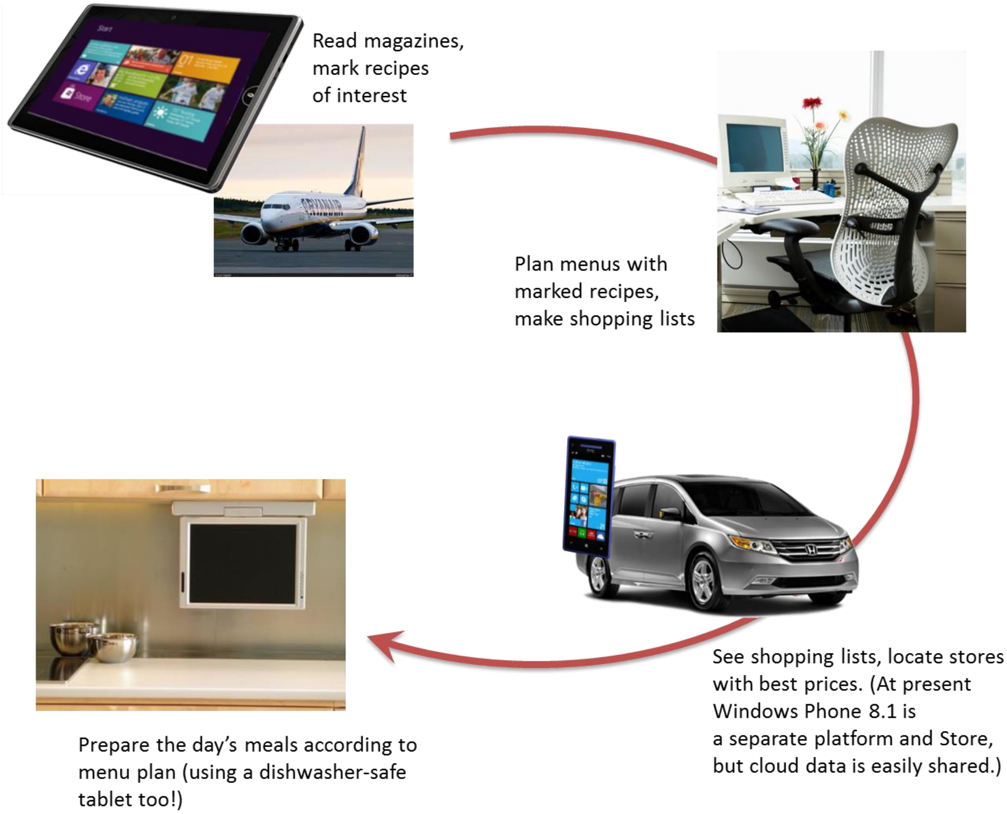
For Windows Store apps written in HTML and JavaScript, you can also use existing caching mechanisms like HTML5 local storage, IndexedDB, and app cache, along with third-party database options like SQLite, which act like local storage.

Sidebar: The Opportunity of Per-User Licensing and Data Roaming

Details aside, I personally find the cross-device roaming aspect of the platform very exciting, because it enables the developer to think about apps as something beyond a single-device or single-situation experience. As I mentioned earlier, a user's collection of apps is highly personal and it personalizes the device; apps themselves are licensed to the user and not the device. In that way, we as developers can think about each app as something that projects itself appropriately onto whatever device and into whatever context it finds itself. On some devices it can be oriented for intensive data entry or production work, while on others it can be oriented for consumption or sharing. The end result is an overall app experience that is simply more *present* in the user's life and appropriate to each context.

An example scenario is illustrated below, where an app can have different personalities or flavors depending on user context and how different devices might be used in that context. It might seem rather pedestrian to think about an app for meal planning, recipe management, and shopping lists, but that's something that happens in a large number of households worldwide. Plus it's something that my wife would like to see me implement if I ever get around to writing more code than text!

This, to me, is the real manifestation of the next era of personal computing, an era in which personal computing expands well beyond, yet still includes, a single device experience and includes embracing the power of cloud-based resources for your personal needs. Devices, then, are merely viewports for your apps and data, each viewport having a distinct role in the larger story of how your move through and interact with the world at large.


Read magazines, mark recipes of interest
Plan menus with marked recipes, make shopping lists
See shopping lists, locate stores with best prices. (At present Windows Phone 8.1 is a separate platform and Store, but cloud data is easily shared.)
Prepare the day's meals according to menu plan (using a dishwasher-safe tablet too!)

Sidebar: Writing Windows Phone Apps with HTML, CSS, and JavaScript

The diagram in the previous sidebar shows that a Windows Phone can be part of an overall app presence across multiple devices. At present, apps for Windows 8.1 and Windows Phone 8.1 are still separate entities, each with their own identity, app package, and storefront. As such, they do not as yet share common roaming data. They can, however, use common backend services such as Windows Azure Mobile Services for data sharing and managing push notifications. You can also use a common live tile service as the tile update XML format is the same on both.

Indeed, much is the same on the platform level, including the fact that Windows Phone 8.1 supports apps written in HTML, CSS, and JavaScript, exactly like those we're talking about in this book. With the appropriate update to Visual Studio 2013, you can create solutions in which you can easily share code between Windows and Windows Phone projects.

On the API level, you'll find that most of WinRT is identical between both platforms, excepting those areas where the underlying hardware can't support a particular feature or the form factor isn't suitable. For example, inking, printing, USB/HID device access, the built-in camera capture

UI, and various aspects of a full file system are not available on Windows Phone. Similarly, phone-specific features like the Wallet API are not available on Windows.

WinJS is available on both platforms as well but differs somewhat in the available controls, which I'll detail in Chapter 5. That said, the app model that WinJS presents is identical across both, as are the mechanisms for dealing with async operations and data binding.

Suffice it to say that much of what you'll learn in this Second Edition will be completely applicable to Windows Phone projects, which is great news because it means you can use all the skills you develop to write apps for a much larger combined market! And as the platforms become increasingly converged in the future, your investments of today will continue to bear fruit.

As I mentioned at the beginning of this chapter, the ability to write Windows Phone apps with HTML, CSS, and JavaScript came quite late in the production cycle of this book, so our focus here will still be Windows Store apps. That said, the Windows Developer Center will have more information for Windows Phone Apps written in JavaScript, such as clear documentation on the variations in WinJS and WinRT. I'll also be working on content for both the Windows/Windows Phone developer blog (where you'll also find announcements about the platforms) and my personal blog. Hope to see you there!

Coming Back Home: Updates and New Opportunities

If you're one of those developers that can write a perfect app the first time, I have to ask why you're actually reading this book! Fact of the matter is that no matter how hard we try to test our apps before they go out into the world, our efforts pale in comparison to the kinds of abuse that customers will heap on them. To be more succinct: expect problems. An app might crash under circumstances we never predicted, or there just might be usability problems because people are finding creative ways to use the app outside of its intended purpose.

Fortunately, the Windows Store dashboard—go to http://dev.windows.com and click the Dashboard tab at the top—makes it easy for you get the kind of feedback that has traditionally been very difficult to obtain. For one, the Store maintains *ratings and reviews* for every app, which will be a source of valuable insight into how well your app fulfills its purpose in life and a source of ideas for your next release. And you might as well accept it now: you're going to get praise (if you've done a decent job), and you're going to get criticism, even a good dose of nastiness (even if you've done a decent job!). Don't take it personally—see every critique as an opportunity to improve, and be grateful that people took the time to give feedback. As a wise man once said upon hearing of the death of his most vocal critic, "I've just lost my best friend!"

The Store will also provide you with *crash analytics* so that you can specifically identify problem areas in your app that evaded your own testing. This is incredibly valuable—maybe you're already clapping your hands in delight!—because if you've ever wanted this kind of data before, you've had to implement the entire mechanism yourself. No longer. This is one of the valuable services you get in exchange for your annual registration with the Store. Of course, crash analytics are only the beginning—you'll typically want to instrument your app for much more detailed telemetry so that you can understand exactly how your customers are using your app and where you want to invest more effort. For this there are a number of third-party solutions found on the Windows Partner Directory, and we'll talk about this a little more in Chapter 20.

With this data in hand and all the other ideas you either had to postpone from your first release or dreamt up in the meantime, you're all set to have your app come home for some new love before its next incarnation.

Updates are onboarded to the Windows Store just like the app's first version. You create and upload an app package (with the same package name as before but a new version number), and then you update your description, graphics, pricing, and other information. After that your updated package goes through the same certification and signing process as before, and when all that's complete your new app will be available in the Store and *automatically installed* for your existing customers (unless they opt out). And remember that with the blockmap business described earlier, only those parts of the app that have actually changed will be downloaded for an update (to a 64K resolution). This means that issuing small fixes (especially if they're placed at the end of files) won't force users to repeat potentially large downloads each time, bringing the update model closer to that of web applications.

When an update gets installed that has the same package name as an existing app, all the settings and appdata for the prior version remain intact. Your updated app should be prepared, then, to migrate a previous version of its state if and when it encounters such.

This brings up an interesting question: what happens with roaming data when a user has different versions of the same app installed on multiple devices? The answer is twofold: first, app state (which includes roaming data) has its own version number independent of the app, and second, Windows will transparently maintain multiple versions of the roaming state so long as there are apps installed on the user's devices that reference those state versions. Once all the devices have updated apps and have converted their state, Windows will delete old versions. We'll talk more of this in Chapter 10.

Another interesting question with updates is whether you can get a list of the customers who have acquired your app from the Store. The answer is no, because of privacy considerations. However, there is nothing wrong with including a registration feature in your app through which users can opt in to receive additional information from you, such as more detailed update notifications. Your Settings panel is a great place to include this.

The last thing to say about the Store is that in addition to basic analytics about your own app—which also includes data like sales figures, of course—it also provides you with marketwide analytics. These help you explore new opportunities to pursue—maybe taking an idea you had for a feature in one app and breaking that out into a new app in a different category. Here you can see what's selling well (and what's not) or where a particular category of app is underpopulated or generally has less than average reviews. For more details, again see the Dashboard at http://dev.windows.com.

And, Oh Yes, Then There's Design

In this first chapter we've covered the nature of the world in which Windows Store apps live and operate. In this book, too, we'll be focusing on the details of how to build such apps with HTML, CSS, and JavaScript. But what we haven't talked about, and what we'll only be treating minimally, is how you decide what your app does—its purpose in the world!—and how it clothes itself for that purpose.

This is really the question of good design for Windows Store apps—all the work that goes into apps before we even start writing code.

I said that we'll be treating this minimally because I simply do not consider myself a designer. I encourage you to be honest about this yourself: if you don't have a good designer working with you, **get one**. Sure, you can probably work out an OK design on your own, but the demands of a consumer-oriented market combined with a newer design language like that employed in Windows—where the emphasis is on simplicity and tailored experiences—underscores the need for professional help. It'll make the difference between a functional app and a great app, between a tool and a piece of art, between apps that consumers accept and those they *love*.

With design, I do encourage developers to peruse the material on the Windows Design Center for a better understanding of design principles. And if you're going to be playing the role of designer, a great place to start is the Category ideas area where you'll find case studies for converting websites, iOS apps, and enterprise LOB apps to a Windows Store app and many idea books that serve as starting points for different kinds of experiences.

But let's be honest: as a developer, do you really want to ponder every design principle and design not just static wireframes but also the dynamic aspects of an app like animations, page transitions, and progress indicators? Do you want to spend your time in graphic design and artwork (which is essential for a great app)? Do you want to haggle over the exact pixel alignment of your layout in all variable views? If not, find someone who does, because the combination of their design sensibilities and your highly productive hacking will produce much better results than either of you working alone. As one of my co-workers puts it, a marriage of "freaks" and "geeks" often produces the most creative, attractive, and inspiring results.

Let me add that design is neither a one-time nor a static process. Developers and designers will need to work together throughout the development experience, as design needs will arise in response to how well the implementation really works. For example, the real-world performance of an app might require the use of progress indicators when loading certain pages or might be better solved with a redesign of page navigation. It may also turn out, as we found with one of our early app partners, that the kinds of graphics called for in the design simply weren't available from the app's backend service. The design was lovely, in other words, but couldn't actually be implemented, so a design change was necessary. So make sure that your ongoing relationship with your designers is a healthy and happy one.

And on that note, let's get into your part of the story: the coding!

Feature Roadmap and Cross-Reference

As a means of indexing the content in this book, the tables below identifies the primary platform features of Windows 8.1, the chapter and section(s) where they're covered, and the purpose that those features serve in apps. Note that this is only a cross-reference for platform features: it does not represent nonfeature topics such as tooling, best practices, custom extensions, and design.

Feature – Chapter 3	**Section**	**Purpose**
Visual assets in the app manifest	"Branding Your App 101"	Defines your app's basic presence on the Start screen and elsewhere, including the splash screen on first launch.
App lifecycle management and basic state	"App Lifecycle Transition Events and Session State"	Allows Windows to manage whether apps are in memory while maintaining the appearance that they're always running.
WinJS Page controls	"Page Controls and Navigation"	Provides an HTML page loader so that apps can maintain a single script context.
WinJS navigation	"Page Controls and Navigation"	Provides a means to dynamically load and unload page controls, maintaining a navigation history.
Asynchronous APIs	"Async Operations"	Avoids unresponsive apps due to blocking the UI thread.
WinJS Scheduler	"Managing the UI Thread with the WinJS Scheduler"	Allows apps to set relative priorities of work on the UI thread to increase app responsiveness.

Feature – Chapter 4	**Section**	**Purpose**
Connectivity and cost awareness	"Network Information and Connectivity"	Enables apps to understand network state to implement great offline support and to prevent bill shock on metered networks.
Webview element	"Hosting Content"	Allows an app to display arbitrary HTML content, including content that's downloaded, dynamically generated, or hosted on the web.
HttpClient API	"HTTP Requests"	The more powerful and flexible way to perform HTTP requests to online services, with support for protocol-level filtering, cookie control, and precaching.
Background Transfer API	"Background Transfers"	Configures download and upload operations that will continue when an app is suspended or terminated, with the ability to set priorities, cost policies, and grouping.
Credential Locker	"The Credential Locker"	Provides secure roaming storage for credentials and other information that should be protected.
Web Authentication Broker	"The Web Authentication Broker" and "Single Sign On"	Provides UI for server-managed login to web services such that the app never touches user credentials.

Feature – Chapter 5	**Section**	**Purpose**
Core HTML controls: button, checkbox, drop-down list, listbox, hyperlink, file upload, slider, progress, and radio button	"HTML Controls" and "Styling Gallery: HTML Controls"	The UI elements that are declared through standard HTML5 and implemented in the app host.
WinJS controls: back button, date picker, time picker, rating, toggle switch, tooltip, HTML control, item container	"WinJS Controls" and "Styling Gallery: WinJS Controls"	The UI elements that are declared using WinJS syntax and implemented in the WinJS library.

Feature – Chapter 6	Section	Purpose
WinJS data binding	"Data Binding"	Facilitates creating automated connections between UI elements and data sources.
WinJS Template controls	"Binding Templates"	Defines blocks of generic HTML that can be bound to specific data sources and then rendered.
WinRT collection classes	"Windows.Foundation.Collection Types"	Used around WinRT to exchange different kinds of collection data.
WinJS.Binding.List class	"WinJS Binding Lists"	An observable (bindable) collection type employed by WinJS collection controls.

Feature – Chapter 7	Section	Purpose
WinJS Repeater control	"Quickstart #1: The WinJS Repeater Control" and "Repeater Features and Styling"	Conveniently renders template or other block of HTML for each item in a data source; serves as a lightweight means to render a noninteractive collection.
WinJS FlipView control	"Quickstart #2: The FlipView Control Sample," "FlipView Features and Styling," and "A FlipView Using the Pictures Library."	Provides a one-at-a-time view of a collection.
WinJS Semantic Zoom control	"The Semantic Zoom Control"	Wraps to other collection controls (such as a ListView) and provides the means to easily switch between them, giving the user two different views of the same data source.
WinJS ListView control	Most other sections in Chapter 7	Implements the most powerful and flexible means to render a collection with support for grouping, interactivity, drag and drop, variable layouts, and virtualization.

Feature – Chapter 8	Section	Purpose
Variable app view sizing including media queries	"Variable View Sizing and Orientation"	Allows a user to arrange multiple apps together on the same display.
Display rotations and orientation locking	"Variable View Sizing and Orientation"	Allows a user to rotate a tablet device for a different app experience.
Multiple app views	"Multiple Views"	Enables an app to create multiple views that the user can manage independently.
Projection API	"Multiple Views"	Enables an app to project a view onto a second display.
Snap points and rails (CSS features)	"Pannable Sections and Styles"	Controls the experience of pannable regions, including continuous vs. sectional panning and control of panning direction. Also includes continuous vs. step-wise zooming.
WinJS Hub control	"The Hub Control and Hub App Template"	Implements a common UI pattern through which an app displays data from multiple heterogeneous sources.
CSS grid, flexbox, multicolumn text, and regions	"Using the CSS Grid" and "Item Layout"	Controls various aspects of page and element layout within the scope of HTML and CSS.

Feature – Chapter 9	Section	Purpose
WinJS app bar and nav bar	"The App Bar and Nav Bar"	Provides the implementation for top and bottom commanding UI common to apps.
WinJS flyouts	"Flyouts and Menus"	Provides local and transient UI, typically used for popup messages and secondary command options.
WinJS menus	"Menus and Menu Commands"	Implements menu UI.
App-modal message boxes	"Message Dialogs"	Used to display messages that block further interaction with the app.

Feature – Chapter 10	Section	Purpose
Local, temp, and roaming state including folders and settings containers	"App Data Locations," "App Data APIs," and "Using App Data APIs for State Management"	Provides the unrestricted locations in which apps save all data whose lifetime is tied to the app, including session data, caches, and roaming state. This is used to maintain continuity across terminate and restart.
File and folder access APIs (StorageFolder, StorageFile)	"Folders, Files, and Streams"	Used to communicate with the local file system as well as with storage providers that present themselves as part of the file system even if they are non-local.
Stream APIs and blobs	"Folders, Files, and Streams"	Supply the low-level API for handing byte transfers, utilized in many areas of WinRT, including cross-compatibility with HTML5 blobs.
App settings pane (via Settings charm)	"Settings Pane and UI"	Provides the space in which an app offers configuration UI, account management, options, its privacy policy, and so forth.

Feature – Chapter 11	Section	Purpose
File Picker API	"Using the File Picker and Access Cache"	Allows the user to navigate to and select files and folders from any location on the local file system, the network, and cloud locations. The act of picking a file or folder grants usage permission to the app.
Access Cache API	"Using the File Picker and Access Cache"	Preserves file or folder permissions across app sessions.
File metadata and thumbnails	"StorageFile Properties and Metadata"	Provides access to all information about a file or folder without reading file data, including the use of thumbnails to display files without loading content.
Media library management	"Known Folders and the StorageLibrary Object"	Capabilities related to the user's Pictures, Videos, and Music libraries.
Searching files and folders	"Folders and Folder Queries"	Taps into the system indexer through which an app can quickly enumerate files and folders that match specific criteria.
File type association	"File Activation and Association"	Allows an app to register itself as able to handle one or more file types. A user can select from such registered apps when opening a file.
Access to removable storage	"Removable Storage"	Provides the ability to work with the contents of flash drives, memory cards, and similar devices.

Feature – Chapter 12	Section	Purpose
Touch input	"Touch, Mouse, and Stylus Inputs," "Unified Pointer Events," "Gesture Events," and "The Gesture Recognizer"	Enables an app to work with touch screens to the degree it wants, whether to treat touch identically with mouse and stylus input or to respond specifically to touch gestures.
Mouse and styling input	"Unified Pointer Events"	Enables an app to work with the mouse or a stylus.
Keyboard input including the on-screen keyboard	"Keyboard Input and the Soft Keyboard"	Provides keystroke information to the app, including how the on-screen keyboard is configured for and responds to different input controls.
Inking APIs	"Inking"	Provides the ability to capture full information about touch input (e.g., stokes, pressure, etc.), which can be used to re-render that input at a later time or implement your own recognition engine.
GPS device access	"Geolocation" and "Geofencing"	Gives apps access to GPS information including the ability to set up a region with entry and exit events.
APIs for sensor input from the accelerometer, compass, inclinometer, gyrometer, orientation sensor, and light sensor.	"Sensors"	Provides readings from various sensors on devices that are so equipped.

Feature – Chapter 13	Section	Purpose
Display of graphical information	"Graphics Elements"	Displays raster images (img), vector images (svg), and dynamically drawn images (canvas), along with PDFs.
Video playback (video element)	"Video Playback and Deferred Loading"	Provides the ability to render video in an app, including the application of effects.
Locking screen orientation and disabling the lock screen	"Disabling Screen Savers and the Lock Screen"	Ensures that video playback is not interrupted by orientation changes or inactivity timeouts.
Audio playback (audio element)	"Audio Playback and Mixing" and "Playlists"	Provides the ability to play audio in an app, including background audio.
System controls for audio and video	"The Media Transport Control UI"	Allows an app to provide textual and graphical information for the system-managed UI that's connected to hardware/software playback features.
Text to Speech API	"Text to Speech"	Supplies the ability to use a synthesized voice to convert text to audio.
Media manipulation including transcoding	"Loading and Manipulating media"	Provides the ability to convert between different media formats and to support custom formats.
Dynamic media generation (video and audio)	"Media Stream Sources"	Enables app code to generate media information on the fly, to manipulate media as it's played or otherwise rendered, and to support custom formats.
Using the webcam and microphones	"Media Capture"	Provides the ability to record video and audio, either through a system-provided UI or an app UI.
PlayTo	"Streaming Media and PlayTo"	Enables media playback to DLNA devices.
Digital Rights Management	"Streaming Media and PlayTo"	Enables an app to control permissions for media rendering.

Feature – Chapter 14	Section	Purpose
The WinJS animations library	"The WinJS Animations Library"	Provides a predefined set of CSS animations and transitions that reflect the Windows personality.
Low-level animations	"CSS Animations and Transitions" and "Rolling Your Own"	Supports custom animations through standard CSS capabilities or rendering frames using interval timers.

Feature – Chapter 15	Section	Purpose
Share contract	"Share"	Connects two apps together through the Share control so that a source provides data that a target can then share however it wants without leaving the context of the source app.
Associating an app with a URI scheme	"Launching Apps with URI Scheme Associations"	Allows an app to register itself as able to handle one or more URI schemes types. A user can select from such registered apps when a URI is activated.
In-app search	"Search" and "The WinJS.UI.SearchBox Control"	Provides the means to easily implement rich in-app searching capabilities.
Search charm interaction (Search contract)	"The Search Charm UI" and "The Search Contract"	In lieu of handing search directly in an app, allows for search interactions through the Search charm.
System indexer and queries against file content and app-provided content	"Indexing and Searching Content"	Allows apps to perform very fast metadata and content queries against app data as well as arbitrary app content.
Contact Cards UI	"Contacts" and "Contact Cards"	Invokes a system-provided UI that displays contact information aggregated in the People app, relieving other apps from having to manage and protect that data.
Contact Picker API	"Using the Contact Pickers"	Allows an app to obtain contact data from any number of course as defined by provider apps; in this case the app invoking the picker is responsible for protecting the contact information.
Appointments APIs	"Appointments"	Enables an app to create and manage entries on the user's calendar through bits of UI supplied by the default calendar app.

Feature – Chapter 16	**Section**	**Purpose**
Live tiles	"Basic Tile Updates" and "Cycling, Scheduled, and Expiring Updates"	Sends information to the app's tile on the Start screen.
Secondary tiles	"Secondary Tiles"	Allows an app to create additional tiles on the Start screen with user consent. Secondary tiles can be live.
Badges	"Badge Updates"	Provides the means to display a small number or glyph on a Start screen tile.
Periodic tile updates from a service	"Periodic Updates"	Links a tile on the Start screen to up to five URIs that provide tile updates even while the app isn't running.
Toast notifications	"Toast Notifications"	Provides the means to display popup messages on the user's display, on top of other apps as well as the lock screen, where activating the toast activates the app.
Push notifications	"Push Notifications and the Windows Push Notification Service"	Enables services to push tile updates, badge updates, and toast notifications to a specific user's device without needing to involve the app.
Raw notifications	"Raw Notifications (Service)" and "Receiving Notifications (App)"	Delivers a custom payload to an app via push notifications, which the running app or a background task can process however it wants.
Background tasks	"Background Tasks and Lock Screen Apps"	Configures small pieces of code (with quotas on CPU time another resources) that can run in response to various triggers and conditions.
Lock screen apps	"Background Tasks and Lock Screen Apps"	Enables apps to identify themselves as lock-screen-capable to the system so that the user can select them through PC Settings to display information on the lock screen and have greater background privileges.

Feature – Chapter 17	**Section**	**Purpose**
Discovery of peripheral devices	"Enumerating and Watching Devices"	Allows an app to determine what devices of desired types are attached to the system, and to watch when such devices are connected and disconnected.
Image scanning	"Image Scanners"	Enables apps to work with image scanners.
Point of Service device APIs	"Barcode and Magnetic Stripe Readers"	Enables apps to receive information from barcode scanners and magnetic stripe readers, typically used for point of service systems.
Accessing virtual smartcards	"Smartcards"	Enables an app to provision a virtual smartcard.
Verifying a user's physical presence	"Fingerprint (Biometric) Readers"	Determines whether the machine is capable of reading fingerprints and can request verification of the physical presence of the logged-in user.
Call control via Bluetooth devices	"Bluetooth Call Control"	Informs an app of events that happen with Bluetooth earpieces and headsets as they affect calls.
Document Printing APIs	"Printing Made Easy"	Provides for printing 2D documents. 3D printing is supported through the Windows platform but not through JavaScript.
Device access via USB, HID, Bluetooth, and Wi-Fi Direct	"Protocol APIs"	Allows apps to communicate with a wide variety of devices through different communication transports.
NFC APIs	"Near Field Communication and the Proximity API"	Enables connections between devices via taps as well as peer discovery over Wi-Fi direct.

Feature – Chapter 18	**Section**	**Purpose**
Mixed-language apps	"Choosing a Mixed Language Approach" and most of the chapter	Allows an app to use C#, Visual Basic, or C++ to implement different functions, providing more flexibility in implementation strategies and access to additional APIs.
Web workers	"JavaScript Workers"	Provides the means to execute JavaScript code off the UI thread.
Creating custom async APIs	"Implementing Asynchronous Methods"	Provides the means to create custom APIs in C#,VB, or C++ like those in WinRT that execute off the UI thread.

Feature – Chapter 19	**Section**	**Purpose**
Supporting screen readers	"Screen Readers and Aria Attributes"	Helps an app fulfill accessibility requirements by properly labeling its markup for screen readers.
Support high contrast	"Handling Contrast Variations"	Helps an app fulfill accessibility requirements by handling high contrast modes.
Globalization APIs	"Globalization"	Enables creation of a culture-neutral app that can handle locale-based variations such as dates, times, and sorting.
Localizing apps with additional language resources	"Preparing for Localization" and "Creating Localized Resources"	Enables creation of multilingual apps for a large number of world markets.

Feature – Chapter 20	**Section**	**Purpose**
Side-loading apps	"Sidebar: Side Loading"	Enables developers and enterprises to install apps without going through the Windows Store.
Paid apps and trial versions	"Paid Apps and Trial Versions" and "Trial Versions and App Purchase"	Sets an app price in the Windows Store with or without a free trial period.
Monetization through ads	"Ad-Supported Apps"	Supports monetization through display of ads within the app.
In-app purchases (durables and consumables)	"In-App Purchases" and "Listing and Purchasing In-App Products"	Configures and manages monetization of an app through feature enablement and other item purchases such as in-app currency.
Large catalogs	"Handling Large Catalogs"	Provides the ability to dynamically configure in-app purchases without having to define the ahead of time.
Verification of purchase	"Receipts"	Enables an app to make a strong determination that a purchase was made and who specifically made it, used to implement additional licensing schemes and control access to services.
Listing apps in the Store	"Releasing Your App to the World"	Defines the process through which an app is made available to customers in markets around the world.
Linking websites to apps	"Connecting Your Website and Web-Mapped Search Results"	Defines a relationship between a website and an app such that visitors to the website or search results that include that site can point to the app.

Feature – Appendices	**Section**	**Purpose**
Extended Splash Screens	Appendix B: "Extended Splash Screens"	Provides a way for an app to customize its splash screen and avoid being terminated if the user switches away while it continues to load.
Custom ListView layouts	Appendix B: "Custom Layouts for the ListView Control"	Allows an app to extend the visual layout capabilities of the WinJS ListView control.
Large and multipart uploads with the background transfer API	Appendix C: "Breaking Up Large Files" and "Multipart Uploads"	Provides for specialized needs with background transfers.
Security (encryption, certificates)	Appendix C: "Notes on Encryption, Decryption, Data Protection, and Certificates"	Supports apps that need to strongly secure data.
RSS, AtomPub, and XML manipulation	Appendix C: "Syndication: RSS, AtomPub, and XML APIs in WinRT"	Provides APIs to work with RSS feeds, to publish via AtomPub, and to create and manage XML documents.
Sockets (including websockets)	Appendix C: "Sockets"	Gives apps the ability to communicate over a variety of socket types.
Credential Picker UI	Appendix C: "The Credential Picker UI"	Provides the ability for an app to collect credentials from a user including domain credentials, smartcard PINs, and other sources that involve a variety of security protocols.
Providers for file pickers, contact cards, contact picker, and appointments	Appendix D (all sections)	Enables apps to serve as providers for a variety of contracts; applicable to apps that service storage services, manage address books, and manage calendars.

Chapter 2

Quickstart

This is a book about developing apps. So, to quote Paul Bettany's portrayal of Geoffrey Chaucer in *A Knight's Tale*, "without further gilding the lily, and with no more ado," let's create some!

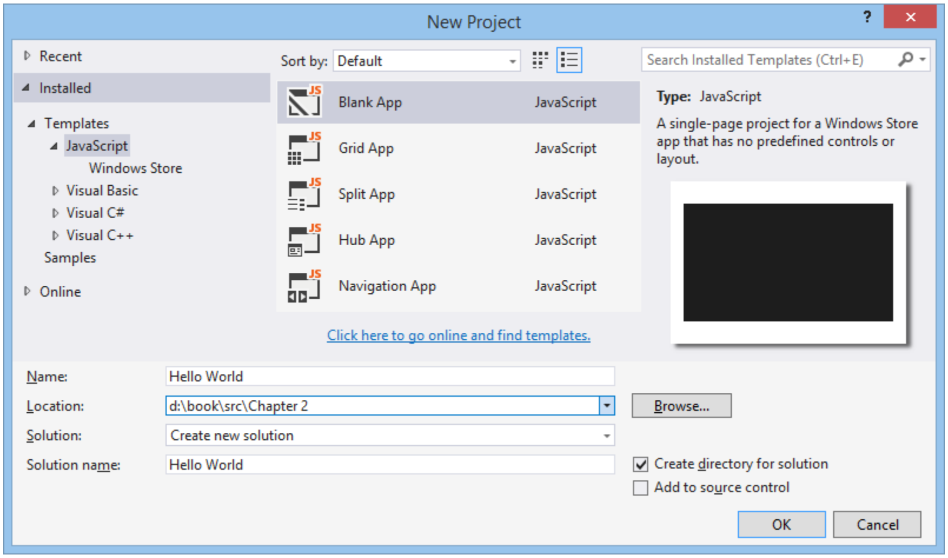
A Really Quick Quickstart: The Blank App Template

We must begin, of course, by paying due homage to the quintessential "Hello World" app, which we can achieve without actually writing any code at all. We simply need to create a new app from a project template in Visual Studio:

1. Run Visual Studio Express for Windows. If this is your first time, you'll be prompted to obtain a developer license. Do this, because you can't go any further without it!

2. Click New Project... in the Visual Studio window, or use the File > New Project menu command.

3. In the dialog that appears (Figure 2-1), make sure you select JavaScript under Templates on the left side, and then select Blank App in the middle. Give it a name (**HelloWorld** will do), a folder, and click OK.



Blank App	JavaScript
Grid App	JavaScript
Split App	JavaScript
Hub App	JavaScript
Navigation App	JavaScript

Type: JavaScript

A single-page project for a Windows Store app that has no predefined controls or layout.

Name: Hello World

Location: d:\book\src\Chapter 2

Solution: Create new solution

Solution name: Hello World

☑ Create directory for solution

□ Add to source control

FIGURE 2-1 Visual Studio's New Project dialog using the light UI theme. (See the Tools > Options menu command, and then change the theme in the Environment/General section). I use the light theme in this book because it looks best against a white page background.

4. After Visual Studio churns for a bit to create the project, click the Start Debugging button (or press F5, or select the Debug > Start Debugging menu command). Assuming your installation is good, you should see something like Figure 2-2 on your screen.

Content goes here

FIGURE 2-2 The only vaguely interesting portion of the Hello World app's display. The message is at least a better invitation to write more code than the standard first-app greeting!

By default, Visual Studio starts the debugger in *local machine* mode, which runs the app full screen on your present system. This has the unfortunate result of hiding the debugger unless you're on a multimonitor system, in which case you can run Visual Studio on one monitor and your Windows Store app on the other. Very handy. See Running apps on the local machine for more on this.[8]

Visual Studio offers two other debugging modes available from the drop-down list on the toolbar (Figure 2-3) or the Debug/[Appname] Properties menu command (Figure 2-4):

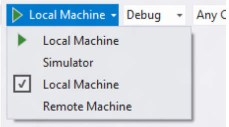
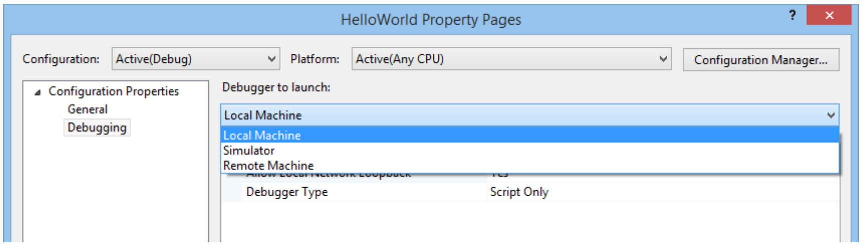


FIGURE 2-3 Visual Studio's debugging options on the toolbar.



HelloWorld Property Pages

Configuration: Active(Debug) Platform: Active(Any CPU) Configuration Manager...

Configuration Properties
General
Debugging

Debugger to launch: Local Machine

Local Machine
Simulator
Remote Machine

Debugger Type: Script Only

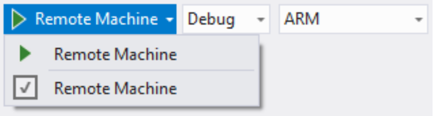
FIGURE 2-4 Visual Studio's debugging options in the app properties dialog.

The Remote Machine option allows you to run the app on a separate device, which is absolutely essential for working with Windows RT devices that can't run desktop apps at all, such as the Microsoft Surface and other ARM devices. Setting this up is a straightforward process, and it works on both Ethernet and wireless networks: see Running apps on a remote machine, and I do recommend that you get familiar with it. Also, when you don't have a project loaded in Visual Studio, the Debug menu offers

[8] For debugging the app and Visual Studio side by side on a single monitor, check out the utility called ModernMix from Stardock that allows you to run Windows Store apps in separate windows on the desktop.

the Attach To Process command, which allows you to debug an already-running app. See [How to start a debugging session (JavaScript)]().

Tip If you ever load a Windows SDK sample into Visual Studio and Remote Machine is the only debugging option that's available, the build target is probably set to ARM (the rightmost drop-down):



Set the build target to Any CPU and you'll see the other options. Note apps written in JavaScript, C#, or Visual Basic that contain no C++ WinRT components (see Chapter 18, "WinRT Components"), should always use the Any CPU target.

Another tip If you ever see a small ⊗ on the tile of one of your app projects, or for some reason it just won't launch from the tile, your developer license is probably out of date. Just run Visual Studio or Blend to renew it. If you have a similar problem on a Windows RT device, especially when using remote debugging, you'll need renew the license from the command line using PowerShell. See [Installing developer packages on Windows RT]() in the section "Obtaining or renewing your developer license" for instructions.

The Simulator, for its part, duplicates your current environment inside a new login session and allows you to control device orientation, set various screen resolutions and scaling factors, simulate touch events, configure network characteristics, and control the data returned by geolocation APIs. Figure 2-5 shows Hello World in the simulator with the additional controls labeled on the right. We'll see more of the simulator as we go along, though you may also want to peruse the [Running apps in the simulator]() topic.

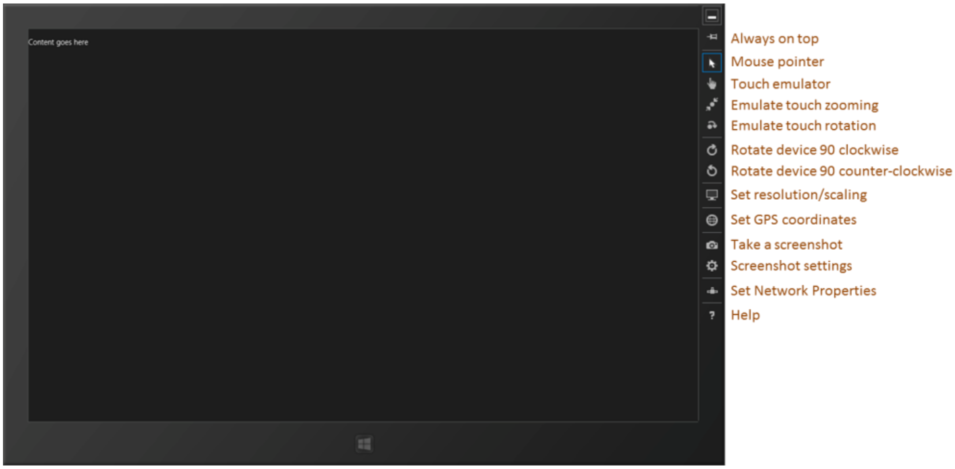


FIGURE 2-5 Hello World running in the simulator, with added labels on the right for the simulator controls. Truly, the "Blank App" template lives up to its name!

Sidebar: How Does Visual Studio Run an App?

Under the covers, Visual Studio is actually deploying the app similar to what would happen if you acquired it from the Store. The app will show up on the Start screen's All Apps view, where you can also uninstall it. Uninstalling will clear out appdata folders and other state, which is very helpful when debugging.

There's no magic involved: deployment can actually be done through the command line. To see the details, use the Store/Create App Package in Visual Studio, select No for a Store upload, and you'll see a dialog in which you can save your package to a folder. In that folder you'll then find an appx package, a security certificate, and a batch file called *Add-AppxDevPackage*. That batch file contains PowerShell scripts that will deploy the app along with its dependencies.

These same files are also what you can share with other developers who have a developer license, allowing them to side-load your app without needing your full source project.

Blank App Project Structure

Although an app created with the Blank template doesn't offer much in the visual department, it lets us see the core structure of all projects you'll use. That structure is found in Visual Studio's Solution Explorer (as shown in Figure 2-6).

In the project root folder:

- **default.html** The starting page for the app.
- **<Appname>_TemporaryKey.pfx** A temporary signature created on first run.
- **package.appxmanifest** The manifest. Opening this file will display Visual Studio's manifest editor (shown later in this chapter). Browse around in this UI for a few minutes to familiarize yourself with what's here: references to the various app images (see below), a checkmark on the *Internet (Client)* capability, default.html selected as the start page, and all the places where you control different aspects of your app. We'll be seeing these throughout this book; for a complete reference, see the App packages and deployment and Using the manifest designer topics. And if you want to explore the manifest XML directly, right-click this file and select View Code. This is occasionally necessary to configure uncommon options that aren't represented in the editor UI. The APIs for accessing package details are demonstrated in the App package information sample.

The **css** folder contains a default.css file that's empty except for a blank rule for the `body` element.

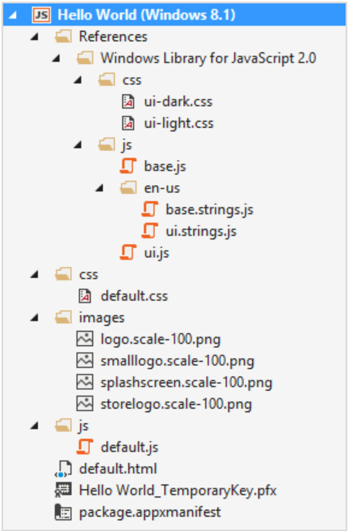
The **images** folder contains four placeholder branding images, and unless you want to look like a real doofus developer, *always* customize these before sending your app to the Store (and to provide scaled versions too, as we'll see in Chapter 3, "App Anatomy and Performance Fundamentals"):

- **logo.scale-100.png** A default 150x150 (100% scale) image for the Start screen.
- **smalllogo.scale-100.png** A 30x30 image for the zoomed-out Start screen and other places at run time.
- **splashscreen.scale-100.png** A 620x300 image that will be shown while the app is loading.
- **storelogo.scale-100.png** A 50x50 image that will be shown for the app in the Windows Store. This needs to be part of an app package but is not used within Windows at run time. For this reason it's easy to overlook—make a special note to customize it.

The **js** folder contains a simple default.js.

The **References** folder points to CSS and JavaScript source files for the WinJS library, which you can open and examine anytime. (If you want to search within these files, you must open and search only within the specific file. These are not included in solution-wide or project-wide searches.)

NuGet Packages If you right-click References you'll see a menu command Manage NuGet Packages.... This opens a dialog box through which you can bring many different libraries and SDKs into your project, including jQuery, knockout.js, Bing Maps, and many more from both official and community sources. For more information, see http://nuget.org/.



```
◢ Hello World (Windows 8.1)
  ◢ References
    ◢ Windows Library for JavaScript 2.0
      ◢ css
          ui-dark.css
          ui-light.css
      ◢ js
          base.js
        ◢ en-us
            base.strings.js
            ui.strings.js
          ui.js
  ◢ css
      default.css
  ◢ images
      logo.scale-100.png
      smalllogo.scale-100.png
      splashscreen.scale-100.png
      storelogo.scale-100.png
  ◢ js
      default.js
    default.html
    Hello World_TemporaryKey.pfx
    package.appxmanifest
```

FIGURE 2-6 A Blank app project fully expanded in Solution Explorer.

As you would expect, there's not much app-specific code for this type of project. For example, the HTML has only a single paragraph element in the body, the one you can replace with "Hello World" if you're really not feeling complete without doing so. What's more important at present are the references to the WinJS components: a core stylesheet (ui-dark.css or ui-light.css), base.js, and ui.js:

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8">
    <title>Hello World</title>

    <!-- WinJS references -->
    <link href="//Microsoft.WinJS.2.0/css/ui-dark.css" rel="stylesheet">
    <script src="//Microsoft.WinJS.2.0/js/base.js"></script>
    <script src="//Microsoft.WinJS.2.0/js/ui.js"></script>

    <!-- HelloWorld references -->
    <link href="/css/default.css" rel="stylesheet">
    <script src="/js/default.js"></script>
</head>
<body>
    <p>Content goes here</p>
</body>
</html>
```

You will generally always have these references in every HTML file of your project (using an appropriate version number, and perhaps using `ui-light.css` instead). The //'s in the WinJS paths refer to shared libraries rather than files in your app package, whereas a single / refers to the root of your package. Beyond that, everything else is standard HTML5, so feel free to play around with adding some additional HTML of your own to see the effects.

> **Tip** When referring to in-package resources, always use a leading / on URIs, which means "package root." This is especially important when using page controls (see Chapter 3) because those pages are typically loaded into a document like default.html whose location is different from where the page exists in the project structure.

Where JavaScript is concerned, default.js just contains the basic WinJS activation code centered on the `WinJS.Application.onactivated` event along with a stub for an event called `WinJS.Application.oncheckpoint` (from which I've omitted a lengthy comment block):

```
(function () {
    "use strict";

    var app = WinJS.Application;
    var activation = Windows.ApplicationModel.Activation;

    app.onactivated = function (args) {
        if (args.detail.kind === activation.ActivationKind.launch) {
            if (args.detail.previousExecutionState !==
                activation.ApplicationExecutionState.terminated) {
                // TODO: This application has been newly launched. Initialize
                // your application here.
            } else {
                // TODO: This application has been reactivated from suspension.
                // Restore application state here.
            }
            args.setPromise(WinJS.UI.processAll());
        }
    };

    app.oncheckpoint = function (args) {
    };

    app.start();
})();
```

We'll come back to checkpoint in Chapter 3. For now, remember from Chapter 1, "The Life Story of a Windows Store App," that an app can be activated in many ways. These are indicated in the args.detail.kind property whose value comes from the Windows.ApplicationModel.Activation.ActivationKind enumeration.

When an app is launched directly from its tile on the Start screen (or in the debugger as we've been doing), the kind is just launch. As we'll see later on, other values tell us when an app is activated to service requests like the search or share contracts, file-type associations, file pickers, protocols, and more. For the launch kind, another bit of information from the Windows.ApplicationModel.Activation.ApplicationExecutionState enumeration tells the app how it was last running. Again, we'll see more on this in Chapter 3, so the comments in the default code above should satisfy your curiosity for the time being.

Now, what is that args.setPromise(WinJS.UI.processAll())for? As we'll see many times, WinJS.UI.processAll instantiates any WinJS controls that are declared in yourHTML—that is, any element (commonly a div or span) that contains a data-win-control attribute whose value is the name of a constructor function. The Blank app template doesn't include any such controls, but because just about every app based on this template *will*, it makes sense to include it by default.[9] As for args.setPromise, that's employing something called a deferral that we'll also defer to Chapter 3.

As short as it is, that little app.start(); at the bottom is also very important. It makes sure that various events that are queued during startup get processed. We'll again see the details in Chapter 3. I'll bet you're looking forward to that chapter now!

Finally, you may be asking, "What on earth is all that ceremonial (function () { … })(); business about?" It's just a convention in JavaScript called a *self-executing anonymous function* that implements the *module pattern*. This keeps the global namespace from becoming polluted, thereby propitiating the performance gods. The syntax defines an anonymous function that's immediately executed, which creates a function scope for everything inside it. So variables like app along with all the function names are accessible throughout the module but don't appear in the global namespace.[10]

[9] There is a similar function WinJS.Binding.processAll that processes data-win-bindattributes (Chapter 6), and WinJS.Resources.processAll that does resource lookup on data-win-res attributes (Chapter 19).

[10] See Chapter 2 of Nicolas Zakas's *High Performance JavaScript* (O'Reilly, 2010) for the performance implications of scoping. More on modules can be found in Chapter 5 of *JavaScript Patterns* by Stoyan Stefanov (O'Reilly, 2010) and Chapter 7 of *Eloquent JavaScript* by Marijn Haverbeke (No Starch Press, 2011).

You can still introduce variables into the global namespace, of course, and to keep it all organized, WinJS offers a means to define your own namespaces and classes (see `WinJS.Namespace.define` and `WinJS.Class.define`), again helping to minimize additions to the global namespace. We'll learn more of these in Chapter 5, "Controls and Control Styling," and Appendix B, "WinJS Extras."

Now that we've seen the basic structure of an app, let's build something more functional and get a taste of the WinRT APIs and a few other platform features.

Get familiar with Visual Studio If you're new to Visual Studio, the tool can be somewhat daunting at first because it supports many features, even in the Express edition. For a quick, roughly 10-minute introduction, Video 2-1 in this chapter's companion content to will show you the basic workflows and other essentials.

QuickStart #1: Here My Am! and an Introduction to Blend for Visual Studio

When my son was three years old, he never—despite the fact that he was born to two engineers parents and two engineer grandfathers—peeked around corners or appeared in a room saying "Hello world!" No, his particular phrase was "Here my am!" Using that variation of announcing oneself to the universe, our next app can capture an image from a camera, locate your position on a map, and share that information through the Windows Share charm. Does this sound complicated? Fortunately, the WinRT APIs actually make it quite straightforward!

Sidebar: How Long Did It Take to Write This App?

This app took me about three hours to write. "Oh sure," you're thinking, "you've already written a bunch of apps, so it was easy for you!" Well, yes and no. For one thing, I also wrote this part of the chapter at the same time, and endeavored to make some reusable code, which took extra time. More importantly, the app came together quickly because I knew how to use my tools—especially Blend—and I knew where I could find code that already did most of what I wanted, namely all the Windows SDK samples on http://code.msdn.microsoft.com/windowsapps/.

As we'll be drawing from many of these most excellent samples in this book, I encourage you to download the whole set—go to the URL above, and click the link for "Windows 8.1 app samples". On that page you can get a .zip file with all the JavaScript samples. Once you unzip these, get into the habit of searching that folder for any API or feature you're interested in (make sure it's being indexed by the Windows file system too). For example, the code I use in this app to implement camera capture and sharing data came directly from a couple of samples.

I also *strongly* encourage you to spend a half-day getting familiar with Visual Studio and Blend for Visual Studio and running samples so that you know what tremendous resources are available. Such small investments will pay huge productivity dividends even in the short term!

Design Wireframes

Before we start on the code, let's first look at design wireframes for this app. Oooh…design? Yes! Perhaps for the first time in the history of Windows, there's a real design *philosophy* to apply to apps in Windows 8. In the past, with desktop apps, it's been more of an "anything goes" scene. There were some UI guidelines, sure, but developers could generally get away with making up any user experience that made sense to them, like burying essential checkbox options four levels deep in a series of modal dialog boxes. Yes, this kind of stuff does make sense to developers; whether it makes sense to anyone else is highly questionable!

If you've ever pretended or contemplated pretending to be a designer, now is the time to surrender that hat to someone with real training or set development aside for a focused time to invest in that training yourself. Simply said, *design matters* for Windows Store apps, and it will make the difference between apps that really succeed and apps that merely exist in the Windows Store and are largely ignored. And having a design in hand will just make it easier to implement because you won't have to make those decisions when you're writing code. (If you still intend on filling designer shoes and communing with apps like Adobe Illustrator, at least be sure to visit Designing UX for apps for the philosophy and details of Windows Store app design, plus design resources.)

> **Note** Traditional wireframes are great to show a static view of the app, but in the "fast and fluid" environment of Windows, the *dynamic* aspects of an app—animations, transitions, and movement—are also very important. Great app design includes consideration of not just where content is placed but how and when it gets there in response to which user actions. Chapter 14, "Purposeful Animations," discusses the different built-in animations that you can use for this purpose.

When I had the idea for this app, I drew up simple wireframes, let a few designers laugh at me behind my back (and offer helpful adjustments), and eventually landed on layouts for the various views as shown in Figures 2-7 through 2-9. These reflect the guidelines of the "grid system" described on Laying out an app page, which defines what's called the layout *silhouette* that includes the size of header fonts, their placement, and specific margins. These suggestions encourage a degree of consistency between apps so that users' eyes literally develop muscle memory for common elements of the UI. That said, they are not hard and fast rules, just a starting point—designers can and do depart from them when there's reason to do so.

Generally speaking, layout is based on a basic 20 pixel unit, with 5 pixel sub-units. In the full landscape view of Figure 2-7, you can see the recommended left margin of 120px, the recommended top margin of 140px above the content region, and placement of the header's baseline at 100px, which for a 42pt font translates to a 44px top margin. For partial landscape views with width <= 1024px, the left margin shrinks to 40px (not shown). In the portrait and narrow views of Figure 2-8 and 2-9, the various margins and gaps get smaller but still align to the grid.

What happened to snapped and filled views? In the first release of Windows 8, app design focused on four view states known as landscape, portrait, filled, and snapped. With Windows 8.1, each view of an app can be arbitrarily sized in the horizontal, so distinct names for these states are deprecated in favor of simply handling different view sizes and aspect ratios—known as *responsive design* on the web. For apps, the minimum design size is now 500x768 pixels, and an app can indicate in the manifest whether it supports a narrower view down to a 320px minimum. The "Here My Am!" app as designed in this section supports all sizes including narrow. Aspect ratios (width/height) of 1 and below (meaning square to tall) use the vertically-oriented layouts; aspect ratios greater than 1 use a horizontally-oriented layout.

To accommodate view sizes, you can use standard CSS3 `orientation` media queries to differentiate aspect ratios; the view state media queries from Windows 8 don't differentiate between the filled state (a narrower landscape) and the 50% split view that will often have an aspect ratio less than 1.

Note, however, that the header font sizes, from which we derive the top header margins, were defined in the WinJS 1.0 stylesheets in Windows 8 but were removed in WinJS 2.0 for Windows 8.1. To adjust the font size for narrow views, then, default.css in Here My Am! has specific rules to set `h1` and `h2` element sizes.

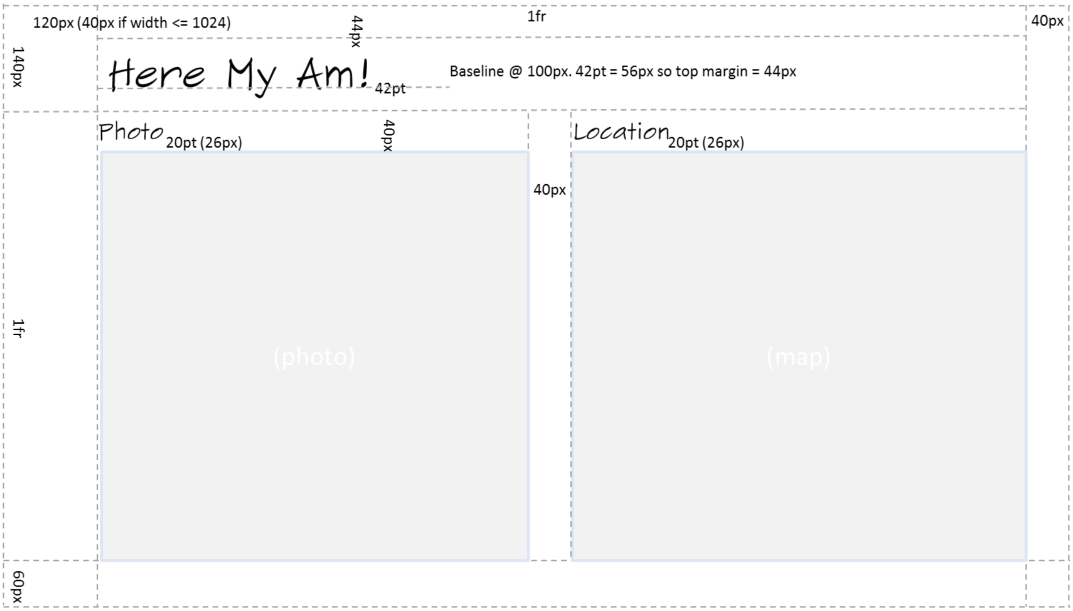


FIGURE 2-7 Wireframe for wide aspect ratios (width/height > 1). The left margin is nominally 120px, changing to 40px for smaller (<1024px) widths. The "1fr" labels denote proportional parts of the CSS grid (see Chapter 8, "Layout and Views") that occupy whatever space remains after the fixed parts are laid out.

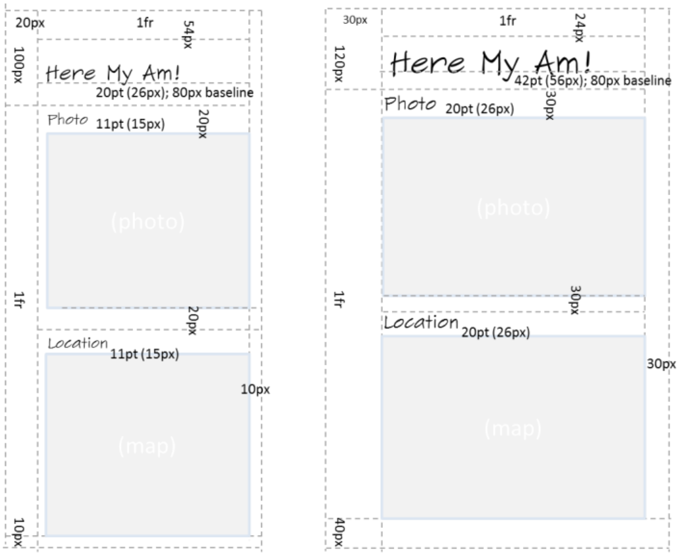


FIGURE 2-8 Wireframes for narrow (320–499px) and portrait (500px or higher) aspect ratios (width/height <= 1). These views also happen to work nicely on a portrait-first Windows Phone.

Sidebar: Design for All Size Variations!

Just as I thought about all size variations for Here My Am!, I encourage you to do the same for one simple reason: *your app will be put into every state whether you design for it or not* (with the exception of the narrow 320–499px view if you don't indicate it in your manifest). Users control the views, not the app, so if you neglect to design for any given state, your app will probably look hideous in that state. You can, as we'll see in Chapter 8, lock the landscape/portrait orientation for your app if you want, but that's meant to enhance an app's experience rather than being an excuse for indolence. So in the end, unless you have a very specific reason not to, every page in your app needs to anticipate all different sizes and dimensions.

This might sound like a burden, but these variations don't affect function: they are simply different views of the same information. Changing the view never changes the *mode* of the app. Handling different views, therefore, is primarily a matter of which elements are visible and how those elements are laid out on the page. It doesn't have to be any more complicated than that, and for apps written in HTML and JavaScript the work can mostly, if not entirely, be handled through CSS media queries.

Enough said! Let's just assume that we have a great design to work from and our designers are off sipping cappuccino, satisfied with a job well done. Our job is now to execute on that great design.

Create the Markup

For the purposes of markup, layout, and styling, one of the most powerful tools you can add to your arsenal is Blend for Visual Studio, which is included for free when you install Visual Studio Express. Blend has full design support for HTML, CSS, *and* JavaScript. I emphasize that latter point because Blend doesn't just load markup and styles: it loads and *executes* your code, right in the "Artboard" (the design surface), because that code so often affects the DOM, styling, and so forth. Then there's Interactive Mode…but I'm getting ahead of myself!

Blend and Visual Studio are very much two sides of a coin: they can share the same projects and have commands to easily switch between them, depending on whether you're focusing on design (layout and styling in Blend) or development (coding and debugging in Visual Studio). To demonstrate that, let's actually start building Here My Am! in Blend. As we did before with Visual Studio, launch Blend, select New Project…, and select the Blank App template. This will create the same project structure as before. (Note: Video 2-2 shows all these steps together.)

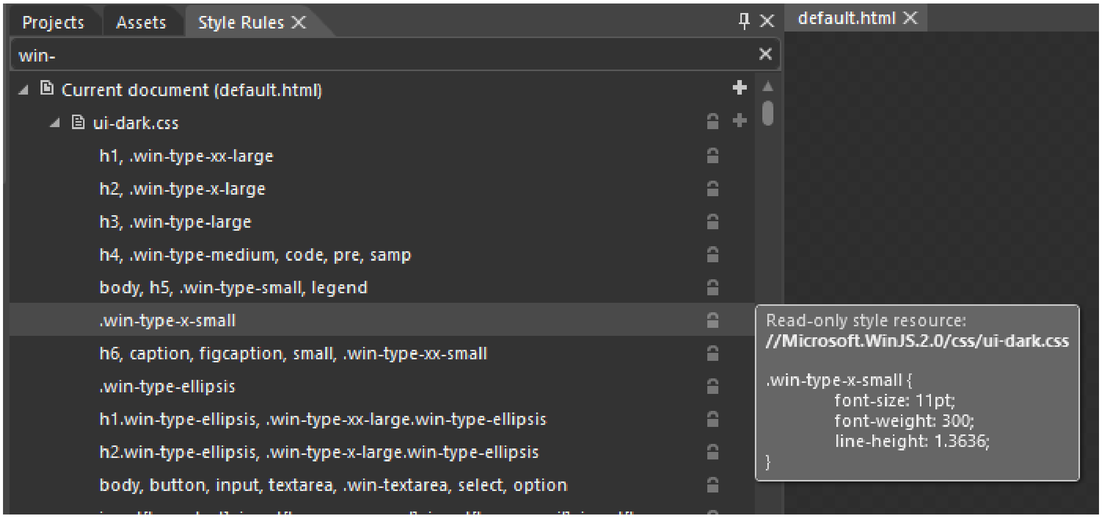
Following the practice of writing pure markup in HTML—with no styling and no code, and even leaving off a few classes we'll need for styling—let's drop the following markup into the `body` element of default.html (replacing the one line of `<p>Content goes here</p>`):

```
<div id="mainContent">
    <header aria-label="Header content" role="banner">
        <h1 class="titlearea win-type-ellipsis">
            <span class="pagetitle">Here My Am!</span>
        </h1>
    </header>
    <section aria-label="Main content" role="main">
        <div id="photoSection" aria-label="Photo section">
            <h2 class="group-title" role="heading">Photo</h2>
            <img id="photo" src="/images/taphere.png"
                alt="Tap to capture image from camera" role="img" />
        </div>
        <div id="locationSection" aria-label="Location section">
            <h2 class="group-title" role="heading">Location</h2>
            <iframe id="map" src="ms-appx-web:///html/map.html" aria-label="Map"></iframe>
        </div>
    </section>
</div>
```

Here we see the five elements in the wireframe: a main header, two subheaders, a space for a photo (for now an `img` element with a default "tap here" graphic), and an `iframe` that specifically houses a page in which we'll instantiate a Bing maps web control.[11]

[11] If you're following the steps in Blend yourself, the taphere.png image should be added to the project in the images folder. Right-click that folder, select Add Existing Item, and then navigate to the complete sample's images folder and select taphere.png. That will copy it into your current project. Note, though, that we'll do away with this later in this chapter.

You'll see that some elements have style classes assigned to them. Those that start with `win-` come from the WinJS stylesheet (among others).[12] You can browse these in Blend on the Style Rules tab, shown in Figure 2-9. Other styles like `titlearea`, `pagetitle`, and `group-title` are meant for you to define in your own stylesheet, thereby overriding the WinJS styles for particular elements.



Current document (default.html)
ui-dark.css
h1, .win-type-xx-large
h2, .win-type-x-large
h3, .win-type-large
h4, .win-type-medium, code, pre, samp
body, h5, .win-type-small, legend
.win-type-x-small
h6, caption, figcaption, small, .win-type-xx-small
.win-type-ellipsis
h1.win-type-ellipsis, .win-type-xx-large.win-type-ellipsis
h2.win-type-ellipsis, .win-type-x-large.win-type-ellipsis
body, button, input, textarea, .win-textarea, select, option

Read-only style resource:
//Microsoft.WinJS.2.0/css/ui-dark.css

```
.win-type-x-small {
        font-size: 11pt;
        font-weight: 300;
        line-height: 1.3636;
}
```

FIGURE 2-9 In Blend, the Style Rules tab lets you look into the WinJS stylesheet and see what each particular style contains. Take special notice of the search bar under the tabs where I've typed "win-". This helps you avoid visually scanning for a particular style—just start typing in the box, and let the computer do the work!

The page we'll load into the `iframe`, map.html, is part of our app package that we'll add in a moment, but note how we reference it. The `ms-appx-web:///` protocol indicates that the `iframe` and its contents will run in the web context (introduced in Chapter 1), thereby allowing us to load the remote script for the Bing maps control. The *triple* slash, for its part—or more accurately the third slash—is shorthand for "the current app package" (a value that you can obtain from `document.location.host`) For more details on this and other protocols, see URI schemes in the documentation.

To indicate that a page should be loaded in the local context, the protocol is just `ms-appx`. It's important to remember that no script is shared between these contexts (including variables and functions), relative paths stay in the same context, and communication between the two goes through the HTML5 `postMessage` function, as we'll see later. All of this prevents an arbitrary website from driving your app and accessing WinRT APIs that might compromise user identity and security.

[12] The two standard stylesheets are `ui-dark.css` and `ui-light.css`. Dark styles are recommended for apps that deal with media, where a dark background helps bring out the graphical elements. We'll use this stylesheet because we're doing photo capture. The light stylesheet is recommended for apps that work more with textual content.

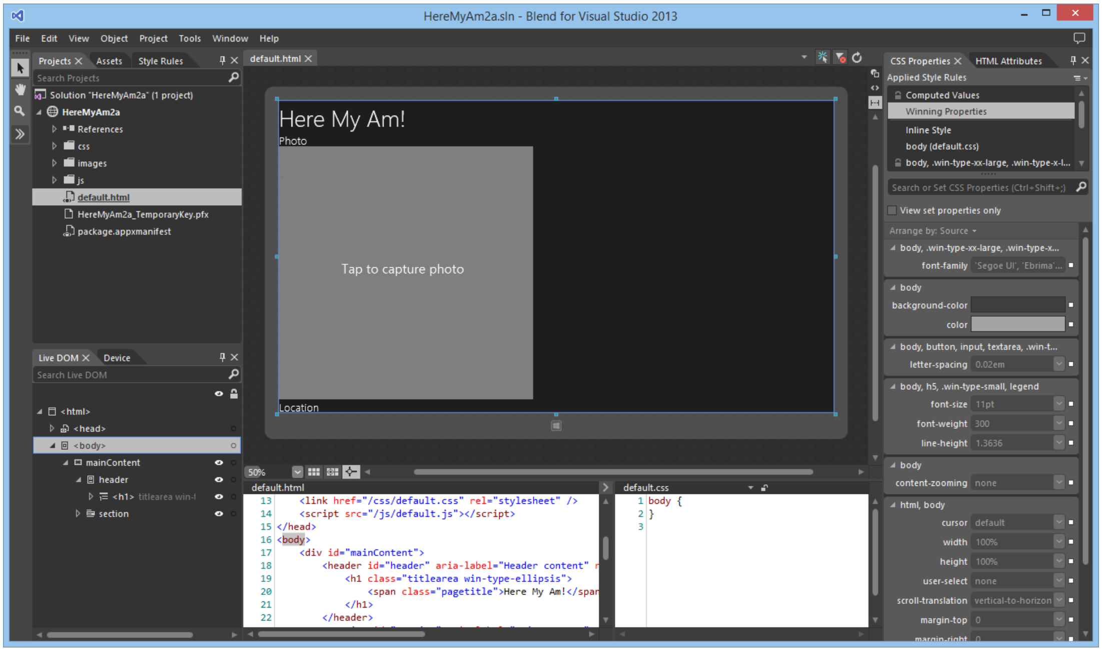
Note I'm using an `iframe` element in this first example because it's probably familiar to most readers. In Chapter 4, "Web Content and Services," we'll change the app to use an `x-ms-webview` element, which is much more flexible than an `iframe` and is the recommended means to host web content.

I've also included various `aria-*` attributes on these elements (as the templates do) that support accessibility. We'll look at accessibility in detail in Chapter 19, "Apps for Everyone, Part 1," but it's an important enough consideration that we should be conscious of it from the start: a majority of Windows users make use of accessibility features in some way. And although some aspects of accessibility are easy to add later on, adding `aria-*` attributes in markup is best done early.

In Chapter 19 we'll also see how to separate strings (including ARIA labels) from our markup, JavaScript, and even the manifest, and place them in a resource file for the purposes of localization. This is something you might want to do from early on, so see the "Preparing for Localization" section in that chapter for the details. Note, however, that resource lookup doesn't work in Blend, so you might want to hold off on the effort until you've done most of your styling.

Styling in Blend

At this point, and assuming you were paying enough attention to read the footnotes, Blend's real-time display of the app shows an obvious need for styling, just like raw markup should. See Figure 2-10.



default.html

```
    <link href="/css/default.css" rel="stylesheet" />
    <script src="/js/default.js"></script>
</head>
<body>
    <div id="mainContent">
        <header id="header" aria-label="Header content" r
            <h1 class="titlearea win-type-ellipsis">
                <span class="pagetitle">Here My Am!</span
            </h1>
        </header>
```

default.css

```
body {
}
```

body, .win-type-xx-large, .win-type-x...
font-family 'Segoe UI', 'Ebrima'...

body
background-color
color

body, button, input, textarea, .win-t...
letter-spacing 0.02em

body, h5, .win-type-small, legend
font-size 11pt
font-weight 300
line-height 1.3636

body
content-zooming none

html, body
cursor default
width 100%
height 100%
user-select none
scroll-translation vertical-to-horizon
margin-top 0
margin-right 0

FIGURE 2-10 The app in Blend without styling, showing a view that is much like the Visual Studio simulator. If the taphere.png image doesn't show after adding it, use the View/Refresh menu command.

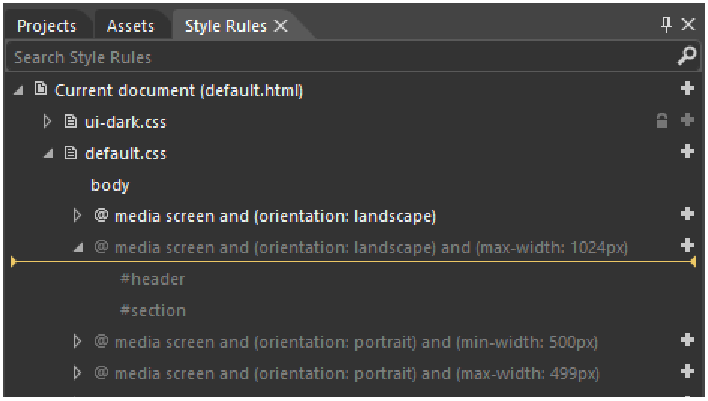
The tabs along the upper left give you access to your Project files, Assets like all the controls you can add to your UI, and a browser for all the Style Rules defined in the environment. On the lower left side, the Live DOM tab lets you browse your element hierarchy and the Device tab lets you set orientation, screen resolution, view sizes and positions, and minimum size. Clicking an element in the Live DOM will highlight it in the designer, and clicking an element in the designer will highlight it in the Live DOM section. Also note the search bar in the Live DOM, where you can enter any CSS selector to highlight those elements that match that selector.

Over on the right side you see what will become a very good friend: the section for HTML Attributes and CSS Properties. With properties, the list at the top shows all the sources for styles that are being applied to the currently selected element and *where*, exactly, those styles are coming from (often a headache with CSS). The location selected in this list, mind you, indicates where changes in the properties pane below will be written, so be very conscious of your selection! That list also contains an item called "Winning Styles," which shows the styles that are actually being applied to the element, and an item called "Computed Values," which will show you the exact *values* applied in the layout engine, such as the actual sizes of rows and columns in a CSS grid and how values like 1.5em translate into pixels.

Now to get our gauche, unstylish page to look like the wireframe, we need to go through the elements and create the necessary selectors and styles. First, I recommend creating a 1x1 grid in the `body` element as this seems to help everything in the app size itself properly. So add `display: -ms-grid; -ms-grid-rows: 1fr; -ms-grid-columns: 1fr;` to default.css for `body`.

CSS grids also make this app's layout fairly simple: we'll just use some of nested grids to place the main sections and the subsections, following the general pattern of styling that works best in Blend:

- Set the insertion point of the style rule within Blend's Style Rules tab by dragging the orange-yellow line control. This determines exactly where any new rule you create will be written. In the image below, new rules would be inserted in default.css at the beginning of the `landscape`/`max-width: 1024px` media query:



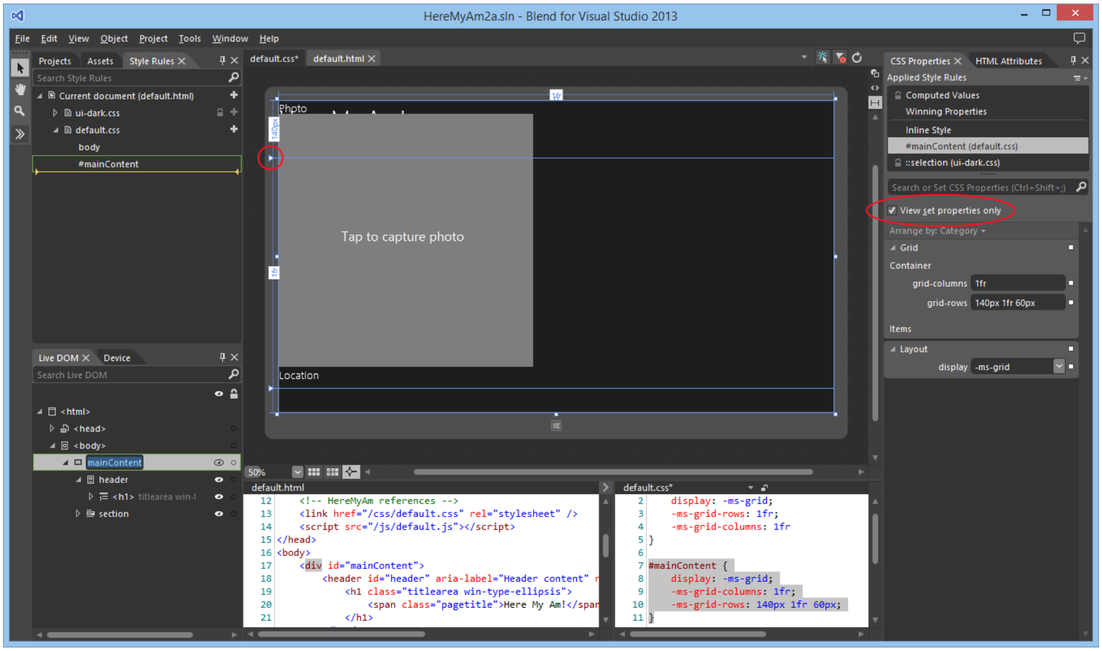
Current document (default.html)
ui-dark.css
default.css
body
@ media screen and (orientation: landscape)
@ media screen and (orientation: landscape) and (max-width: 1024px)
#header
#section
@ media screen and (orientation: portrait) and (min-width: 500px)
@ media screen and (orientation: portrait) and (max-width: 499px)

- In the Live DOM pane (the lower left side in Blend, where you can again search for rules to highlight elements that use them), right-click the element you want to style and select Create Style Rule From Element Id or Create Style Rule From Element Class. This will create a new style rule (at the insertion point indicated in Style Rules above). Then in the CSS properties pane on the right, find the rule that was created and add the necessary style properties.

  > **Note** If the menu items in the Live DOM pane are both disabled, go to the HTML Attributes pane (upper right) and add an id, a class, or both, then return to the menu in the Live DOM. If you do styling without having a distinct rule in place, you'll create inline styles in the HTML, although Blend makes it easy to copy those out and paste them into a rule.

- Repeat with every other element you want to style, which could include `body`, `html`, and so forth, all of which appear in the Live DOM.

So for the *mainContent* `div`, we create a rule from the Id and set it up with `display: -ms-grid; -ms-grid-columns: 1fr; -ms-grid-rows: 140px 1fr 60px;`. (See Figure 2-11.) This creates the basic vertical areas for the wireframes. In general, you won't want to put left or right margins directly in this grid because the lower section will often have horizontally scrolling content that should bleed off the left and right edges. In the case of Here My Am! we could use one grid, but instead we'll add those margins in a nested grid within the `header` and `section` elements.



☑ View set properties only

grid-columns 1fr
grid-rows 140px 1fr 60px
display -ms-grid

default.html

```html
    <!-- HereMyAm references -->
    <link href="/css/default.css" rel="stylesheet" />
    <script src="/js/default.js"></script>
</head>
<body>
    <div id="mainContent">
        <header id="header" aria-label="Header content" r
            <h1 class="titlearea win-type-ellipsis">
                <span class="pagetitle">Here My Am!</spa
            </h1>
```

default.css*

```css
    display: -ms-grid;
    -ms-grid-rows: 1fr;
    -ms-grid-columns: 1fr
}

#mainContent {
    display: -ms-grid;
    -ms-grid-columns: 1fr;
    -ms-grid-rows: 140px 1fr 60px;
}
```

FIGURE 2-11 Setting the grid properties for the *mainContent* `div`. Notice how the View Set Properties Only checkbox (upper right) makes it easy to see what styles are set for the current rule. Also notice how the grid rows and columns appear on the artboard, including sliders (circled) to manipulate rows and columns directly.

Showing this and the rest of the styling—going down into each level of the markup and creating appropriate styles in the appropriate media queries—is best done in video. Video 2-2 (available with this book's companion content) shows this process starting with the creation of the project, styling the different views, and switching to Visual Studio (right-click the project name in Blend and select Edit In Visual Studio) to run the app in the simulator for verification. It also demonstrates the approximate time it takes to style such an app once you're familiar with the tools. (I also highly recommend watching What's New in Blend for HTML Developers from //build 2013, which goes much more in depth with various styling processes.)

The result of all this in the simulator looks just like the wireframes—see Figures 2-12 through 2-14—and all the styling is entirely contained within the appropriate media queries of default.css. Most importantly, the way Blend shows us the results in real time is an enormous time-saver over fiddling with the CSS and running the app over and over again, a painful process that I'm sure you're familiar with! (And the time savings are even greater with Interactive Mode; see Video 5-3 in the companion content created for Chapter 5 and the //build 2013 talk linked above.)

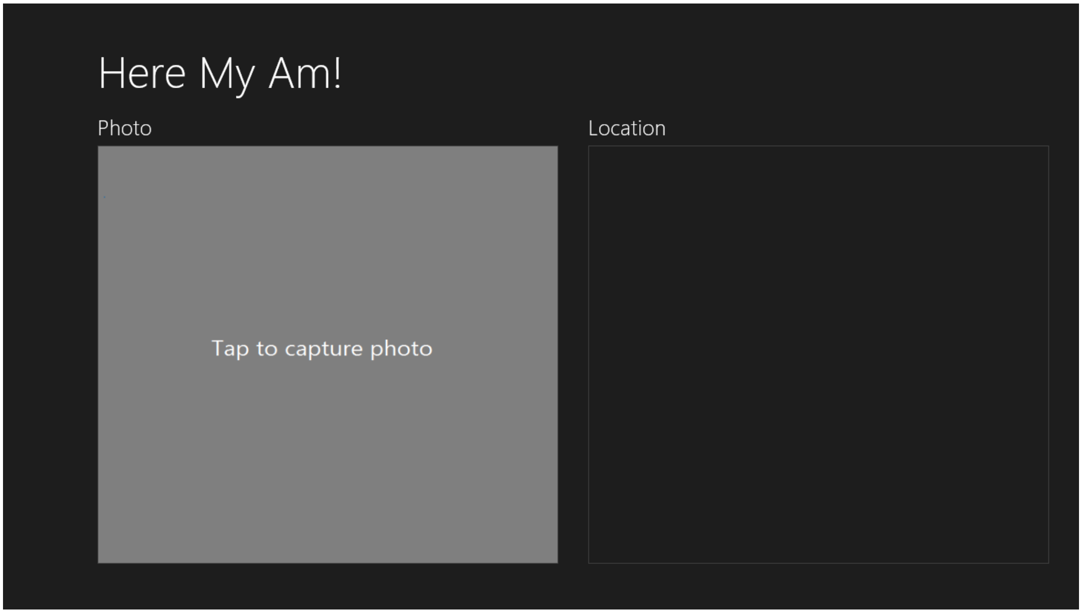


FIGURE 2-12 Full landscape view.

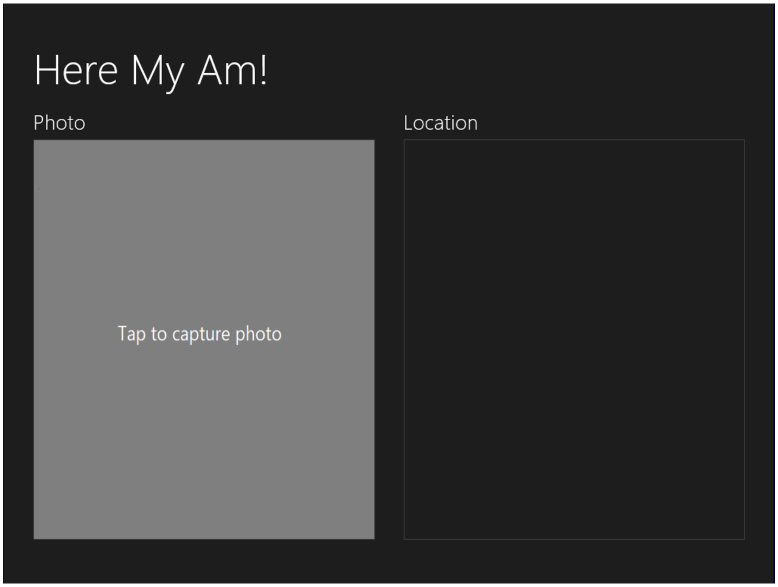


FIGURE 2-13 Partial landscape view (when sharing the screen with other apps).

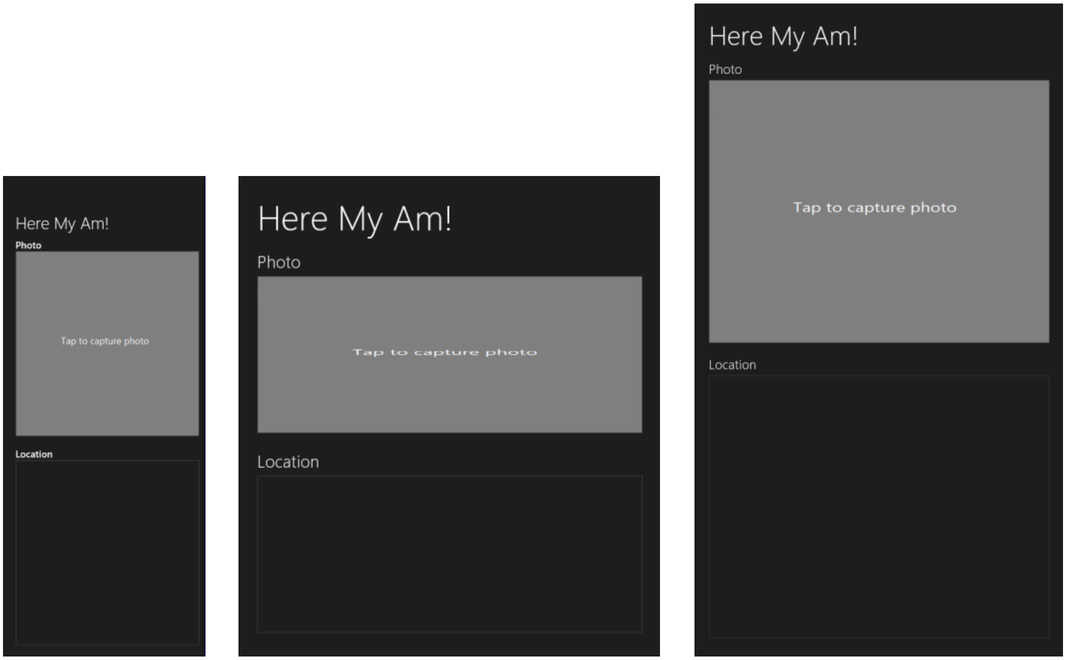


FIGURE 2-14 Narrow aspect ratio views: 320px wide (left), 50% wide (middle), and full portrait (right). These images are not to scale with one another. You can also see that the fixed placeholder image in the Photo section doesn't scale well to the 50% view; we'll solve this later in this chapter in "Improving the Placeholder Image with a Canvas Element."

Adding the Code

Let's complete the implementation now in Visual Studio. Again, right-click the project name in Blend's Project tab and select Edit In Visual Studio if you haven't already. Note that if your project is already loaded into Visual Studio when you switch to it, it will (by default) prompt you to reload changed files. Say yes.[13] At this point, we have the layout and styles for all the necessary views, and our code doesn't need to care about any of it except to make some refinements, as we'll see.

What this means is that, for the most part, we can just write our app's code against the markup and not against the markup plus styling, which is, of course, a best practice with HTML/CSS in general. Here are the features that we'll now implement:

- A Bing maps control in the Location section showing the user's current location. We'll create and display this map automatically.
- Use the WinRT APIs for camera capture to get a photograph in response to a tap on the Photo `img` element.
- Provide the photograph and the location data to the Share charm when the user invokes it.

Figure 2-15 shows what the app will look like when we're done, with the Share charm invoked and a suitable target app like Twitter selected.

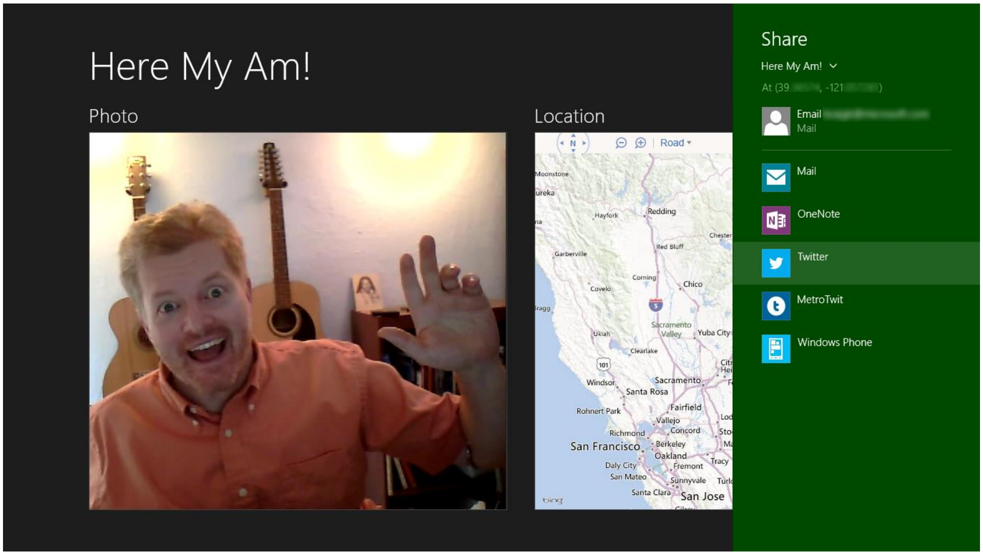


FIGURE 2-15 The completed Here My Am! app with the Share charm invoked (with my exact coordinates blurred out, because they do a pretty accurate job of pinpointing my house).

[13] On the flip side, note that Blend doesn't automatically save files going in and out of Interactive Mode. Be aware, then, if you make a change to the same file open in Visual Studio, switch to Blend, and reload the file, you will lose changes.

Creating a Map with the Current Location

For the map, we're using a Bing maps web control instantiated through the map.html page that's loaded into an `iframe` on the main page (again, we'll switch over to a webview element later on). As we're loading the map control script from a remote source, map.html must be running in the web context. We could employ the Bing Maps SDK here instead, which provides script we can load into the local context. For the time being, I want to use the remote script approach because it gives us an opportunity to work with web content and the web context in general, something that I'm sure you'll want to understand for your own apps. We'll switch to the local control in Chapter 10, "The Story of State, Part 1."

That said, let's put map.html in an *html* folder. Right-click the project and select Add/New Folder (entering **html** to name it). Then right-click that folder, select Add/New Item..., and then select HTML Page. Once the new page appears, replace its contents with the following, and insert your own key for Bing Maps obtained from https://www.bingmapsportal.com/ into the `init` function (highlighted):

```
<!DOCTYPE html>
<html>
    <head>
        <title>Map</title>
        <script type="text/javascript"
            src="http://ecn.dev.virtualearth.net/mapcontrol/mapcontrol.ashx?v=7.0"></script>

        <script type="text/javascript">
            //Global variables here
            var map = null;

            document.addEventListener("DOMContentLoaded", init);
            window.addEventListener("message", processMessage);

            //Function to turn a string in the syntax { functionName: ..., args: [...] }
            //into a call to the named function with those arguments. This constitutes a generic
            //dispatcher that allows code in an iframe to be called through postMessage.
            function processMessage(msg) {
                //Verify data and origin (in this case the local context page)
                if (!msg.data || msg.origin !== "ms-appx://" + document.location.host) {
                    return;
                }

                var call = JSON.parse(msg.data);

                if (!call.functionName) {
                    throw "Message does not contain a valid function name.";
                }

                var target = this[call.functionName];

                if (typeof target != 'function') {
                    throw "The function name does not resolve to an actual function";
                }
                return target.apply(this, call.args);
            }


            //Create the map (though the namespace won't be defined without connectivity)
            function init() {
                if (typeof Microsoft == "undefined") {
                    return;
                }

                map = new Microsoft.Maps.Map(document.getElementById("mapDiv"), {
                    //NOTE: replace these credentials with your own obtained at
                    //http://msdn.microsoft.com/en-us/library/ff428642.aspx
                    credentials: "...",
                    //zoom: 12,
                    mapTypeId: Microsoft.Maps.MapTypeId.road
                });
            }

            function pinLocation(lat, long) {
                if (map === null) {
                    throw "No map has been created";
                }

                var location = new Microsoft.Maps.Location(lat, long);
                var pushpin = new Microsoft.Maps.Pushpin(location, { });
                map.entities.push(pushpin);
                map.setView({ center: location, zoom: 12, });
                return;
            }

            function setZoom(zoom) {
                if (map === null) {
                    throw "No map has been created";
                }

                map.setView({ zoom: zoom });
            }
        </script>
    </head>
    <body>
        <div id="mapDiv"></div>
    </body>
</html>
```

Note that the JavaScript code here could be moved into a separate file and referenced with a relative path, no problem. I've chosen to leave it all together for simplicity.

At the top of the page you'll see a remote script reference to the Bing Maps control. We can again reference remote script here because the page is loaded in the web context within the `iframe` (`ms-appx-web://` in default.html). You can then see that the `init` function is called on `DOMContentLoaded` and creates the map control. Then we have a couple of other methods, `pinLocation` and `setZoom`, which can be called from the main app as needed.[14]

Of course, because this page is loaded in an iframe in the web context, we cannot simply call those functions directly from the local context in which our app code runs. We instead use the HTML5 postMessage function, which raises a message event within the iframe. This is an important point: the local and web contexts are kept separate so that arbitrary web content cannot drive an app or access WinRT APIs (as required by Windows Store certification policy). The two contexts enforce a boundary between an app and the web that can only be crossed with postMessage.

In the code above, you can see that we pick up such messages (the window.onmessage handler) and pass them to the processMessage function, a little generic routine I wrote to turn a JSON string into a local function call, complete with arguments.

To see how this works, let's look at calling pinLocation from within default.js (our local context app code). To make this call, we need some coordinates, which we can get from the WinRT Geolocation APIs. We'll do this within the app's ready event (which fires after the app is fully running). This way the user's location is set on startup and saved in the lastPosition variable for later sharing:

```
//Drop this after the line: var activation = Windows.ApplicationModel.Activation;
var lastPosition = null;
var locator = new Windows.Devices.Geolocation.Geolocator();

//Add this after the app.onactivated handler
app.onready = function () {
    locator.getGeopositionAsync().done(function (geocoord) {
    var position = geocoord.coordinate.point.position;

        //Save for share
        lastPosition = { latitude: position.latitude, longitude: position.longitude };

        callFrameScript(document.frames["map"], "pinLocation",
           [position.latitude, position.longitude]);
    });
}
```

where callFrameScript is another little helper function to turn a target element, function name, and arguments into an appropriate postMessage call:

```
function callFrameScript(frame, targetFunction, args) {
    var message = { functionName: targetFunction, args: args };
    frame.postMessage(JSON.stringify(message), "ms-appx-web://" + document.location.host);
}
```

A few points about all this code. First, in the second parameter to postMessage (both in default.js and map.html) you see ms-appx:// or ms-appx-web:// combined with document.location.host.

[14] Be mindful when using the Bing Maps control that every instance you create is a "billable transaction" that counts against your daily limit depending on your license key. For this reason, avoid creating and destroying map controls across page navigation, as I explain on my blog post, Minimizing billable transactions with Bing Maps.

This essentially means "the current app from the [local or web] context," which is the appropriate origin of the message. We use the same value to check the origin when receiving a message: the code in map.html verifies it's coming from the app's local context, whereas the code in default.js verifies that it's coming from the app's web context. Always make sure to check the origin appropriately; see Validate the origin of postMessage data in Developing secure apps.

Next, to obtain coordinates you can use either the WinRT or HTML5 geolocation APIs. The two are almost equivalent, with the differences described in Chapter 12, "Input and Sensors," in "Sidebar: HTML5 Geolocation." The API exists in WinRT because other supported languages (C# and C++) don't have access to HTML5 APIs. We're focused on WinRT APIs in this book, so we'll just use functions in the `Windows.Devices.Geolocation` namespace.

Note that it's necessary for the WinRT `Geolocation.Geolocator` object to stay in scope while an async location request is happening; otherwise it will cancel the request when a user consent prompt appears (which we'll see shortly). This is why I'm creating it outside the `app.onready` handler.

Finally, the call to `getGeopositionAsync` has an interesting construct, wherein we make the call and chain this function called `done` onto it, whose argument is another function. This is a very common pattern that we'll see while working with WinRT APIs, as any API that might take longer than 50ms to complete runs asynchronously. This conscious decision was made so that the API surface area leads to fast and fluid apps by default.

In JavaScript, async APIs return what's called a *promise* object, which represents results to be delivered at some time in the future. Every promise object has a `done` method whose first argument is the function to be called upon completion, known as the *completed handler* (often an anonymous function). `done` can also take two optional functions to wire up *error* and *progress handlers* as well. We'll see much more about promises as we progress through this book, such as the `then` function that's just like `done` and allows further chaining (Chapter 3), and how promises fit into async operations more generally (Chapter 18). Also, put it deeply into your awareness that anytime you want to stop an uncompleted async operation that's represented by a promise, just call the promise's `cancel` method. It's surprising how often developers forget this!

The argument passed to your completed handler contains the results of the `getGeopositionAsync` call, which in our example above is a `Windows.Geolocation.Geoposition` object containing the last reading. The coordinates from this reading are what we then pass to the `pinLocation` function within the `iframe`, which in turn creates a pushpin on the map at those coordinates and then centers the map view at that same location. (Later in the section "Receiving Messages from the iframe" we'll make the pushpin draggable and show how the app can pick up location changes from the map.)

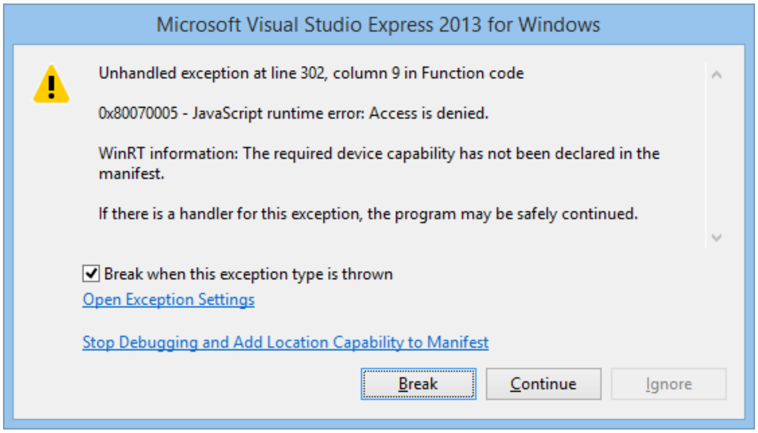
> **Async result types** When reading the docs for an async function, you'll see that the return type is listed like `IAsyncOperation<Geoposition>`; the name within `< >` indicates the actual data type of the results, so refer to the docs on that class for its details. Note also that the `IAsyncOperation` and similar interfaces that exist in WinRT never surface in JavaScript—they are projected as promises.

What's in an (async) name? Within the WinRT API, all async functions have Async in their names. Because this isn't common practice within the DOM API and other JavaScript toolkits, async functions within WinJS don't use that suffix. In other words, WinRT is designed to be language-neutral and follows its own conventions; WinJS consciously follows typical JavaScript conventions.

Oh Wait, the Manifest!

Now you may have tried the code above and found that you get an "Access is denied" exception when you try to call getGeopositionAsync. Why is this? Well, the exception says we neglected to set the *Location* capability in the manifest. Without that capability set, calls that depend on the capability will throw an exception.

If you were running in the debugger, that exception is kindly shown in a dialog box:



Microsoft Visual Studio Express 2013 for Windows

Unhandled exception at line 302, column 9 in Function code

0x80070005 - JavaScript runtime error: Access is denied.

WinRT information: The required device capability has not been declared in the manifest.

If there is a handler for this exception, the program may be safely continued.

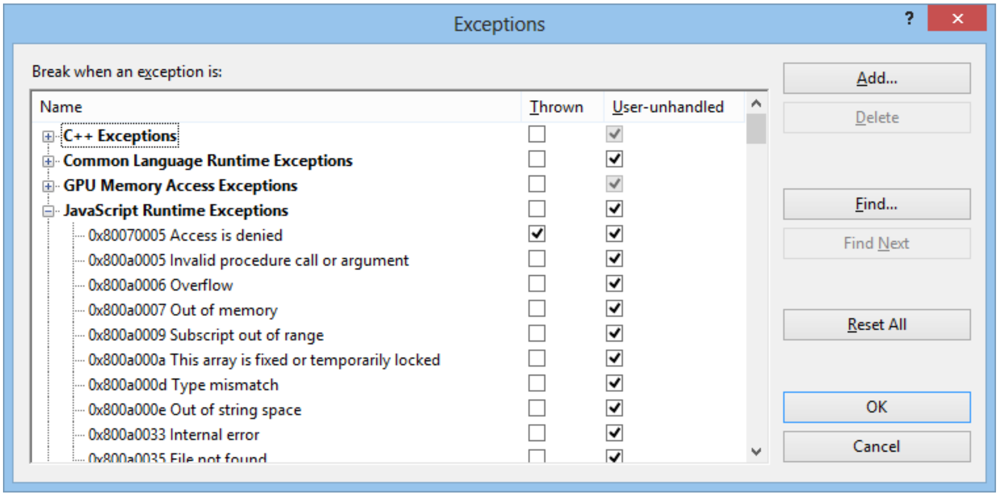
☑ Break when this exception type is thrown

If you run the app outside of the debugger—from the tile on your Start screen—you'll see that the app just terminates without showing anything but the splash screen. This is the default behavior for an unhandled exception. To prevent that behavior, add an error-handling function as the second parameter to the async promise's done method:

```
locator.getGeopositionAsync().done(function (geocoord) {
    //...
}, function(error) {
    console.log("Unable to get location: " + error.message);
});
```

The console.log function writes a string to the *JavaScript Console window* in Visual Studio, which is obviously a good idea (you can also use WinJS.log for this purpose, which allows more customization, as we'll discuss in Chapter 3). Now run the app outside the debugger and you'll see that the app runs, because the exception is now considered "handled." Back in the debugger, set a breakpoint on the console.log line and you'll hit that breakpoint after the exception appears and you press Continue. (This is all we'll do with the error for now; in Chapter 9, "Commanding UI," we'll add a better message and a retry command.)

If the exception dialog gets annoying, you can control which exceptions pop up like this through the Debug > Exceptions dialog box in Visual Studio (shown in Figure 2-16), under JavaScript Runtime Exceptions. If you uncheck the box under User-unhandled, you won't get a dialog when that particular exception occurs.



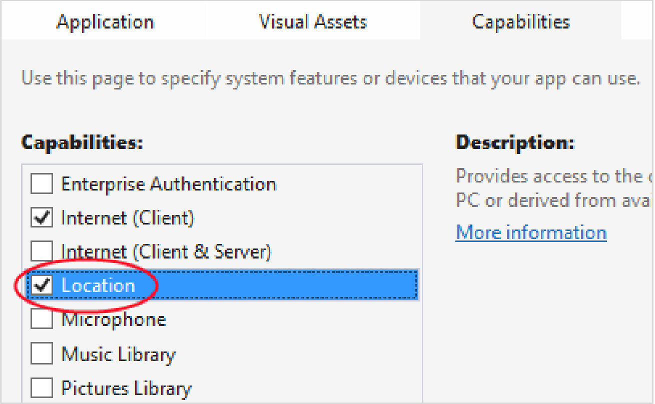
Break when an exception is:

Name	Thrown	User-unhandled
C++ Exceptions	□	☑
Common Language Runtime Exceptions	□	☑
GPU Memory Access Exceptions	□	☑
JavaScript Runtime Exceptions	□	☑
0x80070005 Access is denied	☑	☑
0x800a0005 Invalid procedure call or argument	□	☑
0x800a0006 Overflow	□	☑
0x800a0007 Out of memory	□	☑
0x800a0009 Subscript out of range	□	☑
0x800a000a This array is fixed or temporarily locked	□	☑
0x800a000d Type mismatch	□	☑
0x800a000e Out of string space	□	☑
0x800a0033 Internal error	□	☑
0x800a0035 File not found	□	☑

FIGURE 2-16 JavaScript run-time exceptions in the Debug/Exceptions dialog of Visual Studio.

When the Thrown box is checked for a specific exception (as it is by default for Access is denied to help you catch capability omissions), Visual Studio will always display the "exception occurred" message before your error handler is invoked. If you uncheck Thrown, your error handler will be called without any message.

Back to the capability: to get the proper behavior for this app, open package.appxmanifest in your project, select the Capabilities tab (in the editor UI), and check Location, as shown in Figure 2-17.



Use this page to specify system features or devices that your app can use.

Capabilities:

- □ Enterprise Authentication
- ☑ Internet (Client)
- □ Internet (Client & Server)
- ☑ Location
- □ Microphone
- □ Music Library
- □ Pictures Library

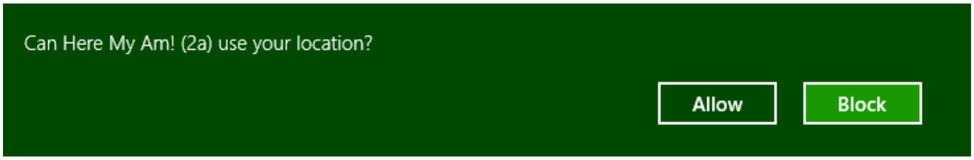
Description:

Provides access to the c
PC or derived from ava

More information

FIGURE 2-17 Setting the *Location* capability in Visual Studio's manifest editor. (Note that Blend supports editing the manifest only as XML.)

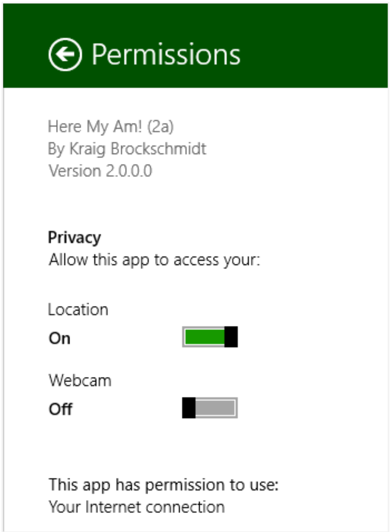
Now, even when we declare the capability, geolocation is still subject to user consent, as mentioned in Chapter 1. When you first run the app with the capability set, then, you should see a popup like this, which appears in the user's chosen color scheme to indicate that it's a message from the system:



Can Here My Am! (2a) use your location?

If the user blocks access here, the error handler will be invoked with an error of "Canceled." (This is also what you get if the Geolocator object goes out of scope while the consent prompt is visible, even if you click Allow, which is again why I create the object outside the `app.onready` handler.)

Keep in mind that this consent dialog will appear only once for any given app, even across debugging sessions (unless you change the manifest or uninstall the app, in which cases the consent history is reset). After that, the user can at any time change their consent in the Settings > Permissions panel as shown in Figure 2-18, and we'll learn how to deal with such changes in Chapter 9. For now, if you want to test your app's response to the consent dialog, go to the Start screen and uninstall the app from its tile. You'll then see the popup when you next run the app.



Here My Am! (2a)
By Kraig Brockschmidt
Version 2.0.0.0

Privacy
Allow this app to access your:

Location
On

Webcam
Off

This app has permission to use:
Your Internet connection

FIGURE 2-18 Any permissions that are subject to user consent can be changed at any time through the Settings Charm > Permissions pane.

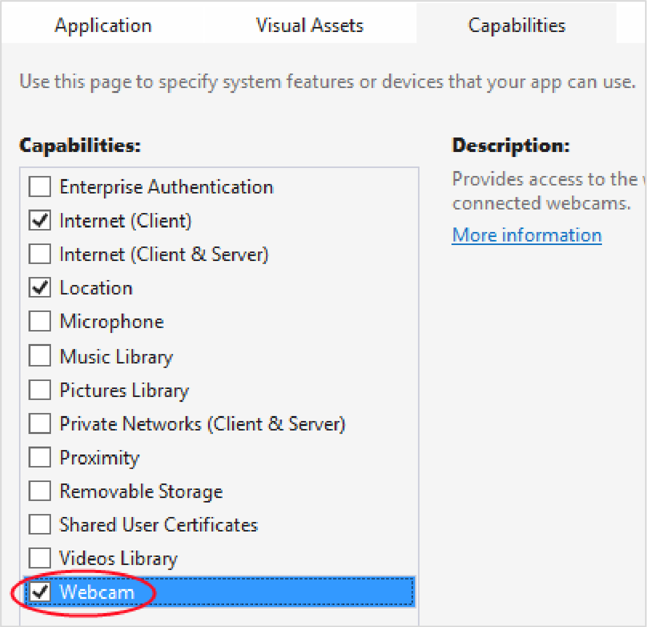
Sidebar: Writing Code in Debug Mode

Because of the dynamic nature of JavaScript, it's impressive that the Visual Studio team figured out how to make the IntelliSense feature work quite well in the Visual Studio editor. (If you're unfamiliar with IntelliSense, it's the productivity service that provides auto-completion for code as well as popping up API reference material directly inline; learn more at JavaScript IntelliSense). That said, a helpful trick to make IntelliSense work even better is to write code while Visual Studio is in debug mode. That is, set a breakpoint at an appropriate place in your code, and then run the app in the debugger. When you hit that breakpoint, you can then start writing and editing code, and because the script context is fully loaded, IntelliSense will be working against instantiated variables and not just what it can derive from the source code. You can also use Visual Studio's Immediate Window to execute code directly to see the results. (You will need to restart the app, however, to execute any new code you write.)

Capturing a Photo from the Camera

In a slightly twisted way, I hope the idea of adding camera capture within a so-called "quickstart" chapter has raised serious doubts in your mind about this author's sanity. Isn't that going to take a whole lot of code? Well, it *used* to, but no longer. The complexities of camera capture have been encapsulated within the `Windows.Media.Capture` API to such an extent that we can add this feature with only a few lines of code. It's a good example of how a little dynamic code like JavaScript combined with well-designed WinRT components—both those in the system and those you can write yourself—are a powerful combination! (You can also write your own capture UI, as we'll see in Chapter 13, "Media," which is presently necessary on Windows Phone as the API we're using here isn't available.)

To implement this feature, we first need to remember that the camera, like geolocation, is a privacy-sensitive device and must also be declared in the manifest, as shown in Figure 2-19.



Use this page to specify system features or devices that your app can use.

Capabilities:

- ☐ Enterprise Authentication
- ☑ Internet (Client)
- ☐ Internet (Client & Server)
- ☑ Location
- ☐ Microphone
- ☐ Music Library
- ☐ Pictures Library
- ☐ Private Networks (Client & Server)
- ☐ Proximity
- ☐ Removable Storage
- ☐ Shared User Certificates
- ☐ Videos Library
- ☑ Webcam

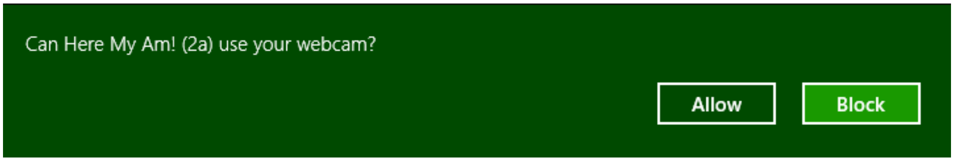
Description:

Provides access to the
connected webcams.

More information

FIGURE 2-19 The camera capability in Visual Studio's manifest editor.

On first use of the camera at run time, you'll see another consent dialog as follows, where again the user can later change their consent in Settings > Permissions (shown earlier in Figure 2-18):


Can Here My Am! (2a) use your webcam?

Next we need to wire up the img element to pick up a tap gesture. For this we simply need to add an event listener for click, which works for all forms of input (touch, mouse, and stylus), as we'll see in Chapter 12:

```
document.getElementById("photo").addEventListener("click", capturePhoto.bind(photo));
```

Here we're providing capturePhoto as the event handler, and using the function object's bind method to make sure the this object inside capturePhoto is bound directly to the img element. The result is that the event handler can be used for any number of elements because it doesn't make any references to the DOM itself:

```
//Place this under var lastPosition = null; (within the app.onactivated handler)
var lastCapture = null;


//Place this after callFrameScript
function capturePhoto() {
    //Due to the .bind() call in addEventListener, "this" will be the image element,
    //but we need a copy for the async completed handler below.
    var captureUI = new Windows.Media.Capture.CameraCaptureUI();
    var that = this;

    //Indicate that we want to capture a JPEG that's no bigger than our target element --
    //the UI will automatically show a crop box of this size.
    captureUI.photoSettings.format = Windows.Media.Capture.CameraCaptureUIPhotoFormat.jpeg;

    captureUI.photoSettings.croppedSizeInPixels =
        { width: that.clientWidth, height: that.clientHeight };

    //Note: this will fail if we're in any view where there's not enough space to display the UI.
    captureUI.captureFileAsync(Windows.Media.Capture.CameraCaptureUIMode.photo)
        .done(function (capturedFile) {
            //Be sure to check validity of the item returned; could be null if the user canceled.
            if (capturedFile) {
                lastCapture = capturedFile;  //Save for Share
                that.src = URL.createObjectURL(capturedFile, { oneTimeOnly: true });
            }
        }, function (error) {
            console.log("Unable to invoke capture UI: " + error.message);
        });
}
```

We *do* need to make a local copy of this within the click handler, though, because once we get inside the async completed handler (the anonymous function passed to captureFileAsync.done) we're in a new function scope and the this object will have changed. The convention for such a copy of this is to call it that. Got that? (You can call it anything, of course.)

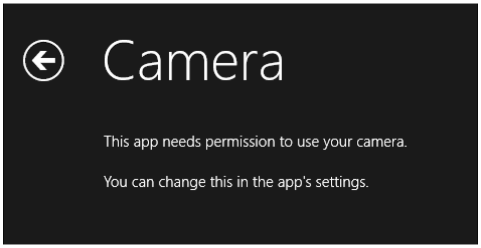
To invoke the camera UI, we only need create an instance of Windows.Media.Capture.-CameraCaptureUI with new (a typical step to instantiate dynamic WinRT objects), configure it with the desired format and size (among many possibilities; see Chapter 13), and then call captureFileAsync. This will check the manifest capability and prompt the user for consent, if necessary (and unlike the Geolocator, a CameraCaptureUI object can go out of scope without canceling the async operation).

This is an async call, so it returns a promise and we hook a .done on the end with our completed handler, which in this case will receive a Windows.Storage.StorageFile object. Through this object you can get to all the raw image data you want, but for our purpose we simply want to display it in the img element. That's easy as well! Data types from WinRT and those in the DOM API are made to interoperate seamlessly, so a StorageFile can be treated like an HTML blob. This means you can hand a WinRT StorageFile object to the HTML URL.createObjectURL method and get back an URI that can be directly assigned to the img.src attribute. The captured photo appears!

Tip The {oneTimeOnly: true} parameter to URL.createObjectURL indicates that the URI is not reusable and should be revoked via URL.revokeObjectURL when it's no longer used, as when we replace img.src with a new picture. Without this, we'd leak memory with each new picture unless you explicitly call URL.revokeObjectURL. (If you've used URL.createObjectURL in the past, you'll see that the second parameter is now a *property bag*, which aligns with the most recent W3C spec.)

Note that captureFileAsync will call the completed handler if the UI was successfully invoked but the user hit the back button and didn't actually capture anything (this includes if you cancel the promise to programmatically dismiss the UI). This is why we do the extra check on the validity of capturedFile. An error handler on the promise will, for its part, pick up failures to invoke the UI in the first place. This will happen if the current view of the app is too small (<500px) for the capture UI to be usable, in which case error.message will say "A method was called at an unexpected time." You can check the app's view size and take other action under such conditions, such as displaying a message to make the view wider. Here we just fail silently; we could also just use the 500px minimum.

Note that a denial of consent will show a message in the capture UI directly, so it's unnecessary to display your own errors with this particular API:



This app needs permission to use your camera.

You can change this in the app's settings.

When this happens, you can again go to Settings > Permissions and give consent to use the camera, as shown in Figure 2-18 earlier.

Sharing the Fun!

Taking a goofy picture of oneself is fun, of course, but sharing the joy with the rest of the world is even better. Up to this point, however, sharing information through different social media apps has meant using the specific APIs of each service. Workable, but not scalable.

Windows 8 instead introduced the notion of the *share contract*, which is used to implement the Share charm with as many apps as participate in the contract. Whenever you're in an app and invoke Share, Windows asks the app for its *source* data, which it provides in one or more formats. Windows then generates a list of *target* apps (according to their manifests) that understand those formats, and displays that list in the Share pane. When the user selects a target, that app is activated and given the source data. In short, the contract is an abstraction that sits between the two, so the source and target apps never need to know anything about each other.

This makes the whole experience all the richer when the user installs more share-capable apps, and it doesn't limit sharing to only well-known social media scenarios. What's also beautiful in the overall experience is that the user never leaves the original app to do sharing—the share target app shows up in its own view as an overlay that only partially obscures the source app (refer back to Figure 2-15). This way, the user remains in the context of the source app and returns there directly when the sharing is completed. In addition, the source data is shared directly with the target app, so the user never needs to save data to intermediate files for this purpose.

So instead of adding code to our app to share the photo and location to a particular target, like Facebook or Twitter, we need only package the data appropriately when Windows asks for it. That asking comes through the `datarequested` event sent to the `Windows.ApplicationModel.DataTransfer.DataTransferManager` object.[15] First we just need to set up an appropriate listener—place this code is in the `activated` event in default.js after setting up the `click` listener on the `img` element:

```
var dataTransferManager =
    Windows.ApplicationModel.DataTransfer.DataTransferManager.getForCurrentView();
dataTransferManager.addEventListener("datarequested", provideData);
```

> **Note** The notion of a *current view* as we see here is a way of saying, "get the singular instance of this system object that's related to the current window," which supports the ability for an app to have multiple windows/views (see Chapter 8). You use `getForCurrentView` instead of creating an instance with `new` because you only ever need one instance of such objects for any given view. `getForCurrentView` will instantiate the object if necessary, or return one that's already available.

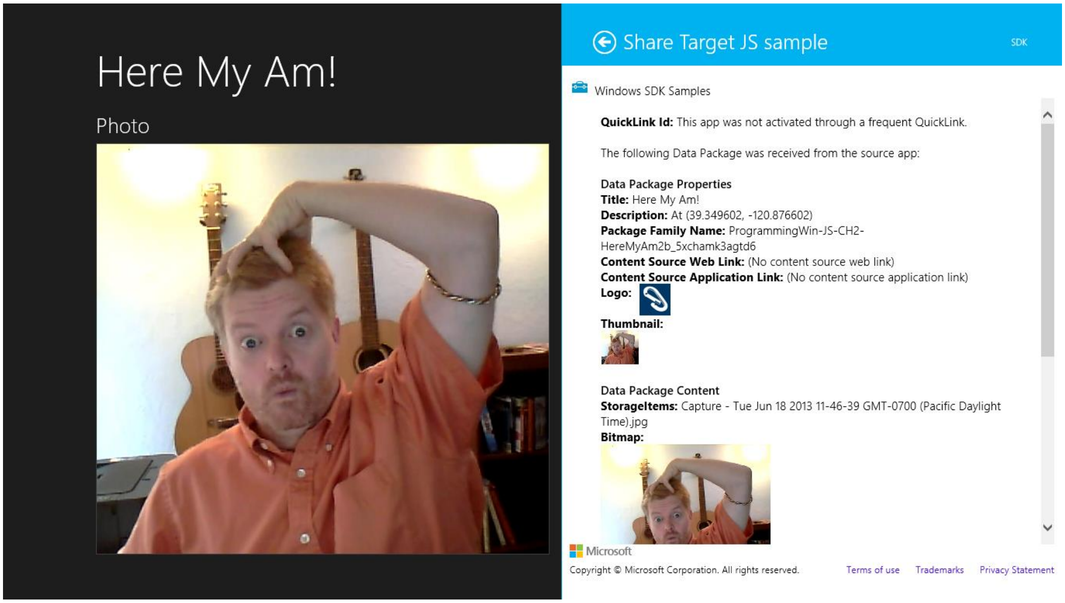
[15] Because we're always listening to `datarequested` while the app is running and add a listener only once, we don't need to worry about calling `removeEventListener`. For details, see "WinRT Events and removeEventListener" in Chapter 3.

For this event, the handler receives a `Windows.ApplicationModel.DataTransfer.DataRequest` object in the event args (`e.request`), which in turn holds a `DataPackage` object (`e.request.data`). To make data available for sharing, you populate this data package with the various formats you have available, as we've saved in `lastPosition` and `lastCapture`. So in our case, we make sure we have position and a photo and then fill in text and image properties (if you want to obtain a map from Bing for sharing purposes, see Get a static map):

```
//Drop this in after capturePhoto
function provideData(e) {
    var request = e.request;
    var data = request.data;

    if (!lastPosition || !lastCapture) {
        //Nothing to share, so exit
        return;
    }

    data.properties.title = "Here My Am!";
    data.properties.description = "At ("
        + lastPosition.latitude + ", " + lastPosition.longitude + ")";

    //When sharing an image, include a thumbnail
    var streamReference =
        Windows.Storage.Streams.RandomAccessStreamReference.createFromFile(lastCapture);
    data.properties.thumbnail = streamReference;

    //It's recommended to always use both setBitmap and setStorageItems for
    // sharing a single image since the target app may only support one or the other.

    //Put the image file in an array and pass it to setStorageItems
    data.setStorageItems([lastCapture]);

    //The setBitmap method requires a RandomAccessStream.
    data.setBitmap(streamReference);
}
```

The latter part of this code is pretty standard stuff for sharing a file-based image (which we have in `lastCapture`). I got most of this code, in fact, directly from the Share content source app sample, which we'll look at more closely in Chapter 15, "Contracts." We'll also talk more about files and streams in Chapter 10.

With this last addition of code, and a suitable sharing target installed (such as the Share content target app sample, as shown in Figure 2-20, or Twitter as shown in Figure 2-21), we now have a very functional app—in all of 35 lines of HTML, 125 lines of CSS, and less than 100 lines of JavaScript!



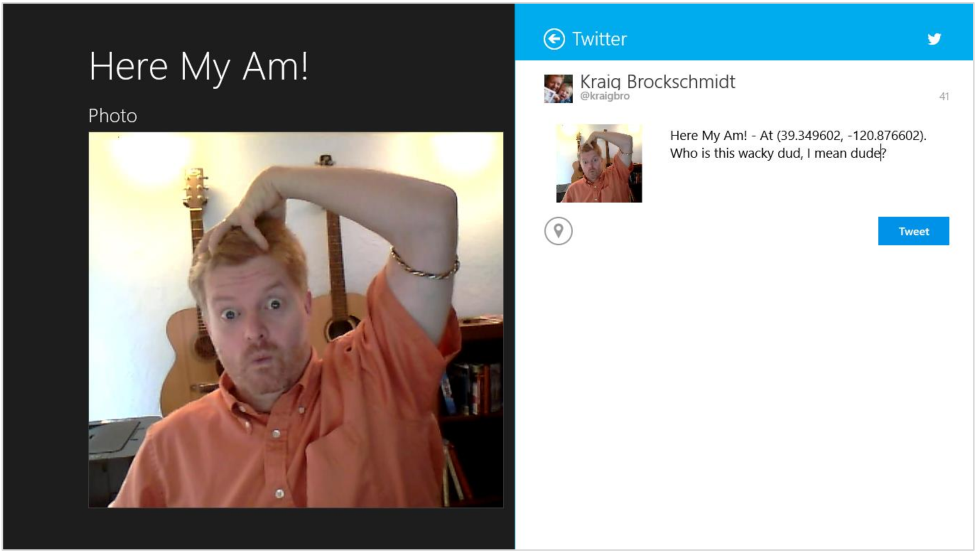
QuickLink Id: This app was not activated through a frequent QuickLink.

The following Data Package was received from the source app:

Data Package Properties
Title: Here My Am!
Description: At (39.349602, -120.876602)
Package Family Name: ProgrammingWin-JS-CH2-HereMyAm2b_5xchamk3agtd6
Content Source Web Link: (No content source web link)
Content Source Application Link: (No content source application link)
Logo:
Thumbnail:

Data Package Content
StorageItems: Capture - Tue Jun 18 2013 11-46-39 GMT-0700 (Pacific Daylight Time).jpg
Bitmap:

FIGURE 2-20 Sharing (monkey-see, monkey-do!) to the Share target sample in the Windows SDK, which is highly useful for debugging as it displays information about all the formats the source app has shared. (And if you still think I've given you coordinates to my house, the ones shown here will send you some miles down the road where you'll make a fine acquaintance with the Tahoe National Forest.)



Here My Am! - At (39.349602, -120.876602). Who is this wacky dud, I mean dude?

FIGURE 2-21 Sharing to Twitter. The fact that Twitter's brand color is nearly identical to the Windows SDK is sheer coincidence. The header color of the sharing pane always reflects the target app's specific color.

Extra Credit: Improving the App

The Here My Am! app as we've built it so far is nicely functional and establishes core flow of the app, and you can find this version in the HereMyAm2a folder of the companion content. However, there are some functional deficiencies that we could improve:

- Because geolocation isn't always as accurate as we'd like, the pushpin location on the map won't always be where we want it. To correct this, we can make the pin draggable and report its updated position to the app via `postMessage` from the `iframe` to the app. This will also complete the interaction story between local and web contexts.
- The placeholder image that reads "Tap to capture photo" works well in some views, but looks terrible in others (such as the 50% view as seen in Figure 2-14). We can correct this, and simplify localization and accessibility concerns later on, by drawing the text on a `canvas` element and using it as the placeholder.
- Auto-cropping the captured image to the size of the photo display area takes control away from users who might like to crop the image themselves. Furthermore, as we change views in the app, the image just gets scaled to the new size of the photo area without any concern for preserving aspect ratio. By keeping that aspect ratio in place, we can then allow the user to crop however they want and adapt well across different view sizes.
- By default, captured images are stored in the app's temporary app data folder. It'd be better to move those images to local app data, or even to the Pictures library, so we could later add the ability to load a previously captured image (as we'll do in Chapter 9 when we implement an app bar command for this purpose).
- Originally we used `URL.createObjectURL` directly on an image's `StorageFile`. Because many images are somewhat larger than most displays, this can use more memory than is necessary. It's better, for consumption scenarios, to use a thumbnail instead.

The sections that follow explore all these details and together produce the HereMyAm2b app in the companion content.

Note For the sake of simplicity, we'll not separate strings (like the text for the `canvas` element) into a resource file as you typically want to do for localization. This gives us the opportunity in Chapter 19 to explore where such strings appear throughout an app and how to extract them. If you're starting your own project now, however, you might want to read the section "World Readiness and Globalization" in Chapter 19 right away so you can properly structure your resources from the get-go.

Sidebar: Debug or Release?

Because JavaScript code is interpreted at run time instead of being compiled, it lacks conditional compilation directives like `#ifdef` that are commonly used in languages like C++ to provide separate code for Debug and Release builds. Fortunately, it's not difficult to edit your project file to detect the Visual Studio build target and then selectively copy a debug or release specific file into the resulting package. This helps you isolate target-specific variables and methods and avoid littering the rest of your code with a bunch of if statements. I explain the details of doing this on my blog, A reliable way to differentiate Debug and Release builds for JavaScript apps.

A similar question is whether you can write generic JavaScript code that could be used in a Windows Store app or a web application. A reasonable way to detect the run-time environment is to check for the existence of the MSApp object and one of its member functions. See my StackOverflow answer on Conditional statement to check for Win8 or iOS.

Receiving Messages from the iframe

Just as app code in the local context can use postMessage to send information to an iframe in the web context, the iframe can use postMessage to send information to the app. In our case, we want to know when the location of the pushpin has changed so that we can update lastPosition.

First, here's a simple utility function I added to map.html to encapsulate the appropriate postMessage calls to the app from the iframe:

```
function notifyParent(event, args) {
    //Add event name to the arguments object and stringify as the message
    args["event"] = event;
    window.parent.postMessage(JSON.stringify(args), "ms-appx://" + document.location.host);
}
```

This function basically takes an event name, adds it to an object containing parameters, stringifies the whole thing, and then posts it back to the parent.

To make a pushpin draggable, we simply add the draggable: true option when we create it in the pinLocation function (in map.html):

```
var pushpin = new Microsoft.Maps.Pushpin(location, { draggable: true });
```

When a pushpin is dragged, it raises a dragend event. We can wire up a handler for this in pinLocation just after the pushpin is created, which then calls notifyParent with a suitable event:

```
Microsoft.Maps.Events.addHandler(pushpin, "dragend", function (e) {
    var location = e.entity.getLocation();
    notifyParent("locationChanged",
        { latitude: location.latitude, longitude: location.longitude });
});
```

Back in default.js (the app), we add a listener for incoming messages inside app.onactivated:

```
window.addEventListener("message", processFrameEvent);
```

where the processFrameEvent handler looks at the event in the message and acts accordingly:

```
function processFrameEvent (message) {
    //Verify data and origin (in this case the web context page)
    if (!message.data || message.origin !== "ms-appx-web://" + document.location.host) {
        return;
    }

    if (!message.data) {
        return;
    }

    var eventObj = JSON.parse(message.data);

    switch (eventObj.event) {
        case "locationChanged":
            lastPosition = { latitude: eventObj.latitude, longitude: eventObj.longitude };
            break;

        default:
            break;
    }
};
```

Clearly, this is more code than we'd need to handle a single message or event from an `iframe`, but I wanted to give you something that could be applied more generically in your own apps. In any case, these additions now allow you to drag the pin to update the location on the map and thus also the location shared through the Share charm.

Improving the Placeholder Image with a Canvas Element

Although our default placeholder image, /images/taphere.png, works well in a number of views, it gets inappropriately squashed or stretched in others. We could create multiple images to handle these cases, but that will bloat our app package and make our lives more complicated when we look at variations for pixel density (Chapter 3) along with contrast settings and localization (Chapter 19). To make a long story short, handling different pixel densities can introduce up to four variants of an image, contrast concerns can introduce four more variants, and localization introduces as many variants as the languages you support. So if, for example, we had three basic variants of this image and multiplied that with four pixel densities, four contrasts, and ten languages, we'd end up with 48 images per language or 480 across all languages! That's too much to maintain, for one, and that many images will dramatically bloat the size of your app package (although the Windows Store manages resource packaging such that users download only what they need).

Fortunately, there's an easy way to solve this problem across all variations, which is to just draw the text we need (for which we can adjust contrast and use a localized string later on) on a `canvas` element and then use the HTML blob API to display that canvas in an `img` element. Here's a routine that does all of that, which we call within `app.onready` (to make sure document layout has happened):

```
function setPlaceholderImage() {
    //Ignore if we have an image (shouldn't be called under such conditions)
    if (lastCapture != null) {
        return;
    }
    var photo = document.getElementById("photo");
    var canvas = document.createElement("canvas");
    canvas.width = photo.clientWidth;
    canvas.height = photo.clientHeight;

    var ctx = canvas.getContext("2d");
    ctx.fillStyle = "#7f7f7f";
    ctx.fillRect(0, 0, canvas.width, canvas.height);
    ctx.fillStyle = "#ffffff";

    //Use 75% height of the photoSection heading for the font
    var fontSize = .75 *
        document.getElementById("photoSection").querySelector("h2").clientHeight;
    ctx.font = "normal " + fontSize + "px 'Arial'";
    ctx.textAlign = "center";
    ctx.fillText("Tap to capture photo", canvas.width / 2, canvas.height / 2);

    var img = photo.querySelector("img");

    //The blob should be released when the img.src is replaced
    img.src = URL.createObjectURL(canvas.msToBlob(), { oneTimeOnly: true });
}u
```

Here we're simply creating a `canvas` element that's the same width and height as the photo display area, but we don't attach it to the DOM (no need). We draw our text on it with a size that's proportional to the photo section heading. Then we obtain a blob for the canvas using its `msToBlob` method, hand it to our friend `URL.createObjectURL`, and assign the result to the `img.src`. Voila!

Because the `canvas` element will be discarded once this function is done (that variable goes out of scope) and because we make a `oneTimeOnly` blob from it, we can call this function anytime the photo section is resized, which we can detect with the `window.onresize` event. We need to use this same event to handle image scaling, so let's see how all that works next.

Handling Variable Image Sizes

If you've been building and playing with the app as we've described it so far, you might have noticed a few problems with the photo area besides the placeholder image. For o ne, if the resolution of the camera is not sufficient to provide a perfectly sized image as indicated by our cropping size, the captured image will be scaled to fit the photo area without concern for preserving the aspect ratio (see Figure 2-22, left side). Similarly, if we change views (or display resolution) after any image is captured, the photo area gets resized and the image is again scaled to fit, without always producing the best results (see Figure 2-22, right side).

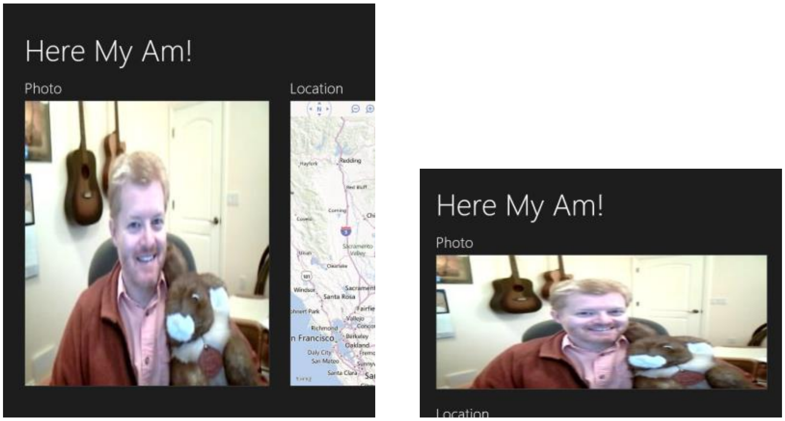


FIGURE 2-22 Poor image scaling with a low-resolution picture from the camera where the captured image isn't inherently large enough for the display area (left), and even worse results in the 50% view when the display area's aspect ratio changes significantly.

To correct this, we'll need to dynamically determine the largest image dimension we can use within the current display area and then scale the image to that size while preserving the aspect ratio and keeping the image centered in the display. For centering purposes, the easiest solution I've found to this is to create a surrounding `div` with a CSS grid wherein we can use row and column centering. So in default.html:

```
<div id="photo" class="graphic">
    <img id="photoImg" src="#" alt="Tap to capture image from camera" role="img" />
</div>
```

and in default.css:

```
#photo {
    display: -ms-grid;
    -ms-grid-columns: 1fr;
    -ms-grid-rows: 1fr;
}

#photoImg {
    -ms-grid-column-align: center;
    -ms-grid-row-align: center;
}
```

The `graphic` style class on the `div` always scales to 100% width and height of its grid cell, so the one row and column within it will also occupy that full space. By adding the centering alignment to the `photoImg` child element, we know that the image will be centered regardless of its size.

To scale the image in this grid cell, then, we either set the image element's `width` style to 100% if its aspect ratio is greater than that of the display area, or set its `height` style to 100% if the opposite is true. For example, on a 1366x768 display, the size of the display area in landscape view is 583x528 for an aspect ratio of 1.1, and let's say we get an 800x600 image back from camera capture with an aspect ratio of 1.33. In this case the image is scaled to 100% of the display area width, making the displayed image 583x437 with blank areas on the top and bottom. Conversely, in 50% view the display area on the same screen is 612x249 with a ratio of 2.46, so we scale the 800x600 image to 100% height, which comes out to 332x249 with blank areas on the left and right.

The size of the display area is readily obtained through the clientWidth and clientHeight properties of the surrounding div we added to the HTML. The actual size of the captured image is then readily available through its StorageFile object's properties.getImagePropertiesAsync method. Putting all this together, here's a function that sets the appropriate style on the img element given its parent div and the captured file:

```
function scaleImageToFit(imgElement, parentDiv, file) {
    file.properties.getImagePropertiesAsync().done(function (props) {
        var scaleToWidth =
            (props.width / props.height > parentDiv.clientWidth / parentDiv.clientHeight);
        imgElement.style.width = scaleToWidth ? "100%" : "";
        imgElement.style.height = scaleToWidth ? "" : "100%";
    }, function (e) {
        console.log("getImageProperties error: " + e.message);
    });
}
```

With this in place, we can simply call this in our existing capturePhoto function immediately after we assign a new image to the element:

```
img.src = URL.createObjectURL(capturedFile, { oneTimeOnly: true });
scaleImageToFit(img, photoDiv, capturedFile);
```

To handle view changes and anything else that will resize the display area, we can add a resize handler within app.onactivated:

```
window.addEventListener("resize", scalePhoto);
```

where the scalePhoto handler can call scaleImageToFit if we have a captured image or the setPlaceholderImage function we created in the previous section otherwise:

```
function scalePhoto() {
    var photoImg = document.getElementById("photoImg");

    //Make sure we have an img element
    if (photoImg == null) {
        return;
    }

    //If we have an image, scale it, otherwise regenerate the placeholder
    if (lastCapture != null) {
        scaleImageToFit(photoImg, document.getElementById("photo"), lastCapture);
    } else {
        setPlaceholderImage();
    }
}
```

With such accommodations for scaling, we can also remove the line from `capturePhoto` that set `captureUI.photoSettings.croppedSizeInPixels`, thereby allowing us to crop the captured image however we like. Figure 2-23 shows these improved results.

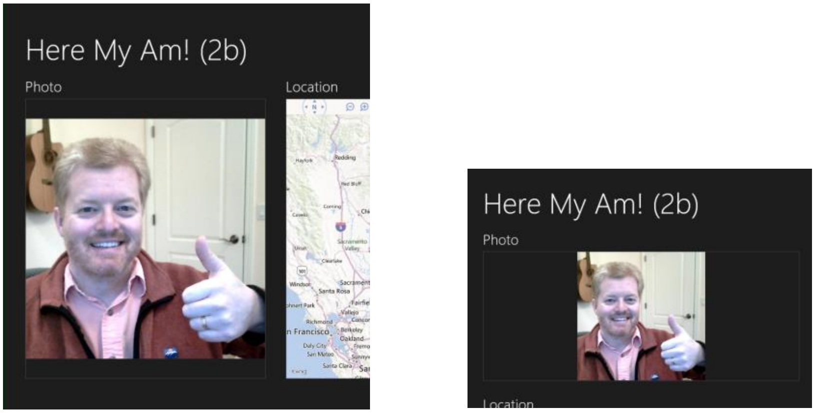


FIGURE 2-23 Proper image scaling after making the improvements.

Moving the Captured Image to AppData (or the Pictures Library)

If you take a look in Here My Am! TempState folder within its appdata, you'll see all the pictures you've taken with the camera capture UI. If you set a breakpoint in the debugger and look at `capturedFile`, you'll see that it has an ugly file path like *C:\Users\kraigb\AppData\Local\Packages\ ProgrammingWin-JS-CH2-HereMyAm2b_5xchamk3agtd6\TempState\picture001.png*. Egads. Not the friendliest of locations, and definitely not one that we'd want a typical consumer to ever see!

Because we'll want to allow the user to reload previous pictures later on (see Chapter 10), it's a good idea to move these images into a more reliable location. Otherwise they could disappear at any time if the user runs the Disk Cleanup tool.

> **Tip** For quick access to the appdata folders for your installed apps, type *%localappdata%/packages* into the path field of Windows Explorer or in the Run dialog (Windows+R key). Easier still, just make a shortcut on your desktop, Start screen, or task bar.

For the purposes of this exercise, we'll move each captured image into a HereMyAm folder within our local appdata and also rename the file in the process to add a timestamp. In doing so, we can also briefly see how to use an `ms-appdata:///local/` URI to directly refer to those images within the `img.src` attribute. (This protocol is described in URI schemes along with its roaming and temp variants, the `ms-appx` protocol for in-package contents, and the `ms-resource` protocol for resources, as described in Chapter 19.) I say "briefly" here because in the next section we'll change this code to use a thumbnail instead of the full image file.

To move the file, we can use its built-in StorageFile.copyAsync method, which requires a target StorageFolder object and a new name, and then delete the temp file with its deleteAsync method.

The target folder is obtained from Windows.Storage.ApplicationData.current.localFolder. The only real trick to all of this is that we have to chain together multiple async operations. We'll discuss this in more detail in Chapter 3, but the way you do this is to have each completed handler in the chain return the promise from the next async operation in the sequence, and to use then for each step except for the last, when we use done. The advantage to this is that we can throw any exceptions along the way and they'll be picked up in the error handler given to done. Here's how it looks in a modified capturePhoto function:

```
var img = photoDiv.querySelector("img");
var capturedFile;

captureUI.captureFileAsync(Windows.Media.Capture.CameraCaptureUIMode.photo)
    .then(function (capturedFileTemp) {
        if (!capturedFileTemp) { throw ("no file captured"); }
        capturedFile = capturedFileTemp;

        //Open the HereMyAm folder, creating it if necessary
        var local = Windows.Storage.ApplicationData.current.localFolder;
        return local.createFolderAsync("HereMyAm",
            Windows.Storage.CreationCollisionOption.openIfExists);

        //Note: the results from the returned promise are fed into the
        //completed handler given to the next then in the chain.
    })
    .then(function (myFolder) {
        //Again, check validity of the result
        if (!myFolder) { throw ("could not create local appdata folder"); }

        //Append file creation time to the filename (should avoid collisions,
        //but need to convert colons)
        var newName = " Capture - " +
            capturedFile.dateCreated.toString().replace(/:/g, "-") + capturedFile.fileType;

        //Make the copy
        return capturedFile.copyAsync(myFolder, newName);
    })
    .then(function (newFile) {
        if (!newFile) { throw ("could not copy file"); }

        lastCapture = newFile;  //Save for Share
        img.src = "ms-appdata:///local/HereMyAm/" + newFile.name;
        //newFile.name includes extension

        scaleImageToFit(img, photoDiv, newFile);

        //Delete the temporary file
        return capturedFile.deleteAsync();
    })
    //No completed handler needed for the last operation
    .done(null, function (error) {
        console.log("Unable to invoke capture UI:" + error.message);
    });
```

This might look a little complicated to you at this point, but trust me, you'll quickly become accustomed to this structure when dealing with multiple async operations. If you can look past all the syntactical ceremony here and simply follow the words *Async* and *then,* you can see that the sequence of operations is simply this:

- Capture an image from the camera capture UI, resulting in a temp file, then…
- Create or open the HereMyAm folder in local appdata, resulting in a folder object, then…
- Copy the captured file to that folder, resulting in a new file, then…
- Delete the temp file, which has no results, and we're done.

To help you follow the chain, I've use different colors in the code above to highlight each set of async calls and their associated `then` methods and results, along with a final call to `done`. What works very well about this chaining structure—which is much cleaner than trying to nest operations within each completed handler—is that any exceptions that occur, whether from WinRT or a direct `throw`, are shunted to the one error handler at the end, so we don't need separate error handlers for every operation (although you can if you want).

Finally, by changing two lines of this code and—very importantly—declaring the *Pictures library* capability in the manifest, you can move the files to the Pictures library instead. Just change the line to obtain `localFolder` to this instead:

```
var local = Windows.Storage.KnownFolders.picturesLibrary;
```

and use `URL.createObjectUrl` with the `img` element instead of the `ms-appdata` URI:

```
img.src = URL.createObjectURL(newFile, {oneTimeOnly: true});
```

as there isn't a URI scheme for the pictures library. Of course, the line above works just fine for a file in local appdata, but I wanted to give you an introduction to the `ms-appdata://` protocol. Again, we'll be removing this line in the next section, so in the example code you'll only see it in comments.

Using a Thumbnail Instead of the Full Image

As we'll learn in Chapter 11, "The Story of State, Part 2," most image consumption scenarios never need to load an entire image file into memory. Images from digital cameras, for example, are often much larger than most displays, so the image will almost always be scaled down even when shown full screen. Unless you're showing the zoomed-in image or providing editing features, then, it's more memory efficient to use thumbnails for image display rather than just passing a `StorageFile` straight to `URL.createObjectURL`. This is especially true when loading many images into a collection control.

To obtain a thumbnail, use either `StorageFile.getThumbnailAsync` or `StorageFile.getScaledImageAsThumbnailAsync`, where the former always relies on the thumbnail cache whereas the latter will use the full image as a fallback. For the purposes of Here My Am!, we'll want to use the latter. First we need to remove the `img.src` assignment inside `capturePhoto`, then have the `scaleImageToFit` function load up the thumbnail:

```
function scaleImageToFit(imgElement, parentDiv, file) {
    file.properties.getImagePropertiesAsync().done(function (props) {
        var requestedSize;
        var scaleToWidth =
            (props.width / props.height > parentDiv.clientWidth / parentDiv.clientHeight);

        if (scaleToWidth) {
            imgElement.style.width = "100%";
            imgElement.style.height = "";
            requestedSize = parentDiv.clientWidth;
        } else {
            imgElement.style.width = "";
            imgElement.style.height = "100%";
            requestedSize = parentDiv.clientHeight;
        }

        //Using a thumbnail is always more memory efficient unless you really need all the
        //pixels in the image file.

        //Align the thumbnail request to known caching sizes (for non-square aspects).
        if (requestedSize > 532) { requestedSize = 1026; }
            else { if (requestedSize > 342) { requestedSize = 532; }
            else { requestedSize = 342; }}

        file.getScaledImageAsThumbnailAsync(
            Windows.Storage.FileProperties.ThumbnailMode.singleItem, requestedSize)
            .done(function (thumb) {
                imgElement.src = URL.createObjectURL(thumb, { oneTimeOnly: true });
            });
    }
}
```

As we'll see in Chapter 11, the `ThumbnailMode.singleItem` argument to `getScaledImageAsThumbnailAsync` is the best mode for loading a larger image, and the second argument specifies the requested size, which works best when aligned to known cache sizes (190, 266, 342, 532, and 1026 for non-square aspects). The resulting thumbnail of this operation is conveniently something you can again pass directly to `URL.createObjectURL`, but ensures that we load only as much image data as we need for our UI.

With that, we've completed our improvements to Here My Am!, which you can again find in the HereMyAm2b example with this chapter's companion content. And I think you can guess that this is only the beginning: we'll be adding many more features to this app as we progress through this book!

The Other Templates: Projects and Items

In this chapter we've worked with only the Blank App template so that we could understand the basics of writing a Windows Store app without any other distractions. In Chapter 3, we'll look more deeply at the anatomy of apps through a few of the other templates, yet we won't cover them all. To close this chapter, then, here's a short introduction to these handy tools to get you started on your own projects.

Navigation App Template

"A project for a Windows Store app that has predefined controls for navigation."
(Blend/Visual Studio description)

The Navigation template builds on the Blank template by adding support for "page" navigation, where the pages in question are more sections of content than distinct pages like we know on the Web. As discussed in Chapter 1, Windows Store apps written in JavaScript are best implemented by having a single HTML page container into which other pages are dynamically loaded. This allows for smooth transitions (as well as animations) between those pages and preserves the script context. Many web apps, in fact, use this single-page approach.

The Navigation template, and the others that remain, employ a Page Navigator control that facilitates loading (and unloading) pages in this way. You need only create a relatively simple structure to describe each page and its behavior. We'll see this in—you guessed it—Chapter 3.

In this model, default.html is little more than a simple container, with everything else in the app coming through subsidiary pages. The Navigation template creates only one subsidiary page, yet it establishes the framework for how to work with multiple pages. Additional pages are easily added to a project through a page item template (right click a folder in your project in Visual Studio and select Add > New Item > Page Control).

Grid App Template

"A three-page project for a Windows Store app that navigates among grouped items arranged in a grid. Dedicated pages display group and item details." (Blend/Visual Studio description)

Building on the Navigation template, the Grid template provides the basis for apps that will navigate collections of data across multiple pages. The home page shows grouped items within the collection, from which you can then navigate into the details of an item or into the details of a group and its items (from which you can then go into individual item details as well).

In addition to the navigation, the Grid template also shows how to manage collections of data through the `WinJS.Binding.List` class, a topic we'll explore much further in Chapter 7, "Collection Controls." It also provides the structure for an app bar and shows how to simplify the app's behavior in narrow views.

The name of the template, by the way, derives from the particular grid *layout* used to display the collection, not from the CSS grid.

Hub App Template

"A three-page project for a Windows Store app that implements the hub navigation pattern by using a hub control on the first page and provides two dedicated pages for displaying group and item details." (Blend/Visual Studio description)

Functionally similar to a Grid Template app, the Hub template uses the WinJS Hub control for a home page with heterogeneous content (that is, where multiple collections could be involved). From there the app navigates to group and item pages. We'll learn about the Hub control in Chapter 8.

Split Template

"A two-page project for a Windows Store app that navigates among grouped items. The first page allows group selection while the second displays an item list alongside details for the selected item." (Blend/Visual Studio description)

This last template also builds on the Navigation template and works over a collection of data. Its home page displays a list of groups, rather than grouped items as with the Grid template. Tapping a group navigates to a group detail page that is split into two sides (hence the template name). The left side contains a vertical list of items; the right side shows details for the currently selected item.

Like the Grid template, the Split template provides an app bar structure and handles different views intelligently. That is, because vertically oriented views don't lend well to splitting the display horizontally, the template shows how to switch to a page navigation model within those views to accomplish the same ends.

Item Templates

In addition to the project templates described above, there are a number of *item* templates that you can use to add new files of particular types to a project, or add groups of files for specific features. Once a project is created, right-click the folder in which you want to create the item in question (or the project file to create something at the root), and select Add > New item. This will present you with a dialog of available item templates, as shown in Figure 2-24 for features specific to Store apps. We'll encounter more of these throughout this book.

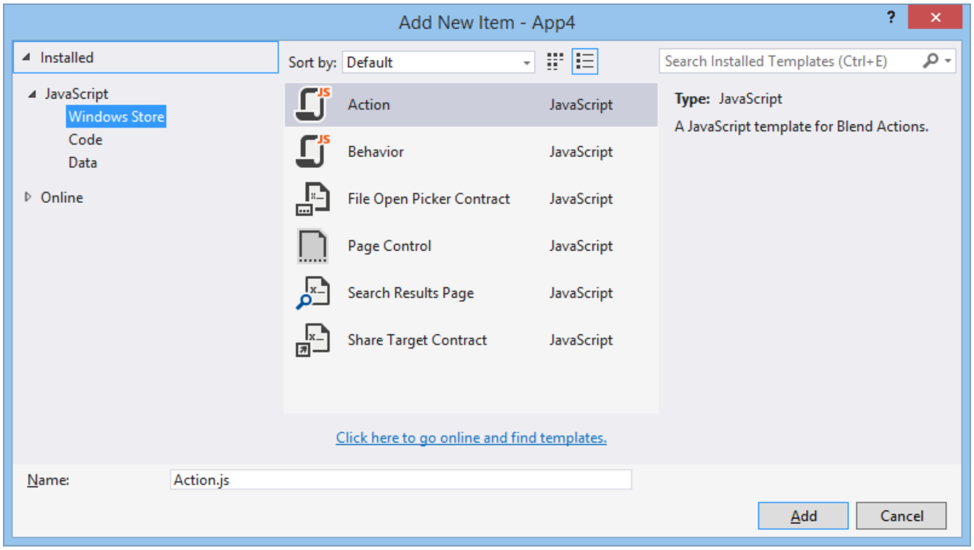

Add New Item - App4
Installed
JavaScript
Windows Store
Code
Data
Online
Sort by: Default
Search Installed Templates (Ctrl+E)
Action JavaScript
Behavior JavaScript
File Open Picker Contract JavaScript
Page Control JavaScript
Search Results Page JavaScript
Share Target Contract JavaScript
Type: JavaScript
A JavaScript template for Blend Actions.
Click here to go online and find templates.
Name: Action.js
Add
Cancel

FIGURE 2-24 Available item templates for a Windows Store app written in JavaScript.

What We've Just Learned

- How to create a new Windows Store app from the Blank app template.
- How to run an app inside the local debugger and within the simulator, as well as the role of remote machine debugging.
- The features of the simulator that include the ability to simulate touch, set views, and change resolutions and pixel densities.
- The basic project structure for Windows Store apps, including WinJS references.
- The core activation structure for an app through the `WinJS.Application.onactivated` event.
- The role and utility of design wireframes in app development, including the importance of designing for all views, where the work is really a matter of element visibility and layout.
- The power of Blend for Visual Studio to quickly and efficiently add styling to an app's markup. Blend also makes a great CSS debugging tool.
- How to safely use web content (such as Bing maps) within a web context `iframe` and communicate between that page and the local context app by using the `postMessage` method.
- How to use the WinRT APIs, especially async methods involving promises such as geolocation and camera capture. Async operations return a promise to which you provide a completed handler (and optional error and progress handlers) to the promise's `then` or `done` method.

- Manifest capabilities determine whether an app can use certain WinRT APIs. Exceptions will result if an app attempts to use an API without declaring the associated capability.
- How to share data through the Share contract by responding to the `datarequested` event.
- How to handle sequential async operations through chained promises.
- How to move files on the file system and work with basic appdata folders.
- The kinds of apps supported through the other app templates: Navigation, Grid, Hub, and Split.

Chapter 3

App Anatomy and Performance Fundamentals

During the early stages of writing this book (the first edition, at least), I was working closely with a contractor to build a house for my family. Although I wasn't on site every day managing the effort, I was certainly involved in nearly all decision-making throughout the home's many phases, and I occasionally participated in the construction itself.

In the Sierra Nevada foothills of California, where I live, the frame of a house is built with the plentiful local wood, and all the plumbing and wiring has to be in the walls before installing insulation and wallboard (aka sheetrock). It amazed me how long it took to complete that infrastructure. The builders spent a lot of time adding little blocks of wood here and there to make it much easier for them to do the finish work later on (like hanging cabinets), and lots of time getting the wiring and plumbing put together properly. All of this disappeared from sight once the wallboard went up and the finish work was in place.

But then, imagine what a house would be like without such careful attention to structural details. Imagine having some light switches that just don't work or control the wrong fixtures. Imagine if the plumbing leaks inside the walls. Imagine if cabinets and trim start falling off the walls a week or two after moving into the house. Even if the house manages to pass final inspection, such flaws would make it almost unlivable, no matter how beautiful it might appear at first sight. It would be like a few of the designs of the famous architect Frank Lloyd Wright: very interesting architecturally and aesthetically pleasing, yet thoroughly uncomfortable to actually live in.

Apps are very much the same story—I've marveled, in fact, just how many similarities exist between the two endeavors! An app might be visually beautiful, even stunning, but once you start using it day to day (or even minute to minute), a lack of attention to the fundamentals will become painfully apparent. As a result, your customers will probably start looking for somewhere else to live, meaning someone else's app! Another similarity is that taking care of core problems early on is *always* less expensive and time-consuming than addressing them after the fact, as anyone who has remodeled a house will know! This is especially true of performance issues in apps—trying to refactor an app at the end of a project to improve the user experience is like adding plumbing and wiring to a house after all the interior surfaces (walls, floors, windows, and ceilings) walls have been covered and painted.

This chapter, then, is about those fundamentals: the core foundational structure of an app upon which you can build something that looks beautiful *and* really works well. This takes us first into the subject of app activation (how apps get running and get running quickly) and then app lifecycle transitions (how they are suspended, resumed, and terminated). We'll then look at page navigation within an app, working with promises, async debugging, and making use of various profiling tools. One subject that we won't talk about here are background tasks; we'll see those in Chapter 16, "Alive with Activity," because there are limits to their use and they are best discussed in the context of the lock screen.

Generally speaking, these anatomical concerns apply strictly to the app itself and its existence on a client device. Chapter 4, "Web Content and Services," expands this story to include how apps reach out beyond the device to consume web-based content and employ web APIs and other services. In that context we'll look at additional characteristics of the hosted environment that we first encountered in Chapter 2, "Quickstart," such as the local and web contexts, basic network connectivity, and authentication. We'll pick up a few other platform fundamentals, like input, in later chapters.

Let me offer you advance warning that this chapter and the next are more intricate than most others because they specifically deal with the software equivalents of framing, plumbing, and wiring. With my family's house, I can completely attest that installing the lovely light fixtures my wife picked out seemed, in those moments, much more satisfying than the framing work I'd done months earlier. But now, actually *living* in the house, I have a deep appreciation for all the nonglamorous work that went into it. It's a place I want to be, a place in which my family and I are delighted, in fact, to spend the majority of our lives. And is that not how you want your customers to feel about your apps? Absolutely! Knowing the delight that a well-architected app can bring to your customers, let's dive in and find our own delight in exploring the intricacies!

App Activation

One of the most important things to understand about any app is how it goes from being a package on disk to something that's up and running and interacting with users. Such activation can happen a variety of ways: through tiles on the Start screen or the desktop task bar, toast notifications, and various contracts, including Search, Share, and file type and URI scheme associations. Windows might also pre-launch the user's most frequently used apps (not visibly, of course), after updates and system restarts. In all these activation cases, you'll be writing plenty of code to initialize your data structures, acquire content, reload previously saved state, and do whatever else is necessary to establish a great experience for the human beings on the other side of the screen.

Tip Pay special attention to what I call the *first experience* of your app, which starts with your app's page in the Store, continues through download and installation (meaning: pay attention to the size of your package), and finished up through first launch and initialization that brings the user to your app's home page. When a user taps an Install button in the Store, he or she clearly wants to try your app, so streamlining the path to interactivity is well worth the effort.

Branding Your App 101: The Splash Screen and Other Visuals

With activation, we first need to take a step back even *before* the app host gets loaded, to the very moment a user taps your tile on the Start screen or when your app is launched some other way (except for pre-launching). At that moment, before any app-specific code is loaded or run, Windows displays your app's splash screen image against your chosen background color, both of which you specify in your manifest.

The splash screen shows for at least 0.75 seconds (so that it's never just a flash even if the ap p loads quickly) and accomplishes two things. First, it guarantees that *something* shows up when an app is activated, even if no app code loads successfully. Second, it gives users an interesting branded experience for the app—that is, your image—which is better than a generic hourglass. (So don't, as one popular app I know does, put a generic hour class in your splash screen image!) Indeed, your splash screen and your app tile are the two most important ways to uniquely brand your app. Make sure you and your graphic artist(s) give full attention to these. (For further guidance, see Guidelines and checklist for splash screens.)

The default splash screen occupies the whole view where the app is being launched (in whatever view state), so it's a much more directly engaging experience for your users. During this time, an instance of the app host gets launched to load, parse, and render your HTML/CSS, and load, parse, and execute your JavaScript, firing events along the way, as we'll see in the next section. When the app's first page is ready, the system removes the splash screen.[16]

Additional settings and graphics in the manifest also affect your branding and overall presence in the system, as shown in the tables on the next page. Be especially aware that the Visual Studio and Blend templates provide some default and thoroughly unattractive placeholder graphics. Take a solemn vow right now that you truly, truly, cross-your-heart will *not* upload an app to the Windows Store with these defaults graphics still in place! (The Windows Store will reject your app if you forget, delaying certification.)

In the second table, you can see that it lists multiple sizes for various images specified in the manifest to accommodate varying pixel densities: 100%, 140%, and 180% scale factors, and even a few at 80% (don't neglect the latter: they are typically used for most desktop monitors and can be used when you turn on Show More Tiles on the Start screen's settings pane). Although you can just provide a single 100% scale image for each of these, it's almost guaranteed that stretching that graphics for higher pixel densities will look bad. Why not make your app look its best? Take the time to create each individual graphic consciously.

Manifest Editor Tab	Text Item or Setting	Use

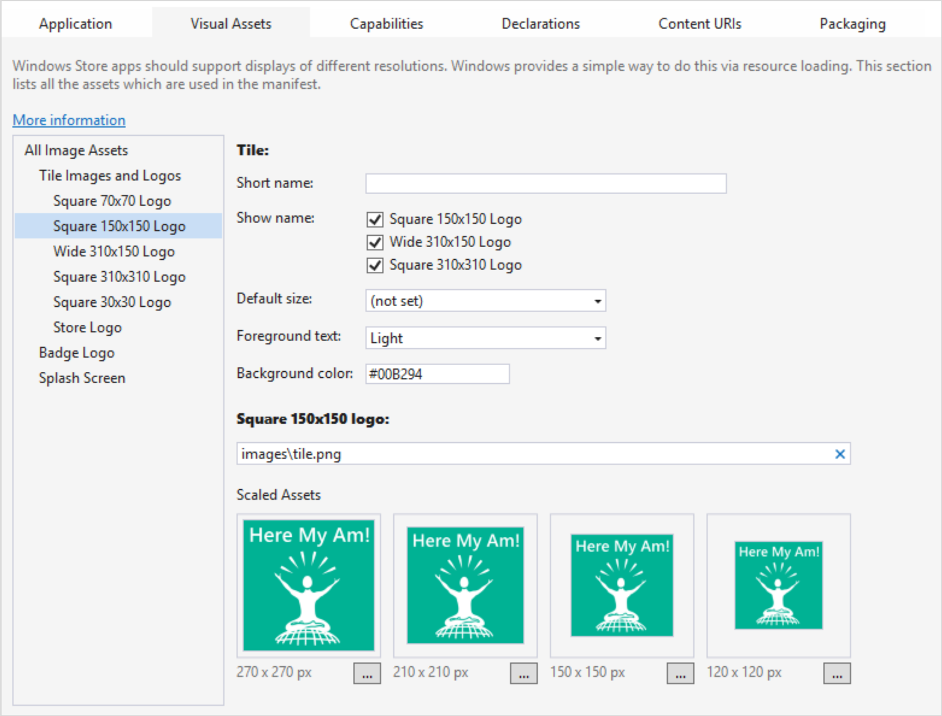
[16] This system-provided splash screen is composed of only your splash screen image and your background color and does not allow any customization. Through an *extended* splash screen (see Appendix B) you can control the entire display.

Application	Display Name	Appears in the "all apps" view on the Start screen, search results, the Settings charm, and in the Store.
Visual Assets	Tile > Short name	Optional: if provided, is used for the name on the tile in place of the Display Name, as Display Name may be too long for a square tile.
	Tile > Show name	Specifies which tiles should show the app name (the small 70x70 tile will never show the name). If none of the items are checked, the name never appears.
	Tile > Default size	Indicates whether to show the square or wide tile on the Start screen after installation.
	Tile > Foreground text	Color of name text shown on the tile if applicable (see Show name). Options are Light and Dark. There must be a 1.5 contrast ratio between this and the background color. Refer to The Paciello Group's Contrast Analyzer for more.
	Tile > Background color	Color used for transparent areas of any tile images, the default background for secondary tiles, notification backgrounds, buttons in app dialogs, borders when the app is a provider for file picker and contact picker contracts, headers in settings panes, and the app's page in the Store. Also provides the splash screen background color unless that is set separately.
	Splash Screen > Background color	Color that will fill the majority of the splash screen; if not set, the App UI Background color is used.

Visual Assets Tab Image	**Use**	**Image Sizes**			
		80%	**100%**	**140%**	**180%**
Square 70x70 logo	A small square tile image for the Start screen. If provided, the user has the option to display this after installation; it cannot be specified as the default. (Note also that live tiles are not supported on this size.)	56x56	70x70	98x98	126x126
Square 150x150 logo	Square tile image for the Start screen.	120z120	150x150	210x210	270x270
Wide 310x150 logo	Optional wide tile image. If provided, this is shown as the default unless overridden by the Default option below. The user can use the square tile if desired.	248x120	310x150	434x210	558x270
Square 310x310 logo	Optional double-size/large square tile image. If provided, the user has the option to display this after installation; it cannot be specified as the default.	248x248	310x310	434x434	558x558
Square 30x30 logo	Tile used in zoomed-out and "all apps" views of the Start screen, and in the Search and Share panes if the app supports those contracts as targets. Also used on the app tile if you elect to show a logo instead of the app name in the lower left corner of the tile. Note that there are also four "Target size" icons that are specifically used in the desktop file explorer when file type associations exist for the app. We'll cover this in Chapter 15, "Contracts."	24x24	30x30	42x42	54x54

Store logo	Tile/logo image used for the app on its product description page in the Windows Store. This image appears only in the Windows Store and is not used by the app or system at run time.	n/a	50x50	70x70	90x90
Badge logo	Shown next to a badge notification to identify the app on the lock screen (uncommon, as this requires additional capabilities to be declared; see Chapter 16).	n/a	24x24	33x33	43x43
Splash screen	When the app is launched, this image is shown in the center of the screen against the Splash Screen > Background color (or Tile > Background color if the other isn't specified). The image can utilize transparency if desired.	n/a	620x300	868x420	1116x540

The Visual Assets tab in the editor shows you which scale images you have in your package, as shown in Figure 3-1. To see all visual elements at once, select All Image Assets in the left-hand list.



Windows Store apps should support displays of different resolutions. Windows provides a simple way to do this via resource loading. This section lists all the assets which are used in the manifest.

Tile:

Short name:

Show name: ☑ Square 150x150 Logo ☑ Wide 310x150 Logo ☑ Square 310x310 Logo

Default size: (not set)

Foreground text: Light

Background color: #00B294

Square 150x150 logo:

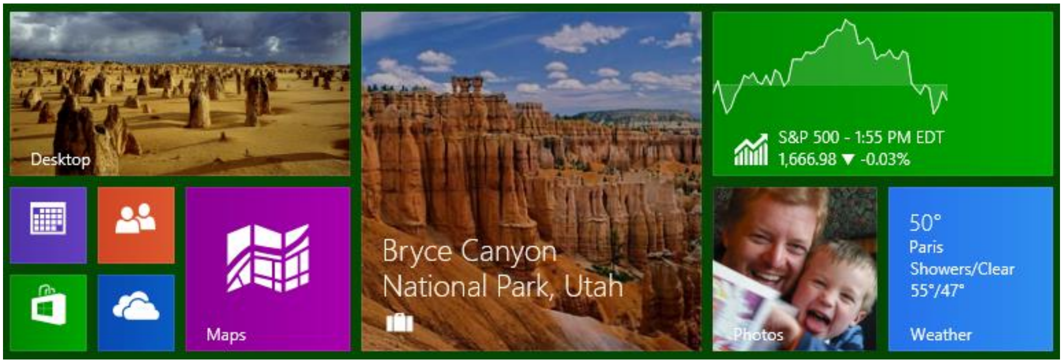
images\tile.png

Scaled Assets

FIGURE 3-1 Visual Studio's Visual Assets tab of the manifest editor. It automatically detects whether a scaled asset exists for the base filename (such as images\tile.png).

In the table, note that 80% scale tile graphics are used in specific cases like low DPI modes (generally when the DPI is less than 130 and the resolution is less than 2560 x 1440) and should be provided with other scaled images. When you upload your app to the Windows Store, you'll also need to provide some additional graphics. See the App images topic in the docs under "Promotional images" for full details.

The combination of small, square, wide, and large square tiles allows the user to arrange the start screen however they like. For example:



Of course, it's not required that your app supports anything other than the 150x150 square tile; all others are optional. In that case Windows will scale your 150x150 tile down to the 70x70 small size to give users at least that option.

When saving scaled image files, append *.scale-80*, *.scale-100*, *.scale-140*, and *.scale-180* to the filenames, before the file extension, as in *splashscreen.scale-140.png* (and be sure to remove any file that doesn't have a suffix). This allows you, both in the manifest and elsewhere in the app, to refer to an image with just the base name, such as *splashscreen.png*, and Windows will automatically load the appropriate variant for the current scale. Otherwise it looks for one without the suffix. No code needed! This is demonstrated in the HereMyAm3a example, where I've added all the various branded graphics (with some additional text in each graphic to show the scale). With all of these graphics, you'll see the different scales show up in the manifest editor, as shown in Figure 3-1 above.

To test these different graphics, use the set resolution/scaling button in the Visual Studio simulator—refer to Figure 2-5 in Chapter 2—or the Device tab in Blend, to choose different pixel densities on a 10.6" screen (1366 x 768 =100%, 1920 x 1080 = 140%, and 2560 x 1440 = 180%), or the 7" or 7.5" screens (both use 140%). You'll also see the 80% scale used on the other display choices, including the 23" and 27" settings. In all cases, the setting affects which images are used on the Start screen and the splash screen, but note that you might need to exit and restart the simulator to see the new scaling take effect.

One thing you might notice is that full-color photographic images don't scale down very well to the smallest sizes (store logo and small logo). This is one reason why Windows Store apps often use simple logos, which also keeps them smaller when compressed. This is an excellent consideration to keep your package size smaller when you make more versions for different contrasts and languages. We'll see more on this in Chapter 19, "Apps for Everyone, Part 1" and Chapter 20, "Apps for Everyone, Part 2."

Package bloat? As mentioned already in Chapters 1 and 2, the multiplicity of raster images that you need to create to accommodate scales, contrasts, and languages will certainly increase the size of the package you upload to the Store. (There are 104 possible variants per language of the manifest image assets alone!) Fortunately, the default packaging model for Windows 8.1 structures your resources into separate packs that are downloaded only as a user needs them, as we'll discuss in Chapters 19 and 20. In short, although the package you upload will contain all possible resources for all markets where your app will be available, most if not all users will be downloading a much smaller subset. That said, it's also good to consider the differences between file formats like JPEG, GIF, and PNG to get the most out of your pixels. For a good discussion, see PNG vs. GIF vs. JPEG on StackOverflow.

Tip Three other branding-related resources you might be interested in are the Branding your Windows Store app topic in the documentation (covering design aspects) the CSS styling and branding your app sample (covering CSS variations and dynamically changing the active stylesheet), and the very useful Applying app theme color (theme roller) sample (which lets you configure a color theme, showing its effect on controls, and which generates the necessary CSS).

Activation Event Sequence

As the app host is built on the same parsing and rendering engines as Internet Explorer, the general sequence of activation events is more or less what a web application sees in a browser. Actually, it's more rather than less! Here's what happens so far as Windows is concerned when an app is launched (refer to the ActivationEvents example in the companion code to see this event sequence as well as the related WinJS events that we'll discuss a little later):

1. Windows displays the default splash screen using information from the app manifest (except for pre-launching).

2. Windows launches the app host, identifying the app's installation folder and the name of the app's Start Page (an HTML file) as indicated in the Application tab of the manifest editor.[17]

3. The app host loads that page's HTML, which in turn loads referenced stylesheets and script (deferring script loading if indicated in the markup with the `defer` attribute). Here it's important that all files are properly encoded for best startup performance. (See the sidebar on the next page.)

4. `document.DOMContentLoaded` fires. You can use this to do early initialization specifically related to the DOM, if desired. This is also the place to perform one-time initialization work that should not be done if the app is activated on multiple occasions during its lifetime.

5. `window.onload` fires. This generally means that document layout is complete and elements will reflect their actual dimensions. (Note: In Windows 8 this event occurs at the end of this

[17] To avoid confusion with the Windows Start *screen*, I'll often refer to this as the app's *home* page unless I'm specifically referring to the entry in the manifest.

list instead.)

6. `Windows.UI.WebUI.WebUIApplication.onactivated` fires. This is typically where you'll do all your startup work, instantiate WinJS and custom controls, initialize state, and so on.

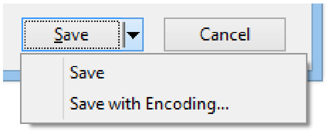
Once the `activated` event handler returns, the default splash screen is dismissed unless the app has requested a deferral, as discussed later in the "Activation Deferrals and setPromise" section.With the latter four events, your app's handling of these very much determines how quickly it comes up and becomes interactive. It almost goes without saying that you should strive to optimize that process, a subject we'll return to a little later in "Optimizing Startup Time."

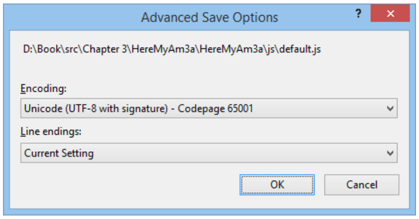
What's also different between an app and a website is that an app can again be activated for many different purposes, such as contracts and associations, even while it's already running. As we'll see in later chapters, the specific page that gets loaded (step 3) can vary by contract, and if a particular page is already running it will receive only the `Windows.UI.WebUI.WebUIApplication.onactivated` event and not the others.

For the time being, though, let's concentrate on how we work with this core launch process, and because you'll generally do your initialization work within the `activated` event, let's examine that structure more closely.

Sidebar: File Encoding for Best Startup Performance

To optimize bytecode generation when parsing HTML, CSS, and JavaScript, which speeds app launch time, the Windows Store requires that all .html, .css, and .js files are saved with Unicode UTF-8 encoding. This is the default for all files created in Visual Studio or Blend. If you're importing assets from other sources including third-party libraries, check this encoding: in Visual Studio's File Save As dialog (Blend doesn't have a Save As feature), select *Save with Encoding* and set that to *Unicode (UTF-8 with signature) – Codepage 65001*. The Windows App Certification Kit will issue warnings if it encounters files without this encoding.





D:\Book\src\Chapter 3\HereMyAm3a\HereMyAm3a\js\default.js

Encoding: Unicode (UTF-8 with signature) - Codepage 65001

Line endings: Current Setting

Along these same lines, minification of JavaScript isn't particularly important for Windows Store apps. Because an app package is downloaded from the Windows Store as a unit and often contains other assets that are much larger than your code files, minification won't make much difference there. Once the package is installed, bytecode generation means that the package's JavaScript has already been processed and optimized, so minification won't have any additional performance impact. If your intent is to obfuscate your code (because it is just there in source form in the installation folder), see "Protecting Your Code" in Chapter 18, "WinRT Components."

Activation Code Paths

As we saw in Chapter 2, new projects created in Visual Studio or Blend give you the following code in js/default.js (a few comments have been removed):

```
(function () {
    "use strict";

    var app = WinJS.Application;
    var activation = Windows.ApplicationModel.Activation;

    app.onactivated = function (args) {
        if (args.detail.kind === activation.ActivationKind.launch) {
            if (args.detail.previousExecutionState !==
                activation.ApplicationExecutionState.terminated) {
                // TODO: This application has been newly launched. Initialize
                // your application here.
            } else {
                // TODO: This application has been reactivated from suspension.
                // Restore application state here.
            }
            args.setPromise(WinJS.UI.processAll());
        }
    };

    app.oncheckpoint = function (args) {
    };

    app.start();
})();
```

Let's go through this piece by piece to review what we already learned and complete our understanding of this core code structure:

- `(function () { … })();` surrounding everything is again the JavaScript module pattern.
- `"use strict"` instructs the JavaScript interpreter to apply Strict Mode, a feature of ECMAScript 5. This checks for sloppy programming practices like using implicitly declared variables, so it's a good idea to leave it in place.
- `var app = WinJS.Application;` and `var activation = Windows.ApplicationModel.Activation;` both create substantially shortened aliases for commonly used namespaces. This is a common practice to simplify multiple references to the same part of WinJS or WinRT, and it also provides a small performance gain.
- `app.onactivated = function (args) {…}` assigns a handler for the `WinJS.UI.onactivated` event, which is a wrapper for `Windows.UI.WebUI.WebUIApplication.onactivated` (but will be fired *after* `window.onload`). In this handler:
    - `args.detail.kind` identifies the type of activation.

    - args.detail.previousExecutionState identifies the state of the app prior to this activation, which determines whether to reload session state.

    - WinJS.UI.processAll instantiates WinJS controls—that is, elements that contain a data-win-control attribute, as we'll cover in Chapter 5, "Controls and Control Styling."

    - args.setPromise instructs Windows to wait until WinJS.UI.processAll is complete before removing the splash screen. (See "Activation Deferrals and setPromise" later in this chapter.)

- app.oncheckpoint, which is assigned an empty function, is something we'll cover in the "App Lifecycle Transition Events" section later in this chapter.

- app.start() (WinJS.Application.start()) initiates processing of events that WinJS queues during startup.

Notice how we're not directly handling any of the events that Windows or the app host is firing, like DOMContentLoaded or Windows.UI.WebUI.WebUIApplication.onactivated. Are we just ignoring those events? Not at all: one of the convenient services that WinJS offers through WinJS.UI.Application is a simplified structure for activation and other app lifetime events. Its use is entirely optional but very helpful.

With its start method, for example, a couple of things are happening. First, the WinJS.Application object listens for a variety of events that come from different sources (the DOM, WinRT, etc.) and coalesces them into a single object with which you register your handlers. Second, when WinJS.Application receives activation events, it doesn't just pass them on to the app's handlers immediately, because your handlers might not, in fact, have been set up yet. So it queues those events until the app says it's really ready by calling start. At that point WinJS goes through the queue and fires those events. That's all there is to it.

As the template code shows, apps typically do most of their initialization work within the WinJS activated event, where there are a number of potential code paths depending on the values in args.details (an IActivatedEventArgs object). If you look at the documentation for activated, you'll see that the exact contents of args.details depends on specific activation kind. All activations, however, share some common properties:

args.details Property	Type (in Windows.Application-Model.Activation)	Description
kind	ActivationKind	The reason for the activation. The possibilities are launch (most common); restrictedLaunch (specifically for app to app launching); search, shareTarget, file, protocol, fileOpenPicker, fileSavePicker, contactPicker, and cachedFileUpdater (for servicing contracts); and device, printTask, settings, and cameraSettings (generally used with device apps). For each supported activation kind, the app will have an appropriate initialization path.

previousExecutionState	ApplicationExecutionState	The state of the app prior to this activation. Values are notRunning, running, suspended, terminated, and closedByUser. Handling the terminated case is most common because that's the one where you want to restore previously saved session state (see "App Lifecycle Transition Events").
splashScreen	SplashScreen	Contains an ondismissed event for when the system splash screen is dismissed along with an imageLocation property (Windows.Foundation.Rect) with coordinates where the splash screen image was displayed. For use of this, see "Extended Splash Screens" in Appendix B, "WinJS Extras."

Additional properties provide relevant data for the activation. For example, launch provides the tileId and arguments from secondary tiles (see Chapter 16). The search kind (the next most commonly used) provides queryText and language, the protocol kind provides a uri, and so on. We'll see how to use many of these in their proper contexts, and sometimes they apply to altogether different pages than default.html. What's contained in the templates (and what we've already used for an app like Here My Am!) is primarily to handle normal startup from the app tile or when launched within Visual Studio's debugger.

WinJS.Application Events

WinJS.Application isn't concerned only with activation—its purpose is to centralize events from several different sources and turn them into events of its own. Again, this enables the app to listen to events from a single source (either assigning handlers via addEventListener(<event>) or on<event> properties; both are supported). Here's the full rundown on those events and when they're fired (if queued, the event is fired within your call to WinJS.Application.start):

- loaded Queued for DOMContentLoaded in both local and web contexts.[18] This is fired before activated.
- activated Queued in the local context for Windows.UI.WebUI.WebUIApplication.onactivated (which fires after window.onload). In the web context, where WinRT is not applicable, this is instead queued for DOMContentLoaded (where the launch kind will be launch and previousExecutionState is set to notRunning).
- ready Queued after loaded and activated. This is the last one in the activation sequence.
- error Fired if there's an exception in dispatching another event. (If the error is not handled here, it's passed onto window.onerror.)
- checkpoint Fired when the app should save the session state it needs to restart from a previous state of terminated. It's fired in response to both the document's beforeunload

[18] There is also WinJS.Utilities.ready through which you can specifically set a callback for DOMContentLoaded. This is used within WinJS, in fact, to guarantee that any call to WinJS.UI.processAll is processed after DOMContentLoaded.

event as well as Windows.UI.WebUI.WebUIApplication.onsuspending.

- unload   Also fired for beforeunload after the checkpoint event is fired.
- settings   Fired in response to Windows.UI.ApplicationSettings.SettingsPane.-oncommandsrequested. (See Chapter 10, "The Story of State, Part 1.")

I think you'll generally find WinJS.Application to be a useful tool in your apps, and it also provides a few more features as documented on the WinJS.Application page. For example, it provides local, temp, roaming, and sessionState properties, which are helpful for managing state. We saw a little of local already in Chapter 2; we'll see more later on in Chapter 10.

The other bits are the queueEvent and stop methods. The queueEvent method drops an event into the queue that will get dispatched, after any existing queue is clear, to whatever listeners you've set up on the WinJS.Application object. Events are simply identified with a string, so you can queue an event with any name you like, and call WinJS.Application.addEventListener with that same name anywhere else in the app. This makes it easy to centralize custom events that you might invoke both during startup and at other points during execution without creating a separate global function for that purpose. It's also a powerful means through which separately defined, independent components can raise events that get aggregated into a single handler. (For an example of using queueEvent, see scenario 2 of the App model sample.)

As for stop, this is provided to help with unit testing so that you can simulate different activation sequences without having to relaunch the app and somehow recreate a set of specific conditions when it restarts. When you call stop, WinJS removes its listeners, clears any existing event queue, and clears the sessionState object, but the app continues to run. You can then call queueEvent to populate the queue with whatever events you like and then call start again to process that queue. This process can be repeated as many times as needed.

Activation Deferrals and setPromise

As noted earlier under "Activation Event Sequence," once you return from your handler for WebUIApplication.onactivated (or WinJS.Application.onactivated), Windows assumes that your home page is ready and that it can dismiss the default splash screen. The same is true for WebUIApplication.onsuspending (and by extension, WinJS.Application.oncheckpoint): Windows assumes that it can suspend the app once the handler returns. More generally, WinJS.Application assumes that it can process the next event in the queue once you return from the current event.

This gets tricky if your handler needs to perform one or more async operations, like an HTTP request, whose responses are essential for your home page. Because those operations are running on other threads, you'll end up returning from your handler while the operations are still pending, which could cause your home page to show before its ready or the app to be suspended before it's finished saving state. Not quite what you want to have happen! (You can, of course, make other secondary requests, in which case it's fine for them to complete after the home page is up—always avoid blocking the home page for nonessentials.)

For this reason, you need a way to tell Windows and WinJS to defer their default behaviors until your most critical async work is complete. The mechanism that provides for this is in WinRT called a *deferral*, and the setPromise method that we've seen in WinJS ties into this.

On the WinRT level, the args given to WebUIApplication.onactivated contains a little method called getDeferral (technically Windows.UI.WebUI.ActivatedOperation.getDeferral). This function returns a deferral object that contains a complete method. By calling getDeferral, you tell Windows to leave the system splash screen up until you call complete (subject to a 15-second timeout as described in "Optimizing Startup Time" below). The code looks like this:

```
//In the activated handler
var activatedDeferral = Windows.UI.WebUI.ActivatedOperation.getDeferral();

someOperationAsync().done(function () {
   //After initialization is complete
    activatedDeferral.complete();
}
```

This same mechanism is employed elsewhere in WinRT. You'll find that the args for WebUIApplication.onsuspending also has a getDeferral method, so you can defer suspension until an async operation completed. So does the DataTransferManager.ondatarequested event that we saw in Chapter 2 for working with the Share charm. You'll also encounter deferrals when working with the Search charm, printing, background tasks, Play To, and state management, as we'll see in later chapters. In short, wherever there's a potential need to do async work within an event handler, you'll find getDeferral.

Within WinJS now, whenever WinJS provides a wrapper for a WinRT event, as with WinJS.Application.onactivated, it also wraps the deferral mechanism into a single setPromise method that you'll find on the args object passed to the relevant event handler. Because you need deferrals when performing async operations in these event handlers, and because async operations in JavaScript are always represented with promises, it makes sense for WinJS to provide a generic means to link the deferral to the fulfillment of a promise. That's exactly what setPromise does.

WinJS, in fact, automatically requests a deferral whether you need it or not. If you provide a promise to setPromise, WinJS will attach a completed handler to it and call the deferral's complete at the appropriate time. Otherwise WinJS will call complete when your event handler returns.

You'll find setPromise on the args passed to the WinJS.Application loaded, activated, ready, checkpoint, and unload events. Again, setPromise both defers Windows' default behaviors for WinRT events and tells WinJS.Application to defer processing the next event in its queue. This allows you, for example, to delay the activated event until an async operation within loaded is complete.

Now we can see the purpose of setPromise within the activation code we saw earlier:

```
var app = WinJS.Application;

app.onactivated = function (args) {
   if (args.detail.kind === activation.ActivationKind.launch) {
       //...
       args.setPromise(WinJS.UI.processAll());
       }
};
```

WinJS.UI.processAll starts an async operation to instantiate WinJS controls. It returns a promise that is fulfilled when all those controls are ready. Clearly, if we have WinJS controls on our home page, we don't want to dismiss the default splash screen until processAll is done. So we defer that dismissal by passing that promise to setPromise.

Oftentimes you'll want to do more initialization work of your own when processAll is complete. In this case, simply call then with your own completed handler, like so:

```
args.setPromise(WinJS.UI.processAll().then(function () {
   //Do more initialization work
}));
```

Here, be sure to use then and not done because the latter returns undefined rather than a promise, which means that no deferral will happen. See "Error Handling Within Promises: then vs. done" later on.

Because setPromise just waits for a single promise to complete, how do you handle multiple async operations? Just pick the one you think will take the longest? No—there are a couple of ways to do this. First, if you need to control the sequencing of those operations, you can chain them together as we already saw in Chapter 2 and as we'll discuss further in this chapter under "Async Operations: Be True to Your Promises." Just be sure that the end result of the chain is a promise that becomes the argument to setPromise—again, use then and not done!

Second, if the sequence isn't important but you need *all* of them to complete, you can combine those promises by using WinJS.Promise.join, passing the result to setPromise. If you need only one of the operations to complete, you can use WinJS.Promise.any instead. Again, see "Be True to Your Promises" later on.

The other means is to register more than one handler with WinJS.Application.onactivated; each handler will get its own event args and its own setPromise function, and WinJS will combine those returned promises together with WinJS.Promise.join.

Optimizing Startup Time

Ideally, an app launches and its home page comes up within one second of activation, with an acceptable upper bound being three seconds. Anything longer begins to challenge most user's patience threshold, especially if they're already pressed for time and swilling caffeine-laden beverages! In fact, the Windows App Certification Toolkit, which we'll meet at the end of this chapter, will give you a warning if your app takes more than a few seconds to get going.

Windows is much more generous here, however. It allows an app to hang out on the default start screen for as long as the user is willing to stare at it. Apparently that willingness peaks out at about 15 seconds, at which point most users will pretty much assume that the app has hung and return to the

Start screen to launch some other app that won't waste the afternoon. For this reason, if an app doesn't get its home page up in that time—that is, return from the `activated` event and complete any deferral—and the user switches away, then boom!: Windows will terminate the app. (This saves the user from having to do the sordid deed in Task Manager.)

Of course, some apps, especially on first run after acquisition, might really need more time to get started. To accommodate this, there is an implementation strategy called an *extended splash screen* wherein you make you home page look just like the default start screen and then place additional controls on it to keep the user informed of progress so that she knows the app isn't hung. Once you're on the extended splash screen, the 15-second limit no longer applies. For more info, see Appendix B.

For most startup scenarios, though, it's best to focus your efforts on minimizing time to interactivity. This means prioritizing work that's necessary for the primary workflows of the home page and deferring everything else until the home page it up. This includes deferring configuration of app bars, nav bars, settings panels, and secondary app pages, as well as acquiring and processing content for those secondary pages. But even before that, let's take a step back to understand what's going on behind the default splash screen to begin with, because there are things you can do to help that process along as well.

When the user taps your tile, Windows first creates a new app host process and points it to the start page specified in your manifest. The app host then loads and parses that file. In doing so, it must also load and parse the CSS and JavaScript files it refers to. This process will fire various events, as we've seen, at which point it enters your activation code.

Up to that point, one thing that really matters is the structure of your HTML markup. As much as possible, avoid inline styles and scripts because these cause the HTML parser to switch from an HTML parsing context into a CSS or JavaScript parsing context, which is relatively expensive. In other words, the separation of concerns between markup, styling, and script is both a good development practice and a good performance practice! Also make sure to place any static markup in the HTML file rather than creating it from JavaScript: it's faster to have the app host's inner engine parse HTML than to make DOM API calls from code for the same purpose. And even if you must create elements dynamically, once you use more than four DOM API calls it's faster to build an HTML string and assign it to an `innerHTML` or similar property (so that the inner engine does the work).

Similarly, minimize the amount of CSS that has to be loaded for your start page to appear; CSS that's needed for secondary pages can be loaded with those pages (see "Page Controls and Navigation" later in this chapter).

Loading JavaScript files can also be deferred, both for secondary pages but also on the start page. That is, you can use the `defer="defer"` attribute on `<script>` tags to delay loading specific .js files until after the first parsing pass, or you can dynamically inject `<script>` tags or call `eval` at a later time in your activation path or after your initial activation is complete.

Review all the resources that your markup references as well, and place any critical ones directly into the app package where you can reference them with `ms-appx:///` URIs. Any remote resources will, of course, require a round trip to the network with possible connectivity failures. Where making HTTP requests is unavoidable, suggest your most critical URIs to the Windows.Networking.BackgroundTransfer.ContentPrefetcher object (see "Prefetching Content" in Chapter 4). If the prefetcher determines that those URIs are among the top requests, it will actively cache requests to those URIs such that requests from your code will draw directly from that cache. This won't help the app the first time it's run, but it can help with subsequent activations.

Consider whether you can also cache such content directly in your app package. That way you have something to work with immediately, even if there's no connectivity when the app is first run. This would mean building a refresh/sync strategy into your data model, but it's certainly doable.

Once you hit your activation code, a new set of considerations come into play. The key thing to consider here is this: *so long as you're on the default or an extended splash screen, go ahead and block the UI thread for high-priority work.* A splash screen, by definition, is noninteractive, so any UI thread work that deals with interactivity is a much lower priority than work that's necessary to initialize controls, retrieve and process data, and otherwise get ready for interactivity. (Page content animations, similarly, should be disabled while the splash screen is up.)

Most important, though, is making sure that your critical non-UI work runs at a higher priority than UI rendering processes, especially while the splash screen is still active. For this you use the WinJS scheduler API, which we'll return to later in "Managing the UI Thread with the WinJS Scheduler." For now, know that you can schedule work to happen at a higher priority than layout and rendering and also at other lower priorities. This way you can kick off a number of HTTP requests, for example, but give your most important ones a high priority while giving your secondary ones a much lower priority so that they happen after layout and rendering. With this API you can also reprioritize work at any time: for example, if the user immediately navigates to a secondary page as soon as the app comes up, you can set that request (or more specifically, the function that processes its results) to high priority.

For a deeper dive on these matters of startup performance, I recommend two talks from //build 2013: Create Fast and Fluid Interfaces with HTML and JavaScript (Paul Gildea) and Web Runtime Performance (Tobin Titus). Also refer to Reducing your app's loading time in the documentation.

WinRT Events and removeEventListener

Before going further, we need to take a slight detour into a special consideration for events that originate from WinRT, such as dismissed. You may have noticed that I'm highlighting these with a different text color than other events.

As we've already been doing in this book, typical practice within JavaScript, especially for websites, is to call addEventListener to specify event handlers or to simply assign an event handler to an on<event> property of some object. Oftentimes these handlers are just declared as inline anonymous functions:

```
var myNumber = 1;
```

```
element.addEventListener(<event>, function (e) { myNumber++; } );
```

Because of JavaScript's particular scoping rules, the scope of that anonymous function ends up being the same as its surrounding code, which allows the code within that function to refer to local variables like *myNumber* as used here.

To ensure that such variables are available to that anonymous function when it's later invoked as an event handler, the JavaScript engine creates a *closure*: a data structure that describes the local variables available to that function. Usually the closure requires only a small bit of memory, but depending on the code inside that event handler, the closure could encompass the entire global namespace—a rather large allocation! Every such active closure increases the memory footprint or *working set* of the app, so it's a good practice to keep closures at a minimum. For example, declaring a separate named function—which has its own scope—rather than using an anonymous function, will reduce the size of any necessary closure.

More important than minimizing closures is making sure that the event listeners themselves—and their associated closures—are properly cleaned up and their memory allocations released. Typically, this is not even something you need to think about. When objects such as HTML elements are destroyed or removed from the DOM, their associated listeners are automatically removed and closures are released. However, in a Windows Store app written in HTML and JavaScript, events can also come from WinRT objects. Because of the nature of the projection layer that makes WinRT available in JavaScript, WinRT ends up holding references to JavaScript event handlers (known also as *delegates*) and the JavaScript closures hold references to those WinRT objects. As a result of these cross-references, the associated closures aren't released unless you do so explicitly with `removeEventListener` (or assignment of `null` to an `on<event>` property).

This is not a problem, mind you, if the app is *always* listening to a particular event. For example, the `suspending` and `resuming` events are two that an app typically listens to for its entire lifetime, so any related allocations will be cleaned up when the app is terminated. It's also not much of a concern if you add a listener only once, as with the splash screen `dismissed` event. (In that case, however, it's good to remove the listener explicitly, because there's no reason to keep any closures in memory once the splash screen is gone.)

Do pay attention, however, when an app listens to a WinRT object event only *temporarily* and neglects to explicitly call `removeEventListener`, and when the app might call `addEventListener` for the same event multiple times (in which case you can end up duplicating closures). With *page controls*, which are used to load HTML fragments into a page (as discussed later in this chapter under "Page Controls and Navigation"), it's common to call `addEventListener` or assign a handler to an `on<event>` property on some WinRT object within the page's `ready` method. When you do this, *be sure to match that call with* `removeEventListener` *(or assign* `null` *to* `on<event>`*) in the page's* `unload` *method to release the closures*.

Note Events from WinJS objects don't need this attention because the library already handles removal of event listeners. The same is true for listeners you might add for `window` and `document` events that persist for the lifetime of the app.

Throughout this book, the WinRT events with which you need to be concerned are highlighted with a special color, as in `datarequested` (except where the text is also a hyperlink). This is your cue to check whether an explicit call to `removeEventListener` or `on<event>=null` is necessary. Again, if you'll always be listening to the event, removing the listener isn't needed, but if you add a listener when loading a page control, or anywhere else where you might add that listener again, be sure to make that extra call. Be especially aware that the samples in the Windows SDK don't necessary pay attention to this detail, so don't duplicate the oversight.

In the chapters that follow, I will remind you of what we've just discussed on our first meaningful encounter with a WinRT event. Keep your eyes open for the WinRT color coding in any case. We'll also come back to the subject of debugging and profiling toward the end of this chapter, where we'll learn about tools that can help uncover memory leaks.

App Lifecycle Transition Events and Session State

Now that we've seen how an app gets activated into a running state, our next concern is with what can happen to it while it's running. To an app—and the app's publisher—a perfect world might be one in which consumers ran that app and stayed in that app forever (making many in-app purchases, no doubt!). Well, the hard reality is that this just isn't reality. No matter how much you'd love it to be otherwise, yours is not the only app that the user will ever run. After all, what would be the point of features like sharing or split-screen views if you couldn't have multiple apps running together? For better or for worse, users will be switching between apps, changing view states, and possibly closing your app, none of which the app can control. But what you *can* do is give energy to the "better" side of the equation by making sure your app behaves well under all these circumstances.

The first consideration is *focus*, which applies to controls in your app as well as to the app itself (the `window` object). Here you can simply use the standard HTML `blur` and `focus` events. For example, an action game or one with a timer would typically pause itself on `window.onblur` and perhaps restart again on `window.onfocus`.

A similar but different condition is *visibility*. An app can be visible but not have the focus, as when it's sharing the screen with others. In such cases an app would continue things like animations or updating a feed, which it would stop when visibility is lost (that is, when the app is actually in the background). For this, use the `visibilitychange` event in the DOM API, and then examine the `visibilityState` property of the `window` or `document` object, as well as the `document.hidden` property. (The event works for visibility of individual elements as well.) A change in visibility is also a good time to save user data like documents or game progress.

For *view state changes*, an app can detect these in several ways. As shown in the Here My Am! example, an app typically uses media queries (in declarative CSS or in code through media query listeners) to reconfigure layout and visibility of elements, which is really all that view states should affect. (Again, view state changes never change the mode of the app.) At any time, an app can also retrieve its view state through `Windows.UI.ViewManagement.ApplicationView.orientation` (returning an ApplicationViewOrientation value of either portrait or landscape), the size of the app window, and other details from ApplicationView like isFullScreen; details in Chapter 8, "Layout and Views."[19]

When your app is *closed* (the user swipes top to bottom and holds, or just presses Alt+F4), it's important to note that the app is first moved off-screen (hidden), then suspended, and then closed, so the typical DOM events like body.unload aren't much use. A user might also kill your app in Task Manager, but this won't generate any events in your code either. Remember also that apps should *not* close themselves nor offer a means for the user to do so (this violates Store certification requirements), but they can use MSApp.terminateApp to close due to unrecoverable conditions like corrupted state.

Suspend, Resume, and Terminate

Beyond focus, visibility, and view states, there are three other critical moments in an app's lifetime:

- **Suspending** When an app is not visible in any view state, it will be suspended after five seconds (according to the wall clock) to conserve battery power. This means it remains wholly in memory but won't be scheduled for CPU time and thus won't have network or disk activity (except when using specifically allowed background tasks, discussed in Chapter 16). When this happens, the app receives the Windows.UI.WebUI.WebUIApplication.onsuspending event, which is also exposed through WinJS.Application.oncheckpoint. Apps must return from this event within the five-second period, or Windows will assume the app is hung and terminate it (period!). During this time, apps save transient session state and should also release any exclusive resources acquired as well, like file streams or device access. (See How to suspend an app.) If you need to do async work in the suspending handler, WinRT provides a deferral object as with activation and WinJS provides the setPromise equivalent. Using the deferral will not, however, extend the suspension deadline.

- **Resuming** If the user switches back to a suspended app, it receives the Windows.UI.WebUI.WebUIApplication.onresuming event. This is *not* surfaced through WinJS.Application, mind you, because WinJS has no value to add, but it's easy enough to use WinJS.Application.queueEvent for this purpose. We'll talk more about this event in coming chapters, as it's used to refresh any data that might have changed while the app was suspended. For example, if the app is connected to an online service, it would refresh that content if enough time has passed while the app was suspended, as well as check connectivity status (Chapter 4). In addition, if you're tracking sensor input of any kind (like compass, geolocation, or orientation, see Chapter 12, "Input and Sensors"), resuming is a good time to get a fresh reading. You'll also want to check license status for your app and in-app purchases if you're using trials and/or expirations (see Chapter 20). There are also times when you might want to refresh your layout (as we'll see in Chapter 8), because it's possible for your app to resume

[19] The Windows 8 view states from ApplicationView.value—namely fullscreen-landscape, fullscreen-portrait, filled, and snapped—are deprecated in Windows 8.1 in favor of just checking orientation and window size.

directly into a different view state than when it was suspended, or resume to a different screen resolution as when the device has been connected to an external monitor. The same goes for enabling/disabling clipboard-related commands (Chapter 9, "Commanding UI"), refreshing any tile updates and push notification channels (see Chapter 16), and checking any saved state that might have been modified by background tasks or roaming (Chapter 10).

- **Terminating** When suspended, an app might be terminated if there's a need for more memory. There is *no event* for this, because by definition the app is already suspended and no code can run. Nevertheless, this is important for the app lifecycle because it affects `previousExecutionState` when the app restarts.

Before we go further, it's essential to know that you can simulate these conditions in the Visual Studio debugger by using the toolbar drop-down shown in Figure 3-2. These commands will trigger the necessary events as well as set up the `previousExecutionState` value for the next launch of the app. (Be very grateful for these controls—there was a time when we didn't have them, and it was painful to debug these conditions!)

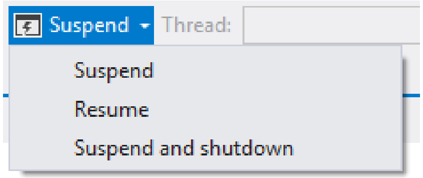


FIGURE 3-2 The Visual Studio toolbar drop-down to simulate suspend, resume, and terminate.

We've briefly listed those previous states before, but let's see how they relate to the events that get fired and the `previousExecutionState` value that shows up when the app is next launched. This can get a little tricky, so the transitions are illustrated in Figure 3-3 and the table below describes how the `previousExecutionState` values are determined.

Value of `previousExecutionState`	Scenarios
`notrunning`	First run after install from Store. First run after reboot or log off. App is launched within 10 seconds of being closed by user (about the time it takes to hide, suspend, and cleanly terminate the app; if the user relaunches quickly, Windows has to immediately terminate it without finishing the suspend operation). App was terminated in Task Manager while running or closed itself with `MSApp.terminateApp`.
`running`	App is *currently running* and then invoked in a way other than its app tile, such as Search, Share, secondary tiles, toast notifications, and all other contracts. When an app is running and the user taps the app tile, Windows just switches to the already-running app and without triggering activation events (though `focus` and `visibilitychange` will both be raised).
`suspended`	App is *suspended* and then invoked in a way other than the app tile (as above for `running`). In addition to focus/visibility events, the app will also receive the `resuming` event.
`terminated`	App was previously suspended and then terminated by Windows due to

	resource pressure. Note that this does not apply to MSApp.terminateApp because an app would have to be running to call that function.
closedByUser	App was closed by an uninterrupted close gesture (swipe down + hold or Alt+F4). An "interrupted" close is when the user switches back to the app within 10 seconds, in which case the previous state will be notrunning instead.

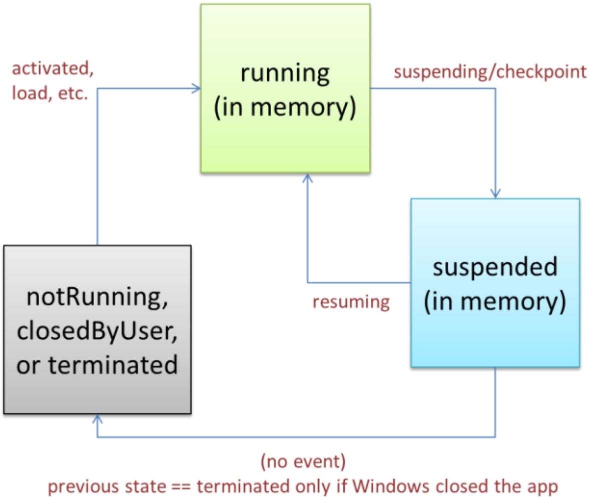


FIGURE 3-3 Process lifecycle events and previousExecutionState values.

The big question for the app, of course, is not so much what determines the value of previousExecutionState as what it should actually *do* with this value during activation. Fortunately, that story is a bit simpler and one that we've already seen in the template code:

- If the activation kind is launch and the previous state is notrunning or closedByUser, the app should start up with its default UI and apply any *persistent* state or settings. With closedByUser, there might be scenarios where the app should perform additional actions (such as updating cached data) after the user explicitly closed the app and left it closed for a while.
- If the activation kind is launch and the previous state is terminated, the app should start up in the same *session state* as when it was last suspended.
- For launch and other activation kinds that include additional arguments or parameters (as with secondary tiles, toast notifications, and contracts), it should initialize itself to serve that purpose by using the additional parameters. The app might already be running, so it won't necessarily initialize its default state again.

In the first two requirements above, *persistent state* refers to state that always applies to an instance of the app, such as user accounts, UI configurations, and similar settings. *Session state*, on the other hand, is the transient state of a particular instance and includes things like unsubmitted form data, page navigation history, scroll position, and so forth.

We'll see the full details of managing state in Chapter 10. What's important to understand at present is the relationship between the lifecycle events and session state, in particular. When Windows terminates a suspended app, *the app is still running in the user's mind*. Thus, when the user activates the app again for normal use (activation kind is launch, rather than through a contract), he or she expects that app to be right where it was before. This means that by the time an app gets suspended, it needs to have saved whatever state is necessary to make this possible. It then rehydrates the app from that state when previousExecutionState is terminated. This creates continuity across the suspend-terminate-restart boundary.

For more on app design where this is concerned, see Guidelines for app suspend and resume. Be clear that if the user directly closes the app with Alt+F4 or the swipe-down+hold gesture, the suspending and checkpoint events will also be raised, so the app still saves session state. However, the app won't be asked to reload session state when it's restarted because previousExecutionState will be notRunning or closedByUser.

It works out best, actually, to save session state as it changes during the app's lifetime, thereby minimizing the work needed within the suspending event (where you have only five seconds). Mind you, this session state does not include persistent state that an app would always reload or reapply in its activation path. The only concern here is maintaining the illusion that the app was always running.

You always save session state to your appdata folders or settings containers, which are provided by the Windows.Storage.ApplicationData API. Again, we'll see all the details in Chapter 10. What I want to point out here are a few helpers that WinJS provides for all this.

First is the WinJS.Application.checkpoint event, which is raised when suspending fires. checkpoint provides a single convenient place to save both session state and any other persistent data you might have, if you haven't already done so. If you need to do any async work in this handler, be sure to pass the promise for that operation to eventArgs.setPromise. This ties into the WinRT deferral mechanism as with activation (and see "Suspending Deferrals" below).

Second is the WinJS.Application.sessionState object. On normal startup, this is just an empty object to which you can add whatever properties you like, including other objects. A typical strategy is to just use sessionState directly as a container for session variables. Within the checkpoint event, WinJS automatically serializes the contents of this object (using JSON.stringify) into a file within your local appdata folder (meaning that all variables in sessionState must have a string representation). Note that because WinJS ensures that its own handler for checkpoint is always called *after* your app gets the event, you can be assured that WinJS will save whatever you write into sessionState at any time before your checkpoint handler returns.

Then, when the app is activated with the previous state of terminated, WinJS automatically rehydrates the sessionState object so that everything you put there is once again available. If you use this object for storing variables, you need only to avoid setting those values back to their defaults when reloading your state.

Finally, if you don't want to use the sessionState object or you have state that won't work with it, the WinJS.Application object makes it easy to write your own files without having to use async WinRT APIs. Specifically, it provides (as shown in the documentation) local, temp, and roaming objects that each have methods called readText, writeText, exists, and remove. These objects each work within their respective appdata folders and provide a simplified API for file I/O, as shown in scenario 1 of the App model sample.

Suspending Deferrals and Deadlines

As noted earlier, the suspending event has a deferral mechanism, like activation, to accommodate async operations in your handler. That is, Windows will normally suspend your app as soon as you return from the suspending event (regardless of whether five seconds have elapsed), unless you request a deferral.

The event args for suspending contains an instance of Windows.UI.WebUI.WebUIApplication.-SuspendingOperation. Its getDeferral method returns a deferral object with a complete method, which you call when your async operations are finished. WinJS wraps this with the setPromise method on the event args object passed to a checkpoint handler. To this you pass whatever promise you have for your async work and WinJS automatically adds a completed handler that calls the deferral's complete method.

Well, hey! All this sounds pretty good—is this perhaps a sneaky way to circumvent the restriction on running Windows Store apps in the background? Will my app keep running indefinitely if I request a deferral by never calling complete?

No such luck, amigo. Accept my apologies for giving you a fleeting moment of exhilaration! Deferral or not, five seconds is the *most* you'll ever get. Still, you might want to take full advantage of that time, perhaps to first perform critical async operations (like flushing a cache) and then to attempt other noncritical operations (like a sync to a server) that might greatly improve the user experience. For such purposes, the suspendingOperation object also contains a deadline property, a Date value indicating the time in the future when Windows will forcibly suspend you regardless of any deferral. Once the first operation is complete, you can check if you have time to start another, and so on.

Note The suspendingOperation object is not surfaced through the WinJS checkpoint event; if you want to work with the deadline property, you must use a handler for the WinRT suspending event.

A basic demonstration of using the suspending deferral can be found in the App activated, resume, and suspend sample. This also shows activation through a custom URI scheme, a subject that we'll be covering later in Chapter 15. An example of handling state, in addition to the updates we'll make to Here My Am! in the next section, can be found in scenario 3 of the App model sample.

Basic Session State in Here My Am!

To demonstrate some basic handling of session state, I've made a few changes to Here My Am! as given in the HereMyAm3b example in the companion content. Here we have two pieces of information we care about: the variables `lastCapture` (a `StorageFile` with the image) and `lastPosition` (a set of coordinates). We want to make sure we save these when we get suspended so that we can properly apply those values when the app gets launched with the previous state of `terminated`.

With `lastPosition`, we can just move this into the `sessionState` object (prepending `app.sessionState.`). If this value exists on startup, we can skip making the call to `getGeopositionAsync` because we already have a location:

```
//If we don't have a position in sessionState, try to initialize
if (!app.sessionState.lastPosition) {
    locator.getGeopositionAsync().done(function (geocoord) {
        var position = geocoord.coordinate.point.position;

        //Save for share
        app.sessionState.lastPosition = {
            latitude: position.latitude, longitude: position..longitude };

        updatePosition();
    }, function (error) {
        console.log("Unable to get location.");
    });
}
```

With this change I've also moved the bit of code to update the map location into a separate function that ensures a location exists in `sessionState`:

```
function updatePosition() {
    if (!app.sessionState.lastPosition) {
        return;
    }

    callFrameScript(document.frames["map"], "pinLocation",
        [app.sessionState.lastPosition.latitude, app.sessionState.lastPosition.longitude]);
}
```

Note also that because `app.sessionState` is initialized to an empty object by default, `{ }`, `lastPosition` will be `undefined` until the geolocation call succeeds. This also works to our advantage when rehydrating the app. Here's what the `previousExecutionState` conditions look like for this:

```
if (args.detail.previousExecutionState !==
    activation.ApplicationExecutionState.terminated) {
    //Normal startup: initialize lastPosition through geolocation API
} else {
    //WinJS reloads the sessionState object here. So try to pin the map with the saved location
    updatePosition();
}
```

Because the contents of `sessionState` are automatically saved in `WinJS.Application.oncheckpoint` and automatically reloaded when the app is restarted with the previous state of `terminated`, our previous location will exist in `sessionState` and `updatePosition` just works.

You can test all this by running the HereMyAm3b app, taking a suitable picture and making sure you have a location. Then use the *Suspend and Shutdown* option on the Visual Studio toolbar to terminate the app. Set a breakpoint on the `updatePosition` call above, and then restart the app in the debugger. You'll see that `sessionState.lastPosition` is initialized at that point.

With the last captured picture, we don't need to save the `StorageFile`, just the URI: we copied the file into our local appdata (so it persists across sessions already) and can just use the `ms-appdata://` URI scheme to refer to it. When we capture an image, we just save that URI into `sessionState.imageURI` (the property name is arbitrary) at the end of the promise chain inside `capturePhoto`:

```
app.sessionState.imageURI = "ms-appdata:///local/HereMyAm/" + newFile.name;
```

Again, because `imageURI` is saved within `sessionState`, this value will be available when the app is restarted after being terminated. We also need to re-initialize `lastCapture` with a `StorageFile` so that the image is available through the Share contract. For this we can use `Windows.Storage.StorageFile.getFileFromApplicationUriAsync`. Here, then, is the code within the `previousExecutionState == terminated` case during activation:

```
//WinJS reloads the sessionState object here: initialize from the saved image URI and location.
if (app.sessionState.imageURI) {
    var uri = new Windows.Foundation.Uri(app.sessionState.imageURI);
    Windows.Storage.StorageFile.getFileFromApplicationUriAsync(uri).done(function (file) {
        lastCapture = file;
        var img = document.getElementById("photoImg");
        scaleImageToFit(img, document.getElementById("photo"), file);
    });
}

updatePosition();
```

As always, the code to set `img.src` with a thumbnail happens inside `scaleImageToFit`. This call is also inside the completed handler here because we want the image to appear only if we can also access its `StorageFile` again for sharing. Otherwise the two features of the app would be out of sync.

In all of this, note again that we don't need to explicitly reload these variables within the `terminated` case because WinJS reloads `sessionState` automatically. If we managed our state more directly, such as storing some variables in roaming settings within the `checkpoint` event, we would reload and apply those values at this time.

> **Note** Using `ms-appdata:///` and `getFileFromApplicationUriAsync` (or its sibling `getFileFromPathAsync`) works because the file exists in a location that we can access programmatically by default. It also works for libraries for which we declare a capability in the manifest. If, however, we obtain a `StorageFile` from the file picker, we need to save that in the `Windows.Storage.AccessCache` to preserve access permissions across sessions. We'll revisit the access cache in Chapter 11, "The Story of State, Part 2."

Page Controls and Navigation

Now we come to an aspect of Windows Store apps that very much separates them from typical web applications but makes them very similar to AJAX-based sites.

To compare, many web applications do page-to-page navigation with `<a href>` hyperlinks or by setting `document.location` from JavaScript. This is all well and good: oftentimes there's little or no state to pass between pages, and even then there are well-established mechanisms for doing so, such as HTML5 `sessionStorage` and `localStorage` (which work just fine in Store apps, by the way).

This type of navigation presents a few problems for Store apps, however. For one, navigating to a new page means a wholly new script context—all the JavaScript variables from your previous page will be lost. Sure, you can pass state between those pages, but managing this across an entire app likely hurts performance and can quickly become your least favorite programming activity. It's better and easier, in other words, for client apps to maintain a consistent in-memory state across pages and also have each individual page be able to load what script it uniquely needs, as needed.

Also, the nature of the HTML/CSS rendering engine is such that a blank screen appears when navigating a hyperlink. Users of web applications are accustomed to waiting a bit for a browser to acquire a new page (I've found many things to do with an extra 15 seconds!), but this isn't an appropriate user experience for a fast and fluid Windows Store app. Furthermore, such a transition doesn't allow animation of various elements on and off the screen, which can help provide a sense of continuity between pages if that fits with your design.

So, although you can use direct links, Store apps typically implement "pages" by dynamically replacing sections of the DOM within the context of a single page like default.html, akin to how "single-page" web applications work. By doing so, the script context is always preserved and individual elements or groups of elements can be transitioned however you like. In some cases, it even makes sense to simply show and hide pages so that you can switch back and forth quickly. Let's look at the strategies and tools for accomplishing these goals.

WinJS Tools for Pages and Page Navigation

Windows itself, and the app host, provides no mechanism for dealing with pages—from the system's perspective, this is merely an implementation detail for apps to worry about. Fortunately, the engineers who created WinJS and the templates in Visual Studio and Blend worried about this a lot! As a result, they've provided some marvelous tools for managing bits and pieces of HTML+CSS+JS in the context of a single container page:

- `WinJS.UI.Fragments` contains a low-level "fragment-loading" API, the use of which is necessary only when you want close control over the process (such as which parts of the HTML fragment get which parent). We won't cover it in this book; see the documentation and the Loading HTML fragments sample.

- WinJS.UI.Pages is a higher-level API intended for general use and is employed by the templates. Think of this as a generic wrapper around the fragment loader that lets you easily define a "page control"—simply an arbitrary unit of HTML, CSS, and JS—that you can easily pull into the context of another page as you do other controls.[20] They are, in fact, implemented like other controls in WinJS (as we'll see in Chapter 5), so you can declare them in markup, instantiate them with WinJS.UI.process[All], use as many of them within a single host page as you like, and even nest them.

These APIs provide *only* the means to load and unload individual "pages"—they pull HTML in from other files (along with referenced CSS and JS) and attach the contents to an element in the DOM. That's it. As such they can be used for any number of purposes, such as a custom control model, depending on how you like to structure your code. See scenario 1 of the HTML Page controls sample.

Page controls and fragments are not gospel To be clear, there's *absolutely no requirement* that you use the WinJS mechanisms described here in a Windows Store app. These are simply convenient *tools* for common coding patterns. In the end, it's just about making the right elements and content appear in the DOM for your user experience, and you can implement that however you like.

Assuming that you'll want to save yourself loads of trouble and use WinJS for page-to-page navigation, you'll need two other pieces. The first is something to manage a navigation stack, and the second is something to hook navigation events to the loading mechanism of WinJS.UI.Pages.

For the first piece, you can turn to WinJS.Navigation, which supplies, through about 150 lines of CS101-level code, a basic navigation stack. This is all it does. The stack itself is just a list of URIs on top of which WinJS.Navigation exposes location, history, canGoBack, and canGoForward properties, along with one called state in which you can store any app-defined object you need. The stack (maintained in history) is manipulated through the forward, back, and navigate methods, and the WinJS.Navigation object raises a few events—beforenavigate, navigating, and navigated—to anyone who wants to listen (through addEventListener).[21]

Tip In the WinJS.Navigation.history.current object there's an initialPlaceholder flag that answers the question, "Can WinJS.Navigation.navigate go to a new page without adding an entry in the history?" If you set this flag to true, subsequent navigations won't be stored in the nav stack. Be sure to set it back to false to reenable the stack.

What this means is that WinJS.Navigation by itself doesn't really do anything unless some other piece of code is listening to those events. That is, for the second piece of the navigation puzzle we need a linkage between WinJS.Navigation and WinJS.UI.Pages, such that a navigation event causes the

[20] If you are at all familiar with user controls in XAML, this is the same idea.

[21] The beforenavigate event can be used to cancel the navigation, if necessary. Either call args.preventDefault (args being the event object), return true, or call args.setPromise where the promise is fulfilled with true.

target page contents to be added to the DOM and the current page contents to be removed.

The basic process is as follows, and it's also shown in Figure 3-4:

1. Create a new `div` with the appropriate size (typically the whole app window).
2. Call `WinJS.UI.Pages.render` to load the target HTML into that element (along with any script that the page uniquely references). This is an async function that returns a promise. We'll take a look at what `render` does later on.
3. When that loading (that is, rendering) is complete, attach the new element from step 1 to the DOM.
4. Remove the previous page's root element from the DOM. If you do this before yielding the UI thread, you won't see both pages on-screen together.

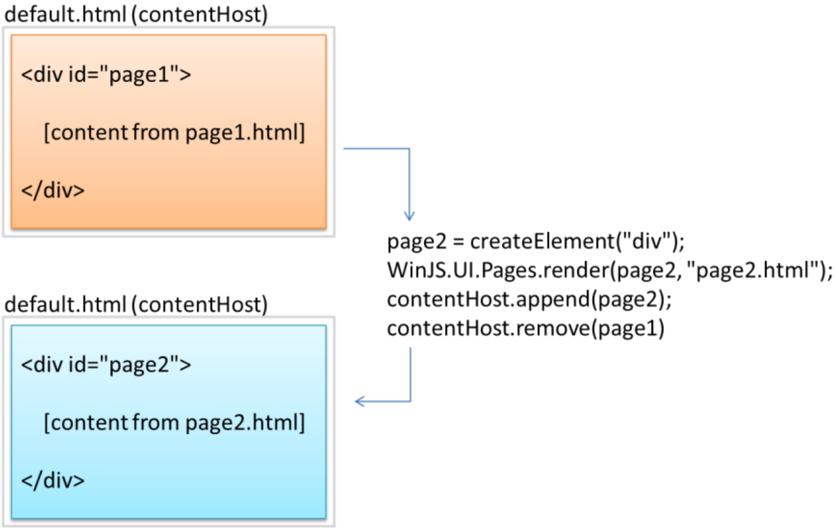

default.html (contentHost)
<div id="page1">
[content from page1.html]
</div>
page2 = createElement("div");
WinJS.UI.Pages.render(page2, "page2.html");
contentHost.append(page2);
contentHost.remove(page1)
default.html (contentHost)
<div id="page2">
[content from page2.html]
</div>

FIGURE 3-4 Performing page navigation in the context of a single host (typically default.html) by replacing appending the content from page2.html and removing that from page1.html. Typically, each page occupies the whole display area, but page controls can just as easily be used for smaller areas.

As with page navigation in general, you're again free to do whatever you want here, and in the early developer previews of Windows 8 that's all that you could do! But as developers built the first apps for the Windows Store, we discovered that most people ended up writing just about the same boilerplate code over and over. Seeing this pattern, two standard pieces of code have emerged. One is the WinJS back button control, `WinJS.UI.BackButton`, which listens for navigation events to enable itself when appropriate. The other is a piece is called the `PageControlNavigator` and is magnanimously supplied by the Visual Studio templates. Hooray!

Because the `PageControlNavigator` is just a piece of template-supplied code and not part of WinJS, it's entirely under your control: you can tweak, hack, or lobotomize it however you want.[22] In any case, because it's likely that you'll often use the `PageControlNavigator` (and the back button) in your own apps, let's look at how it all works in the context of the Navigation App template.

> **Note** Additional samples that demonstrate basic page controls and navigation, along with handling session state, can be found in the following SDK samples: App activate and suspend using WinJS (using the session state object in a page control), App activated, resume and suspend (described earlier; shows using the suspending deferral and restarting after termination), and Navigation and navigation history (showing page navigation along with tracking and manipulating the navigation history). In fact, just about every sample uses page controls to switch between different scenarios, so you have no shortage of examples to draw from!

The Navigation App Template, PageControl Structure, and PageControlNavigator

Taking one step beyond the Blank App template, the Navigation App template demonstrates the basic use of page controls. (The more complex templates build navigation out further.) If you crea te a new project with this template in Visual Studio or Blend, here's what you'll get:

- **default.html** Contains a single container `div` with a `PageControlNavigator` control pointing to pages/home/home.html as the app's home page.
- **js/default.js** Contains basic activation and state checkpoint code for the app.
- **css/default.css** Contains global styles.
- **pages/home** Contains a page control for the "home page" contents, composed of **home.html**, **home.js**, and **home.css**. Every page control typically has its own markup, script, and style files. Note that CSS styles for page controls are cumulative as you navigate from page to page. See "Page-Specific Styling" later in this chapter.
- **js/navigator.js** Contains the implementation of the `PageControlNavigator` class.

To build upon this structure, you can add additional pages to the app with the page control *item* template in Visual Studio. For each page I recommend first creating a specific folder under *pages*, similar to *home* in the default project structure. Then right-click that folder, select Add > New Item, and select Page Control. This will create suitably named .html, .js. and .css files in that folder.

Now let's look at the body of default.html (omitting the standard header and a commented-out AppBar control):

[22] The Quickstart: using single-page navigation topic also shows a clever way to hijack HTML `<a href>` hyperlinks and hook them into `WinJS.Navigation.navigate`. This can be a useful tool, especially if you're importing code from a web app or otherwise want to create page links in declarative markup.

```
<body>
    <div id="contenthost" data-win-control="Application.PageControlNavigator"
        data-win-options="{home: '/pages/home/home.html'}"></div>
</body>
```

All we have here is a single container `div` named *contenthost* (it can be whatever you want), in which we declare the `Application.PageControlNavigator` as a custom WinJS control. (This is the purpose of `data-win-control` and `data-win-options`, as we'll see in Chapter 5.) With this we specify a single option to identify the first page control it should load (/pages/home/home.html). The `PageControlNavigator` will be instantiated within our `activated` handler's call to `WinJS.UI.processAll`.

Within home.html we have the basic markup for a page control. Below is what the Navigation App template provides as a home page by default, and it's pretty much what you get whenever you add a new page control from the item template (with different filenames, of course):

```
<!DOCTYPE html>
<html>
<head>
   <!--... typical HTML header and WinJS references omitted -->
   <link href="/css/default.css" rel="stylesheet">
   <link href="/pages/home/home.css" rel="stylesheet">
   <script src="/pages/home/home.js"></script>
</head>
<body>
   <!-- The content that will be loaded and displayed. -->
   <div class="fragment homepage">
       <header aria-label="Header content" role="banner">
          <button data-win-control="WinJS.UI.BackButton"></button>
          <h1 class="titlearea win-type-ellipsis">
             <span class="pagetitle">Welcome to NavApp!</span>
          </h1>
       </header>
       <section aria-label="Main content" role="main">
          <p>Content goes here.</p>
       </section>
   </div>
</body>
</html>
```

The `div` with *fragment* and *homepage* CSS classes, along with the `header`, creates a page with a standard silhouette and a `WinJS.UI.BackButton` control that automatically wires up keyboard, mouse, and touch events and again keeps itself hidden when there's nothing to navigate back to. (Isn't that considerate of it!) All you need to do is customize the text within the `h1` element and the contents within `section`, or just replace the whole smash with the markup you want. (By the way, even though the WinJS files are referenced in each page control, they aren't actually reloaded; they exist here to allow you to edit a standalone page control in Blend.)

Tip The leading / on what looks like relative paths to CSS and JavaScript files actually creates an absolute reference from the package root. If you omit that /, there are many times—especially with path controls—when the relative path is not what you'd expect, and the app doesn't work. In general, unless you really know you want a relative path, use the leading /.

The definition of the actual page control is in pages/home/home.js; by default, the templates just provide the bare minimum:

```
(function () {
    "use strict";

    WinJS.UI.Pages.define("/pages/home/home.html", {
        // This function is called whenever a user navigates to this page. It
        // populates the page elements with the app's data.
        ready: function (element, options) {
            // TODO: Initialize the page here.
        }
    });
})();
```

The most important part is `WinJS.UI.Pages.define`, which associates a *project-based URI* (the page control identifier, always starting with a /, meaning the project root), with an *object* containing the page control's methods. Note that the nature of `define` allows you to define different members of the page in multiple places: multiple calls to `WinJS.UI.Pages.define` with the same URI will add members to an existing definition and replace those that already exist.

Tip Be mindful that if you have a typo in the URI that creates a mismatch between the URI in `define` and the actual path to the page, the page won't load but there won't be an exception or other visible error. You'll be left wondering what's going wrong! So, if your page isn't loading like you think it should, carefully examine the URI and the file paths to make sure they match exactly.

For a page created with the Page Control item template, you get a couple more methods in the structure (some comments omitted; in this example *page2* was created in the pages/page2 folder):

```
(function () {
    "use strict";

    WinJS.UI.Pages.define("/pages/page2/page2.html", {
        ready: function (element, options) {
        },

        unload: function () {
            // TODO: Respond to navigations away from this page.
        }

        updateLayout: function (element) {
            // TODO: Respond to changes in layout.
        },
    });
})();
```

A page control is essentially just an object with some standard methods. You can instantiate the control from JavaScript with new by first obtaining its constructor function from WinJS.UI.Pages.get(<page_uri>) and then calling that constructor with the parent element and an object containing its options. This operation already encapsulated within WinJS.UI.Pages.render, as we'll see shortly.

Although a basic structure for the ready method is provided by the templates, WinJS.UI.Pages and the PageControlNavigator will make use of the following if they are available, which are technically the members of an interface called WinJS.UI.IPageControlMembers:

PageControl Method	When Called
init	Called before elements from the page control have been created.
processed	Called after WinJS.UI.processAll is complete (that is, controls in the page have been instantiated, which is done automatically), but before page content itself has been added to the DOM. Once you return from this method—or a promise you return is fulfilled—WinJS animates the new page into view with WinJS.UI.Animation.enterPage, so all initialization of properties and data-binding should occur within this method; it's also a good place to load string resources.
ready	Called after the page have been added to the DOM (and before the unload of the previous page; note that in WinJS 1.0 this was called after the previous page's unload).
error	Called if an error occurs in loading or rendering the page.
unload	Called when navigation has left the page. By default, WinJS automatically disposes of controls on a page when that page is unloaded; see "Sidebar: The Ubiquitous dispose Method" in Chapter 5.
updateLayout	Called in response to the window.onresize event, which signals changes between various view states.

Note that WinJS.UI.Pages calls the first four methods; the unload and updateLayout methods, on the other hand, are used only by the PageControlNavigator.

Of all of these, the ready method is the most common one to implement. It's where you'll do further initialization of controls (e.g., populate lists), wire up other page-specific event handlers, and so on. Any processing that you want to do before the page content is added to the DOM should happen in processed, and note that if you return a promise from processed, WinJS will wait until that promise is fulfilled before starting the enterpage animation.

The unload method is also where you'll want to remove event listeners for WinRT objects, as described earlier in this chapter in "WinRT Events and removeEventListener." The updateLayout method is important when you need to adapt your page layout to a new view, as we've been doing in the Here My Am! app.

As for the PageControlNavigator itself, which I'll just refer to as the "navigator," the code in js/navigator.js shows how it's defined and how it wires up navigation events in its constructor:

```
(function () {
    "use strict";

    // [some bits omitted]
    var nav = WinJS.Navigation;

    WinJS.Namespace.define("Application", {
        PageControlNavigator: WinJS.Class.define(
        // Define the constructor function for the PageControlNavigator.
            function PageControlNavigator (element, options) {
                this.element = element || document.createElement("div");
                this.element.appendChild(this._createPageElement());

                this.home = options.home;

                // ...

                // Adding event listeners; addRemovableEventListener is a helper function
                addRemovableEventListener(nav, 'navigating',
                    this._navigating.bind(this), false);
                addRemovableEventListener(nav, 'navigated',
                    this._navigated.bind(this), false);

                // ...
            }, {
    // ...
```

First we see the definition of the `Application` namespace as a container for the `PageControlNavigator` class (see "Sidebar: WinJS.Namespace.define and WinJS.Class.define" later). Its constructor receives the `element` that contains it (the *contenthost* `div` in default.html), or it creates a new one if none is given. The constructor also receives an `options` object that is the result of parsing the `data-win-options` string of that element. The navigator then appends the page control's contents to this root element, adds listeners for the `WinJS.Navigation.onnavigated` event, among others.[23]

The navigator then waits for someone to call `WinJS.Navigation.navigate`, which happens in the `activated` handler of js/default.js, to navigate to either the home page or the last page viewed if previous session state was reloaded:

```
if (app.sessionState.history) {
    nav.history = app.sessionState.history;
}
args.setPromise(WinJS.UI.processAll().then(function () {
    if (nav.location) {
        nav.history.current.initialPlaceholder = true;  // Don't add first page to nav stack
        return nav.navigate(nav.location, nav.state);
    } else {
        return nav.navigate(Application.navigator.home);
    }
}));
```

Notice how this code is using the WinJS `sessionState` object exactly as described earlier in this chapter, taking advantage again of `sessionState` being automatically reloaded when appropriate.

When a navigation happens, the navigator's `_navigating` handler is invoked, which in turn calls `WinJS.UI.Pages.render` to do the loading, the contents of which are then appended as child

[23] If the use of `.bind(this)` is unfamiliar to you, please see my blog post, The purpose of this<event>.bind(this).

elements to the navigator control:

```
_navigating: function (args) {
    var newElement = this._createPageElement();
    var parentedComplete;
    var parented = new WinJS.Promise(function (c) { parentedComplete = c; });

    this._lastNavigationPromise.cancel();

    this._lastNavigationPromise = WinJS.Promise.timeout().then(function () {
        return WinJS.UI.Pages.render(args.detail.location, newElement,
            args.detail.state, parented);
    }).then(function parentElement(control) {
        var oldElement = this.pageElement;
        if (oldElement.winControl && oldElement.winControl.unload) {
            oldElement.winControl.unload();
        }
        WinJS.Utilities.disposeSubTree(this._element);
        this._element.appendChild(newElement);
        this._element.removeChild(oldElement);
        oldElement.innerText = "";
        parentedComplete();
    }.bind(this));

    args.detail.setPromise(this._lastNavigationPromise);
},
```

If you look past all the business with promises that you see here (which essentially makes sure the rendering and parenting process is both asynchronous and yields the UI thread), you can see how the navigator is handling the core process shown earlier in Figure 3-4. It first creates a new page element. Then it calls the previous page's `unload` event, after which it asynchronously loads the new page's content. Once that's complete, the new page's content is added to the DOM and the old page's contents are removed. Note that the navigator uses the WinJS *disposal* helper, `WinJS.Utilities.-disposeSubTree` to make sure that we fully clean up the old page. This disposal pattern invokes the navigator's `dispose` method (also in navigator.js), which makes sure to release any resources held by the page and any controls within it, including event listeners. (More on this in Chapter 5.)

Tip In a page control's JavaScript code you can use `this.element.querySelector` rather than `document.querySelector` if you want to look only in the page control's contents and have no need to traverse the entire DOM. Because `this.element` is just a node, however, it does not have other traversal methods like `getElementById` (which, by the way, operates off an optimized lookup table and actually doesn't traverse anything).

And that, my friends, is how it works! In addition to the HTML Page controls sample, and to show a concrete example of doing this in a real app, the code in the HereMyAm3c sample has been converted to use this model for its single home page. To make this conversion, I started with a new project by using the Navigation App template to get the page navigation structures set up. Then I copied or imported the relevant code and resources from HereMyAm3b, primarily into pages/home/home.html, home.js, and home.css. And remember how I said that you could open a page control directly in Blend (which is why pages have WinJS references)? As an exercise, open the HereMyAm3c project in Blend. You'll first see that everything shows up in default.html, but you can also open home.html by itself and edit just that page.

> **Note** To give an example of calling removeEventListener for the WinRT datarequested event, I make this call in the unload method of pages/home/home.js.

Be aware that WinJS calls WinJS.UI.processAll in the process of loading a page control (before calling the processed method), so we don't need to concern ourselves with that detail when using WinJS controls in a page. On the other hand, reloading state when previousExecutionState == terminated needs some attention. Because this is picked up in the WinJS.Application.onactivated event *before* any page controls are loaded and before the PageControlNavigator is even instantiated, we need to remember that condition so that the home page's ready method can later initialize itself accordingly from app.sessionState values. For this I simply write another flag into app.sessionState called initFromState (true if previousExecutionState is terminated, false otherwise.) The page initialization code, now in the page's ready method, checks this flag to determine whether to reload session state.

The other small change I made to HereMyAm3c is to use the updateLayout method in the page control rather than attaching my own handler to window.onresize. With this I also needed to add a height: 100%; style to the *#mainContent* rule in home.css. In previous iterations of this example, the *mainContent* element was a direct child of the body element and it inherited the full screen height automatically. Now, however, it's a child of the *contentHost*, so the height doesn't automatically pass through and we need to set it to 100% explicitly.

Sidebar: WinJS.Namespace.define and WinJS.Class.define

WinJS.Namespace.define provides a shortcut for the JavaScript namespace pattern. This helps to minimize pollution of the global namespace as each app-defined namespace is just a single object in the global namespace but can provide access to any number of other objects, functions, and so on. This is used extensively in WinJS and is recommended for apps as well, where you define everything you need in a module—that is, within a (function() { ... })() block—and then export selective variables or functions through a namespace. In short, use a namespace anytime you're tempted to add any global objects or functions!

Here's the syntax: var ns = WinJS.Namespace.define(<name>, <members>) where <name> is a string (dots are OK) and <members> is any object contained in { }'s. Also, WinJS.Namespace.defineWithParent(<parent>, <name>, <members>) defines one within the <parent> namespace.

If you call WinJS.Namespace.define for the same <name> multiple times, the <members> are combined. Where collisions are concerned, the most recently added members win. For example:

```
WinJS.Namespace.define("MyNamespace", { x: 10, y: 10 });
WinJS.Namespace.define("MyNamespace", { x: 20, z: 10 });
//MyNamespace == { x: 20, y: 10, z: 10}
```

WinJS.Class.define is, for its part, a shortcut for the object pattern, defining a constructor so that objects can be instantiated with new.

Syntax: var className = WinJS.Class.define(<constructor>, <instanceMembers>, <staticMembers>) where <constructor> is a function, <instanceMembers> is an object with the class's properties and methods, and <staticMembers> is an object with properties and methods that can be directly accessed via <className>.<member> (without using new).

Variants: WinJS.Class.derive(<baseClass>, ...) creates a subclass (... is the same arg list as with define) using prototypal inheritance, and WinJS.Class.mix(<constructor>, [<classes>]) defines a class that combines the instance (but not static) members of one or more other <classes> and initializes the object with <constructor>.

Finally, note that because class definitions just generate an object, WinJS.Class.define is typically used inside a module with the resulting object exported to the rest of the app as a namespace member. Then you can use new <namespace>.<class> anywhere in the app.

For more details on classes in WinJS, see Appendix B.

Sidebar: Helping Out IntelliSense

If you start poking around in the WinJS source code—for example, to see how WinJS.UI.Pages is implemented—you'll encounter certain structures within code comments, often starting with a triple slash, ///. These are used by Visual Studio and Blend to provide rich IntelliSense within the code editors. You'll see, for example, /// <reference path…/> comments, which create a relationship between your current script file and other scripts to resolve externally defined functions and variables. This is explained on the JavaScript IntelliSense page in the documentation. For your own code, especially with namespaces and classes that you will use from other parts of your app, use these comment structures to describe your interfaces to IntelliSense. For details, see Extending JavaScript IntelliSense, and again look around the WinJS JavaScript files for many examples.

The Navigation Process and Navigation Styles

Having seen how page controls, WinJS.UI.Pages, WinJS.Navigation, and the PageControlNavigator all relate, it's straightforward to see how to navigate between multiple pages within the context of a single HTML container (e.g., default.html). With the PageControlNavigator instantiated and a page control defined via WinJS.UI.Pages, simply call WinJS.Navigation.navigate with the URI of that page control (its identifier). This loads that page's contents into a child element inside the

PageControlNavigator, unloading any previous page. That becomes page visible, thereby "navigating" to it so far as the user is concerned. You can also use (like the WinJS BackButton does) the other methods of WinJS.Navigation to move forward and back in the nav stack, which results in page contents being added and removed. The WinJS.Navigation.canGoBack and canGoForward properties allow you to enable/disable navigation controls as needed. Just remember that all the while, you'll still be in the overall context of your host page where you created the PageControlNavigator control.

As an example, create a new project using the Grid App template and look at these particular areas:

- **pages/groupedItems/groupedItems** is the home or "hub" page. It contains a ListView control (see Chapter 6, "Data Binding, Templates, and Collections") with a bunch of default items.
- Tapping a group header in the list navigates to section page (**pages/groupDetail**). This is done in pages/groupedItems/groupedItems.html, where an inline onclick handler event navigates to pages/groupDetail/groupDetail.html with an argument identifying the specific group to display. That argument comes into the ready function of pages/groupDetail/groupDetail.js.
- Tapping an item on the hub page goes to detail page (**pages/itemDetail**). The itemInvoked handler for the items, the _itemInvoked function in pages/groupedItems/groupedItem.js, calls WinJS.Navigation.navigate("/pages/itemDetail/itemDetail.html") with an argument identifying the specific item to display. As with groups, that argument comes into the ready function of pages/itemDetail/itemDetail.js.
- Tapping an item in the section page also goes to the details page through the same mechanism—see the _itemInvoked function in pages/groupDetail/groupDetail.js.
- The back buttons on all pages wire themselves into WinJS.Navigation.back for keyboard, mouse, and touch events.

The Split App template works similarly, where each list item on pages/items is wired to navigate to pages/split when invoked. Same with the Hub App template that has a hub page using the WinJS.UI.Hub control that we'll meet in Chapter 8.

The Grid App and Hub App templates also serve as examples of what's called the *Hub-Section-Item* navigation style (it's most explicitly so in the Hub App). Here the app's home page is the hub where the user can explore the full extent of the app. Tapping a group header navigates to a section, the second level of organization where only items from that group are displayed. Tapping an item (in the hub or in the section) navigates to a details page for that item. You can, of course, implement this navigation style however you like; the Grid App template uses page controls, WinJS.Navigation, and the PageControlNavigator. (Semantic zoom, as we'll see in Chapter 7, "Collection Controls," is also supported as a navigation tool to switch between hubs and sections.)

An alternate navigation choice is the *Flat* style, which simply has one level of hierarchy. Here, navigation happens to any given page at any time through a *navigation bar* (swiped in along with the app bar, as we'll see in Chapter 9). When using page controls and PageControlNavigator, navigation commands or buttons can just invoke `WinJS.Naviation.navigate` for this purpose. Note that in this style, there typically is no back button: users are expected to always swipe in the navigation bar from the top and go directly to the desired page.

These styles, along with many other UI aspects of navigation, can be found on Navigation design for Windows Store apps. This is an essential topic for designers.

Sidebar: Initial Login and In-App Licensing Agreements (EULA) Pages

Some apps might require either a login or acceptance of a license agreement to do anything, and thus it's appropriate that such pages are the first to appear in an app after the splash screen. In these cases, if the user does not accept a license or doesn't provide a login, the app should display a message describing the necessity of doing so, but it should *always* leave it to the user to close the app if desired. Do not close the app automatically. (This is a Store certification requirement.)

Typically, such pages appear only the first time the app is run. If the user provides a valid login, or if you obtain an access token through the Web Authentication Broker (see Chapter 4), those credentials/token can be saved for later use via the `Windows.Security.Credentials.PasswordVault` API. If the user accepts a EULA, that fact should be saved in appdata and reloaded anytime the app needs to check. These settings (login and acceptance of a license) should then always be accessible through the app's Settings charm. Legal notices, by the way, as well as license agreements, should always be accessible through Settings as well. See Guidelines and checklist for login controls.

Optimizing Page Switching: Show-and-Hide

Even with page controls, there is still a lot going on when navigating from page to page: one set of elements is removed from the DOM, and another is added in. Depending on the pages involved, this can be an expensive operation. For example, if you have a page that displays a list of hundreds or thousands of items, where tapping any item goes to a details page (as with the Grid App template), hitting the back button from a detail page will require complete reconstruction of the list (or at least its visible parts if the list is virtualized, which could still take a long time).

Showing progress indicators can help alleviate the user's anxiety, of course, but users are notoriously impatient and will likely want to quickly switch between a list of items and item details. (You've probably already encountered apps that seem to show progress indicators all the time for just about everything—how do they make you feel?) Indeed, the recommendation is that switching between fully interactive pages takes a quarter second or less, if possible, and no more than half a second. In some cases, completely swapping out chunks of the DOM with page controls will just become too time-consuming. (You could use a split master-detail view, of course, but that means splitting the available screen real estate.)

A good alternative is to actually keep the list/master page fully loaded the whole time. Instead of navigating to the item details page in the way we've seen, simply render that details page (using `WinJS.UI.Pages.render` directly) into another `div` that occupies the whole screen and overlays the list (similar to what we do with an extended splash screen), and then make that `div` visible *without* removing the list page from the DOM. When you dismiss the details page, just hide its `div`. This way you get the same effect as navigating between pages but the whole process is much quicker. You can also apply WinJS animations like enterContent and exitContent to make the transition more fluid.

If necessary, you can clear out the details `div` by just setting its `innerHTML` to `""`. However, if each details page has the same structure for every item, you can leave it entirely intact. When you "navigate" to the next details page, you would go through and refresh each element's data and properties for the new item before making that page visible. This could be significantly faster than rebuilding the details page all over again.

Note that because the `PageControlNavigator` implementation in navigator.js is provided by the templates and becomes part of your app, you can modify it however you like to handle these kinds of optimizations in a more structured manner that's transparent to the rest of your code.

Page-Specific Styling

When creating an app that uses page controls, you'll end up with each page having its own .css file in which you place page-specific styles. What's very important to understand here, though, is that while each page's HTML elements are dynamically added to and removed from the DOM, *any and all CSS that is loaded for page controls is cumulative to the app as a whole*. That is, styles behave like script and are preserved across page "navigations." This can be a source of confusion and frustration, so it's essential to understand what's happening here and how to work with it.

Let's say the app's root page is default.html and its global styles are in css/default.css. It then has several page controls defined in pages/page1 (page1.html. page1.js, page1.css), pages/page2 (page2.html. page2.js, page2.css), and pages/page1 (page3.html. page3.js, page3.css). Let's also say that page1 is the "home" page that's loaded at startup. This means that the styles in default.css and page1.css have been loaded when the app first appears.

Now the user navigates to page2. This causes the contents of page1.html to be dumped from the DOM, *but its styles remain in the stylesheet*. So when page2 is loaded, page2.css gets added to the overall stylesheet as well, and any styles in page2.css that have identical selectors to page1.css will overwrite those in page1.css. And when the user navigates to page3 the same thing happens again: the styles in page3.css are added in and overwrite any that already exist. But so far we haven't seen any unexpected effect of this.

Now, say the user navigates back to page1. Because the apphost's rendering engine has already loaded page1.css into the stylesheet, page1.css won't be loaded again. This means that any styles that were overwritten by other pages' stylesheets will not be reset to those in page1.css—basically you get whichever ones were loaded most recently. As a result, you can see some mix of the styles in page2.css and page3.css being applied to elements in page1.[24]

There are two ways to handle CSS files to avoid these problems. The first way is to take steps to avoid colliding selectors: use unique selectors for each page or can scope your styles to each page specifically. For the latter, wrap each page's contents in a top-level `div` with a unique class (as in `<div class="page1">`) so that you can scope every rule in page1.css with the page name. For example:

```
.page1 p {
    font-weight: bold;
}
```

Such a strategy can also be used to define stylesheets that are shared between pages, as with implementing style themes. If you scope the theme styles with a theme class, you can include that class in the top-level `div` to apply the theme.

A similar case arises if you want to use the ui-light.css and ui-dark.css WinJS stylesheets in different pages of the same app. Here, whichever one is loaded second will define the global styles such that subsequent pages that refer to ui-light.css might appear with the dark styles.

Fortunately, WinJS already scopes those styles that differ between the two files: those in ui-light.css are scoped with a CSS class `win-ui-light` and those in ui-dark.css are scoped with `win-ui-dark`. This means you can just refer to whichever stylesheet you use most often in your .html files and then add either `win-ui-light` or `win-ui-dark` to those elements that you need to style differently. When you add either class, note that the style will apply to that element and all its children. For a simple demonstration of an app with one dark page (as the default) and one light page, see the PageStyling example in the companion content.

The other way of avoiding collisions is to specifically unload and reload CSS files by modifying `<link>` tags in the page header. You can either remove one `<link>` tag and add a different one, toggle the `disabled` attribute for a tag between `true` and `false`, or change the `href` attribute of an existing link. These methods are demonstrated for styling an `iframe` in the CSS styling and branding your app sample, which swaps out and enables/disables both WinJS and app-specific stylesheets. Another demonstration for switching between the WinJS stylesheets is in scenario 1 of the HTML NavBar control sample that we'll see more of in Chapter 9 (js/1-CreateNavBar.js):

```
function switchStyle() {
    var linkEl = document.querySelector('link');
    if (linkEl.getAttribute('href') === "//Microsoft.WinJS.2.0 /css/ui-light.css") {
        linkEl.setAttribute('href', "//Microsoft.WinJS.2.0 /css/ui-dark.css");
    } else {
        linkEl.setAttribute('href', "//Microsoft.WinJS.2.0 /css/ui-light.css");
    }
}
```

[24] The same thing happens with .js files, by the way, which are not reloaded if they've been loaded already. To avoid collisions in JavaScript, you either have to be careful to not duplicate variable names or to use namespaces to isolate them from one another.

The downside of this approach is that every switch means reloading and reparsing the CSS files and a corresponding re-rendering of the page. This isn't much of an issue during page navigation, but given the size of the WinJS files I recommend using it only for your own page-specific stylesheets and using the `win-ui-light` and `win-ui-dark` classes to toggle the WinJS styles.

Async Operations: Be True to Your Promises

Even though we've just got our first apps going, we've already seen a lot to do with async operations and promises. We've seen their basic usage, and in the "Moving the Captured Image to AppData (or the Pictures Library)" section of Chapter 2, we saw how to combine multiple async operations into a sequential chain. At other times you might want to combine multiple parallel async operations into a single promise. Indeed, as you progress through this book you'll find that async APIs, and thus promises, seem to pop up as often as dandelions in a lawn (without being a noxious weed, of course)! Indeed, the implementation of the `PageControlNavigator._navigating` method that we saw earlier has a few characteristics that are worth exploring.

To reiterate a very important point, promises are simply how async operations in WinRT are projected into JavaScript, which matches how WinJS and other JavaScript libraries typically handle asynchronous work. And because you'll be using all sorts of async APIs in your development work, you're going to be using promises quite frequently and will want to understand them deeply.

Note There are a number of different specifications for promises. The one presently used in WinJS and the WinRT API is known as Common JS/Promises A. Promises in jQuery also follow this convention and are thus interoperable with WinJS promises.

The subject of promises gets rather involved, however, so instead of burdening you with the details in the main flow of this chapter, you'll find a full treatment of promises in Appendix A, "Demystifying Promises." Here I want to focus on the most essential aspects of promises and async operations that we'll encounter throughout the rest of this book, and we'll take a quick look at the features of the `WinJS.Promise` class. Examples of the concepts can be found in the WinJS Promise sample.

Using Promises

The first thing to understand about a promise is that it's really nothing more than a code construct or a calling convention. As such, promises have no inherent relationship to async operations—they just so happen to be very *useful* in that regard! A promise is simply an object that represents a value that might be available at some point in the future (or might be available already). It's just like we use the term in human relationships. If I say to you, "I promise to deliver a dozen donuts," it doesn't matter when and how I get them (or even whether I have them already in hand), it only matters that I deliver them at some point in the future.

A promise, then, implies a relationship between two people or, to be more generic, two *agents*, as I call them. There is the *originator* who makes the promise—that is, the one who has some goods to deliver—and the *consumer* or recipient of that promise, who will also be the later recipient of the goods. In this relationship, the originator creates a promise in response to some request from the consumer (typically an API call). The consumer can then do whatever it wants with both the promise itself and whatever goods the promise delivers. This includes sharing the promise with other interested consumers—the promise will deliver its goods to each of them.

The way a consumer listens for delivery is by subscribing a *completed handler* through the promise's `then` or `done` methods. (We'll discuss the differences later.) The promise invokes this handler when it has obtained its results. In the meantime, the consumer can do other work, which is exactly why promises are used with async operations. It's like the difference between waiting in line at a restaurant's drive-through for a potentially very long time (the synchronous model) and calling out for pizza delivery (the asynchronous model): the latter gives you the freedom to do other things.

Of course, if the promised value is already available, there's no need to wait: it will be delivered synchronously to the completed handler as soon as `then/done` is called.

Similarly, problems can arise that make it impossible to fulfill the promise. In this case the promise will invoke any *error handlers* given to `then/done` as the second argument. Those handlers receive an error object containing `name` and `message` properties with more details, and after this point the promise is in what's called the *error state*. This means that any subsequent calls to `then`/`done` will immediately (and synchronously) invoke any given error handlers.

A consumer can also cancel a promise if it decides it no longer needs the results. A promise has a `cancel` method for this purpose, and calling it both halts any underlying async operation represented by the promise (however complex it might be) and puts the promise into the error state.

Some promises—which is to say, some async operations—also support the ability to report intermediate results to any *progress handlers* given to `then`/`done` as the third argument. Check the documentation for the particular API in question.[25]

Finally, two static methods on the `WinJS.Promise` object might come in handy when using promises:

- `is` determines whether an arbitrary value is a promise, returning a Boolean. It basically makes sure it's an object with a function named "then"; it does not test for "done".
- `theneach` takes an array of promises and subscribes completed, error, and progress handlers to each promise by calling its `then` method. Any of the handlers can be `null`. The return value of `theneach` is itself a promise that's fulfilled when all the promises in the array are fulfilled. We call this a *join*, as described in the next section.

[25] If you want to impress your friends while reading the WinRT API documentation, know that if an async function shows it returns `IAsync[Action | Operation]WithProgress` (for whatever result type), it will invoke progress handlers. If it lists only `IAsync[Action | Operation]`, progress is not supported.

Tip If you're new to the concept of *static methods*, these refer to functions that exist on an object class that you call directly through the fully-qualified name, such as WinJS.Promise.theneach. These are distinct from *instance methods*, which must be called through a specific instance of the class. For example, if you have a WinJS.Promise object in the variable *p*, you cancel that particular instance with p.cancel().

Joining Parallel Promises

Because promises are often used to wrap asynchronous operations, it's certainly possible that you can have multiple operations going on in parallel. In these cases you might want to know either when one promise in a group is fulfilled or when all the promises in the group are fulfilled. The static functions WinJS.Promise.any and WinJS.Promise.join provide for this. Here's how they compare:

Function	**any**	**join**
Arguments	An array of promises	An array of promises
Fulfilled when	One of the promises is fulfilled (a logical OR)	All of the promises are fulfilled (a logical AND)
Fulfilled result	This is a little odd. It's an object whose key property identifies the promise that was fulfilled and whose value property is an object containing that promise's state. Within that state is a _value property that contains the actual result of that promise.	This isn't clearly documented but can be understood from the source code or simple tests from the consumer side. If the promises in the join all complete, the completed handler receives an array of results from the individual promises (even if those results are null or undefined). If there's an error in the join, the error object passed to the error handler is an array that contains the individual errors.
Progress behavior	None	Reports progress to any subscribed handlers where the intermediate results are an array of results from those individual promises that have been fulfilled so far.
Behavior after fulfillment	All the operations for the remaining promises continue to run, calling whatever handlers might have been subscribed individually.	None—all promises have been fulfilled.
Behavior upon cancellation	Canceling the promise from any cancels all promises in the array, even if the first has already been fulfilled.	Cancels all other promises that are still pending.
Behavior upon errors	Invokes the subscribed error handler for every error in the individual promises. This one error handler, in other words, can monitor conditions of the underlying promises.	Invokes the subscribed error handler with an array of error objects from any failed promises, but the remainder continue to run. In other words, this reports cumulative errors in the way that progress reports cumulative completions.

Appendix A, by the way, has a small code snippet that shows how to use join and the array's reduce method to execute parallel operations but have their results delivered in a specific sequence.

Sequential Promises: Nesting and Chaining

In Chapter 2, when we added code to Here My Am! to copy the captured image to another folder, we got our first taste of using chained promises to run sequential async operations. To review, what makes this work is that any promise's then method returns another promise that's fulfilled when the given completed handler returns. (That returned promise also enters the error state if the first promise has an error.) That completed handler, for its part, returns the promise from the next async operation in the chain, the results of which are delivered to the next completed handler down the line.

Though it may look odd at first, chaining is the most common pattern for dealing with sequential async operations because it works better than the more obvious approach of nesting. Nesting means to call the next async API within the completed handler of the previous one, fulfilling each with `done`. For example (extraneous code removed for simplicity):

```
//Nested async operations, using done with each promise
captureUI.captureFileAsync(Windows.Media.Capture.CameraCaptureUIMode.photo)
    .done(function (capturedFileTemp) {
        //...
        local.createFolderAsync("HereMyAm", ...)
            .done(function (myFolder) {
                //...
                capturedFile.copyAsync(myFolder, newName)
                    .done(function (newFile) {
                    })
            })
      });
```

The one advantage to this approach is that each completed handler will have access to all the variables declared before it. Yet the disadvantages begin to pile up. For one, there is usually enough intervening code between the async calls that the overall structure becomes visually messy. More significantly, error handling becomes much more difficult. When promises are nested, error handling must be done at each level with distinct handlers; if you throw an exception at the innermost level, for instance, it won't be picked up by any of the outer error handlers. Each promise thus needs its own error handler, making real spaghetti of the basic code structure:

```
captureUI.captureFileAsync(Windows.Media.Capture.CameraCaptureUIMode.photo)
    .done(function (capturedFileTemp) {
        //...
        local.createFolderAsync("HereMyAm", ...)
            .done(function (myFolder) {
                //...
                capturedFile.copyAsync(myFolder, newName)
                    .done(function (newFile) {
                    },
                    function (error) {
                    })
            },
            function (error) {
            });
      },

  function (error) {
  });
```

I don't know about you, but I really get lost in all the }'s and)'s (unless I try hard to remember my LISP class in college), and it's hard to see which error function applies to which async call. And just imagine throwing a few progress handlers in as well!

Chaining promises solves all of this with the small tradeoff of needing to declare a few extra temp variables outside the chain for any variables that need to be shared amongst the various completed handlers. Each completed handler in the chain again returns the promise for the next operation, and each link is a call to `then` except for a final call to `done` to terminate the chain. This allows you to indent all the async calls only once, and it has the effect of propagating errors down the chain, as any intermediate promise that's in the error state will be passed through to the end of the chain very quickly. This allows you to have only a single error handler at the end:

```
captureUI.captureFileAsync(Windows.Media.Capture.CameraCaptureUIMode.photo)
   .then(function (capturedFileTemp) {
      //...
      return local.createFolderAsync("HereMyAm", ...);
   })
   .then(function (myFolder) {
      //...
      return capturedFile.copyAsync(myFolder, newName);
   })
   .done(function (newFile) {
   },
   function (error) {
   })
```

To my eyes (and my aging brain), this is a much cleaner code structure—and it's therefore easier to debug and maintain. If you like, you can even end the chain with `done(null, errorHandler)`, as we did in Chapter 2:

```
captureUI.captureFileAsync(Windows.Media.Capture.CameraCaptureUIMode.photo)
   //...
   .then(function (newFile) {
   })
   .done(null, function (error) {
   })
})
```

Remember, though, that if you need to pass a promise for the whole chain elsewhere, as to a `setPromise` method, you'll use `then` throughout.

Error Handling in Promise Chains: then vs. done

This brings us to why we have both `then` and `done` and to why `done` is used at the end of a chain as well as for single async operations. To begin with, `then` returns another promise, thereby allowing chaining, whereas `done` returns `undefined`, so it always occurs at the end of a chain. Second, if an exception occurs within one async operation's `then` method and there's no error handler at that level, the error gets stored in the promise returned by `then` (that is, the returned promise is in the error state). In contrast, if `done` sees an exception and there's no error handler, it throws that exception to the app's event loop. This will bypass any local (synchronous) `try/catch` block, though you can pick them up in either in `WinJS.Application.onerror` or `window.onerror` handlers. (The latter will get the error if the former doesn't handle it.) If you don't have an app-level handler, the app will be terminated and an error report sent to the Windows Store dashboard. For that reason we recommend that you implement an app-level error handler using one of the events above.

In practical terms, then, this means that if you end a chain of promises with a `then` and not `done`, all exceptions in that chain will get swallowed and you'll never know there was a problem! This can place an app in an indeterminate state and cause much larger problems later on. So, unless you're going to pass the last promise in a chain to another piece of code that will itself call `done` (as you do, for example, when using a `setPromise` deferral or if you're writing a library from which you return promises), always use `done` at the end of a chain even for a single async operation.[26]

Promise error events If you look carefully at the `WinJS.Promise` documentation, you'll see that it has an `error` event along with `addEventListener`, `removeEventListener`, and `dispatchEvent` methods. This is primarily used within WinJS itself and is fired on exceptions (but *not* cancellation). Promises from async WinRT APIs, however, do not fire this event, so apps typically use error handlers passed to `then/done` for this purpose.

Managing the UI Thread with the WinJS Scheduler

JavaScript, as you are probably well aware, is a single-threaded execution environment, where any and all of your code apart from web workers and background tasks run on what we call the *UI thread*. The internal working of asynchronous APIs, like those of WinRT, happen on other threads as well, and the internal engines of the app host are also very much optimized for parallel processing.[27] But regardless of how much work you offload to other threads, there's one very important characteristic to always keep in mind:

> *The results from all non-UI threads eventually get passed back to the app on the main UI thread through callback functions such as the completed handler given to a promise.*

Think about this very clearly: if you make a whole bunch of async WinRT calls within a short amount of time, such as to make HTTP requests or retrieve information from files, those tasks will execute on separate threads but each one will pass their results back to the UI thread when the task is complete. What this means is that the UI thread can become quite overloaded with such incoming traffic! Furthermore, what you do (or what WinJS does on your behalf) in response to the completion of each operation—such as adding elements to the DOM or innocently changing a simple layout-affecting style—can trigger more work on the UI thread, all of which competes for CPU time. As a result, your UI can become sluggish and unresponsive, the very opposite of "fast and fluid"!

[26] Some samples in the Windows SDK might still use `then` instead of `done`, especially for single async operations. This came from the fact that `done` didn't yet exist at one point and not all samples have been updated.

[27] In Windows 8 and Internet Explorer 10, most parsing, JavaScript execution, layout, and rendering on a single thread. Rewriting these processes to happen in parallel is one of the major performance improvements for Windows 8.1 and Internet Explorer 11, from which apps also benefit.

This is something we certainly saw with JavaScript apps on Windows 8, and developers created a number of strategies to cope with it such as starting async operations in timed batches to manage their rate of callbacks to the UI thread, and batching together work that triggers a layout pass so as to combine multiple changes in each pass.

Still, after plenty of performance analysis, the WinJS and app host teams at Microsoft found that what was really needed is a way to asynchronously prioritize different tasks *on the UI thread itself*. This meant creating some low-level scheduling APIs in the app host such as `MSApp.executeAtPriority`. But don't use such methods directly—use the `WinJS.Utilities.Scheduler` API instead. The reason for this is that WinJS very carefully manages its own tasks through the `Scheduler`, so by using it yourself you ensure that all the combined work is properly coordinated. This API also provides a simpler interface to the whole process, especially where promises are concerned.

Let's first understand what the different priorities are, then we'll see how to schedule and manage work at those priorities. Keep in mind, though, that using the scheduler is not at all required—it's there to help you tune the performance of your app, not to make your life difficult!

Scheduler Priorities

The relative priorities for the WinJS `Scheduler` are expressed in the `Scheduler.Priority` enumeration, which I list here in descending order: `max`, `high`, `aboveNormal`, `normal` (the default for app code), `belowNormal`, `idle`, and `min`. Here's the general guidance on how to use these:

Priority	Best Usage
`max`, `high`	Use sparingly for truly high priority work as these priorities take priority over layout passes in the rendering engine. If you overuse these priorities, the app can actually become *less* responsive!
`aboveNormal`, `normal`, `belowNormal`	Use these to indicate the relative importance between most of your tasks.
`idle`, `min`	Use for long-running and/or maintenance tasks where there isn't a UI dependency.

Although you need not use the scheduler in your own code, a little analysis of your use of async operations will likely reveal places where setting priorities might make a big difference. Earlier in "Optimizing Startup Time," for example, we talked about how you want to prioritize non-UI work while your splash screen is visible, because the splash screen is noninteractive by definition. If you're doing some initial HTTP requests, for example, set the most critical ones for your home page to `max` or `high`, and set secondary requests to `belowNormal`. This will help those first requests get processed ahead of UI rendering, whereas your handling of the secondary requests will then happen after your home page has come up. This way you won't make the user wait for completion of those secondary tasks before the app becomes interactive. Other requests that you want to start, perhaps to cache data for a secondary leaderboard page, can be set to `belowNormal` or `idle`. Of course, if the user navigates to a secondary page, you'll want to change its task priorities to `aboveNormal` or `high`.

WinJS, for its part, makes extensive use of priorities. For example, it will batch edits to a data-binding source at `high` priority while scheduling cleanup tasks at `idle` priority. In a complex control like the ListView, fetching new items that are necessary to render the visible part of a ListView control is done at max, rendering of the visible items is done at aboveNormal, pre-loading the next page of items forward is set to normal (anticipating that the user will pan ahead), and pre-loading of the previous page (to anticipate a reverse pan) is set to belowNormal.

Scheduling and Managing Tasks

Now that we know about scheduling priorities, the way to asynchronously execute code on the UI thread at a particular priority is by calling the Scheduler.schedule method (whose default priority is normal). This method allows you to provide an optional object to use as this inside the function along with a name to use for logging and diagnostics.[28]

As a simple example, scenario 1 of the HTML Scheduler sample schedules a bunch of functions at different priorities in a somewhat random order (js/schedulesjobscenario.js):

```
window.output("\nScheduling Jobs...");
var S = WinJS.Utilities.Scheduler;

S.schedule(function () { window.output("Running job at aboveNormal priority"); },
    S.Priority.aboveNormal);
window.output("Scheduled job at aboveNormal priority");

S.schedule(function () { window.output("Running job at idle priority"); },
    S.Priority.idle, this);
window.output("Scheduled job at idle priority");

S.schedule(function () { window.output("Running job at belowNormal priority"); },
    S.Priority.belowNormal);
window.output("Scheduled job at belowNormal priority");

S.schedule(function () { window.output("Running job at normal priority"); }, S.Priority.normal);
window.output("Scheduled job at normal priority");

S.schedule(function () { window.output("Running job at high priority"); }, S.Priority.high);
window.output("Scheduled job at high priority");

window.output("Finished Scheduling Jobs\n");
```

The output then shows that the "jobs," as they're called, which execute in the expected order:

```
Scheduling Jobs...
Scheduled job at aboveNormalPriority
Scheduled job at idlePriority
Scheduled job at belowNormalPriority
Scheduled job at normalPriority
Scheduled job at highPriority
Finished Scheduling Jobs
Running job at high priority
```

[28] The Scheduler.execHigh method is also a shortcut for directly calling MSApp.execAtPriority with Priority.high. This method does not accommodate any added arguments.

```
Running job at aboveNormal priority
Running job at normal priority
Running job at belowNormal priority
Running job at idle priority
```

No surprises here, I hope!

When you call `schedule`, what you get back is an object with the `Scheduler.IJob` interface, which defines the following methods and properties:

Properties	Description
`id`	(read-only) A unique id assigned by the scheduler.
`name`	(read-write) The app-provided name assigned to the job, if any. The name argument to `schedule` will be stored here.
`priority`	(read-write) The priority assigned through `schedule`; setting this property will change the priority.
`completed`	(read-only) A Boolean indicating whether the job has completed (that is, the function given to `schedule` has returned and all its dependent async operations are complete).
`owner`	(read-write) An owner token that can be used to group jobs. This is `undefined` by default.
Methods	**Description**
`pause`	Halts further execution of the job.
`resume`	Resumes a previously paused job (no effect if the job isn't paused).
`cancel`	Removes the job from the scheduler.

In practice, if you've scheduled a job at a low priority but navigate to a page that really needs that job to complete before the page is rendered, you simply bump up its `priority` property (and then drain the scheduler as we'll see in a moment). Similarly, if you scheduled some work on a page that you don't need to continue when navigating away, then call the job's `cancel` method within the page's `unload` method. Or perhaps you have an index page from which you typically navigate into a details page, and then back again. In this case you can `pause` any jobs on the index page when navigating to the details, then `resume` them when you return to the index. See scenarios 2 and 3 of the sample for some demonstrations.

Scenario 2 also shows the utility of the `owner` property (the code is thoroughly mundane so I'll leave you to examine it). An owner token is something created through `Scheduler.createOwnerToken` and then assigned to a job's `owner` (which replaces any previous owner). An owner token is simply an object with a single method called `cancelAll` that calls the `cancel` method of whatever jobs are assigned to it, nothing more. It's a simple mechanism—the owner token really does nothing more than maintain an array of jobs—but clearly allows you to group related jobs together and cancel them with a single call. This way you don't need to maintain your own lists and iterate through them for this purpose. (To do the same for pause and resume you can, of course, just duplicate the pattern in your own code.)

The other important feature of the Scheduler is the `requestDrain` method. This ensures that all jobs scheduled at a given priority or higher are executed before the UI thread yields. You typically use this to guarantee that high priority jobs are completed before a layout pass. `requestDrain` returns a promise that is fulfilled when the jobs are drained, at which time you can drain lower priority tasks or schedule new ones.

A simple demonstration is shown in scenario 5 of the sample. It has two buttons that schedule the same set of varying jobs and then call `requestDrain` with either `high` or `belowNormal` priority. When the returned promise completes, it outputs a message to that effect (js/drainingscenario.js):

```
S.requestDrain(priority).done(function () {
    window.output("Done draining");
});
```

Comparing the output of these two side by side (`high` on the left, `belowNormal` on the right), as below, you can see that the promise is fulfilled at different points depending on the priority:

```
Draining scheduler to high priority
Running job2 at high priority
Done draining
Running job1 at normal priority
Running job5 at normal priority
Running job4 at belowNormal priority
Running job3 at idle priority
```

```
Draining scheduler to belowNormal priority
Running job2 at high priority
Running job1 at normal priority
Running job5 at normal priority
Running job4 at belowNormal priority
Done draining
Running job3 at idle priority
```

The other method that exists on the Scheduler is `retrieveState`, a diagnostic aid that returns a descriptive string for current jobs and drain requests. Adding a call to this in scenario 5 of the sample just after the call to `requestDrain` will return the following string:

```
Jobs:
    id: 28, priority: high
    id: 27, priority: normal
    id: 31, priority: normal
    id: 30, priority: belowNormal
    id: 29, priority: idle
Drain requests:
    *priority: high, name: Drain Request 0
```

Setting Priority in Promise Chains

Let's say you have a set of async data-retrieval methods that you want to execute in a sequence as follows, processing their results at each step:

```
getCriticalDataAsync().then(function (results1) {
    var secondaryPages = processCriticalData(results1);
    return getSecondaryDataAsync(secondaryPages);
}).then(function (results2) {
    var itemsToCache = processSecondaryData(results2);
    return getBackgroundCacheDataAsync(itemsToCache);
}).done(function (results3) {
    populateCache(results3);
});
```

By default, all of this would run at the current priority against everything else happening on the UI thread. But you probably want the call to `processCriticalData` to run at a `high` priority, processSecondaryData to run at normal, and populateCache to run at idle. With schedule by itself, you'd have to do everything the hard way:

```
var S = WinJS.Utilities.Scheduler;

getCriticalDataAsync().done(function (results1) {
    S.schedule(function () {
        var secondaryPages = processCriticalData(results1);
        S.schedule(function () {
            getSecondaryDataAsync(secondaryPages).done(function (results2) {
                var itemsToCache = processSecondaryData(results2);
                S.schedule(function () {
                    getBackgroundCacheDataAsync(itemsToCache).done(function (results3) {
                        populateCache(results3);
                    });
                }, S.Priority.idle);
            });
        }, S.Priority.normal);
    }, S.Priority.high);
});
```

Urg. Blech. Ick. It's more fun going to the dentist than writing code like this! To simplify matters, you could encapsulate the process of setting a new priority within another promise that you can then insert into the chain. The best way to do this is to dynamically generate a completed handler that would take the results from the previous step in the chain, schedule a new priority, and return a promise that delivers those same results (see Appendix A for the use of new WinJS.Promise):

```
function schedulePromise(priority) {
    //This returned function is a completed handler.
    return function completedHandler (results) {
        //The completed handler returns another promise that's fulfilled
        //with the same results it received...
        return new WinJS.Promise(function initializer (c) {
            //But the delivery of those results are scheduled according to a priority.
            WinJS.Utilities.Scheduler.schedule(function () {
                c(results);
            }, priority);
        });
    }
}
```

Fortunately we don't have to write this code ourselves. The WinJS.Utilities.Scheduler already has five pre-made completed handlers like this that also automatically cancel a job if there is an error. These are called schedulePromiseHigh, schedulePromiseAboveNormal, schedulePromiseNormal, schedulePromiseBelowNormal, or schedulePromiseIdle.

Because these APIs are pre-made completed handlers rather than methods you call directly, simply insert the appropriate name at those points in a promise chain where you want to change the priority, as highlighted below:

```
var S = WinJS.Utilities.Scheduler;
```

```
getCriticalDataAsync().then(S.schedulePromiseHigh).then(function (results1) {
    var secondaryPages = processCriticalData(results1);
    return getSecondaryDataAsync(secondaryPages);
}).then(S.schedulePromise.normal).then(function (results2) {
    var itemsToCache = processSecondaryData(results2);
    return getBackgroundCacheDataAsync(itemsToCache);
}).then(S.schedulePromiseIdle).done(function (results3) {
    populateCache(results3);
});
```

Long-Running Tasks

All the jobs that we've seen so far are short-running in that we schedule a worker function at a certain priority and it just completes its work when it's called. However, some tasks might take much longer to complete, in which case you don't want to block higher priority work on your UI thread. To help with this, the scheduler has a built-in interval timer of sorts for tasks that are scheduled at aboveNormal priority or lower, so a task can check whether it should cooperatively yield and have itself rescheduled for its next bit of work. Let me stress that word *cooperatively*: nothing forces a task to yield, but because all of this is affecting the UI performance of your app and your app alone, if you don't play nicely you'll just be hurting yourself!

The mechanism for this is provided through a *job info* object that's passed as an argument to the worker function itself. To make sure we're clear on how this fits in, let's first look at everything a worker has available within its scope, which is best explained with a few comments within the basic code structure:

```
var job = WinJS.Utilities.Scheduler.schedule(function worker(jobInfo) {
    //jobInfo.job is the same as the job returned from schedule.
    //Scheduler.currentPriority will match the second argument to schedule.
    //this will be the third argument passed to schedule.
}, S.Priority.idle, this);
```

The members of the *jobInfo* object are defined by Scheduler.IJobInfo:

Properties	Description
job	(read-only) The same job object as returned from schedule.
shouldYield	(read-only) A Boolean flag that is typically false when the worker is first called and then changes to true if the worker should yield the UI thread and reschedule its work.
Methods	**Description**
setWork	Provides the worker for the rescheduled task.
setPromise	Provides a promise that the scheduler will wait upon before rescheduling the task, where the worker to reschedule is the fulfillment value of the promise.

Scenario 4 of the HTML Scheduler sample shows how to work with these. When you press the Execute a Yielding Task button, it schedules a function called *worker* at idle priority that just spins within itself until you press the Complete Yielding Task button, which sets the taskCompleted flag below to true (js/yieldingscenario.js, with the 2s interval changed to 200ms):

```
S.schedule(function worker(jobInfo) {
    while (!taskCompleted) {
        if (jobInfo.shouldYield) {
            // not finished, run this function again
            window.output("Yielding and putting idle job back on scheduler.");
            jobInfo.setWork(worker);
            break;
        }
        else {
            window.output("Running idle yielding job...");
            var start = performance.now();
            while (performance.now() < (start + 200)) {
                // do nothing;
            }
        }
    }

    if (taskCompleted) {
        window.output("Completed yielding task.");
        taskCompleted = false;
    }
}, S.Priority.idle);
```

Provided that the task is active, it does 200ms of work and then checks if `shouldYield` has changed to `true`. If so, the worker calls `setWork` to reschedule itself (or another function if it wants). You can trigger this while the idle worker is running by pressing the Add Higher Priority Tasks to Queue button in the sample. You'll then see how those tasks are run before the next call to the worker. In addition, you can poke around elsewhere in the UI to observe that the idle task is not blocking the UI thread.

Note here that the worker function checks `shouldYield` first thing to immediately yield if necessary. However, it's perfectly fine to do a little work first and then check. Again, this is all about cooperating within your own app code, so such self-throttling is your choice.

As for `setPromise`, this is slightly tricky. Calling `setPromise` tells the scheduler to wait until that promise is fulfilled before rescheduling the task, where the next worker function for the task is provided directly through the promise's fulfillment value. (As such, `IJobInfo.setPromise` doesn't pertain to handling async operations like other `setPromise` methods in WinJS that are tied in with WinRT deferrals. If you called `IJobInfo.setPromise` with a promise from some random async API, the scheduler would attempt to use the fulfillment value of that operation—which could be anything—as a function and thus likely throw an exception.)

In short, whereas `setWork` says "go ahead and reschedule with this worker," `setPromise` says "hold off rescheduling until I deliver the worker sometime later." This is primarily useful to create a work queue composed of multiple jobs with an ongoing task to process that queue. To illustrate, consider the following code for such an arrangement:

```
var workQueue = [];

function addToQueue(worker) {
    workQueue.push(worker);
}

S.schedule(function processQueue(jobInfo) {
    while (work.length) {
        if (jobInfo.shouldYield) {
            jobInfo.setWork(processQueue);
            return;
        }
        work.shift()();  //Pull the first from the FIFO queue and call it.
    }
}}, S.Priority.belowNormal);
```

Assuming that there are some jobs in the queue when you first call schedule, the *processQueue* task will cooperatively empty that queue. And if new jobs are added to the queue in the meantime, *processQueue* will continue to be rescheduled.

The problem, however, is that the *processQueue* worker will finish and exit as soon as the queue is empty, meaning that any jobs you add to the queue later on won't be processed. To fix this you could just have *processQueue* repeatedly call `setWork` on itself again and again even when the queue is empty, but that would be wasteful. Instead, you can use `setPromise` to have the scheduler wait until there is more work in the queue. Here's how that would work:

```
var workQueue = [];
var haveWork = function () { };  //This function is just a placeholder

function addToQueue(worker) {
    workQueue.push(worker);
    haveWork();
}

S.schedule(function processQueue(jobInfo) {
    while (work.length) {
        if (jobInfo.shouldYield) {
            jobInfo.setWork(processQueue);
            return;
        }
        work.shift()();  //Pull the first from the FIFO queue and call it.
    }

    //If we reach here the queue is empty, but we don't want to exit the worker.
    //Instead of calling setWork without work to do, create a promise that's fulfilled
    //when addToQueue is called again, which we do by replacing the haveWork function
    //with one that calls the promise's completed handler.
    jobInfo.setPromise(new WinJS.Promise(function (completeDispatcher) {
        haveWork = function () { completeDispatcher(processQueue) };
    }))
});
```

With this code, say we populate `workQueue` with a number of jobs and then make the call to `schedule`. Up to this point and so long as the queue doesn't become empty, we stay inside the `while` loop of *processQueue*. Any call to the empty `haveWork` function so far is just a no-op.

If the queue becomes empty, however, we'll exit the while loop but we don't want *processQueue* to exit. Instead, we want to tell the scheduler to wait until more work is added to the queue. This is why we have that placeholder function for haveWork, because we can now replace it with a function that will complete the promise with *processQueue*, thereby triggering a rescheduling of that worker function.

Note that an alternate way to accomplish the same goal is to use this assignment for haveWork:

```
haveWork = completeDispatcher.bind(null, processQueue);
```

This accomplishes the same result as an anonymous function and avoids creating a closure.

Debugging and Profiling

As we've been exploring the core anatomy of an app in this chapter along with performance, now's a good time to talk about debugging and profiling. This means, as I like to put it, becoming a doctor of internal medicine for your app and learning to diagnose how well that anatomy is working.

Tip Debug logging, which is local to and only relevant on your development machine, is a very different concern from telemetry logging, with which you monitor and record user activity. See "Instrumenting Your App for Telemetry and Analytics" in Chapter 20.

Debug or release? Because JavaScript is not a compiled language, it lacks conditional compilation directives like #ifdef in C#/C++. There are, however, a few ways to more or less make this determination at run time (with some caveats). See "Sidebar: Debug or Release?" in Chapter 2.

Debug Output and Logging

It's sometimes heartbreaking to developers that window.prompt and window.alert are not available to Windows Store apps as quickie debugging aids. Fortunately, you have two other good options for that purpose. One is Windows.UI.Popups.MessageDialog, which is actually what you use for real user prompts in general (see Chapter 9). The other is console.log, as we've used in our code already, which sends text to Visual Studio's output pane. These messages can also be logged as Windows events, as we'll see shortly.

For readers who are seriously into logging, beyond the kind you do with chainsaws, there are two other options: a more flexible method in WinJS called WinJS.log, and the logging APIs in Windows.Foundation.Diagnostics.

WinJS.log is a curious beast because although it's ostensibly part of the WinJS namespace, it's actually not implemented within WinJS itself! At the same time, it's used all over the place in the library for errors and other reporting. For instance:

```
WinJS.log && WinJS.log(safeSerialize(e), "winjs", "error");
```

This kind of JavaScript syntax, by the way, means "check whether `WinJS.log` exists and, if so, call it." The `&&` is a shortcut for an `if` statement: the JavaScript engine will not execute the part after the `&&` if the first part is `null`, `undefined`, or `false`. It's a very convenient bit of concise syntax.

Anyway, the purpose of `WinJS.log` is to allow you to implement your own logging function and have it pick up WinJS's logging as well as any you add to your own code. What's more, you can turn the logging on and off at any time, something that's not possible with `console.log` unless, well, you write a wrapper like `WinJS.log`!

Your `WinJS.log` function, as described in the documentation, should accept three parameters:

1. The message to log (a string).
2. A string with a tag or tags to categorize the message. WinJS always uses "winjs" and sometimes adds an additional tag like "binding", in which case the second parameter is "winjs binding". I typically use "app" in my own code.
3. A string describing the type of the message. WinJS will use "error", "info", "warn", and "perf".

Conveniently, WinJS offers a basic implementation of this which you set up by calling `WinJS.Utilities.startLog()`. This assigns a function to `WinJS.log` that uses `WinJS.Utilities.formatLog` to produce decent-looking output to the console. What's very useful is that you can pass a list of tags (in a single string) to `startLog` and only those messages with those tags will show up. Multiple calls to `startLog` will aggregate those tags. Then you can call `WinJS.Utilities.stopLog` to turn everything off and start again if desired (`stopLog` is not made to remove individual tags). As a simple example, see the HereMyAm3d example in the companion content.

Tip Although logging will be ignored for released apps that customers will acquire from the Store, it's a good idea to comment out your one call to `startLog` before submitting a package to the Store and thus avoid making any unnecessary calls at run time.

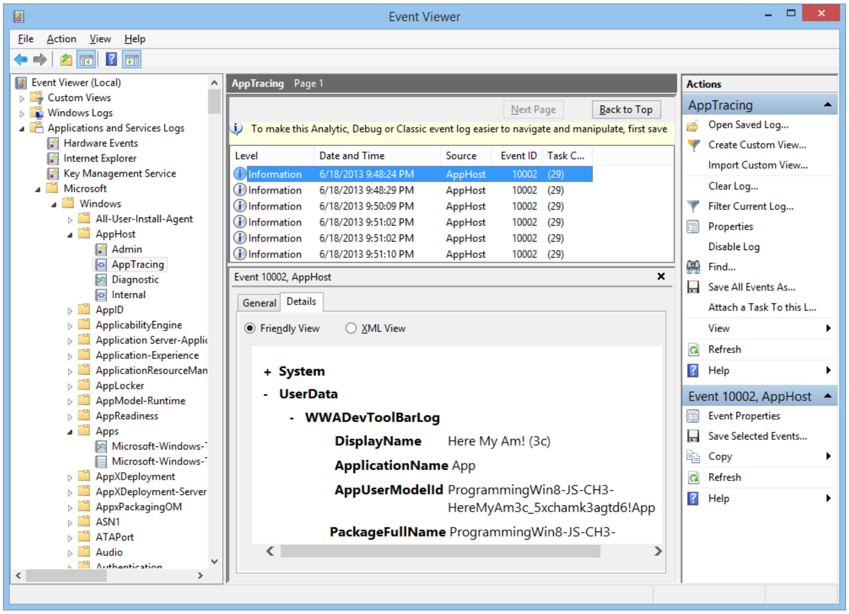
`WinJS.log` is highly useful for generating textual logs, but if you want to go much deeper you'll want to use the WinRT APIs in `Windows.Foundation.Diagnostics`, namely the `LoggingSession` and `FileLoggingSession` classes. These work with in-memory and continuous file-based logging, respectively, and generate binary "Event Trace Log" (ETL) data that can be further analyzed with the Windows Performance Analyzer (wpa.exe) and the Trace Reporter (tracerpt.exe) tools in the Windows SDK. This is a subject well beyond the scope of this book (and this author's experience), so refer to the Windows Performance Analyzer documentation for more, along with the LoggingSession sample and FileLoggingSession sample.

Error Reports and the Event Viewer

Similar to `window.alert`, another DOM API function to which you might be accustomed is `window.close`. You can still use this as a development tool, but in released apps Windows interprets this call as a crash and generates an error report in response. This report will appear in the Store dashboard for your app, with a message telling you to not use it! Generally, Store apps should not provide their own close affordances.

There might be situations, however, when a released app absolutely needs to close itself in response to unrecoverable conditions. Although you can use `window.close` for this, it's better to use `MSApp.terminateApp` because it allows you to also include information as to the exact nature of the error. These details show up in the Store dashboard, making it easier to diagnose the problem.

In addition to the Store dashboard, you should make fast friends with the Windows Event Viewer.[29] This is where error reports, console logging, and unhandled exceptions (which again terminate the app without warning) can be recorded. To enable this, start Event Viewer, navigate to Application And Services Logs on the left side (after waiting for a minute while the tool initializes itself), and then expand Microsoft > Windows > AppHost. Then left-click to *select* Admin (this is important), right-click Admin, and select View > Show Analytic And Debug Logs. This turns on full output, including tracing for errors and exceptions, as shown in Figure 3-5. Then right-click AppTracing (also under AppHost) and select Enable Log. This will trace any calls to `console.log` as well as other diagnostic information coming from the app host.



To make this Analytic, Debug or Classic event log easier to navigate and manipulate, first save

Level	Date and Time	Source	Event ID	Task C...
Information	6/18/2013 9:48:24 PM	AppHost	10002	(29)
Information	6/18/2013 9:48:29 PM	AppHost	10002	(29)
Information	6/18/2013 9:50:09 PM	AppHost	10002	(29)
Information	6/18/2013 9:51:02 PM	AppHost	10002	(29)
Information	6/18/2013 9:51:02 PM	AppHost	10002	(29)
Information	6/18/2013 9:51:10 PM	AppHost	10002	(29)

● Friendly View ○ XML View

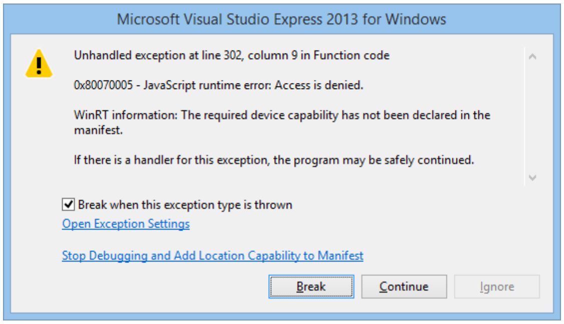
+ **System**
- **UserData**
  - **WWADevToolBarLog**
    DisplayName Here My Am! (3c)
    ApplicationName App
    AppUserModelId ProgrammingWin8-JS-CH3-HereMyAm3c_5xchamk3agtd6!App
    PackageFullName ProgrammingWin8-JS-CH3-

FIGURE 3-5 App host events, such as unhandled exceptions, load errors, and logging can be found in Event Viewer.

We already introduced Visual Studio's Exceptions dialog in Chapter 2; refer back to Figure 2-16. For each type of JavaScript exception, this dialog supplies two checkboxes labeled Thrown and User-

[29] If you can't find Event Viewer, press the Windows key to go to the Start screen and then invoke the Settings charm. Select Tiles, and turn on Show Administrative Tools. You'll then see a tile for Event Viewer on your Start screen.

unhandled. Checking Thrown will display a dialog box in the debugger (see Figure 3-6) whenever an exception is thrown, regardless of whether it's handled and before reaching any of your error handlers.


Microsoft Visual Studio Express 2013 for Windows

Unhandled exception at line 302, column 9 in Function code

0x80070005 - JavaScript runtime error: Access is denied.

WinRT information: The required device capability has not been declared in the manifest.

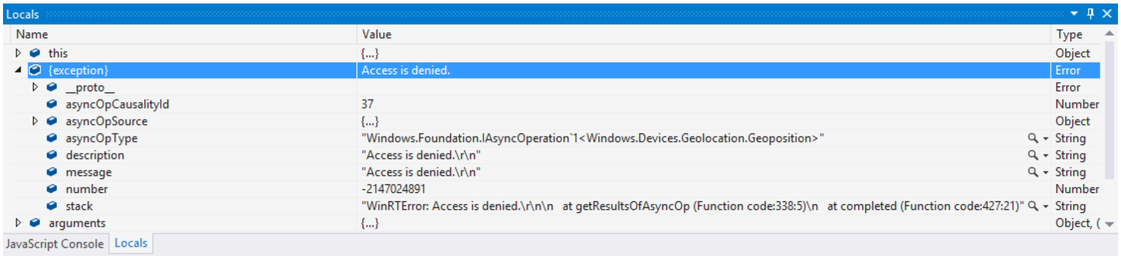
If there is a handler for this exception, the program may be safely continued.

☑ Break when this exception type is thrown

Open Exception Settings
Stop Debugging and Add Location Capability to Manifest
Break
Continue
Ignore

FIGURE 3-6 Visual Studio's exception dialog. As the dialog indicates, it's safe to press Continue if you have an error handler in the app; otherwise the app will terminate. Note that the checkbox in this dialog is a shortcut to toggle the Thrown checkbox for this exception type in the Exceptions dialog.

If you have error handlers in place, you can safely click the Continue button in the dialog of Figure 3-6 and you'll eventually see the exception surface in those error handlers. (Otherwise the app will terminate; see below.) If you click Break instead, you can find the exception details in the debugger's Locals pane, as shown in Figure 3-7.



Name	Value	Type
▷ this	{...}	Object
◢ {exception}	Access is denied.	Error
▷ __proto__		Error
asyncOpCausalityId	37	Number
▷ asyncOpSource	{...}	Object
asyncOpType	"Windows.Foundation.IAsyncOperation`1<Windows.Devices.Geolocation.Geoposition>"	String
description	"Access is denied.\r\n"	String
message	"Access is denied.\r\n"	String
number	-2147024891	Number
stack	"WinRTError: Access is denied.\r\n\n at getResultsOfAsyncOp (Function code:338:5)\n at completed (Function code:427:21)"	String
▷ arguments	{...}	Object, (

FIGURE 3-7 Information in Visual Studio's Locals pane when you break on an exception.

The User-unhandled option (enabled for all exceptions by default) will display a similar dialog whenever an exception is thrown to the event loop, indicating that it wasn't handled by an app-provided error function ("user" code from the system's perspective).

You typically turn on Thrown for only those exceptions you care about; turning them all on can make it very difficult to step through your app! But it's especially helpful if you're debugging an app and end up at the `debugger` line in the following bit of WinJS code, just before the app is terminated:

```
var terminateAppHandler = function (data, e) {
    debugger;
    MSApp.terminateApp(data);
};
```

If you turn on Thrown for all JavaScript exceptions, you'll then see exactly where the exception occurred. You can also just check Thrown for only those exceptions you expect to catch.

Do leave User-unhandled checked for everything else. In fact, unless you have a specific reason not to, make sure that User-unhandled is checked next to the topmost JavaScript Runtime Exceptions item because this includes all exceptions not otherwise listed. This way you can catch (and fix) exceptions that might abruptly terminate the app, which is something your customers should never experience.

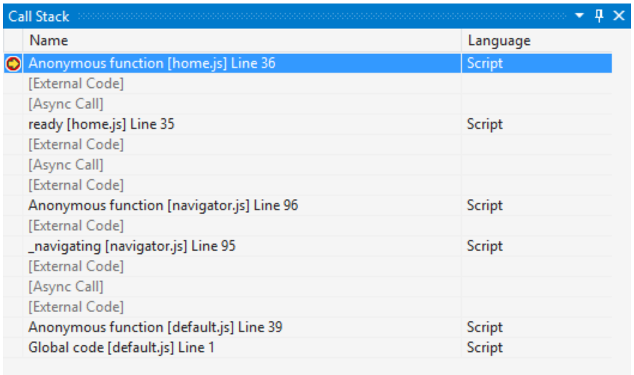
> **WinJS.validation** Speaking of exceptions, if you set `WinJS.validation` to `true` in your app, you'll instruct WinJS to perform a few extra checks on arguments and internal state, and throw exceptions if something is amiss. Just search on "validation" in the WinJS source files for where it's used.

Async Debugging

Working with asynchronous APIs presents a challenge where debugging is concerned. Although we have a means to sequence async operations with promise chains (or nested calls, for that matter), each step in the sequence involves an async call, so you can't just step through as you would with synchronous code. If you try this, you'll step through lots of promise code (in WinJS or the JavaScript projection layer for WinRT) rather than your completed handlers, which isn't particularly helpful.

What you'll need to do instead is set a breakpoint on the first line of each completed handler and on the first line of each error function. As each breakpoint is hit, you can step through that handler. When you reach the next async call in a completed handler, click the Continue button in Visual Studio so that the async operation can run. After that you'll hit the breakpoint in the next completed handler or the breakpoint in the error handler.

When you stop at a breakpoint, or when you hit an exception within an async process, take a look at the debugger's Call Stack pane (typically in the lower right of Visual Studio), as shown here:

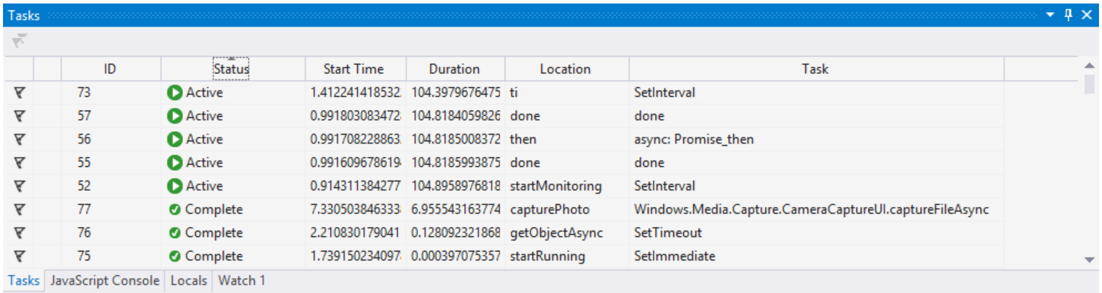


Name	Language
Anonymous function [home.js] Line 36	Script
[External Code]	
[Async Call]	
ready [home.js] Line 35	Script
[External Code]	
[Async Call]	
[External Code]	
Anonymous function [navigator.js] Line 96	Script
[External Code]	
_navigating [navigator.js] Line 95	Script
[External Code]	
[Async Call]	
[External Code]	
Anonymous function [default.js] Line 39	Script
Global code [default.js] Line 1	Script

The Call Stack shows you the sequence of functions that lead up to the point where the debugger stopped, at which point you can double-click any of the lines and examine that function's context. With async calls, this can get really messy with all the generic handlers and other chaining that happens within WinJS and the JavaScript projection layer. Fortunately—very fortunately!—Visual Studio spares you from all that. It condenses such code into the gray [Async Call] and [External Code] markers, leaving only a clear call chain for your app's code. In this example I set a breakpoint in the completed handler for geolocation in HereMyAm3d. That completed handler is an anonymous function, as the first line of the Call Stack indicates, but the next reference to the app code clearly shows that the real context is the `ready` method within home.js, which itself is part of a longer chain that originated in default.js. Double-clicking any one of the app code references will open that code in Visual Studio and update the Locals pane to that context.

The real utility of this comes when an exception occurs somewhere other than within you own handlers, because you can then easily trace the causality chain that led to that point.

The other feature for async debugging is the Tasks pane, as shown below. You turn this on through the Debug > Windows >Tasks menu command. You'll see a full list of active and completed async operations that are part of the current call stack.



	ID	Status	Start Time	Duration	Location	Task
	73	Active	1.412241418532	104.3979676475	ti	SetInterval
	57	Active	0.991803083472	104.8184059826	done	done
	56	Active	0.991708228863	104.8185008372	then	async: Promise_then
	55	Active	0.991609678619	104.8185993875	done	done
	52	Active	0.914311384277	104.8958976818	startMonitoring	SetInterval
	77	Complete	7.330503846333	6.955543163774	capturePhoto	Windows.Media.Capture.CameraCaptureUI.captureFileAsync
	76	Complete	2.210830179041	0.128092321868	getObjectAsync	SetTimeout
	75	Complete	1.739150234097	0.000397075357	startRunning	SetImmediate

Performance and Memory Analysis

Alongside its excellent debugging tools, Visual Studio also offers additional aids to help evaluate the performance of an app, analyze its memory usage, and otherwise discover and diagnose problems that affect the user experience and the app's effect on the system. To close this chapter, I wanted to give you a brief overview of what's available along with pointers to where you can learn more—because this subject could fill a book in itself! (In lieu of that, a general pointer is to filter the //build 2013 videos by the "performance" tag, which turns up a healthy set.)

For starters, the Writing efficient JavaScript topic is well worth a read (as are its siblings under Best practices using JavaScript), because it explains various things you should and should not do in your code to help the JavaScript engine run best. One thing you *shouldn't* worry about is the performance of `querySelector` and `getElementById`, both of which are highly optimized because they're used so often. Keep this in mind, because I know for myself that any function that starts with "query" just sounds like it's going to do a lot of work, but that's not true here.

Next, when thinking about performance, start by setting specific goals for your user experience, such as "the app should become interactive within 1.5 seconds" and "navigating between the gallery and details pages happens in 0.5 seconds or less." In fact, such goals should really be part of the app's design that you discuss with your designers, because they're just as essential to the overall user experience as static considerations like layout. In the end, performance is not about numbers but about creating a great user experience.

Establishing goals also helps you stay focused on what matters. You can measure all kinds of different performance metrics for an app, but if they aren't serving your real goals, you end up with a classic case of what Tom DeMarco, in his book *Why Does Software Cost So Much?* (Dorset House, 1995), calls "measurement dysfunction": lots of data with meaningless results or results that lead to undesired action.[30]

Along the same lines, when running analysis tools, it's important that you *exercise the app like a user would*. That way you get results that are meaningful to the real user experience—that is, the human experience!—rather than results that would be meaningful to a robot. In the end, all the performance analysis in the world won't be worth anything unless is translates into two things: better ratings and reviews in the Windows Store, and greater app revenue.

With your goals in mind, run analysis tools on a regular basis and evaluate the results against your goals. Then adjust your code, run the tools again, and evaluate. In other words, running performance tools to evaluate your performance goals is just another part of making sure you're creating the app according to its design—the static and dynamic parts alike.

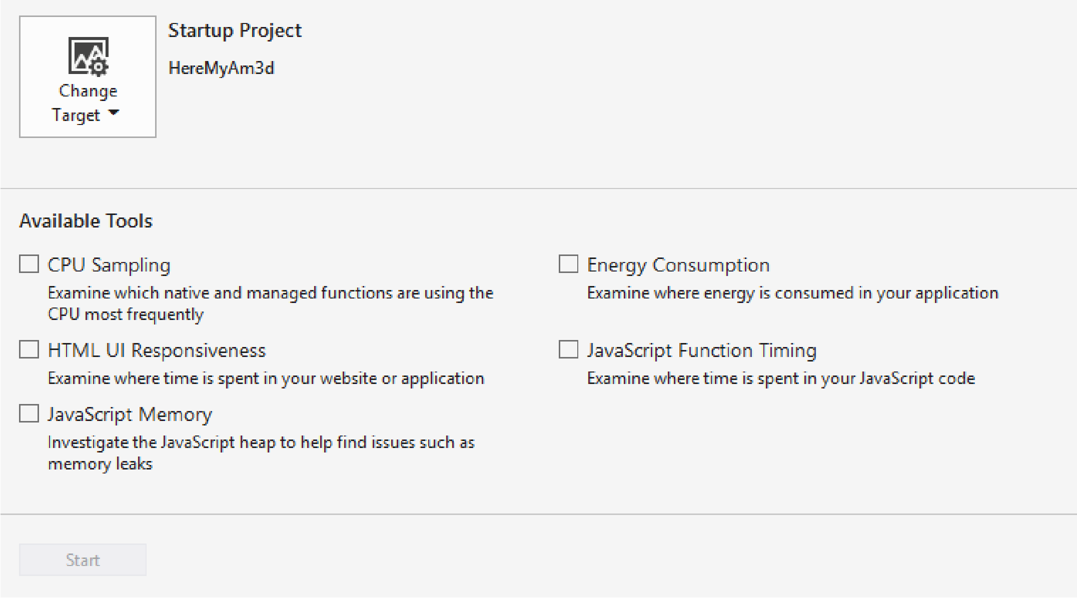
Remember also to *run performance analysis on a variety of hardware*, especially lower-end devices such as ARM tablets that are much more sensitive to performance issues than is your souped-up dev machine. In fact, slower devices are the ones you should be most concerned about, because their users will probably be the first to notice any issues and ding your app ratings accordingly. And yes, you can run the performance tools on a remote machine in the same way you can do remote debugging (but not in the simulator). Also be aware that analysis tools always run *outside* of the debugger for obvious reasons, because stopping at breakpoints and so forth would produce bad performance data!

I very much encourage you, then, to spend a few hours exercising the available tools and getting familiar with the information they provide. Make them a regular part of your coding/testing cycle so that you can catch performance and memory issues early on, when it's easier and less costly to fix them. Doing so will also catch what we call "regressions," where a later change to the code causes performance problems that you fixed a long time ago to rear their ugly heads once again. As the character Alistor Moody of the Harry Potter books says, "Constant vigilance!"

[30] DeMarco tells an amusing story of metrics at their worst: "Consider the case of the Soviet nail factory that was measured on the basis of the number of nails produced. The factory managers hit upon the idea of converting their entire factory to production of only the smallest nails, tiny brads. Some commissar, realizing this as a case of dysfunction, came up with a remedy. He instituted measurement of *tonnage* of nails produced, rather than numbers. The factory immediately switched over to producing only railroad spikes. The image I propose to mark the dysfunction end of the spectrum is a Soviet carpenter, looking perplexed, with a useless brad in one hand and an equally useless railroad spike in the other."

Tip Two topics in the documentation also contain loads of detailed information in these areas: Performance best practices for Windows Store apps using JavaScript and General best practices for performance.

So, on to the tools. These are found on the Debug > Performance And Diagnostics... menu, which brings up the hub shown below with tools that are appropriate to your project's language:



Change Target

Startup Project

HereMyAm3d

Available Tools

☐ CPU Sampling
Examine which native and managed functions are using the CPU most frequently

☐ HTML UI Responsiveness
Examine where time is spent in your website or application

☐ JavaScript Memory
Investigate the JavaScript heap to help find issues such as memory leaks

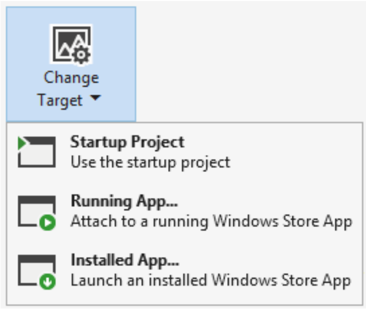
☐ Energy Consumption
Examine where energy is consumed in your application

☐ JavaScript Function Timing
Examine where time is spent in your JavaScript code

Start

Get Visual Studio updates New tools are often released with updates to Visual Studio, so be sure to install them and read the accompanying blogs or release notes to understand what's new.

By default, Visual Studio will set the target to be the currently loaded project. However, you can run the tools on any app by using the options on the Change Target drop-down:



Change Target

Startup Project
Use the startup project

Running App...
Attach to a running Windows Store App

Installed App...
Launch an installed Windows Store App

As the drop-down indicates, the Installed App option will launch an app anew, whereas the Running App option attaches to one that's already been launched. Both are essential for profiling apps on devices where your full project is not present; the latter is also useful if your app is already running and you want to analyze specific user interactions for a set of conditions that you've already set up. This way you won't collect a bunch of extra data that you don't need.

Note that you can run these tools on *any* installed app, not just your own, which means you can gather data from other apps that have the level of performance you'd like to achieve for yours.

The Performance and Diagnostic Hub as a whole is designed to be extensible with third-party tools, giving you a one-stop shop for enabling multiple tools simultaneously. The ones shown above are those built into Visual Studio, and be sure to install new Visual Studio updates because that's often how new tools are released.

Here's a quick overview of what the current tools accomplish:

Tool	**Description**
HTML UI Responsiveness	Provides a graph of Visual Throughput (frames per second) for the rendering engine over time, helping to identify places where your UI is not as responsive as you'd like. It also provides a millisecond breakdown of CPU utilization in various subsystems: loading, scripting, garbage collection, styling, rendering, and image decoding, with various important lifecycle events indicated along the way. This data is also shown on a time line where you can select any part to see the breakdown in more detail. All this is helpful for finding areas where the interactions between subsystems is adding lots of overhead, where there's excessive fragmentation, or where work being done in a particular subsystem is causing a drop in visual throughput. A walkthrough is on HTML UI Responsiveness tool in Visual Studio 2013 (MSDN blogs). Also see Analyze UI responsiveness.
Energy Consumption	Launches the app and collects data about power usage (in milliwatts) over time, split up by CPU, display, and network. This is very important to writing power-efficient apps for tablet devices. It can also help you determine whether it's more power efficient to use the local CPU or a network server for certain tasks, as network I/O can take as much and even more power than a burst of CPU activity. For more, see Energy Consumption tool in Visual Studio 2013.
JavaScript Memory	Launches the app and provides a dynamic graph of memory usage over time as well as the ability to take heap snapshots, allowing you to see memory spikes that occur in response to user activity, and whether that memory is being properly freed. Refer to JavaScript memory anaylsis for Windows Store apps in Visual Studio 2012 (MSDN blogs) and Analyzing memory usage in Windows Store apps.
JavaScript Function Timing (also called the JavaScript Profiler)	Displays data on when and where function calls are being made in JavaScript and how much time is spent in what part of your code. A walkthrough can be found on How to profile a JavaScript App for performance problems (MSDN blogs). Also see Analyizing JavaScript Performance in Windows Store apps, which covers both local and remote machines.
CPU Sampling	Similar to the JavaScript Function Timing tool but works for managed (C#/Visual Basic) and native (C++) code. This is useful only if you're writing a multi-language app with both JavaScript and one of the other languages.

For a video demonstration of most of these, watch the Visual Studio 2013 Performance and Diagnostics Hub video on Channel 9 and Diagnosing Issues in JavaScript Windows Store Apps with Visual Studio 2013 from the //build 2013 conference, both by Andrew Hall, the real expert on these matters. Note that everything you see in these video (with the exception of the console app profiler) is available in the Visual Studio Express edition that we've been using, and if you want to skip the part about XAML UI responsiveness in the first video, you can jump ahead to about 13:30 where he talks about the JavaScript tools.

> **Tip** In the first video, the responsiveness problems for the demo apps written both in XAML/C# and HTML/JavaScript primarily come from loading full image files just to generate thumbnails for gallery views. As the video mentions, you can avoid this entirely and achieve much better performance by using `Windows.Storage.StorageFile.getThumbnailAsync`. This API draws on thumbnail caches and other mechanisms to avoid the memory overhead and CPU cost of loading full image files. We'll see more of this in Chapter 11.

It's important, of course, with all these tools to clearly correlate certain events in the app with the various measurements. This is the purpose of the `performance.mark` function, which exists in the global JavaScript namespace.[31] Events written with this function appear as User Marks in the timelines generated by the different tools, as shown in Figure 3-8. In looking at the figure, note that the resolution of marks on the Memory Analyzer timeline on the scale of *seconds*, so use marks to indicate only significant user interaction events rather than every function entry and exit. (With other tools, however, the resolution is much finer, so you can use `performance.mark` more frequently.)

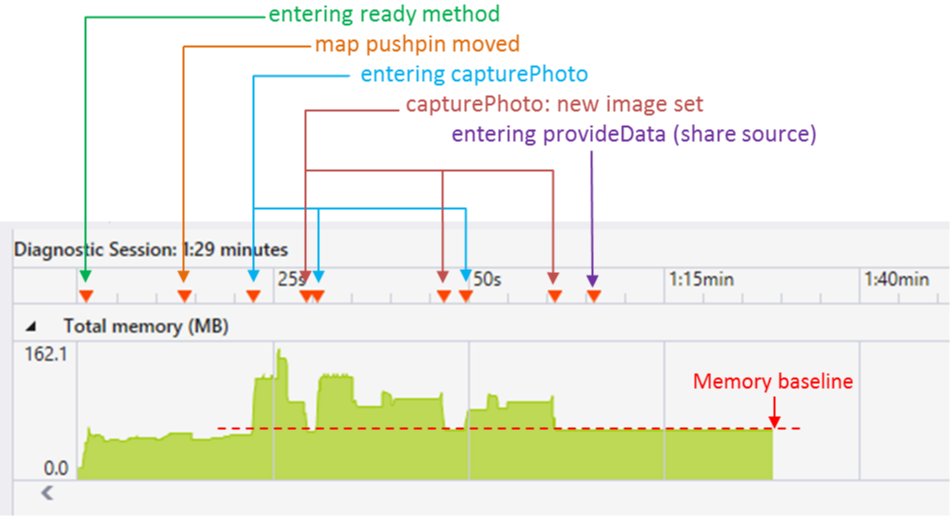


FIGURE 3-8 Output of the JavaScript Memory analyzer annotated with different marks. The red dashed line is also added in this figure to show the ongoing memory footprint; it is not part of the tool's output.

As one example of using these tools, let's run the Here My Am! app through the memory analyzer to see if we have any problems. We'll use the HereMyAm3d example in the companion code where I've

[31] This function is part of a larger group of methods on the `performance` object that reflect developing standards. For more details, see Timing and Performance APIs. `performance.mark` specifically replaces `msWriteProfilerMark`.

added some `performance.mark` calls for events like startup, capturing a new photo, rendering that photo, and exercising the Share charm. Figure 3-8 shows the results. For good measure—logging, actually!—I've also converted `console.log` calls to `WinJS.log`, where I've used a tag of "app" in each call and in the call to `WinJS.Utilities.startLog` (see default.js).

Referring to Figure 3-8, here's what I did after starting up the app in the memory analyzer. Once the home page was up (first mark), I repositioned the map and its pushpin (second mark), and you can see that this increased memory usage a little within the Bing maps control. Next I invoked the camera capture UI (third mark), which clearly increased memory use as expected. After taking a picture and displaying it in the app (fourth mark), you can see that the allocations from the camera capture UI have been released, and that we land at a baseline footprint that now includes a rendered image. I then do into the capture UI two more times, and in each case you can see the memory increase during the capture, but it comes back to our baseline each time we return to the main app. There might be some small differences in memory usage here depending on the size of the image, but clearly we're cleaning up the image when it get replaced. Finally I invoked the Share charm (last mark), and we can see that this causes no additional memory usage in the source app, which is expected because all the work is being done in the target. As a result, I feel confident that the app is managing its memory well. If, on the other hand, that baseline kept increasing over time, then I'd know I have a leak somewhere.

Tip There's no rule anywhere that says you have to profile your full app project. When you're trying to compare different implementation strategies, it can be much easier to create a simple test project and run the profiling tools on it so that you can obtain very focused comparisons for different approaches. Doing so will speed up your investigations and avoid disturbing your main project in the process.

The Windows App Certification Toolkit

The other tool you should run on a regular basis is the Windows App Certification Toolkit (WACK), which is actually one of the first tools that's automatically run on your app when you submit it to the Windows Store. If this toolkit reports failures on your local machine, you can be certain that you'll fail certification very early in the process.

Running the toolkit can be done as part of building an app package for upload, but until then, launch it from your Start screen (it's called Windows App Cert Kit). When it comes up, select Validate Windows Store App, which (after a disk-chewing delay) presents you with a list of installed apps, including those that you've been running from Visual Studio. It takes some time to generate that list if you have lots of apps installed, so you might use the opportunity to take a little stretching break. Then select the app you want to test, and take the opportunity to grab a snack, take a short walk, play a few songs on the guitar, or otherwise entertain yourself while the WACK gives your app a good whacking.

Eventually it'll have an XML report ready for you. After saving it (you have to tell it where), you can view the results. Note that for developer projects it will almost always report a failure on bytecode generation, saying "This package was deployed for development or authoring mode. Uninstall the package and reinstall it normally." To fix this, uninstall it from the Start menu, select a Release target in Visual Studio, and then use the Build > Deploy Solution menu command. But you can just ignore this particular error for now. Any other failure will be more important to address early on—such as crashes, hangs, and launch/suspend problems—rather than waiting until you're ready to submit to the Store.

Note Visual Studio also has a code analysis tool on the Build > Run Code Analysis On Solution menu, which examines source code for common defects and other violation of best practices. However, this tool does not presently work with JavaScript.

What We've Just Learned

- How apps are activated (brought into memory) and the events that occur along the way.
- The structure of app activation code, including activation kinds, previous execution states, and the `WinJS.UI.Application` object.
- Using deferrals when needing to perform async operations behind the splash screen, and optimizing startup time.
- How to handle important events that occur during an app's lifetime, such as focus events, visibility changes, view state changes, and suspend/resume/terminate.
- The basics of saving and restoring state to restart after being terminated, and the WinJS utilities for implementing this.
- How to implement page-to-page navigation within a single page context by using page controls, `WinJS.Navigation`, and the `PageControlNavigator` from the Visual Studio/Blend templates, such as the Navigation App template.
- Details of promises that are commonly used with, but not limited to, async operations.
- How to join parallel promises as well as execute a sequential async operations with chained promises.
- How exceptions are handled within chained promises and the differences between `then` and `done`.
- How to create promises for different purposes.
- Using the APIs in `WinJS.Utilities.Scheduler` for prioritizing work on the UI thread, including the helpers for prioritizing different parts of a promise chain.
- Methods for getting debug output and error reports for an app, within the debugger and the Windows Event Viewer.
- How to debug asynchronous code and how Visual Studio makes it easy to see the causality chain.
- The different performance and memory analysis tools available in Visual Studio.

Chapter 4

Web Content and Services

The classic aphorism, "No man is an island," is a way of saying that all human beings are interconnected within a greater social, emotional, and spiritual reality. And what we see as greatness in a person is very much a matter of how deeply he or she has realized this truth.

The same is apparently also true for apps. The data collected by organizations such as Distmo shows that connected apps—those that reach beyond themselves and their host device rather than thinking of themselves as isolated phenomena—generally rate higher and earn more revenue in various app stores. In other words, just as the greatest of human beings are those who have fully realized their connection to an expansive reality, so also are great apps.

This means that we cannot simply take connectivity for granted or give it mere lip service. What makes that connectivity truly valuable is not doing the obvious, like displaying some part of a web page in an app, downloading some RSS feed, or showing a few updates from the user's social network. Greatness needs to do more than that—it needs to bring online connectedness to life in creative and productive ways that *also* make full use of the local device and its powerful resources. These are "hybrid" apps at their best.

Beyond social networks, consider what can be obtained from thousands of web APIs that are accessible through simple HTTP requests, as listed on sites like http://www.programmableweb.com/. As of this writing, that site lists over 11000 separate APIs, a number that continues to grow monthly. This means not only that there are over 11000 individual sources of interesting data that an app might employ, but that there are literally billions of *combinations* of those APIs. In addition to traditional RSS mashups (combining news feeds), a vast unexplored territory of *API mashups* exists, which means bringing disparate data together in meaningful ways. The Programmable Web, in fact, tracks web applications of this sort, but as of this writing there were several thousand *fewer* such mashups than there were APIs! It's like we've taken only the first few steps on the shores of a new continent, and the opportunities are many.[32]

I think it's pretty clear why connected apps are better apps: as a group, they simply deliver a more compelling and valuable user experience than those that limit themselves to the scope of a client device. Thus, it's worth taking the time early in any app project to make connectivity and web content a central part of your design. This is why we're discussing the subject now, even before considerations

[32] Increasing numbers of entrepreneurs are also realizing that services and web APIs in themselves can be a profitable business. Companies like Mashape and Mashery also exist to facilitate such monetization by managing scalable access plans for developers on behalf of the service providers. You can also consider creating a marketable Windows Runtime Component that encapsulates your REST API within class-oriented structures.

like controls and other UI elements!

Of course, the real creative effort to find new ways to use online content is both your challenge and your opportunity. What we can cover in this chapter are simply the tools that you have at your disposal for that creativity.

We'll begin with the essential topic of network connectivity, because there's not much that can be done without it! Then we'll explore the options for directly hosting dynamic web content within an app's own UI, as is suitable for many scenarios. Then we'll look at the APIs for HTTP requests, followed by those for background transfers that can continue when an app is suspended or not running at all. We'll then wrap up with the very important subject of authentication, which includes working with the user's Microsoft account, user profile, and Live Connect services.

One part of networking that we won't cover here is setting up service connections for live tiles and push notifications, which are covered in Chapter 16, "Alive with Activity." The subject of roaming app state is something we'll pick up in Chapter 10, "The Story of State, Part 1," and navigating to and choosing files from network shares has context with the file pickers that we'll see in Chapter 11, "The Story of State, Part 2."

And there is yet more to say on some web-related and networking-related subjects, such as sockets, but I didn't want those details to intrude on the flow of this chapter. You can find those matters in Appendix C, "Additional Networking Topics."

Sidebar: Debugging Network Traffic with Fiddler

Watching the traffic between your machine and the Internet can be invaluable when trying to debug networking operations. For this, check out the freeware tool from Telerik called Fiddler (http://fiddler2.com/get-fiddler). In addition to inspecting traffic, you can also set breakpoints on various events and fiddle with (that is, modify) incoming and outgoing data.

Sidebar: Windows Azure Mobile Services

No discussion of apps and services is complete without giving mention to the highly useful features of Windows Azure Mobile Services, especially as you can start using them for free and start paying only when your apps become successful and demand more bandwidth.

- **Data:** easy access to cloud-based table storage (SQL Server) without the need to use HTTP requests or other low-level mechanisms. The client-side libraries provide very straightforward APIs for create, insert, update, and delete operations, along with queries. On the server side, you can attach node.js scripts to these operations, allowing you to validate and adjust the data as well as trigger other processes if desired.
- **Authentication:** you can authenticate users with Mobile Services using a Microsoft account

or other identity providers. This supplies a unique user id to Mobile Services as you'll often want with data storage. You can also use server-side node.js scripts to perform other authorization tasks.

- **Push Notifications:** a streamlined back-end for working with the Windows Notification Service to support live tiles, badges, toasts, and raw notifications in your app.
- **Services:** sending email, scheduling backend jobs, and uploading images.

To get started, visit the Mobile Services Tutorials and Resources page. We'll also see some of these features in Chapter 16 when we work with live tiles and notifications. And don't forget all the other features of Windows Azure that can serve all your cloud needs, which have either free trials or limited free plans to get you started.

Network Information and Connectivity

At the time I was writing on the subject of live tiles for the first edition of this book (see Chapter 16) and talking about all the connections that Windows Store apps can have to the Internet, my home and many thousands of others in Northern California were completely disconnected due to a fiber optic breakdown. The outage lasted for what seemed like an eternity by present standards: 36 hours! Although I wasn't personally at a loss for how to keep myself busy, there was a time when I opened one of my laptops, found that our service was still down, and wondered for a moment just what the computer was really good for! Clearly I've grown, as I suspect you have too, to take constant connectivity completely for granted.

As developers of great apps, however, we cannot afford to be so complacent. It's always important to handle errors when trying to make connections and draw from online resources, because any number of problems can arise within the span of a single operation. But it goes much deeper than that. It's our job to make our apps as useful as they can be when connectivity is lost, perhaps just because our customers got on an airplane and switched on airplane mode. That is, don't give customers a reason to wonder about the usefulness of their device in such situations! A great app will prove its worth through a great user experience even if it lacks connectivity.

Indeed, be sure to test your apps early and often, both with and without network connectivity, to catch little oversights in your code. In Here My Am!, for example, my first versions of the script in html/map.html didn't bother to check whether the remote script for Bing Maps had actually been downloaded; as a result, the app terminated abruptly when there was no connectivity. Now it at least checks whether the `Microsoft` namespace (for the `Microsoft.Maps.Map` constructor) is valid. So keep these considerations in the back of your mind throughout your development process.

Be mindful that connectivity can vary throughout an app session, where an app can often be suspended and resumed, or suspended for a long time. With mobile devices especially, one might move between any number of networks without necessarily knowing it. Windows, in fact, tries to make the transition between networks as transparent as possible, except where it's important to inform the user that there may be costs associated with the current provider. It's a good idea, for instance, for an app to be aware of data transfer costs on metered networks and prevent "bill shock" from not-always-generous mobile broadband providers. Just as there are certain things an app can't do when the device is offline, the characteristics of the current network might also cause it to defer or avoid certain operations as well.

Anyway, let's see how to retrieve and work with connectivity details, starting with the different types of networks represented in the manifest, followed by obtaining network information, dealing with metered networks, and providing for an offline experience. And unless noted otherwise, the classes and other APIs that we'll encounter are in the `Windows.Networking` namespace.

> **Note** Network connectivity, by its nature, is an intricate subject, as you'll see in in the sections that follow. But don't feel compelled to think about all these up front! If you want to take connectivity entirely for granted for a while and get right into playing with web content and making HTTP requests, feel free to skip ahead to the "Hosting Content" and "HTTP Requests" sections. You can certainly come back here later.

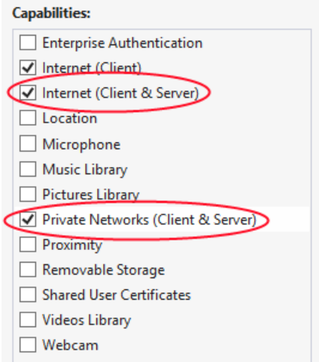
Network Types in the Manifest

Nearly every sample we'll be working with in this book has the *Internet (Client)* capability declared in its manifest, thanks to Visual Studio turning that on by default. This wasn't always the case: early app builders within Microsoft would occasionally scratch their heads wondering just why something really obvious—like making a simple HTTP request to a blog—failed outright. Without this capability, there just isn't any Internet!

Still, *Internet (Client)* isn't the only player in the capabilities game. Some networking apps will also want to act as a server to receive unsolicited incoming traffic from the Internet, and not just make requests to other servers. In those cases—such as file sharing, media servers, VoIP, chat, multiplayer/multicast games, and other bi-directional scenarios involving incoming network traffic, as with sockets—the app must declare the *Internet (Client & Server)* capability, as shown in Figure 4-1. This lets such traffic through the inbound firewall, though critical ports are always blocked.

There is also network traffic that occurs on a private home or business network, where the Internet isn't involved at all, as with line-of-business apps, talking to network-attached storage, and local network games. For this there is the *Private Networks (Client & Server)* capability, also shown in Figure 4-1, which is good for file or media sharing, line-of-business apps, HTTP client apps, multiplayer games on a LAN, and so on. What makes any given IP address part of this private network depends on many factors, all of which are described on How to configure network isolation capabilities. For example, IPv4 addresses in the ranges of 10.0.0.0–10.255.255.255, 172.16.0.0–172.31.255.255, and 192.168.0.0–192.168.255.255 are considered private. Users can flag a network as trusted, and the presence of a domain controller makes the network private as well. Whatever the case, if a device's network endpoint falls into this category, the behavior of apps on that device is governed by this capability rather than those related to the Internet.

Note The *Private Networks* capability isn't necessary when you'll be using the File Picker (see Chapter 11) to allow users to browse local networks. It's necessary only if you're needing to make direct programmatic connections to such resources.



Capabilities:

☐ Enterprise Authentication
☑ Internet (Client)
☑ Internet (Client & Server)
☐ Location
☐ Microphone
☐ Music Library
☐ Pictures Library
☑ Private Networks (Client & Server)
☐ Proximity
☐ Removable Storage
☐ Shared User Certificates
☐ Videos Library
☐ Webcam

FIGURE 4-1 Additional network capabilities in the manifest.

Sidebar: Localhost Loopback

Regardless of the capabilities declared in the manifest, local loopback—that is, using http://localhost URIs—is blocked for apps distributed through the Windows Store. Exceptions are made for side-loaded enterprise apps, and for machines on which a developer license has been installed, as described in "Sidebar: Using the Localhost" in the "Background Transfer" section of this chapter (we'll need to use it with a sample there). The developer exception exists only to simplify debugging apps and services together on the same machine during development. You can disable this allowance in Visual Studio through the Project > Property Pages dialog under Debugging > Allow Local Network Loopback, which helps you test your app as a consumer would experience it.

Network Information (the Network Object Roster)

Regardless of the network involved, everything you want to know about that network is available through the `Connectivity.NetworkInformation` object. Besides a single `networkstatuschanged` event that we'll discuss in "Connectivity Events" a little later, the interface of this object is made up of methods to retrieve more specific details in other objects.

Below is the roster of the methods in `NetworkInformation` and the contents of the objects obtained through them. You can exercise the most common of these APIs through the indicated scenarios of the Network information sample:

- `getInternetConnectionProfile` (Scenario 1) Returns a single `ConnectionProfile` object for the currently active Internet connection. If there is more than one connection, this method returns the preferred profile that's most likely to be used for Internet traffic.

- getConnectionProfiles (Scenario 3) Returns a vector of ConnectionProfile objects, one for each connection, among which will be the active Internet connection as returned by getInternetConnectionProfile. Also included are any wireless connections you've made in the past for which you indicated Connect Automatically. (In this way the sample will show you some details of where you've been recently!) See the next section for more on ConnectionProfile.

- findConnectionProfilesAsync (Scenario 6) Given a ConnectionProfileFilter object, returns a vector of ConnectionProfile objects that match the filter criteria. This helps you find available networks that are suitable for specific app scenarios such as finding a Wi-Fi connection or one with a specific cost policy.

- getHostNames Returns a *vector* (see note below) of HostName objects, one for each connection, that provides various name strings (displayName, canonicalName, and rawName), the name's type (from HostNameType, with values of domainName, ipv4, ipv6, and bluetooth), and an ipinformation property (of type IPInformation) containing prefixLength and networkAdapter properties for IPV4 and IPV6 hosts. (The latter is a NetworkAdapter object with various low-level details.) The HostName class is used in various networking APIs to identify a server or some other endpoint.

- getLanIdentifiers (Scenario 4) Returns a vector of LanIdentifier objects, each of which contains an infrastructureId (LanIdentifierData containing a type and value), a networkAdapterId (a GUID), and a portId (LanIdentifierData).

- getProxyConfigurationAsync Returns a ProxyConfiguration object for a given URI and the current user. The properties of this object are canConnectDirectly (a Boolean) and proxyUris (a vector of Windows.Foundation.Uri objects for the configuration).

- getSortedEndpointPairs Sorts an array of EndpointPair objects according to HostNameSortOptions. An EndpointPair contains a host and service name for local and remote endpoints, typically obtained when you set up specific connections like sockets. The two sort options are none and optimizeForLongConnections, which vary connection behaviors based on whether the app is making short or long duration connection. See the documentation for EndpointPair and HostNameSortOptions for more details.

What is a vector? A vector is a WinRT type that's often used for managing a list or collection. It has methods like append, removeAt, and clear through which you can manage the list. Other methods like getAt and getMany allow retrieval of items, and a vector supports the [ ] operator like an array. For more details, see "Windows.Foundation.Collections Types" in Chapter 6, "Data Binding, Templates, and Collections." In its simplest use, you can treat a vector like a JavaScript array through the [ ] operator.

The ConnectionProfile Object

Of all the information available through the NetworkInformation object, the most important for apps is found in ConnectionProfile, most frequently that returned by getInternetConnectionProfile because that's the one through which an app's Internet traffic will flow. The profile is what contains all the information you need to make decisions about how you're using the network, especially for cost awareness. It's also what you'll typically check when there's a change in network status. Scenarios 1 and 3 of the Network information sample retrieve and display most of these details.

Each profile has a profileName property (a string), such as "Ethernet" or the SSID of your wireless access point, a serviceProviderGuid property (the network operator ID), plus a getNetworkNames method that returns a vector of friendly names for the endpoint. The networkAdapter property contains a NetworkAdapter object for low-level details, should you want them, and the networkSecuritySettings property contains a NetworkSecuritySettings object describing authentication and encryption types.

More generally interesting is the getNetworkConnectivityLevel method, which returns a value from the NetworkConnectivityLevel enumeration: none (no connectivity), localAccess (the level you hate to see when you're trying to get a good connection!), constrainedInternetAccess (captive portal connectivity, typically requiring further credentials as is often encountered in hotels, airports, etc.), and internetAccess (the state you're almost always trying to achieve). The connectivity level is often a factor in your app logic and something you typically watch with network status changes. Related to this is the getDomainConnectivityLevel that provides a DomainConnectivityLevel value of none (no domain controller), unauthenticated (user has not been authenticated by the domain controller), and authenticated.

To check if a connection is on Wi-Fi, check the isWlanConnectionProfile flag and, if it's true, you can look at the wlanConnectionProfileDetails property for more details, such as the SSID. If you're on a mobile connection, on the other hand, the isWwanConnectionProfile flag will be true, in which case the wwlanConnectionProfileDetails property tells you about the type of data service and registration state of the connection. And if for either of these you want to display the connection's strength, the getSignalBars method will give you back a value from 0 to 5.

The ups and downs of a connection's lifetime is retrieved through getConnectivityIntervals-Async, which produces you a vector of ConnectivityInterval objects. Each one describes when this network was connected and how long it remained so.

To track the inbound and outbound traffic on a connection, the getNetworkUsageAsync and method returns a NetworkUsage object that contains bytesReceived, bytesSent, and connectionDuration properties for a given time period and NetworkUsageStates (roaming or shared). Similarly, the getConnectionCost and getDataPlanStatus provide the information an app needs to be aware of how much network traffic is happening and how much it might cost the user. We'll come back to this in "Cost Awareness" shortly, including how to see per-app usage in Task Manager.

Connectivity Events

It is very common for a running app to want to know when connectivity changes. This way it can take appropriate steps to disable or enable certain functionality, alert the user, synchronize data after being offline, and so on. For this, apps need only watch the onnetworkstatuschanged event of the NetworkInformation object, which is fired whenever there's a significant change within the hierarchy of objects we've just seen (and be mindful that this event comes from a WinRT object, so remove your listeners properly). For example, the event will be fired if the connectivity level of a profile changes or the network is disconnected. It fires when new networks are found, in which case you might want to switch from one to another (for instance, from a metered network to a nonmetered one). It will also be fired if the Internet profile itself changes, as when a device roams between different networks, or when a metered data plan is approaching or has exceeded its limit, at which point the user will start worrying about every megabyte of traffic.

In short, you'll generally want to listen for this event to refresh any internal state of your app that's dependent on network characteristics and set whatever flags you use to configure the app's networking behavior. This is especially important for transitioning between online and offline and between unlimited and metered networks; Windows, for its part, also watches this event to adjust its own behavior, as with the Background Transfer APIs.

> **Note** Windows Store apps written in JavaScript can also use the basic window.nagivator.ononline and window.navigator.onoffline events to track connectivity. The window.navigator.onLine property is also true or false accordingly. These events, however, will not alert you to changes in connection profiles, cost, or other aspects that aren't related to the basic availability of an Internet connection. For this reason it's generally better to use the WinRT APIs.

You can play with networkstatuschanged in scenario 5 of the Network information sample. As you connect and disconnect networks or make other changes, the sample will update its details output for the current Internet profile if one is available (code condensed from js/network-status-change.js):

```
var networkInfo = Windows.Networking.Connectivity.NetworkInformation;
// Remember to removeEventListener for this event from WinRT as needed
networkInfo.addEventListener("networkstatuschanged", onNetworkStatusChange);

function onNetworkStatusChange(sender) {
    internetProfileInfo = "Network Status Changed: \n\r";
    var internetProfile = networkInfo.getInternetConnectionProfile();

    if (internetProfile === null) {
        // Error message
    } else {
        internetProfileInfo += getConnectionProfileInfo(internetProfile) + "\n\r";
        // display info
    }

    internetProfileInfo = "";
}
```

Of course, listening for this event is useful only if the app is actually running. But what if it isn't? In that case an app needs to register a *background task* for what's known as the networkStateChange *trigger*, typically applying the internetAvailable or internetNotAvailable *conditions* as needed. We'll talk more about background tasks in Chapter 16; for now, refer to the Network status background sample for a demonstration. The sample itself simply retrieves the Internet profile name and network adapter id in response to this trigger; a real app would clearly take more meaningful action, such as activating background transfers for data synchronization when connectivity is restored. The basic structure is there in the sample nonetheless.

It's also very important to remember that network status might have changed while the app was suspended. Apps that watch the networkstatuschanged event should also refresh their connectivity-related state within their resuming handler.

As a final note, check out the Troubleshooting and debugging network connections topic, which has a little more guidance on responding to network changes as well as network errors.

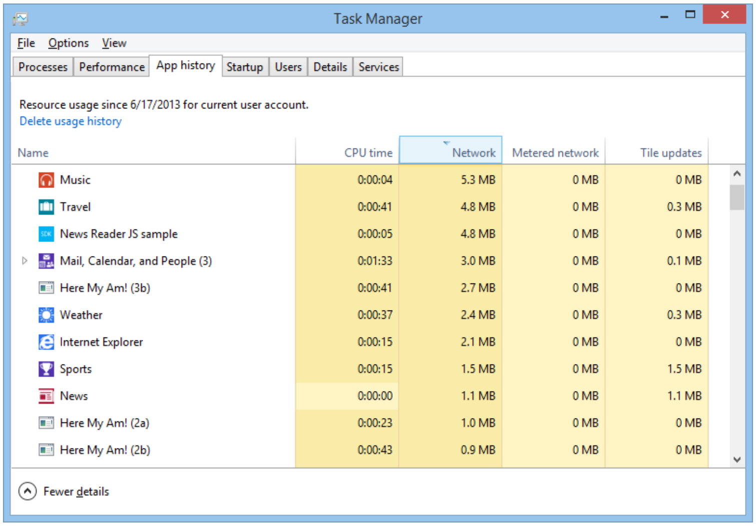
Cost Awareness

If you ever crossed between roaming territories with a smartphone that's set to automatically download email, you probably learned the hard way to disable syncing in such circumstances. I once drove from Washington State into Canada without realizing that I would suddenly be paying $15/megabyte for the privilege of downloading large email attachments. Of course, since I'm a law-abiding citizen I did not look at my phone while driving (wink-wink!) to notice the roaming network. Well, a few weeks later and $100 poorer I knew what "bill shock" was all about!

The point here is that if users conclude that *your* app is responsible for similar behavior, regardless of whether it's actually true, the kinds of rating and reviews you'll receive in the Windows Store won't be good! If your app might transfer any significant data, it's vital to pay attention to changes in the cost of the connection profiles you're using, typically the Internet profile. Always check these details on startup, within your networkstatuschanged event handler, and within your resuming handler.

> **Tip** A powerful way to deal with cost awareness is through what's called a *filter* on which the Windows.Web.Http.HttpClient API is built. This allows you to keep the app logic much cleaner by handling all cost decisions on the lower level of the filter. To see this in action, refer to scenario 11 of the HttpClient sample.

You—and all of your customers, I might add—can track your app's network usage in the App History tab of Task Manager, as shown below. Make sure you've expanded the view by tapping More Details on the bottom left if you don't see this view. You can see that it shows Network and Metered Network usage along with the traffic due to tile updates:


Task Manager
File Options View
Processes Performance App history Startup Users Details Services
Delete usage history
Fewer details

Resource usage since 6/17/2013 for current user account.

Name	CPU time	Network	Metered network	Tile updates
Music	0:00:04	5.3 MB	0 MB	0 MB
Travel	0:00:41	4.8 MB	0 MB	0.3 MB
News Reader JS sample	0:00:05	4.8 MB	0 MB	0 MB
Mail, Calendar, and People (3)	0:01:33	3.0 MB	0 MB	0.1 MB
Here My Am! (3b)	0:00:41	2.7 MB	0 MB	0 MB
Weather	0:00:37	2.4 MB	0 MB	0.3 MB
Internet Explorer	0:00:15	2.1 MB	0 MB	0 MB
Sports	0:00:15	1.5 MB	0 MB	1.5 MB
News	0:00:00	1.1 MB	0 MB	1.1 MB
Here My Am! (2a)	0:00:23	1.0 MB	0 MB	0 MB
Here My Am! (2b)	0:00:43	0.9 MB	0 MB	0 MB

Programmatically, as noted before, the profile supplies usage information through its `getConnectionCost` and `getDataPlanStatus` methods. These return `ConnectionCost` and `DataPlanStatus` objects, respectively, which have the following properties:

ConnectionCost Properties	Description
`networkCostType`	A `NetworkCostType` value, one of `unknown`, `unrestricted` (no extra charges), `fixed` (unrestricted up to a limit), and `variable` (charged on a per-byte basis).
`roaming`	A Boolean indicating whether the connection is to a network outside of your provider's normal coverage area, meaning that extra costs are likely involved. An app should be very conservative with network activity when this is `true` and ask the user for consent for larger data transfers.
`approachingDataLimit`	A Boolean that indicates that data usage on a fixed type network (see `networkCostType`) is getting close to the limit of the data plan.
`overDataLimit`	A Boolean indicating that a fixed data plan's limit has been exceeded and overage charges are definitely in effect. When this is `true`, an app should again be very conservative with network activity, as when `roaming` is `true`.
DataPlanStatus Properties	**Description**
`dataPlanLimitInMegabytes`	The maximum data transfer allowed for the connection in each billing cycle.
`dataPlanUsage`	A `DataPlanUsage` object with an all-important `megabytesUsed` property and a `lastSyncTime` (UTC) indicating when `megabytesUsed` was last updated.
`maxTransferSizeInMegabytes`	The maximum recommended size of a single network operation. This property reflects not so much the capacities of the metered network itself (as its documentation suggests), but rather an appropriate upper limit to transfers on that network.
`nextBillingCycle`	The UTC date and time when the next billing cycle on the plan kicks in and resets `dataPlanUsage` to zero.
`InboundBitsPerSecond` and `outboundBitsPerSecond`	Indicate the nominal transfer speed of the connection.

With all these properties you can make intelligent decisions about your app's network activity, warn the user about possible overage charges, and ask for the user's consent when appropriate. Clearly, when the `networkCostType` is `unrestricted`, you can really do whatever you want. On the other hand, when the type is `variable` and the user is paying for every byte, especially when `roaming` is `true`, you'll want to inform the user of that status and provide settings through which the user can limit the app's network activity, if not halt that activity entirely. After all, the user might decide that certain kinds of data are worth having. For example, they should be able to set the quality of a streaming movie, indicate whether to download email messages or just headers, indicate whether to download images, specify whether caching of online data should occur, turn off background streaming audio, and so on.

Such settings, by the way, might include tile, badge, and other notification activities that you might have established, as those can generate network traffic. If you're also using background transfers, you can set the cost policies for downloads and uploads as well.

An app can, of course, ask the user's permission for any given network operation. It's up to you and your designers to decide when to ask and how often. The Windows Store policy once required that you ask the user for any transfer exceeding one megabyte when `roaming` and `overDataLimit` are both `true` and when performing any transfer over `maxTransferSizeInMegabytes`. This is no longer required, but it's still a good starting point—your customers will clearly appreciate careful consideration, especially if your app is making a number of smaller transfers that might add up to multiple megabytes. At the same time, you don't want to be annoying with consent prompts, so be sure to give the user a way to temporarily disable warnings or ask at reasonable intervals. In short, put yourself in your customer's shoes and design an experience that empowers their ability to control the app's behavior.

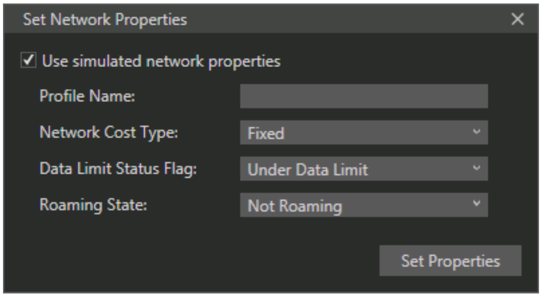
On a `fixed` type network, where data is unrestricted up to `dataPlanLimitInMegabytes`, we find cases where a number of the other properties become interesting. For example, if `overDataLimit` is already `true`, you can ask the user to confirm additional network traffic or just defer certain operations until the `nextBillingCycle`. Or, if `approachingDataLimit` is `true` (or even when it's not), you can determine whether a given operation might exceed that limit. This is where the connection profile's `getNetworkUsageAsync` method comes in handy to obtain a `NetworkUsage` object for a given period (see How to retrieve connection usage data for a specific time period). Call `getNetworkUsageAsync` with the time period between `DataPlanUsage.lastSyncTime` and `DateTime.now()`. Then add that value to `DataPlanUsage.megabytesUsed` and subtract the result from `DataPlanUsage.dataPlanLimitInMegabytes`. This tells you how much more data you can transfer before incurring extra costs, thereby providing the basis for asking the user, "Downloading this file will exceed your data plan limit and dock your wallet. Is that OK or would you rather save your money for something else?"

For simplicity's sake, you can think of cost awareness in terms of three behaviors: *normal*, *conservative*, and *opt-in*, which are described on Managing connections on metered networks and, more broadly, on Developing connected apps. Both topics provide additional guidance on making the kinds of decisions described here already. In the end, saving the user from bill shock—and designing a great user experience around network costs—is definitely an essential investment.

Sidebar: Simulating Metered Networks

You may be thinking, "OK, so I get the need for my app to behave properly with metered networks, but how do I test such conditions?" You can, of course, use a real metered network through a mobile provider, such as the Internet Sharing feature on my phone. However, I do have a data limit and I certainly don't want to test the effect of my app on real roaming fees! Fortunately, you can also simulate the behavior of metered networks with the Visual Studio simulator and, to some extent, directly in Windows with any Wi-Fi connection.

In the simulator, click the Change Network Properties button on the lower right side of the simulator's frame (it's the command above Help—refer back to Figure 2-5 in Chapter 2, "Quickstart"). This brings up the following dialog:


Set Network Properties

☑ Use simulated network properties

Profile Name:

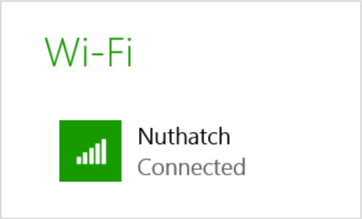
Network Cost Type: Fixed

Data Limit Status Flag: Under Data Limit

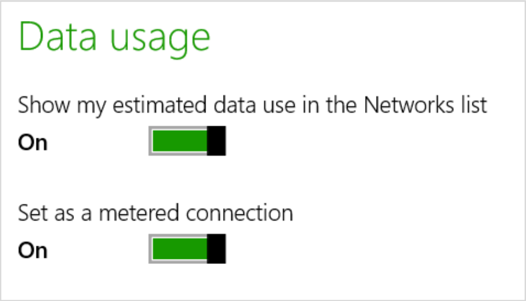
Roaming State: Not Roaming

In this dialog you can create a profile with whatever name and options you'd like. The variations for cost type, data limit status, and roaming allow you to test all conditions that your app might encounter. As such, this is your first choice for working with cost awareness.

To simulate a metered network with a Wi-Fi connection, go to PC Settings > Network > Connections and then tap your current connection under Wi-Fi (as shown below left). On the next page, turn on Set As A Metered Connection under Data Usage (below right):


Wi-Fi

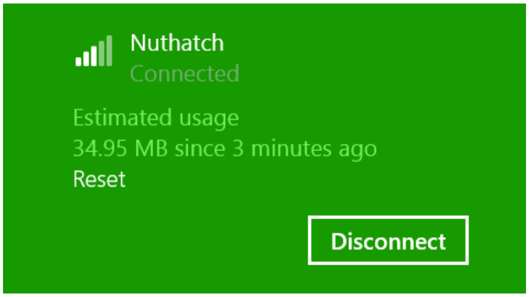
Nuthatch
Connected


Data usage

Show my estimated data use in the Networks list
On

Set as a metered connection
On

Although this option will not set up `DataUsage` properties and all that a real metered network might provide, it will return a `networkCostType` of `fixed`, which allows you to see how your app responds. You can also use the Show My Estimated Data Use in the Networks List option to watch how much traffic your app generates during its normal operation, and you can reset the counter so that you can take some accurate readings:



Nuthatch
Connected
Estimated usage
34.95 MB since 3 minutes ago
Reset

Running Offline

The other user experience that is likely to earn your app a better reputation is how it behaves when there is no connectivity or when there's a change in connectivity. Ask yourself the following questions:

- What happens if your app starts without connectivity, both from tiles (primary and secondary) and through contracts such as search, share, and the file picker?
- What happens if your app runs the first time without connectivity?
- What happens if connectivity is lost while the app is running?
- What happens when connectivity comes back?

As described above in the "Connectivity Awareness" section, use the `networkstatuschanged` event to handle these situations while running and your `resuming` handler to check if connection status changed while the app was suspended. If you have a background task for to the `networkStateChange` trigger, it would primarily save state that your `resuming` handler would then check.

It's perfectly understood that some apps just can't run without connectivity, in which case it's appropriate to inform the user of that situation when the app is launched or when connectivity is lost while the app is running. In other situations, an app might be partially usable, in which case you should inform the user more on a case-by-case basis, allowing them to use unaffected parts of the app. Better still is to cache data that might make the app even more useful when connectivity is lost. Such data might even be built into the app package so that it's always available on first launch.

Consider the case of an ebook reader app that would generally acquire new titles from an online catalog. For offline use it would do well to cache copies of the user's titles locally, rather than rely solely on having a good Internet connection (subject to data transfer limits and appropriate user consent, of course). The app's publisher might also include a number of popular free titles directly in the app package such that a user could install the app while waiting to board a plane and have at least those books ready to go when the app is first launched at 10,000 feet (and you don't like paying for in-flight WiFi). Other apps might include some set of preinstalled data at first and then add to that data over time (perhaps through in-app purchases) when unrestricted networks are available. By following network costs closely, such an app might defer downloading a large data set until either the user confirms the action or a different connection is available.

Tip Caching a set of default data in your app package has several benefits. First, it allows for a good first-run experience when there's no connectivity, because at least some data will appear, even if it's only as current as the last app update in the Store. Second, you can use such cached data to bring the app up very quickly even when there's connectivity, rather than waiting for an HTTP request to respond. Third, you can store the data in your package in its most optimized form so that you don't need to process it as you might an XML or JSON response from a service. What can also work very well is implementing a data model (classes that hide the details of your data management) within your app data that is initially populated from your in-package data and then transparently refreshed and updated with data from HTTP requests. This way the most current data is always used on subsequent runs and is always available offline.

How and when to cache data from online resources is probably one of the fine arts of software development. When do you download it? How much do you acquire? Where do you store it? What might you include as default data in the app package? Should you place an upper limit on the cache? Do you allow changes to cached data that would need to be synchronized with a service when connectivity is restored? These are all good questions ask, and certainly there are others to ask as well. Let me at least offer a few thoughts and suggestions.

First, you can use any network transport to acquire data to cache, such as the various HTTP request APIs we'll discuss later, the background transfer API, as well as the HTML5 AppCache mechanism. Separately, other content acquired from remote resources, such as images and even script (downloaded within `x-ms-webview` or `iframe` elements), are also cached automatically like typical temporary Internet files. Note that this caching mechanism and AppCache are subject to the storage limits defined by Internet Explorer (whose subsystems are shared with the app host). You can also exercise some control over caching through the `HttpClient` API.

How much data you cache depends, certainly, on the type of connection you have and the relative importance of the data. On an unrestricted network, feel free to acquire everything you feel the user might want offline, but it would be a good idea to provide settings to control that behavior, such as overall cache size or the amount of data to acquire per day. I mention the latter because even though my own Internet connection appears to the system as unrestricted, I'm charged more as my usage reaches certain tiers (on the order of gigabytes). As a user, I would appreciate having a say in matters that involve significant network traffic.

Even so, if caching specific data will greatly enhance the user experience, separate that option to give the user control over the decision. For example, an ebook reader might automatically download a whole title while the reader is perhaps just browsing the first few pages. Of course, this would also mean consuming more storage space. Letting users control this behavior as a setting, or even on a per-book basis, lets them decide what's best. For smaller data, on the other hand—say, in the range of several hundred kilobytes—if you know from analytics that a user who views one set of data is highly likely to view another, automatically acquiring and caching those additional data sets could be the right design.

The best places to store cached data are your app data folders, specifically the LocalFolder and TemporaryFolder. Don't use the RoamingFolder to cache data acquired from online sources: besides running the risk of exceeding the roaming quota (see Chapter 10), it's also quite pointless. Because the system would have to roam such data over the network anyway, it's better to just have the app re-acquire it when it needs to.

Whether you use the LocalFolder or TemporaryFolder depends on how essential the data is to the operation of the app. If the app cannot run without the cache, use local app data. If the cache is just an optimization such that the user could reclaim that space with the Disk Cleanup tool, store the cache in the TemporaryFolder and rebuild it again later on.

In all of this, also consider that what you're caching really might be user data that you'd want to store outside of your app data folders. That is, be sure to think through the distinction between app data and user data! We'll think about this more in Chapters 10 and 11.

Finally, you might again have the kind of app that allows offline activity (like processing email) where you will have been caching the results of that activity for later synchronization with an online resource. When connectivity is restored, then, check if the network cost is suitable before starting your sync process.

Hosting Content: the WebView and iframe Elements

One of the most basic uses of online content is to load and render an arbitrary piece of HTML (plus CSS and JavaScript) into a discrete element within an app's overall layout. The app's layout is itself, of course, defined using HTML, CSS, and JavaScript, where the JavaScript code especially has full access to both the DOM and WinRT APIs. For security considerations, however, such a privilege cannot be extended to arbitrary content—it's given only to content that is part of the app's package and has thus gone through the process of Store certification. For everything else, then, we need ways to render content within a more sandboxed environment.

There are two ways to do this, as we'll see in this section. One is through the HTML `iframe` element, which is very restricted in that it can display only in-package pages (`ms-appx[-web]:///` URIs) and secure online content (`https://`). The other more general-purpose choice is the `x-ms-webview` element, which I'll just refer to as the *webview* for convenience. It works with `ms-appx-web`, `http[s]`, and `ms-appdata` URIs, and it provides a number of other highly useful features such as using your own link resolver. The caveats with the webview is that it does not at present support IndexedDB, geolocation, clipboard access, or the HTML5 AppCache, which the `iframe` does. If you require these capabilities, you'll need to use an `iframe` through an `https` URI. At the same time, the webview also has integrated SmartScreen filtering support to protect your app from phishing attacks. Such choices!

In earlier chapters we've already encountered the `ms-appx-web` URI scheme and made mention of the local and web contexts. We'll start this section by exploring these contexts and other security considerations in more detail, because they apply directly to `iframe` and webview elements alike.

Wrapping a web experience `iframe` and `x-ms-webview` elements enable you to easily present a website in an app frame, and this is perfectly allowable. Ideally, you want to do a little more than just show a website in a webview. As an app, your content should look like an app, navigate like an app, work well with touch, use the app bar, support contracts like Search and Share, have a live tile, and draw on other system capabilities. There is a project called the Web app template that helps accomplish some of these basics through configuration data, but think of it as a starting point. The best kind of hybrid app will make the best of both the web and the native platform.

Local and Web Contexts (and iframe Elements)

As described in Chapter 1, "The Life Story of a Windows Store App," apps written with HTML, CSS, and JavaScript are not directly executable like their compiled counterparts written in C#, Visual Basic, or C++. In our app packages, there are no EXEs, just .html, .css, and .js files that are, plain and simple, nothing but text. So something has to turn all this text that defines an app into something that's actually running in memory. That something is again the *app host*, wwahost.exe, which creates what we call the *hosted environment* for Store apps.

Let's review what we've already learned in Chapters 1 and 2 about the characteristics of the hosted environment:

- The app host (and the apps in it) use brokered access to sensitive resources, controlled both by declared capabilities in the manifest and run-time user consent.
- Though the app host provides an environment very similar to that of Internet Explorer (10+), there are a number of changes to the DOM API, documented on HTML and DOM API changes list and HTML, CSS, and JavaScript features and differences. A related topic is Windows Store apps using JavaScript versus traditional web apps.
- HTML content in the app package can be loaded into the *local* or *web context*, depending on the hosting element. `iframe` elements can use the `ms-appx:///` scheme to refer to in-package pages loaded in the local context or `ms-appx-web:///` to specify the web context. (The third `/` again means "in the app package"; the Here My Am! app uses this to load its map.html file into a web context `iframe`.) Remote `https` content in an `iframe` and all content in a webview always runs in the web context.
- Any content within a web context can refer to in-package resources (such as images and other media) with `ms-appx-web` URIs. For example, a page loaded into a webview from an `http` source can refer to an app's in-package logo. (Such a page, of course, would not work in a browser!)
- The local context has access to the WinRT API, among other things, but cannot load remote script (referenced via `http://`); the web context is allowed to load and execute remote script but cannot access WinRT.

- ActiveX control plug-ins are generally not allowed in either context and will fail to load in both iframe and webview elements. The few exceptions are noted on Migrating a web app.
- In the local context, strings assigned to innerHTML, outerHTML, adjacentHTML, and other properties where script injection can occur, as well as strings given to document.write and similar methods, are filtered to remove script. This does not happen in the web context.
- Every iframe and webview element—in either context—has its own JavaScript global namespace that's entirely separate from that of the parent page. Neither can access the other.
- The HTML5 postMessage function can be used to communicate between an iframe and its containing parent across contexts; with a webview such communication happens with the invokeScriptAsync method and window.external.notify. These capabilities can be useful to execute remote script within the web context and pass the results to the local context; script acquired in the web context should not be itself passed to the local context and executed there. (Again, Windows Store policy disallows this, and apps submitted to the Store are analyzed for such practices.)
- Further specifics can be found on Features and restrictions by context, including which parts of WinJS don't rely on WinRT and can thus be used in the web context. (WinJS, by the way, is not supported for use on web *pages* outside of an app, just the web *context* within an app.)

An app's home page—the one you point to in the manifest in the Application > Start Page field—*always* runs in the local context, and any page to which you navigate directly (via <a href> or document.location) must also be in the local context. When using page controls to load HTML fragments into your home page, those fragments are of course rendered into the local context.

Next, a local context page can contain any number of webview and iframe elements. For the webview, because it always loads its content in the web context and cannot refer to ms-appx URIs, it pretty much acts like an embedded web browser where navigation is concerned.

Each iframe element, on the other hand, can load in-package content in either local or web context. (By the way, programmatic read-only access to your package contents is obtained via Windows.ApplicationModel.Package.Current.InstalledLocation.) Referring to a remote location (https) will always place the iframe in the web context.

Here are some examples of different URIs and how they get loaded in an iframe:

```
<!-- iframe in local context with source in the app package -->
<!-- these forms are allowed only from inside the local context -->
<iframe src="/frame-local.html"></iframe>
<iframe src="ms-appx:///frame-local.html"></iframe>

<!-- iframe in web context with source in the app package -->
<iframe src="ms-appx-web:///frame-web.html"></iframe>

<!-- iframe with an external source automatically assigns web context -->
<iframe src="https://my.secure.server.com"></iframe>
```

Also, if you use an `<a href="..." target="...">` tag with `target` pointing to an `iframe`, the scheme in `href` determines the context. And once in the web context, an `iframe` can host only other web context `iframes` such as the last two above; the first two elements would not be allowed.

> **Tip** Some web pages contain frame-busting code that prevents the page from being loaded into an `iframe`, in which case the page will be opened in the default browser and not the app. In this case, use a webview if you can; otherwise you'll need to work with the site owner to create an alternate page that will work for you.

Although Windows Store apps typically don't use `<a href>` or `document.location` for page navigation, similar rules apply if you do happen to use them. The whole scene here, though, can begin to resemble overcooked spaghetti, so I've simplified the exact behavior for these variations and for `iframes` in the following table:

Target	Result in Local Context Page	Result in Web Context Page
`<iframe src="ms-appx:///">`	`iframe` in local context	Not allowed
`<iframe src="ms-appx-web:///">`	`iframe` in web context	`iframe` in web context
`<iframe src="https:// ">`	`iframe` in web context	`iframe` in web context

`<a href="[uri]" target="myFrame">`
`<iframe name="myFrame">`
	`iframe` in local or web context depending on [uri]	`iframe` in web context; [uri] cannot begin with `ms-appx`.

`<a href="ms-appx:///">`	Links to page in local context	Not allowed unless explicitly specified (see below)
`<a href="ms-appx-web:///">`	Not allowed	Links to page in web context
`<a href="[uri]">` with any other protocol including `http[s]`	Opens default browser with [uri]	Opens default browser with [uri]

The last two items in the table really mean that a Windows Store app cannot navigate from its top-level page (in the local context) directly to a web context page of any kind (local or remote) and remain within the app—the app host will launch the default browser instead. That's just life in the app host! Such content must be placed in an `iframe` or a webview. Similarly, navigating from a web context page to a local context page is not allowed by default but can be enabled, as we'll see shortly.

In the meantime, let's see a few simpler `iframe` examples. Again, in the Here My Am! app we've already seen how to load an in-package HTML page in the web context and communicate with the parent page through `postMessage` (We'll change this to a webview in a later section.) Very similar and more isolated examples can also be found in scenarios 2 and 4 of the [Integrating content and controls from web services sample](#).

Scenario 3 of that same sample demonstrates how calls to WinRT APIs are allowed in the local context but blocked in the web context. It loads the same page, callWinRT.html, into a separate `iframe` in each context, which also means the same JavaScript is loaded (and isolated) in both. When running this scenario you can see that WinRT calls will fail in the web context.

A good tip to pick up from this sample is that you can use the `document.location.protocol` property to check which context you're running in, as done in js/callWinRT.js:

```
var isWebContext = (document.location.protocol === "ms-appx-web:");
```

Checking against the string "ms-appx:" will, of course, tell you if you're running in the local context.

Scenarios 5 and 6 of the sample are very interesting because they help us explore matters around inserting HTML into the DOM and navigating from the web to the local context. Each of these subjects, however, needs a little more context of their own (forgive the pun!), as discussed in the next two sections.

> **Tip** To prevent selection of content in an `iframe`, style the `iframe` with `-ms-user-select: none` or set its `style.msUserSelect` property to `"none"` in JavaScript. This does not, however, work for the webview control; its internal content would need to be styled instead.

Dynamic Content

As we've seen, the `ms-appx` and `ms-appx-web` schema allow an app to navigate `iframe` and webview elements to pages that exist inside the app package. This begs a question: can an app point to content on the local file system that exists outside its package, such as a dynamically created file in an appdata folder? Can, perchance, an app use the `file://` protocol to navigate to and/or access that content?

Well, as much as I'd love to tell you that this just works, the answer is somewhat mixed. First, the `file` protocol—along with custom protocols—are wholly blocked by design for various security reasons, even for your appdata folders to which you otherwise have full access. Fortunately, there is a substitute, `ms-appdata:///`, that fulfills part of the need (the third `/`again allows you to omit the specific package name). Within the local context of an app, `ms-appdata` is a shortcut to your appdata folder wherein exist local, roaming, and temp folders. So, if you created a picture called image65.png in your appdata local folder, you can refer to it by using `ms-appdata:///local/image65.png`. Similar forms work with `roaming` and `temp` and work wherever a URI can be used, including within a CSS style like `background`.

Within `iframes`, `ms-appdata` can be used only for resources, namely with the `src` attribute of `img`, `video`, and `audio` elements. It cannot be used to load HTML pages, CSS stylesheets, or JavaScript, nor can it be used for navigation purposes (`iframe`, hyperlinks, etc.). This is because it wasn't feasible to create a sub-sandbox environment for such pages, without which it would be possible for a page loaded with `ms-appdata` to access everything in your app. Fortunately, you *can* navigate a webview to app data content, as we'll see shortly, thereby allowing you to generate and display HTML pages dynamically without having to write your own rendering engine (whew!).

You can also load bits of HTML, as we've seen with page controls, and insert that markup into the DOM through `innerHTML`, `outerHTML`, `adjacentHTML` and related properties, as well as `document.write` and `DOMParser.parseFromString`. But remember that automatic filtering is applied in the local context to prevent injection of script and other risky markup (and if you try, the app host will throw exceptions, as will `WinJS.Utilities.setInnerHTML`, `setOuterHTML`, and `insertAdjacentHTML`). This is not a concern in the web context, of course.

This brings us to whether you can generate and execute script on the fly in the local context at all.

The answer is again qualified. Yes, you can take a JavaScript string and pass it to the `eval` or `execScript` functions, even inject script through properties like `innerHTML`. But be mindful that requirement 3.9 of the App certification requirements (as of this writing) disallows dynamically downloading code or data that changes how the app interacts with the WinRT API. This is admittedly a bit of a gray area—downloading data to configure a game level, for instance, doesn't quite fall into this category. Nevertheless, this requirement is taken seriously, so be careful about making assumptions.

That said, there are situations where you, the developer, really know what you're doing and enjoy juggling chainsaws and flaming swords (or maybe you're just trying to use a third-party library; see the sidebar below). Acknowledging that, Microsoft provides a mechanism to consciously circumvent script filtering: `MSApp.execUnsafeLocalFunction`. For all the details regarding this, refer to Developing secure apps, which covers this along with a few other obscure topics that I'm not including here (like the numerous variations of the `sandbox` attribute for `iframes`, which is also demonstrated in the JavaScript iframe sandbox attribute sample).

And curiously enough, WinJS actually makes it *easier* for you to juggle chainsaws and flaming swords! `WinJS.Utilities.setInnerHTMLUnsafe`, `setOuterHTMLUnsafe`, and `insertAdjacentHTMLUnsafe` are wrappers for calling DOM methods that would otherwise strip out risky content. Alternately, if you want to sanitize HTML before attempting to inject it into an element (and thereby avoid exceptions), you can use the `toStaticHTML` method, as demonstrated in scenario 5 of the Integrating content and controls from web services sample.

Sidebar: Third-Party Libraries and the Hosted Environment

In general, Windows Store apps can employ libraries like jQuery, Angular, Prototype, Dojo, and so forth, as noted in Chapter 1. However, there are some limitations and caveats.

First, because local context pages in an app cannot load script from remote sources, apps typically need to include such libraries in their packages unless they're only being used from the web context. WinJS, mind you, doesn't need bundling because it's provided by the Windows Store, but such "framework packages" are not enabled for third parties.

Second, DOM API changes and app container restrictions might affect the library. For example, using `window.alert` won't work. One library also cannot load another library from a remote source in the local context. Crucially, anything in the library that assumes a higher level of trust than the app container provides (such as open file system access) will have issues.

The most common problem comes up when libraries inject elements or script into the DOM (as through `innerHTML`), a widespread practice for web applications that is not automatically allowed within the app container. You can get around this on the app level by wrapping code within `MSApp.execUnsafeLocalFunction`, but that doesn't solve injections coming from deeper inside the library. In these cases you really need to work with the library author.

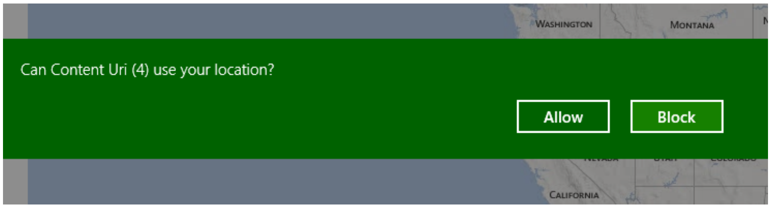
In short, you're free to use third-party libraries so long as you're aware that they might have been written with assumptions that don't always apply within the app container. Over time, of course, fully Windows-compatible versions of such libraries, like jQuery 2.0, will emerge. Note also that for any libraries that include binary components, those must be targeted to Windows 8.1 for use with a Windows 8.1 app.

App Content URIs

When drawing on a variety of web content, it's important to understand the degree to which you trust that content. That is, there's a huge difference between web content that you control and that which you do not, because by bringing that content into the app, the app essentially takes responsibility for it. This means that you want to be careful about what privileges you extend to that web content. In an `iframe`, those privileges include cross-context navigation, geolocation, IndexedDB, HTML5 AppCache, clipboard access, and navigating to web content with an `https` URI. In a webview, it means the ability for remote content to raise an event to the app.[33]

If you ask nicely, in other words, Windows will let you enable such privileges to web pages that the app knows about. All it takes is an affidavit signed by you and sixteen witnesses, and…OK, I'm only joking! You simply need to add what are called *application content URI rules* to your manifest in the Content Uri tab. Each rule—composed of an exact `https` URI or one with wildcards (*)—says that content from some URI is known and trusted by your app and can thus act on the app's behalf. You can also exclude URIs, which is typically done to exclude specific pages that would otherwise be allowed by another rule.

For instance, the very simple ContentUri example in this chapter's companion content has an `iframe` pointing to https://www.bing.com/maps/ (Bing allows an `https://` connection), and this URI is included in the in the content URI rules. This allows the app to host the remote content as partially shown belowNow click or tap the geolocation crosshair circle on the upper left of the map next to World. Because the rules say we trust this content (and trust that it won't try to trick the user), a geolocation request invokes a consent dialog (as shown below) just as if the request came from the app. (Note: When run inside the debugger, the ContentUri example will probably show exceptions on startup. If so, press Continue within Visual Studio; this doesn't affect the app running outside the debugger.)



Can Content Uri (4) use your location?

Such brokered capabilities require a content URI rule because web content loaded into an `iframe` can easily provide the means to navigate to other arbitrary pages that could potentially be malicious.

[33] At whatever point the webview supports IndexedDB or AppCache, these features will likely require such permissions.

Lacking a content URI rule for that target page, the iframe will not navigate there at all.

In some app designs you might have occasion to navigate from a web context page in the app to a local context page. For example, you might host a page on a server where it can keep other server-side content fully secure (that is, not bring it onto the client). You can host the page in an iframe, of course, but if for some reason you need to directly navigate to it, you'll probably need to navigate back to a local context page. You can enable this by calling the super-secret function MSApp.addPublicLocalApplicationUri from code in a local page (and it actually is well-documented) for each specific URI you need. Scenario 6 of the Integrating content and controls from web services sample gives an example of this. First it has an iframe in the web context (html/addPublicLocalUri.html):

```
<iframe src="ms-appx-web:///navigateToLocal.html"></iframe>
```

That page then has an <a href> to navigate to a local context page that calls a WinRT API for good measure; see navigateToLocal.html in the project root:

```
<a href="ms-appx:///callWinRT.html">Navigate to ms-appx:///callWinRT.html</a>
```

To allow this to work, we then have to call addPublicLocalApplicationUri from a local context page and specify the trusted target (js/addPublicLocalUri.js):

```
MSApp.addPublicLocalApplicationUri("callWinRT.html");
```

Typically it's a good practice to include the ms-appx:/// prefix in the call for clarity:

```
MSApp.addPublicLocalApplicationUri("ms-appx:///callWinRT.html");
```

Be aware that this method is very powerful without giving the appearance of such. Because the web context can host any remote page, be especially careful when the URI contains query parameters. For example, you don't want to allow a website to navigate to something like ms-appx:///delete.html?file=superimportant.doc and just accept those parameters blindly! In short, always consider such URI parameters (and any information in headers) to be untrusted content.

The <x-ms-webview> Element

Whenever you want to display some arbitrary HTML page within the context of your app—specifically pages that exists outside of your app package—then the x-ms-webview element is your best friend.[34] This is a native HTML element that's recognized by the rendering engine and basically works like the core of a web browser (without the surrounding business of navigation, favorites, and so forth). Anything loaded into a webview runs in the web context, so it can be used for arbitrary URIs except those using the ms-appx schema. It also supports ms-appdata URIs and rendering string literals, which

[34] The inclusion of the webview element is one of the significant improvements for Windows 8.1. In Windows 8, apps written in HTML, CSS, and JavaScript have only iframe elements at their disposal. However, iframes don't work with web pages that contain frame-busting code, can't load local (appdata) pages, and have some subtle security issues. For this reason, Windows 8.1 has the native x-ms-webview HTML element for most uses and limits iframe to in-package ms-appx[-web] and https URIs exclusively.

means you can easily display HTML/CSS/JavaScript that you generate dynamically as well as content that's downloaded and stored locally. This includes the ability to do your own link resolution, as when images are stored in a database rather than as separate files. Webview content again always runs in the web context (without WinRT access), there aren't restrictions as to what you can do with script and such so far as Store certification is concerned. And the webview even supports additional features like rendering its contents to a stream from which you can create a bitmap. So let's see how all that works!

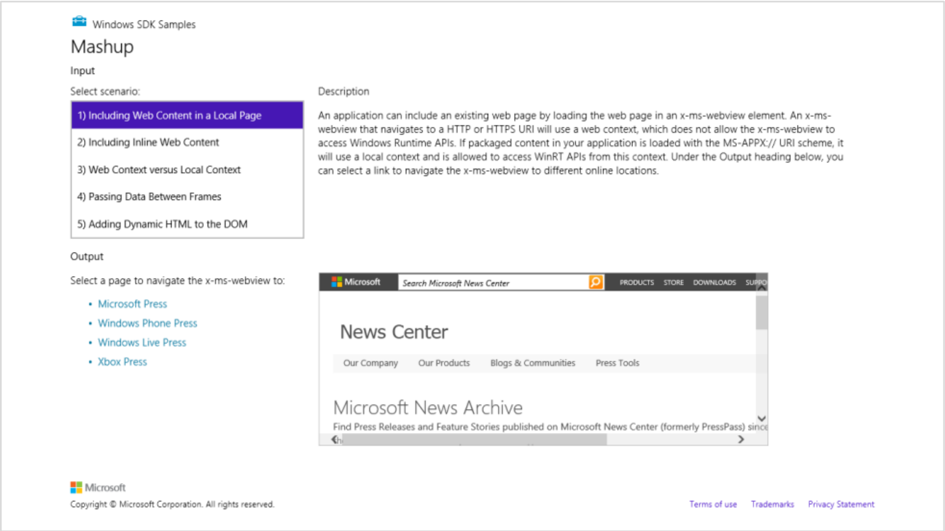
> **What's with the crazy name?** You're probably wondering why the webview has this oddball `x-ms-webview` tag. This is to avoid any future conflict with emerging standards, at which point a vendor-prefixed implementation could become `ms-webview`.

Because the webview is an HTML element like any other, you can style it with CSS however you want, animate the element around, and so forth. Its JavaScript object also has the full set of properties, methods, and events that are shared with other HTML elements, along with a few unique ones of its own. Note, however, that the webview does not have or support any child content of its own, so properties like `innerHTML` and `childNodes` are empty and have no effect if you set them.

The simplest use case for the webview (and I call it this because it's tiresome to type out the funky element name every time) is to just point it to a URI through its `src` attribute. One example is in scenario 1 of the Integrating content and controls from web services sample (html/webContent.html), with the results shown in Figure 4-2:

```
<x-ms-webview id="webContentHolder"
src="http://www.microsoft.com/presspass/press/NewsArchive.mspx?cmbContentType=PressRelease">
</x-ms-webview>
```

The sample lets you choose different links, which are then rendered in the webview by again simply setting its `src` attribute.



Windows SDK Samples

Mashup

Input

Select scenario:

1) Including Web Content in a Local Page
2) Including Inline Web Content
3) Web Context versus Local Context
4) Passing Data Between Frames
5) Adding Dynamic HTML to the DOM

Description

An application can include an existing web page by loading the web page in an x-ms-webview element. An x-ms-webview that navigates to a HTTP or HTTPS URI will use a web context, which does not allow the x-ms-webview to access Windows Runtime APIs. If packaged content in your application is loaded with the MS-APPX:// URI scheme, it will use a local context and is allowed to access WinRT APIs from this context. Under the Output heading below, you can select a link to navigate the x-ms-webview to different online locations.

Output

Select a page to navigate the x-ms-webview to:

- Microsoft Press
- Windows Phone Press
- Windows Live Press
- Xbox Press

Copyright © Microsoft Corporation. All rights reserved.

FIGURE 4-2 Displaying a webview, which is an HTML element like any others within an app layout. The webview runs within the web context and allows navigation within its own content.

Clicking links inside a webview will navigate to those pages. In many cases with live web pages, you'll see JavaScript exceptions if you're running the app in the debugger. Such exceptions will *not* terminate the app as a whole, so they can be safely ignored or left unhandled. Outside of the debugger, in fact, a user will never see these—the webview ignores them.

As we see in this example, setting the `src` attribute is one way to load content into the webview. The webview object also supports four other methods:

- `navigate` Navigates the webview to a supported URI (`http[s]`, `ms-appx-web`, and `ms-appdata`). That page can contain references to other URIs except for `ms-appx`.
- `navigateWithHttpRequestMethod` Navigates to a supported URI with the ability to set the HTTP verb and headers.
- `navigateToString` Renders an HTML string literal into the webview. References can again refer to supported URIs except for `ms-appx`.
- `navigateToLocalStreamUri` Navigates to a page in local appdata using an app-provided object to resolve relative URIs and possibly decrypt the page content.

Examples of the most of these can be found in the HTML Webview control sample. Scenario 1 shows `navigate`, starting with an empty webview and then calling `navigate` with a URI string (js/1_NavToUrl.js):

```
var webviewControl = document.getElementById("webview");
webviewControl.navigate("http://go.microsoft.com/fwlink/?LinkId=294155");
```

Navigating through navigateWithHttpRequestMessage is a little more involved. Though not included in the sample, relevant code can be found on the App Builders Blog in Blending Apps and Sites with the HTML x-ms-webview:

```
//The site to which we navigate
var siteUrl = new Windows.Foundation.Uri("http://www.msn.com");

//Specify the type of request (get)
var httpRequestMessage = new
Windows.Web.Http.HttpRequestMessage(Windows.Web.Http.HttpMethod.get, siteUrl);

// Append headers to request the server to check against the cache
httpRequestMessage.headers.append("Cache-Control", "no-cache");
httpRequestMessage.headers.append("Pragma", "no-cache");

// Navigate the WebView with the request info
webview.navigateWithHttpRequestMessage(httpRequestMessage);
```

Scenario 2 of the SDK sample shows `navigateToString` by loading an in-package HTML file into a string variable, which is like calling `navigate` with the same `ms-appx-web` URI. Of course, if you have the content in an HTML file already, just use navigate! It's more common, then, to use navigateToString with dynamically-generated content. For example, let's say I create a string as follows, which you'll notice includes a reference to an in-package stylesheet. You can find this in scenario 1 of the WebviewExtras example in this chapter's companion content (js/scenario1.js):

```
var baseURI = "http://www.kraigbrockschmidt.com/images/";
var content = "<!doctype HTML><head><style>";
//Refer to an in-package stylesheet (or one in ms-appdata:/// or http[s]://)
content +=
    "<head><link rel='stylesheet' href='ms-appx-web:///css/localstyles.css' /></head>";
content += "<html><body><h1>Dynamically-created page</h1>";
content += "<p>This document contains its own styles as well as a remote image references.</p>"
content += "<img src='" + baseURI + "Cover_ProgrammingWinApps-2E.jpg' />" + space;
content += "<img src='" + baseURI + "Cover_ProgrammingWinApps-1E.jpg' />" + space;
content += "<img src='" + baseURI + "Cover_MysticMicrosoft.jpg' />" + space;
content += "<img src='" + baseURI + "Cover_FindingFocus.jpg' />" + space;
content += "<img src='" + baseURI + "Cover_HarmoniumHandbook2.jpg' />"
content += "</body></html>";
```

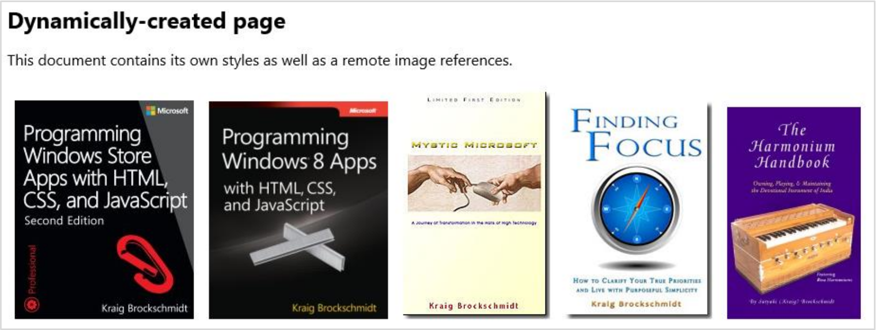
With this we can then just load this string directly:

```
var webview = document.getElementById("webview");
webview.navigateToString(content);
```

We could just as easily write this text to a file in our appdata and use navigate with an ms-appdata URI (this is what's shown in js/scenario1.js):

```
var local = Windows.Storage.ApplicationData.current.localFolder;

local.createFolderAsync("pages",
        Windows.Storage.CreationCollisionOption.openIfExists).then(function (folder) {
    return folder.createFileAsync("dynamicPage.html",
        Windows.Storage.CreationCollisionOption.replaceExisting);
}).then(function (file) {
    return Windows.Storage.FileIO.writeTextAsync(file, content);
}).then(function () {
    var webview = document.getElementById("webview");
    webview.navigate("ms-appdata:///local/pages/dynamicPage.html");
}).done(null, function (e) {
    WinJS.log && WinJS.log("failed to create dynamicPage.html, err = " + e.message, "app");
});
```

In both of these examples, the output (styled with the in-package stylesheet) is the following shameless display of my current written works:



Dynamically-created page

This document contains its own styles as well as a remote image references.

Take careful note of the fact that I create this dynamic page in a subfolder within local appdata. The webview specifically disallows navigation to pages in a *root* local, roaming, or temp appdata folder to protect the security of other appdata files and folders. That is, because the webview runs in the web context and can contain any untrusted content you might have downloaded from the web, and because the webview allows that content to `exec` script and so forth, you don't want to risk exposing potentially sensitive information elsewhere within your appdata. By forcing you to place appdata content in a subfolder, you would have to consciously store other appdata in that same folder to allow the webview to access it. It's a small barrier, in other words, to give you pause to think clearly about exactly what you're doing!

In the example I also include a link to an in-package image (not shown), just to show that you can use ms-appx-web URIs for this purpose:

```
content += "<img src='ms-appx-web:///images/logo.png' />";
```

Scenario 3 of the SDK's HTML WebView control sample (js/scenario3.js) also shows an example of using `ms-appdata` URIs, in this case copying an in-package file to local appdata and navigating to that. Another likely scenario is that you'll download content from an online service via an HTTP request, store that in an appdata file, and navigate to it. In such cases you're just building the necessary file structure in a folder and navigating to the appropriate page. So, for example, you might make an HTTP request to a service to obtain multimedia content in a single compressed fil e. You can then expand that file into your appdata and, assuming that the root HTML page has relative references to other files, the webview can load and render it.

But what if you want to download a single file in a private format (like an ebook) or perha ps acquire a potentially encrypted HTML page along with a single database file for media resources? This is the purpose of `navigateToLocalStream`, which lets you inject your own content handlers and link resolvers into the rendering process. This method takes two arguments:

- A content URI that's created by calling the webview's `buildLocalStreamUri` method with an app-defined content identifier and the relative reference to resolve.
- A *resolver object* that implements an interface called `IUriToStreamResolver`, whose single method `UriToStreamAsync` takes a relative URI and produces a WinRT `IInputStream` through

which the rendering engine can then load the media.

Scenario 4 of the HTML WebView control sample demonstrates this with resolver objects implemented via WinRT components in C# and C++. (See Chapter 18, "WinRT Components," for how these are structured.) Here's how one is invoked:

```
var contentUri = document.getElementById("webview").buildLocalStreamUri("NavigateToStream",
    "simple_example.html");
var uriResolver = new SDK.WebViewSampleCS.StreamUriResolver();
document.getElementById("webview").navigateToLocalStreamUri(contentUri, uriResolver);
```

In this code, `contentUri` will be an `ms-local-stream` URI, such as *ms-local-stream://microsoft.sdksamples.controlswebview.js_4e617669676174465546f53747265616d/simple_example.html*. Because this starts with `ms-local-stream`, the webview will immediately call the resolver object's `UriToStreamAsync` to generate a stream for this page as a whole. So if you had a URI to an encrypted file, the resolver object could perform the necessary decryption to get the first stream of straight HTML for the webview, perhaps applying DRM in the process.

As the webview renders that HTML and encounters other relative URIs, it will call upon the resolver object for each one of those in turn, allowing that resolver to stream media from a database or perform any other necessary steps in the process.

The details of doing all this are beyond the scope of this chapter, so do refer again to the HTML WebView control sample.

Webview Navigation Events

The idea of navigating to a URI is one that certainly conjures up thoughts of a general purpose web browser and, in fact, the web view can serve reasonably well in such a capacity because it both maintains an internal navigation history and fires events when navigation happens.

Although the contents of the navigation history are not exposed, two properties and methods give you enough to implement forward/back UI buttons to control the webview:

- `canGoBack` and `canGoForward` Boolean properties that indicate the current position of the web view within its navigation history.

- `goBack` and `goForward` Methods that navigate the webview backwards or forwards in its history.

When you navigate the webview in any way, it will fire the following events:

- `MSWebViewNavigationStarting` Navigation has started.

- `MSWebViewContentLoading` The HTML content stream has been provided to the webview (e.g., a file is loaded or a resolver object has provided the stream).

- `MSWebViewDOMContentLoaded` The webview's DOM has been constructed.

- MSWebViewNavigationCompleted The webview's content has been fully loaded, including any referenced resources.

If a problem occurs along the way, the webview will raise an MSWebViewUnviewableContentIdentified event instead. It's also worth mentioning that the standard change event will also fire when navigation happens, but this also happens when setting other properties, so it's not as useful for navigation purposes.

Scenario 1 of the HTML WebView control sample, which we saw earlier for navigate, essentially gives you a simple web browser by wiring these methods and events to a couple of buttons. Note that any popups from websites you visit will open in the browser alongside the app.

Tip You'll find when working with the webview in JavaScript that the object does *not* provide equivalent on* properties for these events. This omission was a conscious choice to avoid potential naming conflicts with emerging standards. At present, then, you must use addEventListener to wire up these events.

In addition to the navigating/loading events for the webview's main content, it also passes along similar events for iframe elements within that content: MSWebViewFrameNavigationStarting, MSWebViewFrameContentLoading, MSWebViewFrameDOMContentLoaded, and MSWebViewFrameNavigationCompleted, each of which clearly has the same meaning as the related webview events but also include the URI to which the frame is navigated in eventArgs.uri.

Calling Functions and Receiving Events from Webview Content

The other event that can come from the webview is MSWebViewScriptNotify. This is how JavaScript code in the webview can raise a custom event to its host, similar to how we've used postMessage from an iframe in the Here My Am! app to notify the app of a location change. On the flip side of the equation, the webview's invokeScriptAsync method provides a means for the app to call a function within the webview.

Invoking script in a webview is demonstrated in scenario 5 of the HTML WebView control sample, where the following content of html/script_example.html (condensed here) is loaded into the webview:

```
<!DOCTYPE html><html><head>
    <title>Script Example</title>
    <script type="text/javascript">
        function changeText(text) {
            document.getElementById("myDiv").innerText = text;
        }
    </script>
</head><body>
    <div id="myDiv">Call the changeText function to change this text</div>
</body></html>
```

The app calls changeText as follows:

```
document.getElementById("webview").invokeScriptAsync("changeText",
    document.getElementById("textInput").value).start();
```

The second parameter to invokeScriptAsync method is *always a string* (or will be converted to a string). If you want to pass multiple arguments, use JSON.stringify on an object with suitably named properties and JSON.parse it on the other end.

> **Take note!** Notice the all-important start() tacked onto the end of the invokeScriptAsync call. This is necessary to actually run the async calling operation. Without it, you'll be left wondering just why exactly the call didn't happen! We'll talk more of this in a moment with another example, including how we get a return value from the function.

Receiving an event from a webview is demonstrated in scenario 6 of the sample. An event is raised using the window.external.notify method, whose single argument is again a string. In the sample, the html/scriptnotify_example.html page contains this bit of JavaScript:

```
window.external.notify("The current time is " + new Date());
```

which is picked up in the app as follows, where the event arg's value property contains the arguments from window.external.notify:

```
document.getElementById("webview").addEventListener("MSWebViewScriptNotify", scriptNotify);

function scriptNotify(e) {
    var outputArea = document.getElementById("outputArea");
    outputArea.value += ("ScriptNotify event received with data:\n" + e.value + "\n\n");
    outputArea.scrollTop = outputArea.scrollHeight;
}
```

> **Requirement** MSWebViewScriptNotify will be raised only from webviews loaded with ms-appx-web, ms-local-stream, and https content, where https also requires a content URI rule in your manifest, otherwise that event will be blocked. ms-appdata is also allowed if you have a URI resolver involved. Note that a webview loaded through navigateToString does not have this requirement.

As another demonstration of this call/event mechanism with webview, I've made some changes to Here My Am! in the HereMyAm4 example in this chapter's companion content. First, I've replaced the iframe we've been using to load the map page with x-ms-webview. Then I replaced the postMessage interactions to set a location and pick up the movement of a pin with invokeScriptAsync and MSWebViewScriptNotify. The code structure is essentially the same, and it's still useful to have some generic helper functions with all this (though we don't need to worry about setting the right origin strings as we do with postMessage).

One piece of code we can wholly eliminate is the handler in html/map.html that converted the contents of a message event into a function call. Such code is unnecessary as invokeScriptAsync goes straight to the function; just note again that the arguments are passed as a single string so the invoked function (like our pinLocation in html/map.html) needs to account for that.

The piece of code we want to look at specifically is the new `callWebviewScript` helper, which replaces the previous `callFrameScript` function. Here's the core code:

```
var op = webview.invokeScriptAsync(targetFunction, args);
op.oncomplete = function (args) { /* args.target.result contains script return value */ };
op.onerror = function (e) { /* ... */ };

//Don't forget this, or the script function won't be called!
op.start();
```

What might strike you as odd as you look at this code is that the return value of `invokeScriptAsync` is *not* a promise, but rather a DOM object that has `complete` and `error` events (and can have multiple subscribers to those, of course). In addition, the operation does not actually start until you call this object's `start` method. What gives? Well, remember that the webview is not part of WinRT: it's a native HTML element supported by the app host. So it behaves like other HTML elements and APIs (like `XMLHttpRequest`) rather than WinRT objects. Ah sweet inconsistencies of life!

The reason why `start` must be called separately, then, is so you can attach completed and error handlers to the object *before* the operation gets started, otherwise they won't be called.

Fortunately, it's not too difficult to wrap such an operation within a promise. Just place the same code structure above within the initialization function passed to `new WinJS.Promise`, and call the complete and error dispatchers within the operation's `complete` and `error` events (refer to Appendix A, "Demystifying Promises," on using `WinJS.Promise`). Notice here that the return value from the script function is in `args.target.result`, so we use that value to complete the promise:

```
return new WinJS.Promise(function (completeDispatch, errorDispatch) {
    var op = webview.invokeScriptAsync(targetFunction, args);

    op.oncomplete = function (args) {;
       //Return value from the invoked function (always a string) is in args.target.result
        completeDispatch(args.target.result);
    };

    op.onerror = function (e) {
        errorDispatch(e);
    };

    op.start();
});
```

This works because the promise initializer is attaching completed/error handlers before calling `start`, where those handlers invoke the appropriate dispatchers. Thus, if you call `then` or `done` on the promise after it's already finished, it will call your completed/error handlers right away. You won't miss out on anything!

For errors that occur outside this operation (such having an invalid `targetFunction`), be sure to create an error object with `WinJS.ErrorFromName` and return a promise in the error state by using `WinJS.Promise.wrapError`. You can see the complete code in HereMyAm4 (pages/home/home.js).

Capturing Webview Content

The other very useful feature of the webview that really sets it apart is the ability to capture its content, something that you simply cannot do with an iframe. There are three ways this can happen.

First is the src attribute. Once MSWebViewNavigationCompleted has fired, src will contain a URI to the content as the webview sees it. For web content, this will be an http[s] URI, which can be opened in a browser. Local content (loaded from strings or app data files) will start with ms-local-web, which can be rendered into another webview using navigateToLocalStream. Be aware that while navigation is happening prior to MSWebViewNavigationCompleted, the state of the src property is indeterminate; use the uri property in those handlers instead.

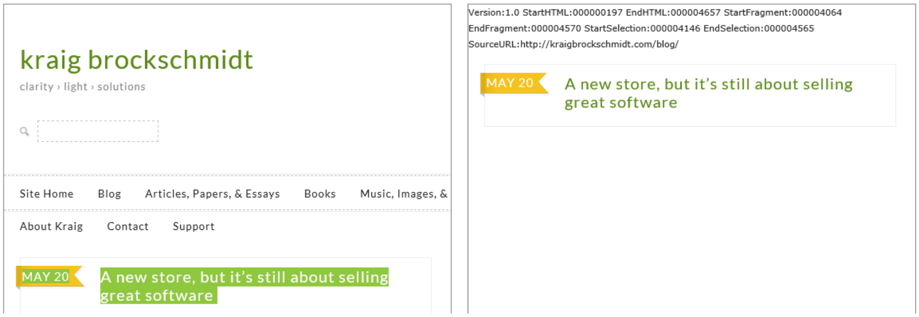
Second is the webview's captureSelectedContentToDataPackageAsync method, which reflects whatever selection the user has made in the webview directly. The fact that a data package is part of this API suggests its primary use: the Share contract. From a user's perspective, any web content you're displaying in the app is really part of the app. So if they make a selection there and invoke the Share charm, they'll expect that their selected data is what gets shared, and this method lets you obtain the HTML for that selection. Of course, you can use this anytime you want the selected content—the Share charm is just one of the potential scenarios.

As with invokeScriptAsync, the return value from captureSelectedContentToDataPackageAsync is again a DOM-ish object with a start method (don't forget to call this!) along with complete and error events. If you want to wrap this in a promise, you can use the same structure as shown in the last section for invokeScriptAsync. In this case, the result you care about within your complete handler, within its args.target.result, is a Windows.ApplicationModel.DataTransfer.DataPackage object, the same as what we encountered in Chapter 2 with the Share charm. Calling its getView method will produce a DataPackageView whose availableFormats object tells you what it contains. You can then use the appropriate get* methods like getHtmlFormatAsync to retrieve the selection data itself. Note that if there is no selection, args.target.result will be null, so you'll need to guard against that. Here, then, is code from scenario 2 of the WebviewExtras example in this chapter's companion content that copies the selection from one webview into another, showing also how to wrap the operation in a promise (js/scenario2.js):

```
function captureSelection() {
    var source = document.getElementById("webviewSource");

    //Wrap the capture method in a promise
    var promise = new WinJS.Promise(function (cd, ed) {
        var op = source.captureSelectedContentToDataPackageAsync();
        op.oncomplete = function (args) { cd(args.target.result); };
        op.onerror = function (e) { ed(e); };
        op.start();
    });

    //Navigate the output webview to the selection, or show an error
    var output = document.getElementById("webviewOutput");

    promise.then(function (dataPackage) {
        if (dataPackage == null) { throw "No selection"; }

        var view = dataPackage.getView();
        return view.getHtmlFormatAsync();
    }).done(function (text) {
        output.navigateToString(text);
    }, function (e) {
        output.navigateToString("Error: " + e.message);
    });
}
```

The output of this example is shown in Figure 4-3. On the left is a webview-hosted page (my blog), and on the right is the captured selection. Note that the captured selection is an HTML *clipboard* format that includes the extra information at the top before the HTML from the webview. If you need to extract just the straight HTML, you'll need to strip off this prefix text up to `<!DOCTYPE html>`.

Generally speaking, `captureSelectedContentToDataPackageAsync` will produce the formats *AnsiText*, *Text*, *HTML Format*, *Rich Text Format*, and *msSourceUrl*, but not a bitmap. For this you need to use the third method, `capturePreviewToBlobAsync`, which again has a `start` method and `complete`/`error` events. The results of this capture (in `args.target.result` within the `complete` handler) is a blob object for whatever content is contained within the webview's display area.


kraig brockschmidt
clarity › light › solutions
Site Home
Blog
Articles, Papers, & Essays
Books
Music, Images, &
About Kraig
Contact
Support
MAY 20
A new store, but it's still about selling great software

```
Version:1.0 StartHTML:000000197 EndHTML:000004657 StartFragment:000004064
EndFragment:000004570 StartSelection:000004146 EndSelection:000004565
SourceURL:http://kraigbrockschmidt.com/blog/
```

MAY 20
A new store, but it's still about selling great software

FIGURE 4-3 Example output from the WebviewExtras example, showing that the captured selection from a webview includes information about the selection as well as the HTML itself.

You can do a variety of things with this blob. If you want to display it in an `img` element, you can use `URL.createObjectURL` on this blob directly. This means you can easily load some chunk of HTML in an offscreen webview (make sure the display style is *not* "none") and then capture a blob and display the results in an `img`. Besides preventing interactivity, you can also animate that image much more efficiently than a full webview, applying 3D CSS transforms, for instance. Scenario 3 of my WebviewExtras example demonstrates this.

For other purposes, like the Share charm, you can call this blob's `msDetachStream` method, which conveniently produces exactly what you need to provide to a data package's `setBitmap` method. This is demonstrated in scenario 7 of the SDK's HTML Webview control sample and more completely (and accurately) in scenario 4 of the Webview Extras example. For more about the Share contract in general, see Chapter 15, "Contracts."

HTTP Requests

Rendering web content directly into your layout with the webview element, as we saw in the previous section, is fabulous provided that, well, you want such content directly in your layout! In many cases you instead want to retrieve data from the web via HTTP requests. Then you can further manipulate, combine, and process it either for display in other controls or to simply drive the app's experience. You'll also have many situations where you need to send information to the web via HTTP requests as well, where one-way elements like the webview aren't of much use.

Windows gives you a number of ways to exchange data with the web. In this section we'll look at the APIs for HTTP requests, which generally require that the app is running. One exception is that Windows lets you indicate web content that it might automatically cache, such that requests you make the next time the app starts (or resumes) can be fulfilled without having to hit the web at all. This takes advantage of the fact that the app host caches web content just like a browser to reduce network traffic and improve performance. This pre-caching capability simply takes advantage of that but is subject to some conditions and is not guaranteed for every requested URI.

Another exception is what we'll talk about in the next section, "Background Transfers." Windows can do background uploads and downloads on your behalf, which continue to work even when the app is suspended or terminated. So, if your scenarios involve data transfers that might test the user's patience for staring at lovely but oh-so-tiresome progress indicators, and which tempt them to switch to another app, use the background transfer API instead of doing it yourself through HTTP requests.

HTTP requests, of course, are the foundation of the RESTful web and many web APIs through which you can get to an enormous amount of interesting data, including web pages and RSS feeds, of course. And because other protocols like SOAP are essentially built on HTTP requests, we'll be focused on the latter here. There are separate WinRT APIs for RSS and AtomPub as well, details for which you can find in Appendix C.

Right! So I said that there are a number of ways to do HTTP requests. Here they are:

- `XMLHttpRequest` This intrinsic JavaScript object works just fine in Windows Store apps, which is very helpful for third-party libraries. Results from this async function come through its `readystatechanged` event.
- `WinJS.xhr` This wrapper provides a promise structure around `XMLHttpRequest`, as we did in the last section with the webview's async methods. `WinJS.xhr` provides quite a bit of flexibility in setting headers and so forth, and by returning a promise it makes it easy to chain `WinJS.xhr` calls with other async operations like WinRT file I/O. You can see a simple example in scenario 1 of the HTML Webview control sample we worked with earlier.
- `HttpClient` The most powerful, high-performance, and flexible API for HTTP requests is found in WinRT in the `Windows.Web.Http` namespace and is recommended for new code. Its primary advantages are that it performs better, works with the same cache as the browser,

serves a wider spectrum of HTTP scenarios, and allows for cookie management, filtering, and flexible transports. You can also create multiple HttpClient instances with different configurations and use them simultaneously.

We'll be focusing here primarily on HttpClient here. For the sake of contrast, however, let's take a quick look at WinJS.xhr in case you encounter it in other code.

Note If you have some experience with the .NET framework, be aware that the HttpClient API in Windows.Web.Http is different from .NET's System.Net.Http.HttpClient API.

Downloadable posters Microsoft's networking team has made some API posters for the HttpClient, Background Transfer, and Sockets APIs, which make handy at-a-glance references.

Using WinJS.xhr

Making a WinJS.xhr call is quite easy, as demonstrated in the SimpleXhr1 example for this chapter. Here we use WinJS.xhr to retrieve the RSS feed from the Windows App Builder blog, noting that the default HTTP verb is GET, so we don't have to specify it explicitly:

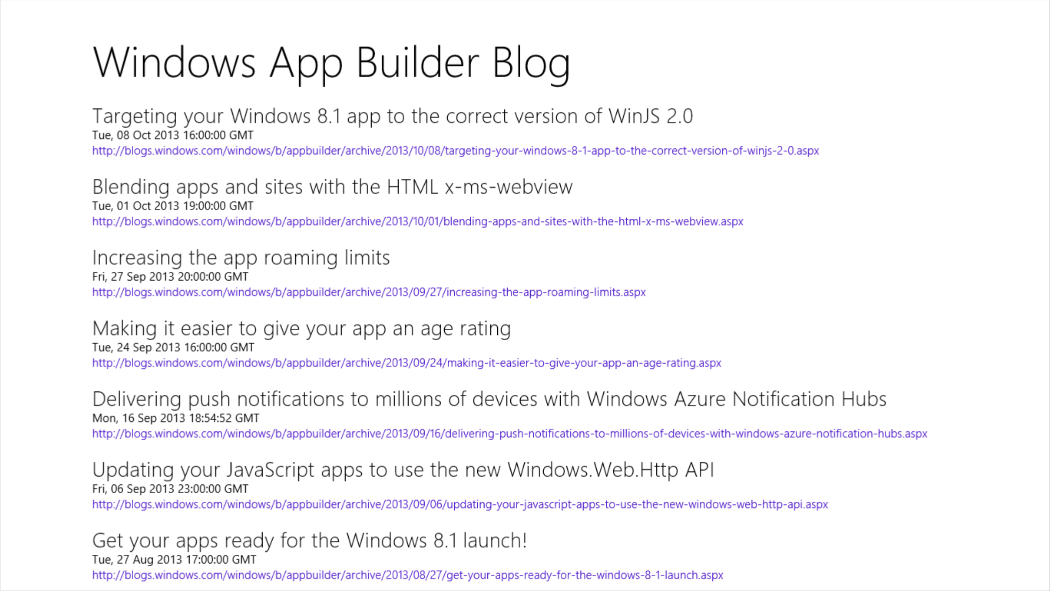
```
WinJS.xhr({ url: "http://blogs.msdn.com/b/windowsappdev/rss.aspx" })
    .done(processPosts, processError, showProgress);
```

That is, give WinJS.xhr a URI and it gives back a promise that delivers its results to your completed handler (in this case processPosts) and will even call a progress handler if provided. With the former, the result contains a responseXML property, which is a DomParser object. With the latter, the event object contains the current XML in its response property, which we can easily use to display a download count:

```
function showProgress(e) {
    var bytes = Math.floor(e.response.length / 1024);
    document.getElementById("status").innerText = "Downloaded " + bytes + " KB";
}
```

The rest of the app just chews on the response text looking for item elements and displaying the title, pubDate, and link fields. With a little styling (see default.css), and utilizing the WinJS typography style classes of win-type-x-large (for title), win-type-medium (for pubDate), and win-type-small (for link), we get a quick app that looks like Figure 4-4. You can look at the code to see the details.[35]

[35] Again, WinRT has a specific API for dealing with RSS feeds in Windows.Web.Syndication, as described in Appendix C. You can use this if you want a more structured means of dealing with such data sources. As it is, JavaScript has intrinsic APIs to work with XML, so it's really your choice. In a case like this, the syndication API along with Windows.Web.AtomPub and Windows.Data.Xml are very much needed by Windows Store apps written in other languages that don't have the same built-in features as JavaScript.



Windows App Builder Blog

Targeting your Windows 8.1 app to the correct version of WinJS 2.0
Tue, 08 Oct 2013 16:00:00 GMT
http://blogs.windows.com/windows/b/appbuilder/archive/2013/10/08/targeting-your-windows-8-1-app-to-the-correct-version-of-winjs-2-0.aspx

Blending apps and sites with the HTML x-ms-webview
Tue, 01 Oct 2013 19:00:00 GMT
http://blogs.windows.com/windows/b/appbuilder/archive/2013/10/01/blending-apps-and-sites-with-the-html-x-ms-webview.aspx

Increasing the app roaming limits
Fri, 27 Sep 2013 20:00:00 GMT
http://blogs.windows.com/windows/b/appbuilder/archive/2013/09/27/increasing-the-app-roaming-limits.aspx

Making it easier to give your app an age rating
Tue, 24 Sep 2013 16:00:00 GMT
http://blogs.windows.com/windows/b/appbuilder/archive/2013/09/24/making-it-easier-to-give-your-app-an-age-rating.aspx

Delivering push notifications to millions of devices with Windows Azure Notification Hubs
Mon, 16 Sep 2013 18:54:52 GMT
http://blogs.windows.com/windows/b/appbuilder/archive/2013/09/16/delivering-push-notifications-to-millions-of-devices-with-windows-azure-notification-hubs.aspx

Updating your JavaScript apps to use the new Windows.Web.Http API
Fri, 06 Sep 2013 23:00:00 GMT
http://blogs.windows.com/windows/b/appbuilder/archive/2013/09/06/updating-your-javascript-apps-to-use-the-new-windows-web-http-api.aspx

Get your apps ready for the Windows 8.1 launch!
Tue, 27 Aug 2013 17:00:00 GMT
http://blogs.windows.com/windows/b/appbuilder/archive/2013/08/27/get-your-apps-ready-for-the-windows-8-1-launch.aspx

FIGURE 4-4 The output of the SimpleXhr1 and SimpleXhr2 apps.

In SimpleXhr1 too, I made sure to provide an error handler to the `WinJS.xhr` promise so that I could at least display a simple message.

For a fuller demonstration of `XMLHttpRequest`/`WinJS.xhr` and related matters, refer to the XHR, handling navigation errors, and URL schemes sample and the tutorial called How to create a mashup in the docs. Additional notes on `XMLHttpRequest` and `WinJS.xhr` can be found in Appendix C.

Using Windows.Web.Http.HttpClient

Let's now see the same app implemented with `Windows.Web.Http.HttpClient`, which you'll find in SimpleXhr2 in the companion content. For our purposes, the `HttpClient.getStringAsync` method is sufficient:

```
var htc = new Windows.Web.Http.HttpClient();
htc.getStringAsync(new Windows.Foundation.Uri("http://blogs.msdn.com/b/windowsappdev/rss.aspx"))
    .done(processPosts, processError, showProgress);
```

This function delivers the response body text to our completed handler (`processPosts`), so we just need to create a `DOMParser` object to talk to the XML document. After that we have the same thing as we received from `WinJS.xhr`:

```
var parser = new window.DOMParser();
var xml = parser.parseFromString(bodyText, "text/xml");
```

The `HttpClient` object provides a number of other methods to initiate various HTTP interactions with a web resource, as illustrated in Figure 4-5.

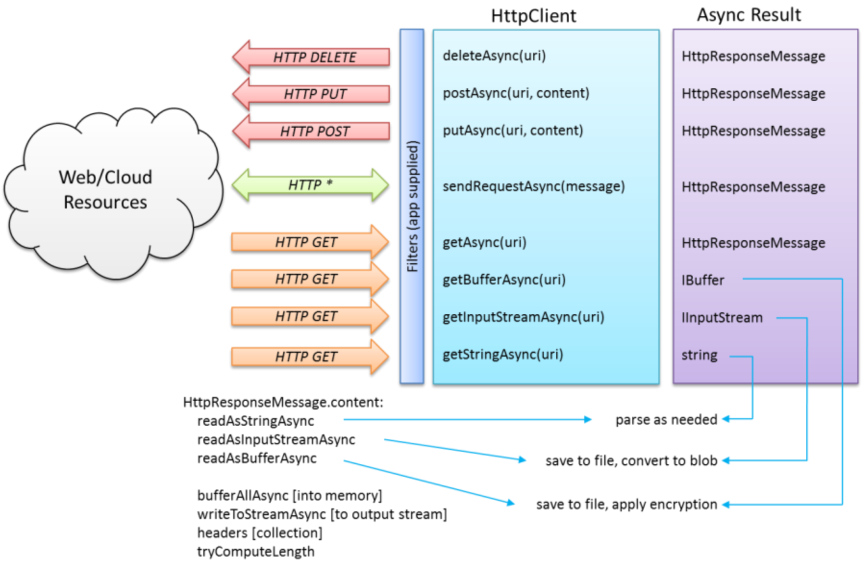


FIGURE 4-5 The methods in the HttpClient object and their associated HTTP traffic. Note how all traffic is routed through an app-supplied filter (or a default), which allows fine-grained control on a level underneath the API.

In all cases, the URI is represented by a Windows.Foundation.Uri object, as we saw in the earlier code snippet. All of the specific get* methods fire off an HTTP GET and deliver results in a particular form: a string, a buffer, and an input stream. All of these methods (as well as sendRequestAsync) support progress, and the progress handler receives an instance of Windows.Web.Http.HttpProgress that contains various properties like bytesReceived.

Working with strings are easy enough, but what are these buffer and input streams? These are specific WinRT constructs that can then be fed into other APIs such as file I/O (see Windows.Storage.Streams and Windows.Storage.StorageFile), encryption/decryption (see Windows.Security. -Cryptography), and also the HTML blob APIs. For example, an IInputStream can be given to MSApp.createStreamFromInputStream, which results in an HTML MSStream object. This can then be given to URL.createObjectURL, the result of which can be assigned directly to an img.src attribute. This is how you can easily fire off an HTTP request for an image resource and show the results in your layout without having to create an intermediate file in your appdata. For more details, see "Q&A on Files, Streams, Buffers, and Blobs" in Chapter 10.

The getAsync method creates a generic HTTP GET request. Its message argument is an HttpRequestMessage object, where you can construct whatever type of request you need, setting the requestUri, headers, transportInformation,[36] and other arbitrary properties that you want to communicate to the filter and possibly the server. The completed handler for getAsync will receive an

[36] This read-only property works with certificates for SSL connections and contains the results of SSL negotiations; see HttpTransportInformation. To set a client certificate, there's a property on the HttpBaseProtocolFilter.

HttpResponseMessage object, as we'll see in a moment.

Handle exceptions! It's very important with HTTP requests that you handle exceptions, that is, provide an error handler for methods like getAsync. Unhandled exceptions arising from HTTP requests has been found to be one of the leading causes of abrupt app termination!

For other HTTP operations, you can see in Figure 4-5 that we have putAsync, postAsync, and deleteAsync, along with the wholly generic sendRequestAsync. With the latter, its *message* argument is again an HttpRequestMessage as used with getAsync, only here you can also set the HTTP method that will be used (this is an HttpMethod object that also allows for additional options). deleteAsync, for its part, works completely from the URI parameters.

In the cases of put and post, the arguments to the methods are the URI and *content,* which is an object that provides the relevant data through methods and properties of the IHttpContent interface (see the lower left of Figure 4-5). It's not expected that you create such objects from scratch (though you can)—WinRT provides built-in implementations called HttpBufferContent, HttpStringContent, HttpStreamContent, HttpMultipartContent, HttpMultipartFormDataContent, and HttpFormUrlEncodedContent.

What you then get back from getAsync, sendRequestAsync, and the delete, put, and post methods is an HttpResponseMessage object. Here you'll find all that bits you would expect:

- statusCode, reasonPhrase, and some helper methods for handling errors—namely, ensureSuccessStatusCode (to throw an exception if a certain code is not received) and isSuccessStatusCode (to check for the range of 200–299).
- A collection of headers (of type HttpResponseHeaderCollection, which then leads to many other secondary classes).
- The original requestMessage (an HttpRequestMessage).
- The source, a value from HttpResponseMessageSource that tells you whether the data was received over the network or loaded from the cache.
- The response content, an object with the IHttpContent interface as before. Through this you can obtain the response data as a string, buffer, input stream, and an in-memory array (bufferAllAsync).

It's clear, then, that the HttpClient object really gives you complete control over whatever kind of HTTP requests you need to make to a service, including additional capabilities like cache control and cookie management as described in the following two sections. It's also clear that HttpClient is still somewhat of a low-level API. For any given web service that you'll be working with, then, I very much recommend creating a layer or library that encapsulates requests to that API and the process of converting responses into the data that the rest of the app wants to work with. This way you can also isolate the rest of the app from the details of your backend, allowing that backend to change as necessary without breaking the app. It's also helpful if you want to incorporate additional features of the `Windows.Web.Http` API, such as filtering, cache control, and cookie management.

I'd love to talk about cookies first (it's always nice to eat dessert before the main meal!) but it's all part of *filtering*. Filtering is a mechanism through which you can control how the `HttpClient` manages its requests and responses. A filter is either an instance of the default `HttpBaseProtocolFilter` class (in the `Windows.Web.Http.Filters` namespace) configured for your needs or an instance of a derived class. You pass this filter object to the `HttpClient` constructor, which will use `HttpBaseProtocolFilter` as a default if none is supplied. To do things like cache control, though, you create an instance of `HttpBaseProtocolFilter` directly, set properties, and then create the `HttpClient` with it.

Tip It's perfectly allowable and encouraged, even, to create multiple instances of `HttpClient` when you need different filters and configurations for different services. There is no penalty in doing so, and it can greatly simplify your programming model.

The filter is essentially a black box that takes an HTTP request and produces an HTTP response—refer to Figure 4-5 again for its place in the whole process. Within the filter you can handle details like credentials, proxies, certificates, and redirects, as well as implement retry mechanisms, caching, logging, and so forth. This keeps all those details in a central place underneath the `HttpClient` APIs such that you don't have to bother with them in the code surrounding `HttpClient` calls.

With cache control, a filter contains a `cacheControl` property that can be set to an instance of the `HttpCacheControl` class. This object has two properties, `readBehavior` and `writeBehavior`, which determine how caching is applied to requests going through this filter. For reading, `readBehavior` is set to a value from the `HttpCacheReadBehavior` enumeration: `default` (works like a web browser), `mostRecent` (does an if-modified-since exchange with the server), and `onlyFromCache` (for offline use). For writing, `writeBehavior` can be a value from `HttpCacheWriteBehavior`, which supports `default` and `noCache`.

Managing cookies happens on the level of the filter as well. By default—through the `HttpBaseProtocolFilter`—the `HttpClient` automatically reads incoming set-cookie headers, saves the resulting cookies as needed, and then adds cookies to outgoing headers as appropriate. To access these cookies, create the `HttpClient` with an instance of `HttpBaseProtocolFilter`. Then you can access the filter's `cookieManager` property (that sounds like a nice job!). This property is an instance of `HttpCookieManager` and has three methods: `getCookies`, `setCookie`, and `deleteCookie`. These allow you to examine specific cookies to be sent for a request or to delete specific cookies for privacy concerns.

Cookie behavior in general follows the same patterns as the browser. A cookie will persist across app sessions depending on the normal cookie rules: the cookie must be marked as persistent, is subject to the normal per-app limitations, and so on. Also note that cookies are isolated between apps for normal security reasons, even if those apps are using the same online resource.

For additional thoughts on `HttpClient`, refer to Updating Your JavaScript Apps to Use the New Windows Web HTTP API on the Windows App Builder blog. Demonstrations of the API, including filtering, can then be found in the HttpClient sample in the Windows SDK. Here's a quick run-down of what its scenarios demonstrate:

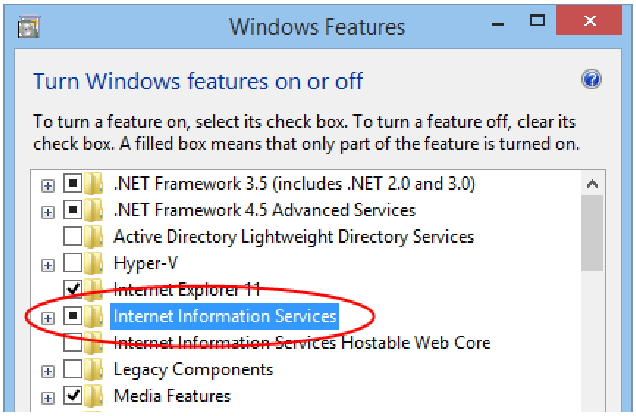
- **Scenarios 1–3** GET requests for text (with cache control), stream, and an XML list.
- **Scenarios 4–7** POST requests for text, stream, multipart MIME form, and a stream with progress.
- **Scenarios 8–10** Getting, setting, and deleting cookies.
- **Scenario 11** A metered connection filter that implements cost awareness on the level of the filter.
- **Scenario 12** A retry filter that automatically handles 503 errors with Reply-After headers.

To run this sample you must first set up a `localhost` server along with a data file and an upload target page. To do this, make sure you have Internet Information Services installed on your machine, as described below in "Sidebar: Using the Localhost." Then, from an administrator command prompt, navigate to the sample's Server folder and run the command **powershell –file setupserver.ps1**. This will install the necessary server-side files for the sample on the localhost (*c:\inetpub\wwwroot*).

Sidebar: Using the Localhost

The localhost is a server process that runs on your local machine, making it possible to debug both sides of client-server interactions. For this you can use a server like Apache or you can use the solution that's built into Windows and integrated with the Visual Studio tools: Internet Information Services (IIS).

To turn on IIS in Windows, go to Control Panel > Programs and Features > Turn Windows Features On or Off. Check the Internet Information Services box at the top level, as shown below, to install the core features:



Turn Windows features on or off

To turn a feature on, select its check box. To turn a feature off, clear its check box. A filled box means that only part of the feature is turned on.

⊞ ■ .NET Framework 3.5 (includes .NET 2.0 and 3.0)
⊞ ■ .NET Framework 4.5 Advanced Services
□ Active Directory Lightweight Directory Services
⊞ □ Hyper-V
☑ Internet Explorer 11
⊞ ■ Internet Information Services
□ Internet Information Services Hostable Web Core
⊞ □ Legacy Components
⊞ ☑ Media Features

Once IIS is installed, the local site addressed by `http://localhost/` is found in the folder *c:\inetpub\wwwroot*. That's where you drop any server-side page you need to work with.

With that page running on the local machine, you can hook it into whatever tools you have available for server-side debugging. Here it's good to know that access to localhost URIs—also known as local loopback—is normally blocked for Windows Store apps unless you're on a machine with a developer license, which you are if you're been running Visual Studio or Blend. This won't be true for your customer's machines, though! In fact, the Windows Store will reject apps that attempt to do so.[37]

To install other server-side features on IIS, like PHP or Visual Studio Express for Web (which allows you to debug web pages), use Microsoft's Web platform installer. We'll make use of these when we work with live tiles in Chapter 16.

Suspend and Resume with Online Content

Now that we've seen the methods for making HTTP requests to any URI, you really have the doors of the web wide open to you. As many web APIs provide REST interfaces, interacting with them is just a matter of putting together the proper HTTP requests as defined by the API documentation. I must leave such details up to you because processing that data within your app has little to do with the Windows platform (except for creating UI with collection controls, but that's for a later chapter).

Instead, what concerns us here are the implications of suspend and resume. In particular, an app cannot predict how long it will stay suspended before being resumed or before being terminated and restarted.

In the first case, an app that gets resumed will have all its previous data still in memory. It very much needs to decide, then, whether that data has become stale since the app was suspended and whether sessions with other servers have exceeded their timeout periods. You can also think of it this way: after what period of time will users not remember nor care what was happening the last time they saw your app? If it's a week or longer, it might be reasonable to resume or restart in a default state. Then again, if you pick up right back where they were, users gain increasing confidence that they *can* leave apps running for a long time and not lose anything. Or you can compromise and give the user options to choose from. You'll have to think through your scenarios, of course, but if there's any doubt, resume where the app left off.

To check elapsed time, save a timestamp on suspend (from `new Date().getTime()`), get another timestamp in the `resuming` event, take the difference, and compare that against your desired refresh period. A Stock app, for example, might have a very short period. With the Windows App Builder blog, on the other hand, new posts don't show up more than once per day, so a much longer period on the

[37] Visual Studio enables local loopback by default for a project. To change it, right-click the project in Solution Explorer, select Properties, select Configuration Properties > Debugging on the left side of the dialog, and set Allow Local Network Loopback to No. For more on the subject of loopback, see How to enable loopback and troubleshoot network isolation.

order of hours is sufficient to keep up-to-date and to catch new posts within a reasonable timeframe.

This is implemented in SimpleXhr2 by first placing the getStringAsync call into a separate function called downloadPosts, which is called on startup. Then we register for the resuming event with WinRT:

```
Windows.UI.WebUI.WebUIApplication.onresuming = function () {
    app.queueEvent({ type: "resuming" });
}
```

Remember how I said in Chapter 3, "App Anatomy and Performance Fundamentals," we could use WinJS.Application.queueEvent to raise our own events to the app object? Here's a great example. WinJS.Application doesn't automatically wrap the resuming event because it has nothing to add to that process. But the code above accomplishes exactly the same thing, allowing us to register an event listener right alongside other events like checkpoint:

```
app.oncheckpoint = function (args) {
    //Save in sessionState in case we want to use it with caching
    app.sessionState.suspendTime = new Date().getTime();
};

app.addEventListener("resuming", function (args) {
    //This is a typical shortcut to either get a variable value or a default
    var suspendTime = app.sessionState.suspendTime || 0;

    //Determine how much time has elapsed in seconds
    var elapsed = ((new Date().getTime()) - suspendTime) / 1000;

    //Refresh the feed if > 1 hour (or use a small number for testing)
    if (elapsed > 3600) {
        downloadPosts();
    }
});
```

To test this code, run it in Visual Studio's debugger and set breakpoints within these events. Then click the suspend button in the toolbar, and you should enter the checkpoint handler. Wait a few seconds and click the resume button (play icon), and you should be in the resuming handler. You can then step through the code and see that the elapsed variable will have the number of seconds that have passed, and if you modify that value (or change 3600 to a smaller number), you can see it call downloadPosts again to perform a refresh.

What about launching from the previously terminated state? Well, if you didn't cache any data from before, you'll need to refresh it again anyway. If you do cache some of it, your saved state (including the timestamp) helps you decide whether to use the cache or simply load data anew. You can also take a hybrid approach of drawing on your own cache as much as you can and then updating it with whatever new data comes from the service.

What helps in this context is that you can ask Windows to prefetch the responses for various URIs, such that when you make the request it is fulfilled from that prefetch cache. And that's our next topic.

Prefetching Content

HTTP requests made through the `XMLHttpRequest`, `WinJX.xhr`, and `HttpClient` APIs all interoperate with the internet cache, such that repeated requests for the same remote resource, whether from an app or Internet Explorer, can be fulfilled from the cache. (`HttpClient` also gives you control over how the cache is used.) Caching works great for offline scenarios and improving performance generally.[38]

One of the first things that many connected apps do upon launch is to make HTTP requests for their home page content. If such a request has not been made previously, however, or if the response data has changed since the last request, the user will have to wait for that data to arrive. This clearly affects the app's startup performance. What would really help, then, is having a way to get that content into the internet cache before the app makes the request directly.

Apps can do this by asking Windows to *prefetch* online content (any kind of data) into the cache, which will take place even when the app itself isn't running. Of course, Windows won't just fulfill such requests indiscriminately, so it applies these limits:

- Prefetching happens only when power and network conditions are met (Windows won't prefetch on metered networks or when battery power is low).
- Prefetching is prioritized for apps that the user runs most often.
- Prefetching is prioritized for content that apps actually request later on. That is, if an app makes a prefetch request but seldom asks for it, the likelihood of the prefetch decreases.
- Windows limits the overall number of requests to 40.
- Resources are cached only for the length of time indicated in the response headers.

In other words, apps don't have control over *whether* their prefetching requests are fulfilled—Windows optimizes the process so that users see increased performance for the apps they use and the content they access most frequently. Apps simply continue to make HTTP requests, and if prefetching has taken place those requests will just be fulfilled right away without hitting the network.

There are two ways to make prefetching requests. The first is to insert `Windows.Foundation.Uri` objects into the `Windows.Networking.BackgroundTransfer.ContentPrefetcher.contentUris` collection. This collection is a vector (see Chapter 6), so you use methods like `append` to add the URIs and `removeAt` to delete them. Note that you can modify this list both from the running app and from a background task. The latter especially lets you periodically refresh the list without having the user run the app.

Here's a quick example from scenario 1 of the ContentPrefetcher sample (js/S1-direct-content-

[38] For readers familiar with .NET languages, note that the .NET `System.Net.HttpClient` API does *not* benefit from the cache or precaching.

uris.js, with some error handling omitted):

```
uri = new Windows.Foundation.Uri(uriToAdd);
Windows.Networking.BackgroundTransfer.ContentPrefetcher.contentUris.append(uri);
```

The second means is to give the prefetcher the URI of an XML file (local or remote) that contains your list. You store this in the `ContentPrefetcher.indirectContentUri` property, as shown in scenario 2 of the sample (js/S2-indirect-content-uri.js).

```
uri = new Windows.Foundation.Uri("http://example.com/prefetchlist.xml");
Windows.Networking.BackgroundTransfer.ContentPrefetcher.indirectContentUri = uri;
```

This allows your service to maintain a dynamic list of URIs (like those of a news feed) such that your prefetching stays very current. The XML in this case should be structured as follows, with as many URIs as are needed (the exact schema is on the `indirectContentUri` page linked above):

```
<?xml version="1.0" encoding="utf-8"?>
<prefetchUris>
  <uri>http://example.com/2013-02-28-headlines.json</uri>
  <uri>http://example.com/2013-02-28-img1295.jpg</uri>
  <uri>http://example.com/2013-02-28-img1296.jpg</uri>
  <uri>http://example.com/2013-02-28-ad_config.xml</uri>
</prefetchUris>
```

> **Note** Prefetch requests will include *X-MS-RequestType: Prefetch* in the headers if services need to differentiate the request from others. Existing cookies will also be included in the request, but beyond that there are no provisions for authentication.

Lastly, the `ContentPrefetcher.lastSuccessfulPrefetchTime` property tells you just how fresh the content really is. Scenario 3 of the sample retrieves this timestamp (js/S3-last-prefetch-time.js):

```
var lastPrefetchTime =
    Windows.Networking.BackgroundTransfer.ContentPrefetcher.lastSuccessfulPrefetchTime;
```

You can use this to decide whether you still want to make a direct request, in which case you'll need to use the `HttpCacheReadBehavior.mostRecent` flag with the `HttpClient` object's `CacheControl` to make sure you have the latest data. Note that you must use `HttpClient` rather than `WinJS.xhr` to exercise this degree of control.

Background Transfer

A common use of HTTP requests is to transfer potentially large files to and from an online repository. For even moderately sized files, however, this presents a challenge: very few users typically want to stare at their screen to watch file transfer progress, so it's highly likely that they'll switch to another app to do something far more interesting while the transfer is taking place. In doing so, the app that's doing the transfer will be suspended and possibly even terminated. This does not bode well for trying to complete such operations using a mechanism like `HttpClient`!

One solution would be to provide a background task for this purpose, which was a common request with early previews of Windows 8. However, there's little need to run app code for this common purpose, so WinRT provides a specific API, `Windows.Networking.BackgroundTransfer` (which includes the prefetcher, as we just saw at the end of the previous section). This API supports up to 500 scheduled transfers systemwide and typically runs five transfers in parallel. It offers built-in cost awareness and resiliency to changes in connectivity (switching seamlessly to the user's preferred network), relieving apps from such concerns. Transfers continue when an app is suspended and will be paused if the app is terminated (including if the user terminates the app with a gesture or Alt+F4), except for uploads (HTTP POST) which cannot be paused. When the app is resumed or launched again, it can then check the status of background transfers it previously initiated and take further action as necessary—processing downloaded information, noting successful uploads in its UI (issuing toasts and tile updates is built into the API), and enumerating pending transfers, which will restart any that were paused or otherwise interrupted.

In short, use the background transfer API whenever you expect the operation to exceed your customer's tolerance for waiting. This clearly depends on the network's connection speed and whether you think the user will switch away from your app while such a transfer is taking place. For example, if you initiate a transfer operation but the user can continue to be productive (or entertained) in your app while that's happening, using HTTP requests directly might be a possibility, though you'll still be responsible for cost awareness and handling connectivity. If, on the other hand, the user cannot do anything more until the transfer is complete, you might choose to use background transfer for perhaps any data larger than 500K or some other amount based on the current network speed.

In any case, when you're ready to employ background transfer in your app, the `BackgroundDownloader` and `BackgroundUploader` objects will become your fast friends. Both objects have methods and properties through which you can enumerate pending transfers as well as perform general configuration of credentials, HTTP request headers, transfer method, cost policy (for metered networks), and grouping. Each individual operation is then represented by a `DownloadOperation` or `UploadOperation` object, through which you can control the operation (pause, cancel, etc.) and retrieve status. With each operation you can also set priority, credentials, cost policy, and so forth, overriding the general settings in the `BackgroundDownloader` and `BackgroundUploader` classes. Both operation classes also have a constructor to which you can pass a `StorageFile` that contains a request body, in case the service you're working with requires something like form data for the transfer.

Note In both download and upload cases, the connection request will be aborted if a new TCP/SSL connection is not established within five minutes. Once there's a connection, any other HTTP request involved with the transfer will time out after two minutes. Background transfer will retry an operation up to three times if there's connectivity and will defer retries if there's no connectivity.

One of the primary reasons why we have the background transfer API is to allow Windows to automatically manage transfers according to systemwide considerations. Changes in network cost, for example, can cause some transfers to be paused until the device returns to an unlimited network. To save battery power, long-running transfers can be slowed (throttled) or paused altogether, as when the system goes into standby. In the latter case, apps can keep the process going by requesting an *unconstrained transfer*. This way a user can let a very large download run all day, if desired, rather than coming back some hours later only to find that the transfer was paused. (Note that a user consent prompt appears if the device is on battery power.)

To see the background transfer API in action, let's start by looking at the Background transfer sample. Note that this sample depends on having the localhost set up on your machine as we did with the HttpClient sample earlier. Refer back to "Sidebar: Using the localhost" for instructions, and be sure to run **powershell –file setupserver.ps1** in the sample's Server folder to set up the necessary files.

Basic Downloads

Scenario 1 (js/downloadFile.js) of the Background transfer sample lets you download any file from the localhost server and save it to the Pictures library. By default the URI entry field is set to a specific localhost URI and the control is disabled. This is because the sample doesn't perform any validation on the URI, a process that you should always perform in your own app. If you'd like to enter other URIs in the sample, of course, just remove `disabled="disabled"` from the *serverAddressField* element in html/downloadFile.html.

By default, scenario 1 here makes a request to *http://localhost/BackgroundTransferSample/download.aspx*, which serves up a stream of 5 million 'a' characters. The sample saves this content in a text file, so you won't see any image showing up on the display, but you will see progress. Change the URI to an image file[39] and you'll see that image appear on the display. (You can also copy an image file to *c:\inetpub\wwwroot* and point to it there.) Note that you can kick off multiple transfers to observe how they are all managed simultaneously; the cancel, pause, and resume buttons help with this.

Three flavors of download are supported in the WinRT API and reflected in the sample:

- A normal download at normal priority. Such a transfer continues to run when the app is suspended, but if it's a long transfer it could be slowed (throttled) or paused depending on system conditions like battery life and network type. The system supports up to five parallel transfers at normal priority.
- A normal download at high priority. Typically an app will set its most important download at a higher priority than others it starts at the same time. A high-priority transfer will start even if there are already five normal-priority downloads running. If you schedule multiple high-priority transfers, up to six of them will run in parallel (one plus replacing all of the five normal-priority downloads) until the high-priority queue is cleared, then normal-priority transfers are resumed.
- An *unconstrained* download at either priority. As noted before, an unconstrained download will continue to run (subject to user consent) even in modes like connected standby. You use this feature in scenarios where you know the user would want a transfer to continue possibly for a

[39] Might I suggest http://kraigbrockschmidt.com/images/photos/kraigbrockschmidt-dot-com-122-10-S.jpg?

long period of time and not have it interrupted or paused.

Starting a download happens as follows. First create a StorageFile to receive the data (though this is not required, as we'll see later in this section). Then create a DownloadOperation object for the transfer using BackgroundDownloader.createDownload, to which you pass the URI of the data, the StorageFile in which to store it, and an optional StorageFile containing a request body to send to the server when starting the transfer (more on this later). In the operation object you can then set its priority, method, costPolicy, and transferGroup properties to override the defaults supplied by the BackgroundDownloader. The priority is a BackgroundTransferPriority value (default or high), and method is a string that identifies the type transfer being used (normally GET for HTTP or RETR for FTP). We'll come back to the other two properties later in the "Setting Cost Policy" and "Grouping Transfers" sections.

Once the operation is configured as needed, the last step is to call its startAsync method, which returns a promise to which you attach your completed, error, and progress handlers via then or done. Here's code from js/downloadFile.js:[40]

```
// Asynchronously create the file in the pictures folder (capability declaration required).
Windows.Storage.KnownFolders.picturesLibrary.createFileAsync(fileName,
    Windows.Storage.CreationCollisionOption.generateUniqueName)
    .done(function (newFile) {
        // Assume uriString is the text URI of the file to download
        var uri = Windows.Foundation.Uri(uriString);
        var downloader = new Windows.Networking.BackgroundTransfer.BackgroundDownloader();

        // Create a new download operation.
        var download = downloader.createDownload(uri, newFile);

        // Start the download
        var promise = download.startAsync().done(complete, error, progress);
    }
```

While the operation underway, the following properties provide additional information on the transfer:

- requestedUri and resultFile The same as those passed to createDownload.
- guid A unique identifier assigned to the operation.
- progress A BackgroundDownloadProgress structure with bytesReceived, totalBytesToReceive, hasResponseChanged (a Boolean, see the getResponseInformation method below), hasRestarted (a Boolean set to true if the download had to be restarted), and status (a BackgroundTransferStatus value: idle, running, pausedByApplication, pausedCostedNetwork, pausedNoNetwork, canceled, error, and completed).

[40] The code in the sample has more structure than shown here. It defines its own DownloadOperation class that unfortunately has the same name as the WinRT class, so I'm electing to omit mention of it.

A few methods of DownloadOperation can also be used with the transfer:

- pause and resume Control the download in progress. We'll talk more of these in the "Suspend, Resume, and Restart with Background Transfers" section below.
- getResponseInformation Returns a ResponseInformation object with properties named headers (a collection of response headers from the server), actualUri, isResumable, and statusCode (from the server). Repeated calls to this method will return the same information until the hasResponseChanged property is set to true.
- getResultStreamAt Returns an IInputStream for the content downloaded so far or the whole of the data once the operation is complete.

In scenario 1 of the sample, the progress function—which is given to the promise returned by startAsync—uses getResponseInformation and getResultStreamAt to show a partially downloaded image:

```
var currentProgress = download.progress;

// ...

// Get Content-Type response header.
var contentType = download.getResponseInformation().headers.lookup("Content-Type");

// Check the stream is an image.
if (contentType.indexOf("image/") === 0) {
    // Get the stream starting from byte 0.
    imageStream = download.getResultStreamAt(0);

    // Convert the stream to a WinRT type
    var msStream = MSApp.createStreamFromInputStream(contentType, imageStream);
    var imageUrl = URL.createObjectURL(msStream);

    // Pass the stream URL to the HTML image tag.
    id("imageHolder").src = imageUrl;

    // Close the stream once the image is displayed.
    id("imageHolder").onload = function () {
        if (imageStream) {
            imageStream.close();
            imageStream = null;
        }
    };
}
```

All of this works because the background transfer API is saving the downloaded data into a temporary file and providing a stream on top of that, hence a function like URL.createObjectURL does the same job as if we provided it with a StorageFile object directly. Once the DownloadOperation object goes out of scope and is garbage collected, however, that temporary file will be deleted.

The existence of this temporary file is also why, as I noted earlier, it's not actually necessary to provide a StorageFile object in which to place the downloaded data. That is, you can pass null as the second argument to createDownload and work with the data through DownloadOperation.getResultStreamAt. This is entirely appropriate if the ultimate destination of the data in your app isn't a separate file.

As mentioned earlier, there is a variation of createDownload that takes a second StorageFile argument whose contents provide the body of the HTTP GET or FTP RETR request that will be sent to the server URI before the download is started. This accommodates some websites that require you to fill out a form to start the download. Similarly, createDownloadAsync supplies the request body through an IInputStream instead of a file, if that's better suited to your needs.

Sidebar: Where Is Cancel?

You might have already noticed that neither DownloadOperation nor UploadOperation have cancellation methods. So how is this accomplished? You cancel the transfer by canceling the startAsync operation, which means calling the cancel method of the *promise* returned by startAsync. Thus, you need to hold on to the promises for each transfer you initiate if you want to possibly cancel them later on.

Requesting an Unconstrained Download

To request an unconstrained download, you use pretty much the same code as in the previous section except for one additional step. With the DownloadOperation from BackgroundDownloader.createDownload, don't call startAsync right away. Instead, place that operation object (and others, if desired) into an array, then pass that array to BackgroundDownloader.requestUnconstrainedDownloadsAsync. This async function will complete with an UnconstrainedTransferRequestResult object, whose single isContrained member will tell you whether the request was granted. Here's the code from the sample for that case (js/downloadFile.js):

```
Windows.Networking.BackgroundTransfer.BackgroundDownloader
    .requestUnconstrainedDownloadsAsync(requestOperations)
.done(function (result) {
    printLog("Request for unconstrained downloads has been " +
        (result.isUnconstrained ? "granted" : "denied") + "<br/>");

    promise = download.startAsync().then(complete, error, progress);
}, error);
```

As you can see, you still call startAsync after making the request, which the sample here does regardless of the request result. In your own app, however, you can make other decisions, such as setting a higher priority for the download even if the request was denied.

Basic Uploads

Scenario 2 (js/uploadFile.js) of the Background transfer sample exercises the background upload capability, specifically sending some file (chosen through the file picker) to a URI that can receive it. By default the URI points to *http://localhost/BackgroundTransferSample/upload.aspx*, a page installed with the PowerShell script that sets up the server. As with scenario 1, the URI entry control is disabled because the sample performs no validation, as you would again always want to do if you accepted any URI from an untrusted source (user input in this case). For testing purposes, of course, you can remove `disabled="disabled"` from the *serverAddressField* element in html/uploadFile.html and enter other URIs that will exercise your own upload services. This is especially handy if you run the server part of the sample in Visual Studio Express for Web where the URI will need a localhost port number as assigned by the debugger.

In addition to a button to start an upload and to cancel it, the sample provides another button to start a *multipart* upload. For more on breaking up large files and multipart uploads, see Appendix C.

In code, an upload happens very much like a download. Assuming you have a `StorageFile` with the contents to upload, create an `UploadOperation` object for the transfer with `BackgroundUploader.createUpload`. If, on the other hand, you have data in a stream (`IInputStream`), create the operation object with `BackgroundUploader.createUploadFromStreamAsync` instead. This can also be used to break up a large file into discrete chunks, if the server can accommodate it; see "Breaking Up Large Files" in Appendix C.

With the operation object in hand, you can customize a few properties of the transfer, overriding the defaults provided by the `BackgroundUploader`. These are the same as for downloads: `priority`, `method` (HTTP POST or PUT, or FTP STOR), `costPolicy`, and `transferGroup`. For the latter two, again see "Setting Cost Policy" and "Grouping Transfers" below.

Once you're ready, the operation's `startAsync` starts the upload:[41]

```
// Assume uri is a Windows.Foundation.Uri object and file is the StorageFile to upload
var uploader = new Windows.Networking.BackgroundTransfer.BackgroundUploader();
var upload = uploader.createUpload(uri, file);
promise = upload.startAsync().then(complete, error, progress);
```

While the operation is underway, the following properties provide additional information on the transfer:

- `requestedUri` and `sourceFile`   The same as those passed to `createUpload` (an operation created with `createUploadFromStreamAsync` supports only `requestedUri`).
- `guid`   A unique identifier assigned to the operation.
- `progress`   A `BackgroundUploadProgress` structure with `bytesReceived`,

[41] As with downloads, the code in the sample has more structure than shown here and again defines its own `UploadOperation` class with the same name as the one in WinRT, so I'm omitting mention of it.

totalBytesToReceive, bytesSent, totalBytesToSend, hasResponseChanged (a Boolean, see the getResponseInformation method below), hasRestarted (a Boolean set to true if the upload had to be restarted), and status (a BackgroundTransferStatus value, again with values of idle, running, pausedByApplication, pausedCostedNetwork, pausedNoNetwork, canceled, error, and completed).

Unlike a download, an UploadOperation does not have pause or resume methods but does have the same getResponseInformation and getResultStreamAt methods. In the upload case, the response from the server is less interesting because it doesn't contain the transferred data, just headers, status, and whatever body contents the upload page cares to return. If that page returns some interesting HTML, though, you might use the results as part of your app's output for the upload.

As noted before, to cancel an UploadOperation, call the cancel method of the promise returned from startAsync. You can also see that the BackgroundUploader also has a requestUnconstrainedUploadsAsync method like that of the downloader, to which you can pass an array of UploadOperation objects for the request. Again, the result of the request tells you whether or not the request was granted, allowing you to decide what you might want to change before calling each operation's startAsync.

Completion and Error Notifications

With long transfer operations, users typically want to know when those transfers are complete or if an error occurred along the way. However, those transfers might finish or fail while the app is suspended, so the app itself cannot directly issue such notifications. For this purpose, the app can instead supply toast notifications and tile updates to the BackgroundDownloader and BackgroundUploader classes. Notice that you're not setting notifications on individual *operation* objects, which means that the content of these notifications should describe all active transfers as a whole. If you have only a single transfer, then of course your language can reflect that, but otherwise you'll want to be more generic with messages like "Your new photo gallery of 108 images has finished uploading."

The downloader and uploader objects each have four different notification objects you can set:

- successToastNotification and failureToastNotification Instances of the Windows.UI.Notifications.ToastNotification class.
- successTileNotification and failureTileNotification Instances of the Windows.UI.Notification.TileNotification class.

For details on using these classes, including all the different templates you can use, refer to Chapter 16. Basically you create these instances as if you intend to issue notifications directly from the app, but hand them off to the downloader and uploader objects so that they can do it on your behalf.

Providing Headers and Credentials

Within the BackgroundDownloader and BackgroundUploader you have the ability to set values for individual HTTP headers by using their setRequestHeader methods. Both take a header name and a value, and you call them multiple times if you have more than one header to set.

Similarly, both the downloader and uploader objects have two properties for credentials: serverCredential and proxyCredential, depending on the needs of your server URI. Both properties are Windows.Security.Credentials.PasswordCredential objects. As the purpose in a background transfer operation is to provide credentials to the server, you'd typically create a PasswordCredential as follows:

```
var cred = new Windows.Security.Credentials.PasswordCredential(resource, userName, password);
```

where the resource in this case is just a string that identifies the resource to which the credentials applies. This is used to manage credentials in the credential locker, as we'll see in the "Authentication, the Microsoft Account, and the User Profile" section later. For now, just creating a credential in this way is all you need to authenticate with your server when doing a transfer.

Note At present, setting the serverCredential property doesn't work with URIs that specify an FTP server. To work around this, include the credentials directly in the URI with the form *ftp://<user>:<password>@server.com/file.ext* (for example, ftp://admin:password1@server.com/file.bin).

Setting Cost Policy

As mentioned earlier in the "Cost Awareness" section, the Windows Store policy requires that apps are careful about performing large data transfers on metered networks. The Background Transfer API takes this into account, based on values from the BackgroundTransferCostPolicy enumeration:

- default Allow transfers on costed networks.
- unrestrictedOnly Do not allow transfers on costed networks.
- always Always download regardless of network cost.

To apply a policy to subsequent transfers, set the value of BackgroundDownloader.costPolicy and/or BackgroundUploader.costPolicy. The policy for individual operations can be set through the DownloadOperation.costPolicy and UploadOperation.costPolicy properties.

Basically, you would change the policy if you've prompted the user accordingly or allow them to set behavior through your settings. For example, if you have a setting to disallow downloads or uploads on a metered network, you'd set the general costPolicy to unrestrictedOnly. If you know you're on a network where roaming charges would apply and the user has consented to a transfer, you'd want to change the costPolicy of that *individual* operation to always. Otherwise the API would not perform the transfer because doing so on a roaming network is disallowed by default.

When a transfer is blocked by policy, the operation's `progress.status` property will contain `BackgroundTransferStatus.pausedCostedNetwork`.

Grouping Transfers

Grouping multiple transfers together lets you enumerate and control related transfers. For example, a photo app that organizes pictures into albums or album pages can present a UI through which the user can pause, resume, or cancel the transfer of an entire album, rather than working on the level of individual files. The grouping features of the background transfer API makes the implementation of this kind of experience much easier, as the app doesn't need to maintain its own grouping structures.

> **Note** Grouping has no bearing on the individual transfers themselves, nor is grouping information communicated to servers. Grouping is simply a client-side management mechanism.

Grouping is set through the `transferGroup` property that's found in the `BackgroundDownloader`, `BackgroundUploader`, `DownloadOperation`, and `UploadOperation` objects. This property is a `BackgroundTransferGroup` object created through the static `BackgroundTransferGroup.createGroup` method using whatever name you want to use for that group. Note that the `transferGroup` property can be set only through `BackgroundDownloader` and `BackgroundUploader`; you would assign this prior to creating a series of individual operations in that group. Each individual operation object will then have that same `transferGroup` as a read-only property.

In addition to its assigned `name`, a `transferGroup` object has a `transferBehavior` property, which is a value from the `BackgroundTransferBehavior` enumeration. This allows you to control whether the operations in the group happen serially or in parallel. A video player for a TV series, for example, could place all the episodes in the same group and then set the behavior to `BackgroundTransferBehavior.serialized`. This ensures that the group's operations are done one at a time, reflecting how the user is likely to consume that content. A photo gallery app that download a composite page of large images, on the other hand, might use `BackgroundTransferBehavior.parallel` (the default). As for pausing, resuming, and cancelling groups, that's best discussed in the context of app lifecycle events, which is the subject of the next section.

Suspend, Resume, and Restart with Background Transfers

Earlier I mentioned that background transfers will continue while an app is suspended, and paused if the app is terminated by the system. Because apps will be terminated only in low-memory conditions, it's appropriate to also pause background transfers in that case.

When an app is resumed from the suspended state, it can check on the status of pending transfers by using the `BackgroundDownloader.getCurrentDownloadsAsync` and `BackgroundUploader.getCurrentUploadsAsync` methods. To limit that list to a specific `transferGroup`, use the `getCurrentDownloadsForTransferGroupAsync` and `getCurrentUploadsForTransferGroupAsync` methods instead.[42]

The list that comes back from these methods is a vector of DownloadOperation and UploadOperation objects, which can be iterated like an array:

```
Windows.Networking.BackgroundTransfer.BackgroundDownloader.getCurrentDownloadsAsync()
    .done(function (downloads) {
        for (var i = 0; i < downloads.size; i++) {
            var download = downloads[i];
        }
    });

Windows.Networking.BackgroundTransfer.BackgroundUploader.getCurrentUploadsAsync()
    .done(function (uploads) {
         for (var i = 0; i < uploads.size; i++) {
             var upload = uploads[i];
         }
    });
```

In each case, the progress property of each operation will tell you how far the transfer has come along. The progress.status property is especially important. Again, status is a BackgroundTransferStatus value and will be one of idle, running, pausedByApplication, pausedCostedNetwork, pausedNoNetwork, canceled, error, and completed). These are clearly necessary to inform users, as appropriate, and to give them the ability to restart transfers that are paused or experienced an error, to pause running transfers, and to act on completed transfers.

Speaking of which, when using the background transfer API, an app should always give the user control over pending transfers. Downloads can be paused through the DownloadOperation.pause method and resumed through DownloadOperation.resume. (There are no equivalents for uploads.) Download and upload operations are canceled by canceling the promises returned from startAsync. Again, if you requested a list of transfers for a particular group, iterate over the results to affect the operations in that group.

This brings up an interesting situation: if your app has been terminated and later restarted, how do you restart transfers that were paused? The answer is quite simple. By enumerating transfers through getCurrentDownloads[ForTransferGroup]Async and getCurrentUploads[ForTransferGroup]-Async, incomplete transfers are automatically restarted. But then how do you retrieve the promises originally returned by the startAsync methods? Those are not values that you can save in your app state and reload on startup, and yet you need them to be able to cancel those operations, if necessary, and also to attach your completed, error, and progress handlers.

For this reason, both DownloadOperation and UploadOperation objects provide a method called attachAsync, which returns a promise for the operation just like startAsync did originally. You can then call the promise's then or done methods to provide your handlers:

[42] The optional group argument for the other methods is obsolete and replaced with these that work with a transferGroup argument.

```
promise = download.attachAsync().then(complete, error, progress);
```

and call `promise.cancel` if needed. In short, when Windows restarts a background transfer and essentially calls `startAsync` on your app's behalf, it holds that promise internally. The `attachAsync` methods simply return that new promise.

Authentication, the Microsoft Account, and the User Profile

If you think about it, just about every online resource in the world has some kind of credentials or authentication associated with it. Sure, we can read many of those resources without credentials, but having permission to upload data to a website is more tightly controlled, as is access to one's account or profile in a database managed by a website. In many scenarios, then, apps need authenticate with services in some way, using service-specific credentials or perhaps using accounts from other providers like Facebook, Twitter, Microsoft, and so on.

There are two approaches for dealing with credentials. First, you can collect credentials directly through your own UI, which means the app is fully responsible for protecting those credentials. For this there are a number of design guidelines for different login scenarios, such as when an app requires a login to be useful and when a login is simply optional. These topics, as well as where to place login and account/profile management UI, are discussed in Guidelines for login controls.

For storage purposes, the Credential Locker API in WinRT will help you out here—you can securely save credentials when you collect them and retrieve them in later sessions so that you don't have to pester the user again. Transmitting those credentials to a server, on the other hand, will require encryption work on your part, and there are many subtleties that can get complicated. For a few notes on encryption APIs in WinRT, as well as a few other security matters, see Appendix C.

The simpler and more secure approach—one that we highly recommend—is to use the Web Authentication Broker API. This lets the user authenticate directly with a server in the broker's UI, keeping credentials entirely on the server, after which the app receives back a token to use with later calls to the service. The Web Authentication Broker works with any service that's been set up as a provider. This can be your own service, as we'll see, or an OAuth/OpenID provider.

Tip When thinking about providers that you might use for authentication, remember that non-domain-joined users sign into Windows with a Microsoft account to begin with. If you can leverage that Microsoft account with your own services, signing into Windows means they won't have to enter any additional credentials or create a separate account for your service, providing a delightfully transparent experience. The Microsoft account also provides access to other features, as we'll see in "Using the Microsoft Account" later on.

One of the significant benefits of the Web Authentication Broker is that authentication for any given service transfers across apps as well as websites, providing a very powerful *single sign-on* experience for users. That is, once a user signs in to a service—either in the browser or in an app that uses the broker—they're already signed into other apps and sites that use that same service (again, signing into Windows with a Microsoft account also applies here). To make the story even better, those credentials also roam across the user's trusted devices (unless they opt out) so that they won't even have to authenticate again when they switch machines. Personally I've found this marvelously satisfying—when setting up a brand new device, for example, all those credentials are immediately in effect!

Sidebar: User Verification via Fingerprints

If your scenario calls for verification that the user that is logged in to the current Microsoft account is physically present (as opposed to someone who just happens to know a password), check out the API in Windows.Security.Credentials.UI.UserConsentVerifier. This object has just two methods, checkAvailabilityAsync and requestVerificationAsync, and yet provides one of the strongest methods to authenticate a specific human being. For more details, see "Fingerprint (Biometric) Readers" in Chapter 17, "Devices and Printing."

The Credential Locker

One of the reasons that apps might repeatedly ask a user for credentials is simply because they don't have a truly secure place to store and retrieve those credentials that's also isolated from all other apps. This is entirely the purpose of the credential locker, a function that's also immediately clear from the name of this particular API: Windows.Security.Credentials.PasswordVault. It's designed to store credentials, of course, but you can use it to store other things like tokens as well.

With the locker, any given credential itself is represented by a PasswordCredential object, as we saw briefly with the background transfer API. You can create an initialized credential as follows:

```
var cred = new Windows.Security.Credentials.PasswordCredential(resource, userName, password);
```

Another option is to create an uninitialized credential and set its properties individually:

```
var cred = new Windows.Security.Credentials.PasswordCredential();
cred.resource = "userLogin"
cred.userName = "username";
cred.password = "password";
```

A credential object also contains an IPropertySet value named properties, through which the same information can be managed.

In any case, when you collect credentials from a user and want to save them, create a PasswordCredential and pass it to PasswordVault.add:

```
var vault = new Windows.Security.Credentials.PasswordVault();
vault.add(cred);
```

Note that if you add a credential to the locker with a resource and userName that already exist, the new credential will replace the old. And if at any point you want to delete a credential from the locker, call the PasswordVault.remove method with that credential.

Furthermore, even though a PasswordCredential object sees the world in terms of usernames and passwords, that password can be anything you need to store securely, such as an access token. As we'll see in the next section, authentication through OAuth providers might return such a token, in which case you store something like "Facebook_Token" in the credential's resource property, your app name in userName, and the token in password. This is a perfectly legitimate and expected use.

Once a credential is in the locker, it will remain there for subsequent launches of the app until you call the remove method or the user explicitly deletes it through Control Panel > User Accounts and Family Safety >Credential Manager. On a trusted PC (which requires sign-in with a Microsoft account), Windows will also automatically and securely roam the contents of the locker to the user's other devices (unless turned off in PC Settings > OneDrive > Sync Settings > Other Settings > Passwords). This help to create a seamless experience with your app as the user moves between devices.[43]

So, when you launch an app—even when launching it for the first time—always check if the locker contains saved credentials. There are several methods in the PasswordVault class for doing this:

- findAllByResource Returns an array (vector) of credential objects for a given resource identifier. This is how you can obtain the username and password that's been roamed from another device, because the app would have stored those credentials in the locker on the other machine under the same resource.
- findAllByUserName Returns an array (vector) of credential objects for a given username. This is useful if you know the username and want to retrieve all the credentials for multiple resources that the app connects to.
- retrieve Returns a single credential given a resource identifier and a username. Again, there will only ever be a single credential in the locker for any given resource and username.
- retrieveAll Returns a vector of all credentials in the locker for this app. The vector contains a snapshot of the locker and will not be updated with later changes to credentials in the locker.

There is one subtle difference between the findAll and retrieve methods in the list above. The retrieve method will provide you with fully populated credentials objects. The findAll methods, on the other hand, will give you objects in which the password properties are still empty. This avoids performing password decryption on what is potentially a large number of credentials. To populate that property for any individual credential, call the PasswordCredential.retievePassword method.

For further demonstrations of the credential locker—the code is very straightforward—refer to the Credential locker sample. This shows variations for single user/single resource (scenario 1), single user/multiple resources (scenario 2), multiple users/multiple resources (scenario 3), and clearing out the locker entirely (scenario 4).

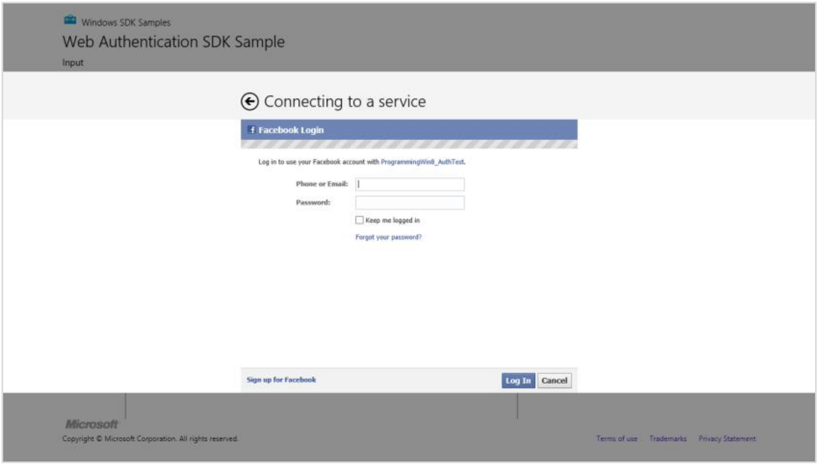
[43] Such roaming will not happen, however, if a credential is *first* stored in the locker on a domain joined machine. This protects domain credentials from leaking to the cloud.

The Web Authentication Broker

As described earlier, keeping the whole authentication process on a server is the most secure and trusted way to authenticate with a service, whether you're using a service-specific account or leveraging one from any number of other OAuth providers (OAuth, in other words, is *not* a requirement). The Web Authentication Broker provides a means of doing this authentication within the context of an app while yet keeping the authentication process completely isolated from the app.

It works like this. An app provides the URI of the authenticating page of the external site (which must use the `https://` URI scheme; otherwise you get an invalid parameter error). The broker then creates a new web host process in its own app container, into which it loads the indicated web page. The UI for that process is displayed as an overlay dialog on the app, as shown in Figure 4-6, for which I'm using scenario 1 of the Web authentication broker sample.

Provider guidance To create authentication pages for your own service to work with the web authentication broker, see Web authentication broker for online providers on the dev center.



Log in to use your Facebook account with ProgrammingWin8_AuthTest.

Phone or Email:

Password:

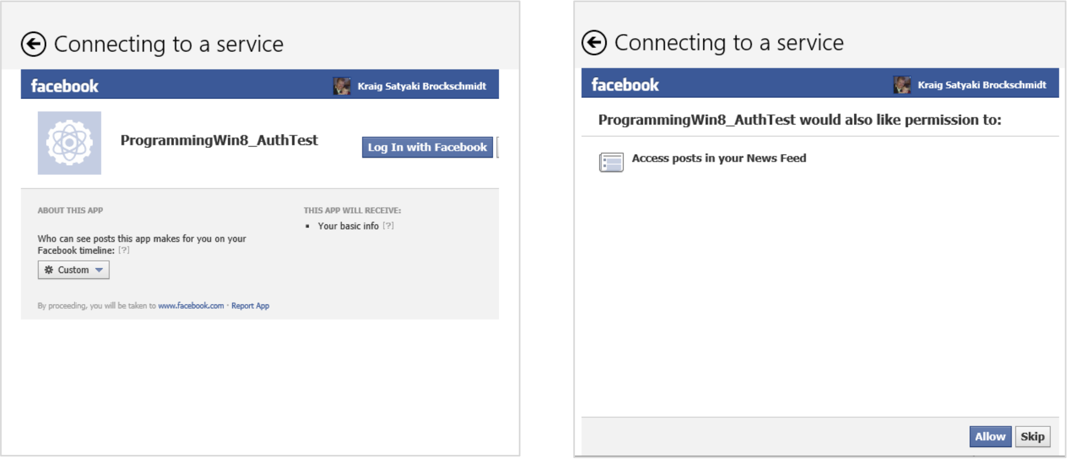
□ Keep me logged in

Copyright © Microsoft Corporation. All rights reserved.

FIGURE 4-6 The Web authentication broker sample using a Facebook login page.

Note To run the sample you'll need an app ID for each of the authentication providers in the various scenarios. For Facebook in scenario 1, visit http://developers.facebook.com/setup and create an App ID/API Key for a test app.

In the case of Facebook, the authentication process involves more than just checking the user's credentials. It also needs to obtain permission for other capabilities that the app wants to use (which the user might have independently revoked directly through Facebook). As a result, the authentication process might navigate to additional pages, each of which still appears within the web authentication broker, as shown in Figure 4-7. In this case the app identity, *ProgrammingWin8_AuthTest*, is just one that I created through the Facebook developer setup page for the purposes of this demonstration.



Connecting to a service

facebook Kraig Satyaki Brockschmidt

ProgrammingWin8_AuthTest

Log In with Facebook

ABOUT THIS APP

Who can see posts this app makes for you on your Facebook timeline: [?]

Custom

THIS APP WILL RECEIVE:

- Your basic info [?]

By proceeding, you will be taken to www.facebook.com · Report App

Connecting to a service

facebook Kraig Satyaki Brockschmidt

ProgrammingWin8_AuthTest would also like permission to:

Access posts in your News Feed

Allow Skip

FIGURE 4-7 Additional authentication steps for Facebook within the web authentication broker.

Within the broker UI—the branding of which is under the control of the provider—the user might be taken through multiple pages on the provider's site (but note that the back button next to the "Connecting to a service" title dismisses the dialog entirely). But this begs a question: how does the broker know when authentication is actually complete? In the second page of Figure 4-7, clicking the Allow button is the last step in the process, after which Facebook would normally show a login success page. In the context of an app, however, we don't need that page to appear—we want the broker's UI taken down so that we return to the app with the results of the authentication. What's more, many providers don't even have such a page—so what do we do?

Fortunately, the broker takes this into account: the app simply provides the URI of that final page of the provider's process. When the broker detects that it's navigated to that page, it removes its UI and gives the response to the app, where that response contains the appropriate token with which the app can access the service API.

As part of this process, Facebook saves these various permissions in its own back end for each particular user and token, so even if the app started the authentication process again, the user would not see the same pages shown in Figure 4-7. The user can, of course, manage these permissions when visiting Facebook through a web browser. If the user deletes the app information there, these additional authentication steps would reappear (a good way to test the process, in fact).

The overall authentication flow, showing how the broker serves as an intermediary between the app and a service, is illustrated in Figure 4-8. The broker itself creates a separate app container in which to load the service's pages to ensure complete isolation from the app. But then note how the broker is only an intermediary for authentication: once the service provides a token, which the broker returns to the app, the app can talk directly with the service. Oftentimes a service will also provide for renewing the token as needed.

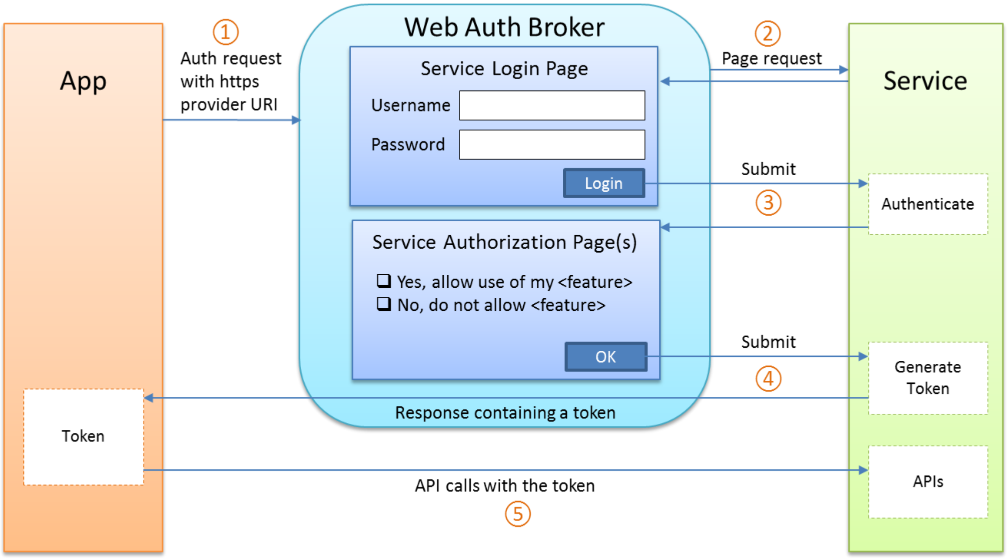


FIGURE 4-8 The authentication flow with the web authentication broker.

In WinRT, the broker is represented by the [Windows.Security.Authentication.Web.WebAuthenticationBroker] class. Authentication happens through its authenticateAsync methods. I say "methods" here because there are two variations. We'll look at one here and return to the second in the next section, "Single Sign-On."

This first variant of authenticateAsync method takes three arguments:

- options Any combination of values from the [WebAuthenticationOptions] enumeration (combined with bitwise OR). Values are none (the default), silentMode (no UI is shown), useTitle (returns the window title of the webpage in the results), useHttpPost (sends the authentication token through HTTP POST rather than on the URI, to accommodate long tokens that would make the URI exceed 2K), and useCorporateNetwork (to render the web page in an app container with the *Private Networks (Client & Server)*, *Enterprise Authentication*, and *Shared User Certificates* capabilities; the app must have also declared these).
- requestUri The URI (Windows.Foundation.Uri) for the provider's authentication page along with the parameters required by the service; again, this must use the https:// URI scheme.
- callbackUri The URI (Windows.Foundation.Uri) of the provider's final page in its authentication process. The broker uses this to determine when to take down its UI.[44]

[44] As described on How the web authentication broker works, requestUri and callbackUri "correspond to an Authorization Endpoint URI and Redirection URI in the OAuth 2.0 protocol. The OpenID protocol and earlier versions of OAuth have similar concepts."

The results given to the completed handler for authenticateAsync is a WebAuthenticationResult object. This contains properties named responseStatus (a WebAuthenticationStatus with either success, userCancel, or errorHttp), responseData (a string that will contain the page title and body if the useTitle and useHttpPost options are set, respectively), and responseErrorDetail (an HTTP response number).

> **Tip** Web authentication events are visible in the Event Viewer under *Application and Services Logs > Microsoft > Windows > WebAuth > Operational*. This can be helpful for debugging because it brings out information that is otherwise hidden behind the opaque layer of the broker. The Fiddler tool is also very helpful for debugging. For more details, see Troubleshooting web authentication broker.

Generally speaking, the app is most interested in the contents of responseData, because it will contain whatever tokens or other keys that might be necessary later on. Let's look at this again in the context of scenario 1 of the Web authentication broker sample. Set a breakpoint within the completed handler for authenticateAsync (line 59 or thereabouts), and then run the sample, enter an app ID you created earlier, and click Launch. (Note that the callbackUri parameter is set to *https://www.facebook.com/connect/login_success.html*, which is where the authentication process finishes up.)

In the case of Facebook, the responseData contains a string in this format:

```
https://www.facebook.com/connect/login_success.html#access_token=<token>&expires_in=<timeout>
```

where <token> is a bunch of alphanumeric gobbledygook and <timeout> is some period defined by Facebook. If you're calling any Facebook APIs—which is likely because that's why you're authenticating through Facebook in the first place—the <token> is the real treasure you're after because it's how you authenticate the user when making later calls to that API. (This is true of web APIs in general too.)

This token is what you then save in the credential locker for later use when the app is relaunched after being closed or terminated. With Facebook, you don't need to worry about the expiration of that token because the API generally reports that as an error and has a built-in renewal process. You'd do something similar with other services, referring, of course, to their particular documentation on what information you'll receive with the response and how to use and/or renew keys or tokens. The Web authentication broker sample, for its part, shows how to also work with Twitter (scenario 2), Flickr (scenario 3), and Google/Picasa (scenario 4), and it also provides a generic interface for any other service (scenario 5). The sample also shows the recommended UI for managing accounts (scenario 6) and how to use an OAuth filter with the HttpClient API to separate authentication concerns from the rest of your app logic.

It's instructive to look through these various scenarios. Because Facebook and Google use the OAuth 2.0 protocol, the requestUri for each is relatively simple (ignore the word wrapping):

```
https://www.facebook.com/dialog/oauth?client_id=<client_id>&redirect_uri=<redirectUri>&
scope-read_stream&display=popup&response_type=token
```

```
https://accounts.google.com/o/oauth2/auth?client_id=<client_id>&redirect_uri=<redirectUri>&
response_type=code&scope=http://picasaweb.google.com/data
```

where <client_id> and <redirectUri> are replaced with whatever is specific to the app. Twitter and Flickr, for their parts, use OAuth 1.0a protocol instead, so much more ceremony goes into creating the lengthy OAuth token to include with the *requestUri* argument to `authenticateAsync`. I'll leave it to the sample code to show those details.

Single Sign-On

What we've seen so far with the credential locker and the web authentication broker works very well to minimize how often the app needs to pester the user for credentials. Where a single app is concerned, it would ideally only ask for credentials once until such time as the user explicitly logs out. But what about multiple apps? Imagine over time that you acquire some dozens, or even hundreds, of apps from the Windows Store that use services that all require authentication. Even if those services exclusively use well-known OAuth providers, it'd still mean that you'd have to enter your Facebook, Twitter, Google, LinkedIn, Tumblr, Yahoo, or Yammer credentials in each and every app. At that point, the fact that you only need to authenticate each app once gets lost in the overall tedium!

From the user's point of view, once they've authenticated through a given provider in one app, it makes sense that other apps should benefit from that authentication if possible. Yes, some apps might need to prompt for additional permissions and some providers may not support the process, but the ideal is again to minimize the fuss and bother where we can.

The concept of *single sign-on* is exactly this: authenticating the user in one app (or the system in the case of a Microsoft account) effectively logs the user in to other apps that use the same provider. To make this work, the web authentication broker keep persisted logon cookies for each service in a special app container that's completely isolated from apps but yet allows those cookies to be shared between apps (like cookies are shared between websites in a browser). At the same time, each app must often acquire its own access keys or tokens, because these should not be shared between apps. So the real trick is to effectively perform the same kind of authentication we've already seen, only to do it without showing any UI unless it's really necessary. (An example of this experience can be seen on the Windows App Builder's blog post on Facebook login.)

This is the purpose of the variation of `authenticateAsync` that takes only the *options* and *requestUri* arguments (and not an explicit *callbackUri*). In this case *options* is often set to `WebAuthenticationOptions.silentMode` to prevent the broker's UI from appearing (this isn't required). But then how does the broker know when authentication is complete? That is, what *callbackUri* does it use for comparison, and how does the provider know that itself? It sounds like a situation where the broker would just sit there, forever hidden, while the provider patiently waits for input to a web page that's equally invisible!

What actually happens is that `authenticateAsync` watches for the provider to navigate to a special *callbackUri* in the form of *ms-app://<SID>*, where <SID> is a security identifier that uniquely identifies the calling app. This SID URI, as we'll call it, is obtained in two ways. In code, call the static method `WebAuthenticationBroker.getCurrentApplicationCallbackUri`. This returns a `Windows.Foundation.Uri` object whose `absoluteUri` property is the string you need. The second means is through the Windows Store Dashboard. When viewing info for the app in question, go to the "Services" section. There you'll see a link to the "Live Services site" (rooted at https://account.live.com). On that site, click the link "Authenticating your service" and you'll see the URI listed here under Package Security Identifier (SID).

To understand how it's used, let's follow the entire flow of the silent authentication process:

5. The app registers its SID URI with the service. From code, this could be done through some service API or other endpoint that's been set up for this purpose. A service could have a page (like Facebook) where you, the developer, registers your app directly and provides the SID URI as part of the process.
6. When constructing the `requestUri` argument for `authenticateAsync`, the app inserts its SID URI as the value of the *&redirect_uri*= parameter. The SID URI will need to be appropriately encoded as other URI parameters, of course, using encodeURIComponent.
7. The app calls `authenticateAsync` with the `silentMode` option.
8. When the provider processes the `requestUri` parameters, it checks whether the *redirect_uri* value has been registered, responding with a failure if it hasn't.
9. Having validated the app, the provider then silently authenticates (if possible) and navigates to the *redirect_uri*, making sure to include things like access keys and tokens in the response data.
10. The web authentication broker will detect this navigation and match it to the app's SID URI. Finding a match, the broker can complete the async operation and provide the response data to the app.

With all of this, it's still possible that the authentication might fail for some other reason. For example, if the user has not set up permissions for the app in question (as with Facebook), it's not possible to silently authenticate. So, an app attempting to use single sign-on would call this form of `authenticateAsync` first and, failing that, would then revert to calling its longer form (with UI), as described in the previous section.

Using the Microsoft Account

Because various Microsoft services are OAuth providers, it is possible to use the web authentication broker with a Microsoft account such as Hotmail, Live, and MSN. (I still have the same @msn.com email account I've had since 1996!) Details can be found on the OAuth 2.0 page on the Live Connect Developer Center.

Live Connect accounts—also known as Microsoft accounts—are in a somewhat more privileged position because they can also be used to sign in to Windows or can be connected to a domain account used for the same purpose. Many of the built-in apps such as Mail, Calendar, OneDrive, People, and the Windows Store itself work with this same account. Thus, it's something that many other apps might want to take advantage of. Such apps automatically benefit from single sign-on and have access to the same Live Services that the built-in apps draw from themselves (including Skype, which has taken the place of Live Messenger).

The whole gamut of what's available can be found on the Live Connect documentation.[45] You can access Live Connect features directly through its REST API as well as through the client side libraries of the Live SDK. When you install the SDK and add the appropriate references to your project, you'll have a `WL` namespace available in JavaScript. Signing in, for example, is accomplished through the `WL.login` method.

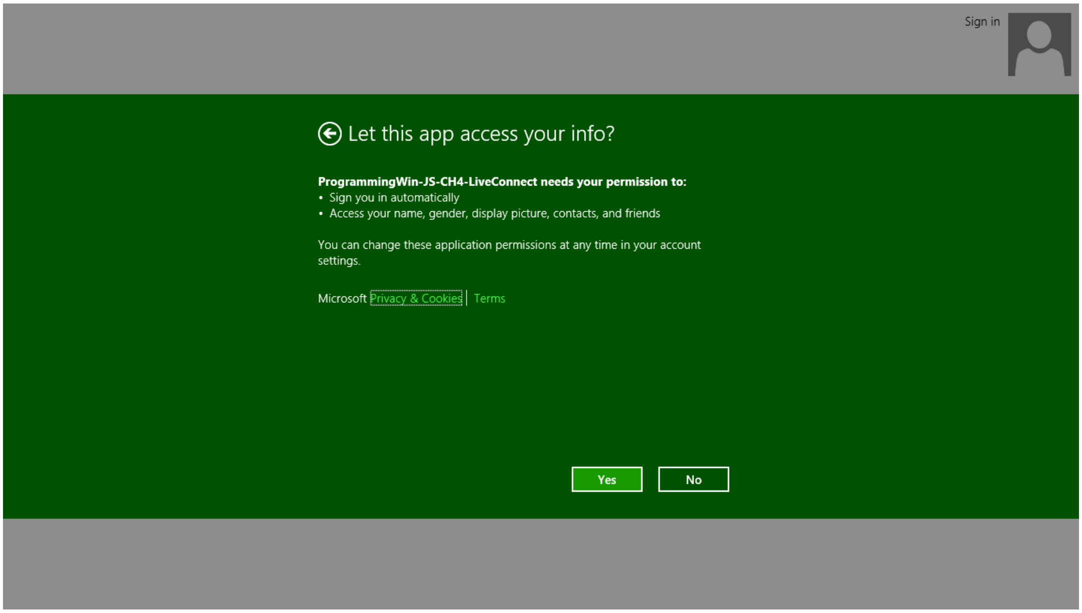
To explore Live Services a little, we'll first walk through the user experience that applies here and then we'll turn to the LiveConnect example in this chapter's companion content, which demonstrates using the Live SDK library. Note that when using Live Services, the app's package information in its manifest must match what exists in the Windows Store dashboard for your app. To ensure this, create the app profile in the dashboard (to what extent you can), go to Visual Studio, select the Store > Associate App with the Store menu command, sign in to the Store, and select your app.

> **The OnlineId API in WinRT** The `Windows.Security.Authentication.OnlineId` namespace contains an API that has some redundancy with the Live SDK, providing another route to log in and obtain an access token. The Windows account authorization sample demonstrates this, using the token when making HTTP requests directly to the Live REST API. Although the sample includes a JavaScript version, the API is primarily meant for apps written in C++ where there isn't another option like the Live SDK. However, the API is also useful when the user logs into Windows with something other than a Microsoft account, such as a domain account. The `OnlineIdAuthenticator.canSignOut` property, for example, is set to `true` if the Microsoft account is not the primary login, and thus apps that use it should provide a means to sign out. The `OnlineId` API also provides for authenticating multiple accounts together (e.g., multiple OneDrive accounts) and can also work with provider like Windows Azure Active Directory and OneDrive Pro.

The Live Connect User Experience

Whenever an app attempts to log in to Live Connect for the first time, a consent dialog such as that in Figure 4-9 will automatically appear to make sure the user understands the kinds of information the app might access. If the user denies consent, then of course the login will fail. For this reason the app should provide a means through which the user can sign in again. (Also see Guidelines for the Microsoft account sign-in experience for additional requirements.)

[45] Additional helpful references include Live Connect (Windows Store apps), Single sign-on for apps and websites, Using Live Connect to personalize apps, and Guidelines for the Microsoft account sign-in experience. Also see Bring single sign-on and OneDrive to your Windows apps with the Live SDK and Best Practices when adding single sign-on to your app with the Live SDK on the Windows 8 Developer Blog (prior to the Windows App Builder blog).



Let this app access your info?

ProgrammingWin-JS-CH4-LiveConnect needs your permission to:

- Sign you in automatically
- Access your name, gender, display picture, contacts, and friends

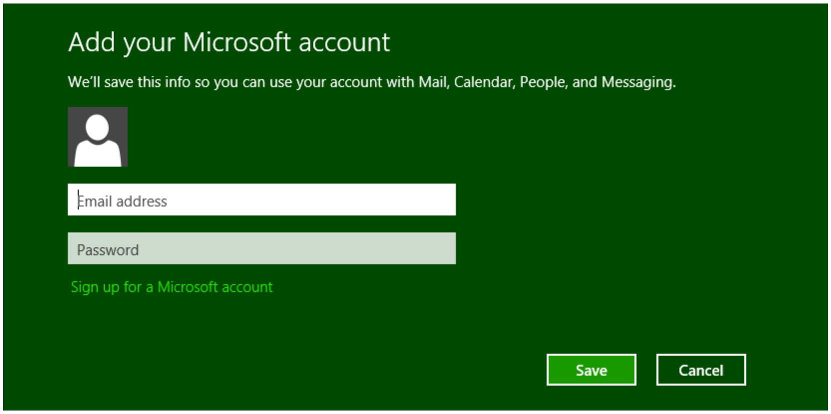
You can change these application permissions at any time in your account settings.

Microsoft Privacy & Cookies | Terms

FIGURE 4-9 The Live Connect consent dialog that appears when you first attempt to log in.

With this Live Connect login, the information that appears here (and in the other UI described below) comes through a configuration that's specific to Live Connect. You can do this in two ways, assuming you've created a profile for the app in the Windows Store dashboard. One way is to visit https://account.live.com/Developers/Applications/ and find your app there. The other is to go to the Windows Store dashboard, open your app's profile, and click Services. There you should see a Live Services Site link. Click that, and then find the link that reads Representing Your App to Live Connect Users. Click *that* one (talk about runaround!) and you'll finally arrive at a page where you can set your app's name, provide URIs for your terms of service and privacy statement, and upload a logo. All of this is independent of other info that exists in your app or in the Store dashboard, though you'll probably use the same URIs for your terms and privacy policy.

Note that if the user signed in to Windows with a domain account that has not been connected to a Microsoft account (through PC Settings > Accounts > Your Account), the first login attempt will prompt the user for those account credentials, as shown in Figure 4-10. Fortunately, the user will have to do this only once for all apps that use the Microsoft account, thanks to single sign-on.

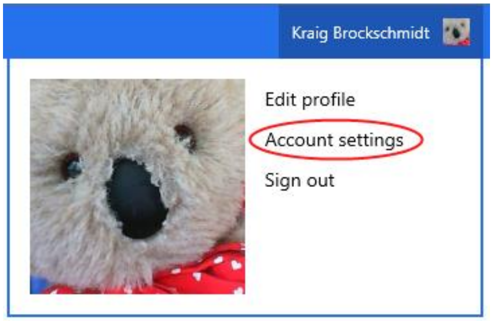


Add your Microsoft account

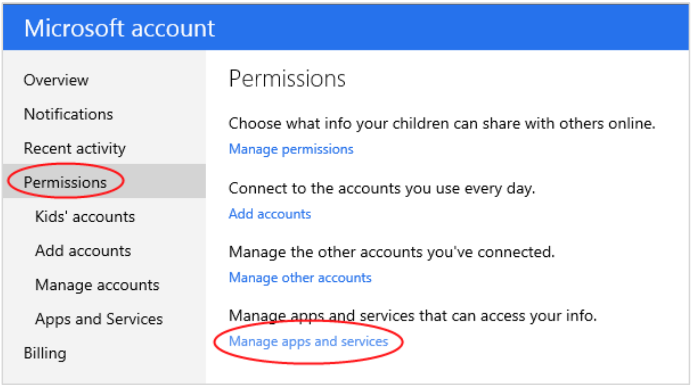
We'll save this info so you can use your account with Mail, Calendar, People, and Messaging.

FIGURE 4-10 The Microsoft account login dialog if the user logged in to Windows with a domain account.

Once you've consented to any request from an app, those permissions can be managed through the Microsoft Account portal, https://account.live.com. You can also get there from http://www.live.com by clicking your name on the upper right. This will pop up some options (as shown below), where Account Settings takes you to the account management page.



On the management page, select Permissions on the left side, and then click the Manage Apps And Services link:



Permissions

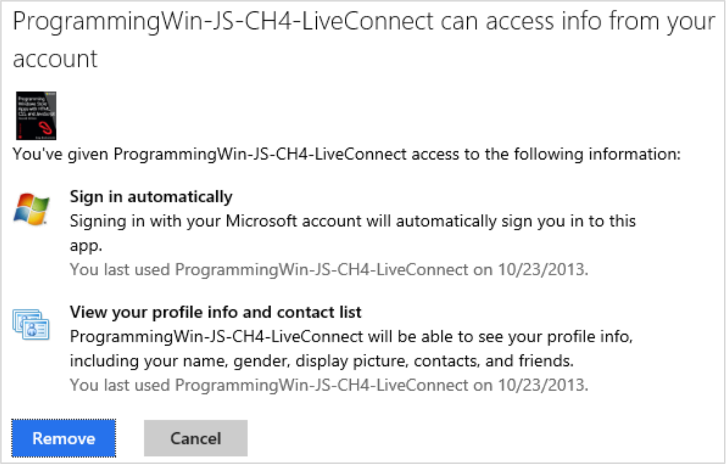
Choose what info your children can share with others online.

Connect to the accounts you use every day.

Manage the other accounts you've connected.

Manage apps and services that can access your info.

Now you'll see what permissions you've granted to all apps that use the Microsoft account, and clicking an app name (or the Edit link shown under it) takes you to a page where you can manage permissions, including revoking those to which you've consented earlier:



ProgrammingWin-JS-CH4-LiveConnect can access info from your account

You've given ProgrammingWin-JS-CH4-LiveConnect access to the following information:

Sign in automatically
Signing in with your Microsoft account will automatically sign you in to this app.
You last used ProgrammingWin-JS-CH4-LiveConnect on 10/23/2013.

View your profile info and contact list
ProgrammingWin-JS-CH4-LiveConnect will be able to see your profile info, including your name, gender, display picture, contacts, and friends.
You last used ProgrammingWin-JS-CH4-LiveConnect on 10/23/2013.

If permissions are revoked, the consent dialog will appear again when the app is next run (though you might need to sign out of Windows first—single sign on is sometimes quite sticky!). It does not appear (from my tests) to affect an app that is already running; those permissions are likely cached for the duration of the app session.

Live SDK Library Basics

Assuming that your app has been defined in Windows Store dashboard and that you've associated your Visual Studio project to it as mentioned before (the Store > Associate App with the Store menu command), the first thing you do in code is call `WL.init`. This can accept various configuration properties, if desired. After this you can subscribe to various events using `WL.Event.subscribe`; the LiveConnect example watches the `login`, `sessionChange`, and `statusChange` events:

```
WL.init();
WL.Event.subscribe("auth.login", onLoginComplete);
WL.Event.subscribe("auth.sessionChange", onSessionChange);
WL.Event.subscribe("auth.statusChange", onStatusChange);
```

Signing in with the Microsoft account, which provides a token, is then done with the `WL.login` method (js/default.js):

```
WL.login({ scope: ["wl.signin", "wl.basic"] }).then(
    function (response) {
        WinJS.log && WinJS.log("Authorization response: " + JSON.stringify(response), "app");
    },
    function (response) {
        WinJS.log && WinJS.log("Authorization error: " + JSON.stringify(response), "app");
    }
);
```

WL.login takes an object argument with a *scope* property that provides the list of scopes—features, essentially—that we want to use in the app (these can also be given to WL.init). WL.login returns a promise to which we then attach completed and error handlers that log the response. (Note that promises from WL methods support only a then method; they don't have done.)

Again, when you run the app the first time, you'll see the consent dialog shown earlier in Figure 4-9. Assuming that consent is given and the login succeeds, the response that's delivered to the completed handler for WL.login will contain status and session properties, the latter of which contains the access token. In the LiveConnect example, the response is output to the JavaScript console:

```
Authorization response: {"status":"connected","session":{"access_token":"<token_string>"}}
```

The token itself is easily accessed through the login result. Assuming we call that variable response, as in the code above, the token would be in response.session.access_token.

Note that there really isn't any need to save the token into persistent storage like the Credential Locker because you'll always attempt to login when the app starts. If that succeeds, you'll get the token again; if it fails, you wouldn't be able to get to the service anyway. If the login fails, by the way, the response object given to your error handler will contain error and error_description properties:

```
{ "error": "access_denied",
   "error_description": "The authentication process failed with error: Canceled" }
```

Note also that attempting to log out of the Microsoft account with WL.logout, if that's how the user logged in to Windows, will generate an error to this effect.

Anyway, a successful login will also trigger sessionChange events as well as the login event. In the LiveConnect example, the login handler (a function called onLoginComplete) retrieves the user's name and profile picture by using the Live API as follows (js/default.js, code condensed and error handlers omitted):

```
var loginImage = document.getElementById("loginImage");
var loginName = document.getElementById("loginName");

WL.api({ path: "me/picture?type=small", method: "get" }).then(
   function (response) {
      if (response.location) { img.src = response.location; }
   },
);

WL.api({ path: "me", method: "get" }).then(
   function (response) { name.innerText = response.name; },
);
```

Methods in the Live API are invoked, as you can see, with the WL.api function. The first argument to WL.api is an object that specifies the *path* (the data or API object we want to talk to), an optional *method* (specifying what to do with it, with "get" as the default), and an optional *body* (a JSON object with the request body for "post" and "put" methods). It's not too hard to think of WL.api as essentially generating an HTTP request using *method* and *body* to *https://apis.live.net/v5.0/<path>?access_token=<token>*, automatically using the token that came back from WL.login. But of course you don't have to deal with those details.

In any case, if all goes well, the app shows your username and image in the upper right, similar to what you see in various apps:

The User Profile (and the Lock Screen Image)

Any discussion about user credentials brings up the question of accessing additional user information that Windows itself maintains (this is separate from anything associated with the Microsoft account). What is available to Windows Store apps is provided through the Windows.System.UserProfile API. Here we find three classes of interest.

The first is the LockScreen class, through which you can get or set the lock screen image or configure an image feed (a slideshow). The image is available through the originalImageFile property (returning a StorageFile) and the getImageStream method (returning an IRandomAccessStream). Setting the image can be accomplished through setImageFileAsync (using a StorageFile) and setImageStreamAsync (using an IRandomAccessStream). This would be utilized in a photo app that has a command to use a picture for the lock screen. For an image feed you use requestSetImageFeedAsync and tryRemoveImageFeedAsync. See the Lock screen personalization sample for a demonstration.

The second is the GlobalizationPreferences object, which contains the user's specific choices for language and cultural settings. We'll return to this in Chapter 19, "Apps for Everyone, Part 1."

Third is the UserInformation class, whose capabilities are clearly exercised within PC Settings > Accounts > Your Account > Account Picture:

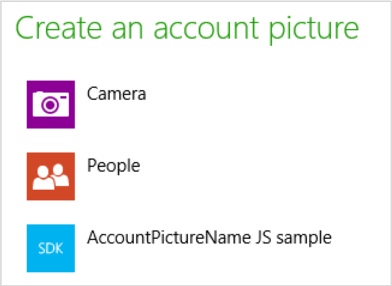
- **User name** If the nameAccessAllowed property is true, an app can then call getDisplayNameAsync, getFirstNameAsync, and getLastNameAsync, all of which provide a string to your completed handler. If nameAccessAllowed is false, these methods will complete but provide an empty result. Also note that the first and last names are available only from a Microsoft account.
- **User picture or video** Retrieved through getAccountPicture, which returns a StorageFile for the image or video. The method takes a value from AccountPictureKind: smallImage, largeImage, and video.
- If the accountPictureChangeEnabled property is true, you can use one of four methods to set the image(s): setAccountPictureAsync (for providing one image from a StorageFile),

setAccountPicturesAsync (for providing small and large images as well as a video from StorageFile objects), and setAccountPictureFromStreamAsync and setAccountPicturesFromStreamAsync (which do the same given IRandomAccessStream objects instead). In each case the async result is a [SetAccountPictureResult](#) value: success, failure, changeDisabled (accountPictureChangeEnabled is false), largeOrDynamicError (the picture is too large), fileSizeError (file is too large), or videorameSizeError (video frame size is too large),

- The accountpicturechanged event signals when the user picture(s) have been altered. Remember that because this event originates within WinRT, you should call removeEventListener if you aren't listening for this event for the lifetime of the app.

These features are demonstrated in the [Account picture name sample](#). Scenario 1 retrieves the display name, scenario 2 retrieves the first and last name (if available), scenario 3 retrieves the account pictures and video, and scenario 4 changes the account pictures and video and listens for picture changes.

One other bit that this sample demonstrates is the Account Picture Provider declaration in its manifest, which causes the app to appear within PC Settings > Accounts > Your Account, under Create an Account Picture:



Create an account picture

Camera

People

AccountPictureName JS sample

In this case the sample doesn't actually provide a picture directly but launches into scenario 4. A real app, like the Camera app that's also in PC Settings by default, will automatically set the account picture when one is acquired through its UI. How does it know to do this? The answer lies in a special URI scheme through which the app is activated. That is, when you declare the Account Picture Provider declaration in the manifest, the app will be activated with the activation kind of protocol (see Chapter 15), where the URI scheme specifically starts with ms-accountpictureprovider. You can see how this is handled in the sample's js/default.js file:

```
if (eventObject.detail.kind === Windows.ApplicationModel.Activation.ActivationKind.protocol) {
    // Check if the protocol matches the "ms-accountpictureprovider" scheme
    if (eventObject.detail.uri.schemeName === "ms-accountpictureprovider") {
        // This app was activated via the Account picture apps section in PC Settings.
        // Here you would do app-specific logic for providing the user with account
        // picture selection UX
    }
```

Returning to the `UserInformation` class, it also provides a few more details for domain accounts provided that the app has declared the *Enterprise Authentication* capability in its manifest:

- `getDomainNameAsync` Provides the user's fully qualified domain name as a string in the form of <domain>\<user> where <domain> is the full name of the domain controller, such as *mydomain.corp.ourcompany.com*.
- `getPrincipalNameAsync` Provides the principal name as a string. In Active Directory parlance, this is an Internet-style login name (known as a user principal name or UPN) that is shorter and simpler than the domain name, consolidating the email and login namespaces. Typically, this is an email address like *user@ourcompany.com*.
- `getSessionInitiationProtocolUriAsync` Provides a *session initiation protocol URI* that will connect with this user; for background, see Session Initiation Protocol (Wikipedia).

The use of these methods is demonstrated in the User domain name sample.

What We've Just Learned

- Networks come in a number of different forms, and separate capabilities in the manifest specifically call out *Internet (Client)*, *Internet (Client & Server)*, and *Private Networks (Client & Server)*. Local loopback within these is normally blocked for apps but may be used for debugging purposes on machines with a developer license.
- Rich network information is available through the `Windows.Networking.Connectivity.NetworkInformation` API, including the ability to track connectivity, be aware of network costs, and obtain connection profile details.
- Connectivity can be monitored from a background task by using the `networkStateChange` trigger and conditions such as `internetAvailable` and `internetNotAvailable`.
- The ability to run offline can be an important consideration that can make an app much more attractive to customers. Apps need to design and implement such features themselves, using local or temporary app data folders to store the necessary caches.
- Web content can be hosted in an app both in webview and `iframe` elements, depending on requirements. The local and web contexts for use with `iframe` elements provide different capabilities for hosted content, whereas the webview can host local dynamically-generated content (using `ms-appdata` URIs) and untrusted web content.
- To make HTTP requests, you can choose between `XMLHttpRequest`, `WinJS.xhr`, and `Windows.Web.Http.HttpClient`, the latter of which is the most powerful. In all cases, the `resuming` event if often used to refresh online content as appropriate.
- `Windows.Networking.BackgroundTransfer` provides for prefetching online content as well as

managing transfers while an app isn't running. It include cost-awareness, credentials, grouping, and multipart uploads, and is recommended over using your own HTTP requests for larger transfers.

- The Credential Locker is the place to securely store any credentials or sensitive tokens that an app might collect.
- To ideally keep credentials off the client device entirely, apps can log into services through the Web Authentication Broker API, which also provides for single sign-on across apps that use the same identity provider.
- Though the user's Microsoft account and the Live SDK, apps can access all the information available in Live Services, including OneDrive, contacts, and calendar.
- Apps can obtain and manage some of the user's profile data, including the user image or video and the lock screen image and video.

Chapter 5

Controls and Control Styling

Controls are one of those things you just can't seem to get away from, especially within technology-addicted cultures like those that surround many of us. Even low-tech devices like bicycles and various gardening tools have controls. But this isn't a problem—it's actually a necessity. Controls are the means through which human intent is translated into the realm of mechanics and electronics, and they are entirely made to invite interaction. As I write this, in fact, I'm sitting on an airplane and noticing all the controls that are in my view. The young boy in the row ahead of me seems to be doing the same, and that big "call attendant" button above him is just begging to be pressed!

Controls are certainly essential to Windows Store apps, and they will invite consumers to poke, prod, touch, click, and swipe them. (They will also invite the oft-soiled hands of many small toddlers as well; has anyone made a dishwasher-safe tablet PC yet?) Windows, of course, provides a rich set of controls for apps written in HTML, CSS, and JavaScript. What's most notable in this context is that from the earliest stages of design, Microsoft wanted to avoid forcing HTML/JavaScript developers to use controls that were incongruous with what those developers already know—namely, the use of HTML control elements like `<button>` that can be styled with CSS and wired up in JavaScript by using functions like `addEventListener` and `on<event>` properties.

You can, of course, use those intrinsic HTML controls in a Store app because those apps run on top of the same HTML/CSS rendering engine as Internet Explorer. No problem. There are even special classes, pseudo-classes, and pseudo-elements that give you fine-grained styling capabilities, as we'll see. But the real question was how to implement Windows-specific controls like the toggle switch and list view that would allow you to work with them in the same way—that is, declare them in markup, style them with CSS, and wire them up in JavaScript with `addEventListener` and `on<event>` properties.

The result of all this is that for you, the HTML/JavaScript developer, you'll be looking to WinJS for these controls rather than WinRT. Let me put it another way: if you've noticed the large collection of APIs in the `Windows.UI.Xaml` namespace (which constitutes about 40% of WinRT), guess what? You get to completely ignore all of them! Instead, you'll use the WinJS controls that support declarative markup, styling with CSS, and so on, which means that Windows controls (and custom controls that follow the same model) ultimately show up in the DOM along with everything else, making them accessible in all the ways you already know and understand.

The story of Windows controls is actually larger than a single chapter. We've already explored the webview control in Chapter 4, "Web Content and Services." Here we'll now look primarily at those controls that represent or work with simple data (single values) and that participate in page layout as elements in the DOM. Participating in the DOM, in fact, is exactly why you can style and manipulate all the controls (HTML and WinJS alike) through standard mechanisms, and a big part of this chapter is to just visually show the styling options you have available.

In Chapter 6, "Data Binding, Templates, and Collections," we'll explore the related subject of data binding: creating relationships between properties of data objects and properties of controls (including styles) so that the controls reflect what's happening in the data. Binding is frequently used with collections of data objects, so Chapter 6 will also delve into those details along with the WinJS templating mechanism that's often used to render data items.

That brings us to the next part of the story in Chapter 7, "Collection Controls," where we meet those controls that exist to display and interact with potentially large data sets. These are the Repeater, ItemContainer, FlipView, and ListView controls. Later on we'll also give special attention to media elements (image, audio, and video) and their unique considerations in Chapter 13, aptly titled "Media," and pick up the details of the SearchBox control in Chapter 15, "Contracts." Similarly, those elements that are primary for defining layout (like grid and flexbox) are the subject of Chapter 8, "Layout and Views," and we also have a number of UI elements that don't participate in layout at all, like app bars, navigation bars, and flyouts, as we'll see in Chapter 9, "Commanding UI."

In short, having covered much of the wiring, framing, and plumbing of an app in Chapter 3, "App Anatomy and Performance Fundamentals," and obtaining content from remote sources in Chapter 4, we're ready to start enjoying the finish work like light switches, doorknobs, and faucets—the things that make an app and its content really come to life and engage with human beings.

Sidebar: Essential References for Controls

Before we go on, you'll want to know about two essential topics on the Windows Developer Center that you'll likely refer to time and time again. First is the comprehensive Controls list that identifies all the controls that are available to you, as we'll summarize later in this chapter. The second are comprehensive UX Guidelines for Windows Store apps, which describes the best use cases for most controls and scenarios in which not to use them. This is a very helpful resource for both you and your designers.

The Control Model for HTML, CSS, and JavaScript

Again, when Microsoft designed the developer experience for Windows Store apps, we strove for a high degree of consistency between intrinsic HTML control elements, WinJS controls, and custom controls. I like to refer to all of these as "controls" because they all result in a similar user experience: some kind of widget with which the user interacts with an app. In this sense, every such control has three parts:

- Declarative markup (producing elements in the DOM).

- Applicable CSS (styles as well as special pseudo-class and pseudo-element selectors); also see the sidebar coming up on WinJS stylesheets.
- Methods, properties, and events accessible through JavaScript.

Standard HTML controls, of course, already have dedicated markup to declare them, like `<button>`, `<input>`, and `<progress>`. Styles are applied to them as with any other HTML element, and once you obtain the control's object in JavaScript you can programmatically set styles, modify properties, call methods, and attach events handlers, as you well know.

WinJS and custom controls follow nearly all of these same conventions with the exception that they don't have dedicated markup. You instead declare these controls by using some root element, typically a `<div>` or `<span>`, with two custom `data-*` attributes: `data-win-control` and `data-win-options`. The value of `data-win-control` (required) specifies the fully qualified name of a public constructor function that creates the necessary child elements of the root that make up the control. The second, `data-win-options`, is an optional JSON string containing key-value pairs separated by commas: `{ <key1>: <value1>, <key2>: <value2>, ... }`. These values are used to initialize the control.

Headache relief #1 Avoid using self-closing `div` or `span` elements for controls. Self-closing tags, like `<div … />` are not valid HTML5 and will behave as if you left off the `/` entirely. This will cause great confusion when subsequent controls aren't instantiated properly. In short, always match `<div>` with `</div>` and `<span>` with `</span>`.

Headache relief #2 If you've just made changes to `data-win-options` and your app seems to terminate without reason (and without an exception) when you next launch it, check for syntax errors in the options string. Forgetting the closing `}`, for example, will cause this behavior.

The constructor function itself takes two parameters: the root (parent) element and an options object. Conveniently, `WinJS.Class.define` produces functions that look exactly like this, making it very handy for defining controls. Indeed, WinJS defines all of its controls using `WinJS.Class.define`, and you can do the same for custom controls. The only real difference, in fact, between WinJS controls and custom controls is that the former's constructors already exist in WinJS whereas you need to implement the latter's.

Because `data-*` attributes are, according to the HTML5 specifications, completely ignored by the HTML/CSS rendering engine, some additional processing is necessary to turn an element with these attributes into an actual control in the DOM. And this, as I've hinted at before, is exactly the life purpose of the `WinJS.UI.process` and `WinJS.UI.processAll` methods. As we'll see shortly, these methods parse the `data-win-options` string and pass the resulting object and the root element to the constructor function identified in `data-win-control`. The constructor then does all the rest.

The result of this simple declarative markup plus `WinJS.UI.process/processAll` is that WinJS and custom controls are just elements in the DOM like any others. They can be referenced by DOM-traversal APIs and targeted for styling using the full extent of CSS selectors (as we'll see in the styling gallery later on). They can listen for external events like other elements and can surface events of their own by implementing `[add/remove]EventListener` and `on<event>` properties. (WinJS again provides standard implementations of `addEventListener`, `removeEventListener`, and `dispatchEvent` for this purpose.)

Let's now look at the controls we have available for Windows Store apps, starting with the HTML controls and then the WinJS controls. In both cases we'll look at their basic appearance, how they're instantiated, and the options you can apply to them.

Sidebar: WinJS stylesheets: ui-light.css, ui-dark.css, and win-* styles

As you've seen by now with various code samples, WinJS comes with two parallel stylesheets that provide many default styles and style classes for Windows Store apps: ui-light.css and ui-dark.css. You'll typically use one or the other; however, as described in Chapter 3 in the section "Page Specific Styling," you can apply them to specific pages. In any case, the light stylesheet is intended for apps that are oriented around text, because dark text on a light background is generally easier to read (so this theme is often used for news readers, books, magazines, etc., including figures in published books like this!). The dark theme, on the other hand, is intended for media-centric apps like picture and video viewers where you want the richness of the media to stand out.

Both stylesheets define a number of `win-*` style classes, which I like to think of as style packages (containing styles and CSS-based behaviors like the `:hover` pseudo-class). Most of these classes apply only to WinJS controls, but some specifically turn standard HTML controls into a Windows-specific variant. These are `win-backbutton` and `win-navigation-backbutton` for buttons, `win-ring`, `win-medium`, and `win-large` for circular `progress` controls, `win-error` and `win-paused` for progress controls generally, `win-small` for a rating control, `win-vertical` for a vertical slider (range) control, and `win-textarea` for a content editable `div`. There are also a number of `win-type-*` classes to centralize font sizes. If you want to see the details, search on their names in the Style Rules tab in Blend.

HTML Controls

HTML controls, I hope, don't need much explaining. They are described in HTML5 references, such as http://www.w3schools.com/html/html5_intro.asp, and shown with default "light" styling in Figure 5-1 and Figure 5-2. (See the next section for more on WinJS stylesheets.) It's worth mentioning that most embedded objects are not supported, except for specific ActiveX controls; see Migrating a web app.

Creating or instantiating HTML controls works as you would expect. You can declare them in markup by using attributes to specify options, the rundown of which is given in the table following Figure 5-2. You can also create them procedurally from JavaScript by calling `new` with the appropriate constructor, configuring properties and listeners as desired, and adding the element to the DOM wherever it's needed. Nothing new here at all where Store apps are concerned.

For examples of creating and using these controls, refer to the [HTML essential controls sample]() in the Windows SDK, from which the images in Figure 5-1 and Figure 5-2 were obtained. In the sample you'll find different scenarios that show many variations for the individual controls, including styling options as I'll summarize later in the chapter. Note also that scenario 13 shows how to layout a form, and all the scenarios provide a radiobutton to toggle between the light and dark stylesheets (modifying the CSS links as discussed in Chapter 3). Scenario 14 lets you also see how this looks any page background color of your choice.

Tip For more on creating, validating, and submitting forms built with HTML controls, refer to the [HTML5 form validation sample]().

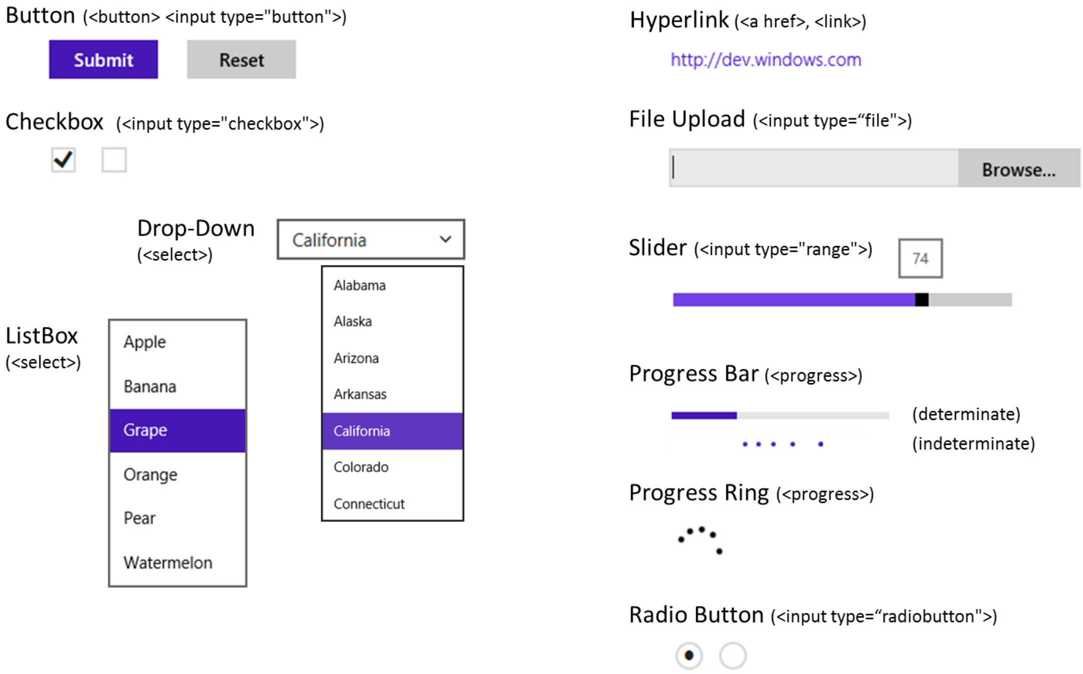
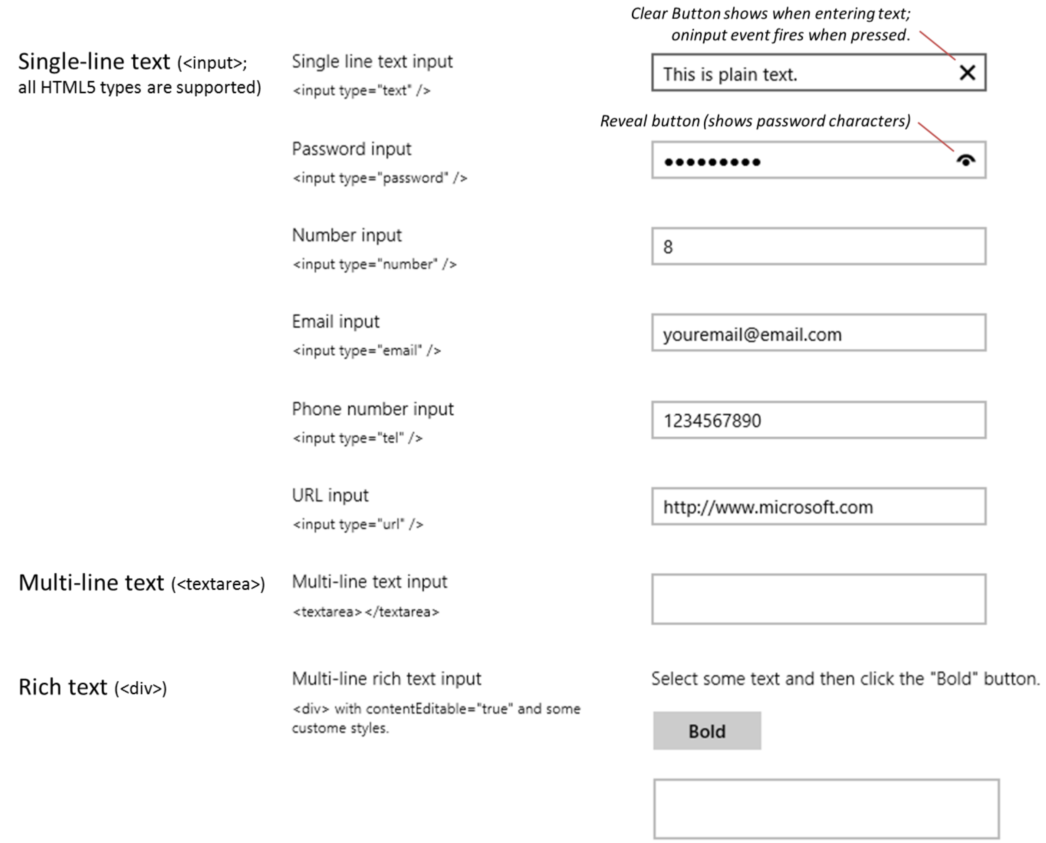

Button (<button> <input type="button">)
Submit
Reset
Checkbox (<input type="checkbox">)
☑ □
Drop-Down (<select>)
California
Alabama
Alaska
Arizona
Arkansas
California
Colorado
Connecticut
ListBox (<select>)
Apple
Banana
Grape
Orange
Pear
Watermelon
Hyperlink (<a href>, <link>)
http://dev.windows.com
File Upload (<input type="file">)
Browse...
Slider (<input type="range">)
74
Progress Bar (<progress>)
(determinate)
(indeterminate)
Progress Ring (<progress>)
Radio Button (<input type="radiobutton">)
● ○

FIGURE 5-1 Standard HTML5 controls with default "light" styles (the ui-light.css stylesheet of WinJS).



Single-line text (<input>; all HTML5 types are supported)

Single line text input
<input type="text" />

Clear Button shows when entering text; oninput event fires when pressed.

This is plain text. ✕

Password input
<input type="password" />

Reveal button (shows password characters)

•••••••••

Number input
<input type="number" />

8

Email input
<input type="email" />

youremail@email.com

Phone number input
<input type="tel" />

1234567890

URL input
<input type="url" />

http://www.microsoft.com

Multi-line text (<textarea>)

Multi-line text input
<textarea></textarea>

Rich text (<div>)

Multi-line rich text input
<div> with contentEditable="true" and some custome styles.

Select some text and then click the "Bold" button.

Bold

FIGURE 5-2 Standard HTML5 text input controls with default "light" styles (the ui-light.css stylesheet of WinJS).

Control	Markup	Common Option Attributes	Element Content (inner text/HTML)
Button	`<button type="button">`	(Note that without type, the default is `"submit"`)	button text

Button	`<input type="button">`
	`<input type="submit">`
	`<input type="reset">`
		`value` (button text)	n/a

Checkbox	`<input type=“checkbox”>`	`value`, `checked`	n/a (use a label element around the input control to add clickable text)
Drop Down List	`<select>`	`size=“1”` (default), `multiple`, `selectedIndex`	multiple `<option>` elements
Email	`<input type="email">`	`value` (initial text)	n/a
File Upload	`<input type="file">`	`accept` (mime types), `mulitple`	n/a
Hyperlink	`<a>`	`href`, `target`	Link text

ListBox	`<select> with size > 1`	`size` (a number greater than 1), `multiple`, `selectedIndex`	multiple `<option>` elements
Multi-line Text	`<textarea>`	`cols`, `rows`, `readonly`, `data-placeholder` (because `placeholder` has a bug)	initial text content
Number	`<input type="number">`	`value` (initial text)	n/a
Password	`<input type=“password">`	`value` (initial text)	n/a
Phone Number	`<input type=“tel">`	`value` (initial text)	n/a
Progress	`<progress>`	`value` (initial position), `max` (highest position; `min` is 0); no value makes it inderterminate	n/a
Radiobutton	`<input type="radiobutton">`	`value`, `checked`, `defaultChecked`	radiobutton label
Rich Text	`<div>`	`contentEditable=“true”`	HTML content
Slider	`<input type="range">`	`min`, `max`, `value` (initial position), `step` (increment)	n/a
URI	`<input type="url">`	`value` (initial text)	n/a

When to show progress controls? A `progress` element is clearly intended to inform the user that something is happening in the app. They can be used to indicate background progress, like syncing, or to indicate that some processing is blocking further interactivity. The latter case is best avoided entirely, if possible, so that the app is always interactive. If you must block interactivity, however, the recommendation is to show a progress indicator after two seconds and then provide a means to cancel the operation after ten seconds.

Extensions to HTML Elements

As you probably know already, there are many developing standards for HTML and CSS. Until these are brought to completion, implementations of those standards in various browsers are typically made available ahead of time with vendor-prefixed names. In addition, browser vendors sometimes add their own extensions to the DOM API for various elements.

With Windows Store apps, of course, you don’t need to worry about the variances between browsers, but because these apps essentially run on top of the Internet Explorer engine, it helps to know about those extensions that still apply. The key ones are summarized in the table below, and you can find the full details in the Elements and Cascading Style Sheets references in the documentation. We’ll also encounter a few others in later chapters. Another simple way to determine what extensions are available on any given object is to examine that object directly in the Visual Studio debugger at run time. Visual Studio will show the properties directly; expand the object’s Methods node to see all the functions it supports, and expand its styles node for style-related properties.

If you’ve been working with HTML5 and CSS3 in Internet Explorer already (especially Internet

Explorer 9), you might be wondering why the table doesn't show the various animation (msAnimation*), transition (msTransition*), and transform properties (msPerspective* and msTransformStyle), along with msBackfaceVisibility and msFlex* (among others). This is because these standards are now far enough along that they no longer need vendor prefixes with Internet Explorer 10+ or Store apps (though the ms* variants still work).

Methods	Description
msMatchesSelector	Determines if the control matches a selector.
ms[Set | Get | Release]PointerCapture	Captures, retrieves, and releases pointer capture for an element.
msZoomTo	Scrolls or zooms an element to a given coordinate with animation.
Style properties (on element.style)	Description
msGrid*, msRow*	Gets or sets placement of element within a CSS grid.
msContentZoom*, msContentZooming	Enables programmatic control of zooming; see Touch:Zooming and Panning.
msScroll*, msOverflowStyle	Enables programmatic control of panning and scrollbar scylting; see Touch:Zooming and Panning.
msTouchAction	Specifies whether a given region can be zoomed or panned
msTouchSelect	Toggles the "gripper" visuals that enable touch text selection.
msUserSelect	When set to 'element' allows the element to be selected; 'text' allows inline selection of the element's contents; the default is 'none'. Note that this replaces an earlier data-win-selectable attribute from early developer previews of Windows 8.
Events (add "on" for event properties)	Description
mscontentzoom	Fires when a user zooms an element (Ctrl+ +/-, Ctrl + mousewheel), pinch gestures.
msgesture[change | end | hold | tap | pointercapture]	Gesture input events (see Chapter 12, "Input and Sensors").
msinertiastart	Gesture input events (see Chapter 12).
msgotpointercapture, mslostpointercapture	Element acquired or lost capture (set with msSetPointerCapture.
mspointer[cancel | down | enter| leave | | move | out | over | up]	Pointer input events (see Chapter 12).
msmanipulationstatechanged	State of a manipulated element has changed.

WinJS Controls

Windows defines a number of controls that help apps fulfill various design guidelines. As noted before, these are implemented in WinJS for apps written in HTML, CSS, and JavaScript, rather than WinRT; this allows those controls to integrate naturally with other DOM elements. Each control is defined as part of the `WinJS.UI` namespace using `WinJS.Class.define`, where the constructor name matches the control name. So the full constructor name for a control like the Rating is `WinJS.UI. Rating`. As noted before in the control model, this is the name you specify within a `data-win-control` attribute for the control's root element such that `WinJS.UI.process`/`processAll` can instantiate that control.

Tip Once a WinJS control is instantiated, the root element object will have a `winControl` property attached to it. This `winControl` object is the result of calling `new` on the control's constructor and gives access to the control's specific methods, properties, and events. The `winControl` object also has an `element` property to go the other direction.

The simpler UI controls are `BackButton`, `DatePicker`, `Rating`, `TimePicker`, and `ToggleSwitch`, the default styling for which are shown in Figure 5-3. I trust that the purpose of each control is obvious!

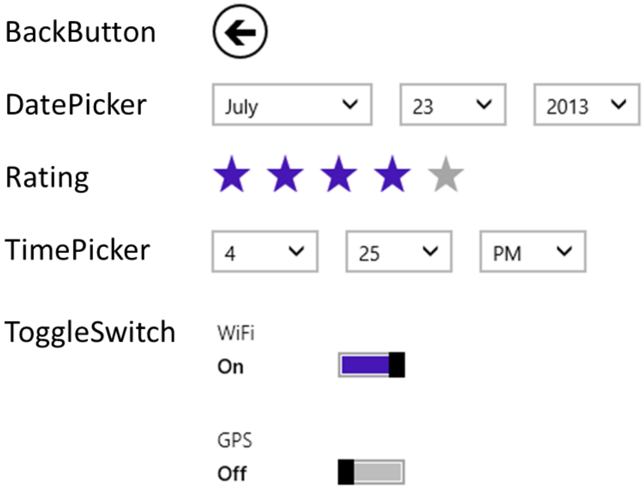


FIGURE 5-3 Default (light) styles for the simple WinJS controls.

There are then three other WinJS controls whose purpose is to contain other content and provide some other behavior around it. These are `HtmlControl`, `Tooltip`, and `ItemContainer`.[46]

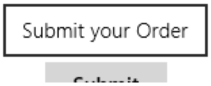
WinJS controls on Windows Phone As of this writing, the WinJS library that's available for Windows Phone 8.1 includes the `HtmlControl`, `ToggleSwitch`, and `ItemContainer` controls but not the others described in this chapter. It does include the collection controls described in other chapters, including the `FlipView`, `ListView`, `Repeater`, and `SemanticZoom`, along with the `AppBar` and its unique `Pivot` control (which you use instead of `Hub`). For full details, refer to the documentation for the `WinJS.UI` namespace.

[46] There is also the `WinJS.UI.TabContainer` control that is primarily used within the ListView control implementation. It has limited direct utility for apps and does not support declarative processing, so I won't talk more of it here.

The `HtmlControl` is a simpler form of the page controls we saw in Chapter 3, allowing you to easily load some HTML fragment from your app package (along with its referenced CSS and JavaScript) as a control. The `WinJS.UI.HtmlControl` constructor, in fact, simply calls `WinJS.UI.Pages.render`, loading that content into the control. Indeed, page controls themselves can be used in the same way—you can, for example, load up multiple page controls within the same host page. I prefer to keep these ideas separate, however, using page controls for full-page DOM replacement and the `HtmlControl` for control-like fragments.

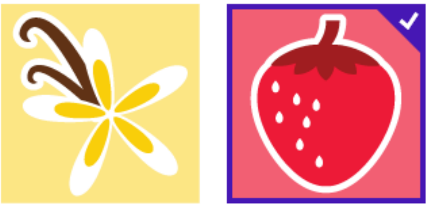
Because the purpose of the `HtmlControl` is to simply encapsulate other pieces of HTML, the control itself has no inherent visuals or styling. If you want an example, simply look at the header and footer of any Windows SDK sample!

The `Tooltip` control, for its part, is a step above the plain text tooltip that HTML provides automatically for the `title` attribute. Its behavior is such that when you attach it to some other control, it will appear automatically for hover states. With default styling and text, it appears just like the default tooltips (I've cropped the button to which the tooltip is attached—it's not part of the tooltip):


Submit your Order

Unlike the default tooltip, however, the WinJS `Tooltip` control can use any HTML including other controls. We'll see this shortly in the section "Example: WinJS.UI.Tooltip." The default rectangle share can also be fully styled, but I'll also make you wait for that!

The `ItemContainer` control is also a container—as befits its name!—for some other piece of HTML that acts as a single unit. The HTML within it can again include other controls, but the behaviors apply to the item as a whole. Those behaviors, as determined by various options, include swipe (to select), tap (to invoke), and HTML5 drag-and-drop support. Here's an example item in both selection states:



We'll see more about these options a little later with specific examples.

As noted in this chapter's introduction, I'm saving other WinJS controls for later chapters because they each need additional context. We'll see the Repeater in Chapter 6; the `FlipView`, `ListView`, and `SemanticZoom` that work with collections are covered in Chapter 7; the `Hub` we'll see in Chapter 8 as it's pertinent to layout; AppBar, NavBar, and Flyout are transient commanding UI that we'll come to in Chapter 9 (because they don't participate in layout); and media controls and the `SearchBox` we'll encounter in Chapters 13, and 15, respectively. Much to look forward to!

Now to reiterate the control model, a WinJS control is declared in markup by attaching data-win-control and data-win-options attributes to some root element. That element is typically a div (block element) or span (inline element), because these don't bring much other baggage, but any element, such as a button, can be used. These elements can, of course, have id and class attributes as needed.

The available options for the WinJS controls we're discussing in this chapter are summarized in the table below. The table includes those events that can be wired up declaratively through the data-win-options string, if desired, or through JavaScript. For full documentation on all these options, start with the Controls list in the documentation and go to the control-specific topics linked from there, or use the links given below. For details on the syntax of the data-win-options string, see the next section in this chapter.

Fully-qualified constructor name as used in data-win-control	**Options in data-win-options (note that event names use the 'on' prefix in the attribute syntax)**
WinJS.UI.BackButton (sample)	Events: none. The BackButton instead calls WinJS.Navigation.back directly when clicked or tapped (or the user presses Alt+Left) and listens for WinJS.Navigation.onnavigated to enable or disable itself appropriately, depending on the navigation stack. Methods: refresh
WinJS.UI.DatePicker (sample)	Properties: calendar, current, datePattern, disabled, maxYear, minYear, monthPattern, yearPattern Events: onchange
WinJS.UI.HtmlControl (see default.html in any SDK sample)	Properties: uri (referring to an in-package fragment), such as "/controls/mycontrol.html".
WinJS.UI.ItemContainer (sample)	Properties: draggable, selected, selectionDisabled, swipeBehavior, swipeOrientation, tapBehavior Events: oninvoked, onselectionchanging, onselectionchanged Methods: forceLayout
WinJS.UI.Rating (sample)	Properties: averageRating, disabled, enableClear, maxRating, tooltipStrings (an array of strings the size of maxRating), userRating Events: oncancel, onchange, onpreviewchange
WinJS.UI.TimePicker (sample)	Properties: clock, current, disabled, hourPattern, minuteIncrement, minutePattern, periodPattern. (Note that the date portion of current will always be July 15, 2011 because there are no known daylight savings time transitions on this day.) Events: onchange
WinJS.UI.ToggleSwitch (sample)	Properties: checked, disabled, labelOff, labelOn, title Events: onchange
WinJS.UI.Tooltip (sample)	Properties: contentElement, innerHTML, infotip, extraClass, placement Events: onbeforeclose, onbeforeopen, onclosed, onopened Methods: open, close

Note All WinJS controls have the methods addEventListener, removeEventListener, dispatchEvent, and dispose, along with an element property that identifies the control's root element.

When setting WinJS control options from JavaScript, be sure to set them on the winControl object and not on the containing element itself. (This is very important to remember when doing data-binding to control properties, as we'll see in Chapter 6.) Event handlers can be assigned through winControl.on<event> properties or through winControl.addEventListener. If you want to set multiple options at the same time, you can use the WinJS.UI.setOptions method, which accepts the winControl object (*not* the root element) and a second object containing the options. Such an object is exactly what WinJS produces when it parses a data-win-options string.

Sidebar: The Ubiquitous dispose Method

If you look at the documentation for any of these controls, you'll see that they each have a method named dispose whose purpose is to release any resources held by the control. In addition, every page control you define through WinJS.UI.Pages.define also has a dispose method. All together, these methods allow WinJS to make sure that the pages and the controls on those pages release any necessary resources when they're removed from the DOM (correcting memory leaks that were observed in Windows 8). That is, when you navigate away from a page, the PageControlNavigator we learned about in Chapter 3 calls the page's dispose method, which in turn calls WinJS.Utilities.disposeSubTree on its root element. That method calls dispose on every child control that is marked with the win-disposable class (as all WinJS controls are) and that has a dispose method (as all WinJS controls do).

If you are using PageControlNavigator, then, you don't need to worry much about calling dispose yourself. However, if you implement your own navigation controller or page-loading mechanism, be sure to replicate this behavior and call dispose on any controls that are being removed from the DOM. Additionally, custom controls should also implement a dispose method and mark themselves with the win-disposable class to participate in the process. Custom controls that can contain other controls (like a ListView or the PageControlNavigator) must also be mindful to manage control disposal. We'll talk more of this later with custom controls, but if you want a quick demonstration refer to the Dispose model sample.

Syntax for data-win-options

As noted earlier, the data-win-options attribute is a string containing key-value pairs, one for each property or event, separated by commas, in the form { <key1>: <value1>, <key2>: <value2>, ... }. When a control is instantiated through WinJS.UI.process[All], WinJS parses the options string into the equivalent JavaScript object and passes it to the control's constructor. Easy enough! (Remember to be mindful of syntax errors in your data-win-options string, though, because such errors will cause an app to abruptly terminate during control instantiation.)

Most of the time you'll just be using literal values in the options string, as with this example that we'll see again later for the WinJS.UI.Rating control:

```
<div id="rating1" data-win-control="WinJS.UI.Rating"
    data-win-options="{averageRating: 3.4, userRating: 4, onchange: changeRating}">
</div>
```

Notice how WinJS control events use key names in the form of on<event>—those are the equivalent JavaScript property names on the control object.

There are other possibilities for values, however, many of which you can find if you search through the SDK JavaScript samples for "data-win-options". To summarize:

- Values can dereference objects, namespaces, and arrays by using dot notation, brackets ([ ]'s), or both.
- Values can be array literals as well as objects surrounded by { }, which is how you assign values to complex properties.
- Values can be an id of another element in the DOM, provided that element has an id attribute.
- Values can use the syntax select('<selector>') to declaratively reference an element in the DOM.

Let's see some examples. Dot notation is typically used to reference members of an enumeration or a namespace. The latter is common with ListView controls (see Chapter 7), where the data source is defined in a separate namespace, or with app bars (see Chapter 9), where its event handlers are defined in a namespace:

```
data-win-options = "{ selectionMode: WinJS.UI.SelectionMode.multi,
    tapBehavior: WinJS.UI.TapBehavior.toggleSelect }"

data-win-options = "{ itemDataSource: DefaultData.bindingList.dataSource }"

data-win-options = "{ id: 'tenDay', label: 'Ten day', icon: 'calendarweek', type: 'toggle',
    onclick: FluidAppLayout.transitionPivot }"
```

The Adaptive layout with CSS sample in html/app.html shows the use of both dot notation and array deferences together:

```
data-win-options = "{ data: FluidAppLayout.Data.mountains[0].weatherData[0] }"

data-win-options = "{ data: FluidAppLayout.Data.mountains[0].weatherData }"
```

Object notation is commonly used with ListViews to specify its layout object, along with dot notation to identify the layout object:

```
data-win-options = "{ layout: {type: WinJS.UI.GridLayout} }"

data-win-options = "{ selectionMode: 'none', tapBehavior: 'none', swipeBehavior: 'none',
    layout: { type: WinJS.UI.GridLayout, maxRows: 2 } }
```

```
data-win-options = "{ itemDataSource: Data.list.dataSource,
    layout: {type: SDKSample.Scenario1.StatusLayout} }
```

To declaratively refer to another element in the DOM—which must be instantiated before the control that's trying to reference it, of course—you have two choices.

First is to use the id of the element—that is, the same name that is in that element's `id` attribute. Note that you do *not* make that name a string literal, but you specify it directly as an identifier. For example, if somewhere earlier in markup we declare a binding template (as we'll see in Chapter 6):

```
<div id="smallListItemTemplate" data-win-control="WinJS.Binding.Template"><!-- ... --></div>
```

then we can reference that control's root element in an options string as follows:

```
data-win-options = "{ itemTemplate: smallListItemTemplate }"
```

This works because the app host automatically creates variables in the global namespace for elements with `id` attributes, and thus the value we're using *is* that variable.

The other way is using the *select* syntax, which is a way of inserting the equivalent of `document.querySelector` into the options string (technically `WinJS.UI.scopedSelect`, which calls `querySelector`). So the previous options string could also be written like this:

```
data-win-options = "{ itemTemplate: select('smallListItemTemplate') }"
```

Alternately, if we declared a template with a class instead of an id:

```
<div class="templateSmall" data-win-control="WinJS.Binding.Template"><!-- ... --></div>
```

then we can refer to it as follows:

```
data-win-options = "{ itemTemplate: select('templateSmall') }"
```

The HTML AppBar control sample in html/custom-content.html and the HTML flyout control sample in html/appbar-flyout.html provide a couple more examples:

```
data-win-options = "{ id: 'list', type: 'content', section: 'selection',
    firstElementFocus:select('.dessertType'), lastElementFocus:select('.dessertType') }"
data-win-options = "{ id: 'respondButton', label: 'Respond', icon: 'edit', type: 'flyout',
    flyout:select('#respondFlyout') }"
```

If you're wondering, *select* is presently the only method that you can use in this way.

WinJS Control Instantiation

As we've seen a number of times already, WinJS controls declared in markup with `data-*` attributes are not instantiated until you call `WinJS.UI.process(<element>)` for a single control or `WinJS.UI.processAll` for all such elements currently in the DOM or, optionally, for all elements contained in a given element. In the case of `processAll`, if you've just added a bunch of markup to the DOM (through an `innerHTML` assignment or `WinJS.UI.Pages.render`, for instance), you can call `WinJS.UI.processAll` to instantiate any WinJS controls in that markup.

To understand this process, here's what `WinJS.UI.process` does for a single element:

11. Parse the `data-win-options` string into an options object, throwing exceptions if parsing fails.
12. Extract the constructor specified in `data-win-control` and call `new` on that function passing the root element and the options object.
13. The constructor creates whatever child elements it needs within the root element.
14. The object returned from the constructor—the control object—is stored in the root element's `winControl` property.

Clearly, then, the bulk of the work really happens in the constructor. Once this takes place, other JavaScript code (as in your app's `activated` method or a page control's `ready` method) can call methods, manipulate properties, and add listeners for events on both the root element and the `winControl` object.

`WinJS.UI.processAll`, for its part, simply traverses the DOM (starting at the document root or an optional element, if given) looking for `data-win-control` attributes and does `WinJS.UI.process` for each. How you use both of these is really your choice: `processAll` does a deep traversal whereas `process` lets you instantiate controls one at a time as you dynamically insert markup. Note that in both cases the return value is a promise, so if you need to take additional steps after processing is complete, provide a completed handler to the promise's `then` or `done` method.

It's also good to understand that `process` and `processAll` are really just helper functions. If you need to, you can just directly call `new` on a control constructor with an element and options object. This will create the control and attach it to the given element automatically. You can also pass `null` for the element, in which case the WinJS control constructors create a new `div` element to contain the control that is otherwise unattached to the DOM. This would allow you, for instance, to build up a control offscreen and attach it to the DOM only when needed.

Also note that `process` and `processAll` will check whether any given control has already been instantiated (the element already has a `winControl` property), so you can call either method repeatedly on the same root element or the whole document without issue.

To see all this in action, we'll look at some examples in a moment. First, however, we need to discuss a matter referred to as *strict processing*.

Strict Processing and processAll Functions

WinJS has three DOM-traversing functions: `WinJS.UI.processAll`, `WinJS.Binding.processAll` (which we'll see later in Chapter 6), and `WinJS.Resources.processAll` (which we'll see in Chapter 19, "Apps for Everyone, Part 1"). Each of these looks for specific `data-win-*` attributes and then takes additional actions using those contents. Those actions, however, can involve calling a number of different types of functions:

- Functions appearing in a "dot path" for control processing and binding sources

- Functions appearing in the left-hand side for binding targets, resource targets, or control processing
- Control constructors and event handlers
- Binding initializers or functions used in a binding expression
- Any custom layout used for a ListView control

Such actions introduce a risk of injection attack if a processAll function is called on untrusted HTML, such as arbitrary markup obtained from the web. To mitigate this risk, WinJS has a notion of *strict processing* that is enforced within all HTML/JavaScript apps. The effect of strict processing is that any functions indicated *in markup* that processAll methods might encounter must be "marked for processing" or else processing will fail. The mark itself is simply a property named supportedForProcessing on the function object that is set to true.

Functions returned from WinJS.Class.define, WinJS.Class.derive, WinJS.UI.Pages.define, and WinJS.Binding.converter are automatically marked in this manner. For other functions, you can either set a supportedForProcessing property to true directly or use any of the following marking functions:

```
WinJS.Utilities.markSupportedForProcessing(myfunction);
WinJS.UI.eventHandler(myHandler);
WinJS.Binding.initializer(myInitializer);

//Also OK
<namespace>.myfunction = WinJS.UI.eventHandler(function () {
});
```

Note also that appropriate functions coming directly from WinJS, such as all WinJS.UI.* control constructors, as well as WinJS.Binding.* functions, are marked by default.

So, if you reference custom functions from your markup, be sure to mark them accordingly. But this is *only* for references from *markup*: you don't need to mark functions that you assign to on<event> properties in JavaScript or pass to addEventListener.

Example: WinJS.UI.HtmlControl

OK, now that we got the strict processing stuff covered, let's see some concrete examples of working with a WinJS control.

For starters, you can find examples of using the HtmlControl in the default.html file of just about any Windows SDK sample (such as the HTML Rating control sample). In that file you'll see the following construct:

```
<div id="header" data-win-control="WinJS.UI.HtmlControl"
   data-win-options="{uri: '/sample-utils/header.html'}"></div>
<div id="content">
   <h1 id="featureLabel"></h1>
```

```
    <div id="contentHost">
        <!-- Sample-specific content -->
    </div>
</div>
<div id="footer" data-win-control="WinJS.UI.HtmlControl"
    data-win-options="{uri: '/sample-utils/footer.html'}"></div>
```

This uses two `HtmlControl` instances for static header and footer content, while the sample's JavaScript builds up its scenario selectors within the *contentHost* element. Clearly, each `HtmlControl` allows the app to keep certain content isolated in separate files; in the case of the SDK samples, it makes it very easy to globally update the headers and footers of hundreds of samples.

Example: WinJS.UI.Rating (and Other Simple Controls)

For starters, here's some markup for a `WinJS.UI.Rating` control, where the options specify two initial property values and an event handler:

```
<div id="rating1" data-win-control="WinJS.UI.Rating"
    data-win-options="{averageRating: 3.4, userRating: 4, onchange: changeRating}">
</div>
```

To instantiate this control, we need either of the following calls:

```
WinJS.UI.process(document.getElementById("rating1")); //Or any ancestor element
WinJS.UI.processAll();
```

Again, both of these functions return a promise, but it's unnecessary to call `then`/`done` unless we need to do additional post-instantiation processing or handle exceptions that might have occurred (and that are otherwise swallowed). Also, note that the `changeRating` function specified in the markup must be globally visible and marked for processing, or else the control will fail to instantiate.

Alternately, we can instantiate the control and set the options procedurally. In markup:

```
<div id="rating1" data-win-control="WinJS.UI.Rating"></div>
```

And in code:

```
var element = document.getElementById("rating1");
WinJS.UI.process(element);
element.winControl.averageRating = 3.4;
element.winControl.userRating = 4;
element.winControl.onchange = changeRating;
```

The last three lines above could also be written as follows using the `WinJS.UI.setOptions` method, but this isn't recommended because it's harder to debug:

```
var options = { averageRating: 3.4, userRating: 4, onchange: changeRating };
WinJS.UI.setOptions(element.winControl, options);
```

We can also just instantiate the control directly. In this case the markup is nonspecific:

```
<div id="rating1"></div>
```

and we call new on the constructor ourselves:

```
var newControl = new WinJS.UI.Rating(document.getElementById("rating1"));
newControl.averageRating = 3.4;
newControl.userRating = 4;
newControl.onchange = changeRating;
```

Or, as mentioned before, we can skip the markup entirely, have the constructor create an element for us (a div), and attach it to the DOM at our leisure:

```
var newControl = new WinJS.UI.Rating(null,
    { averageRating: 3.4, userRating: 4, onchange: changeRating });
newControl.element.id = "rating1";
document.body.appendChild(newControl.element);
```

> **Hint** If you see strange errors on instantiation with these latter two cases, check whether you forgot the new and are thus trying to directly invoke the constructor function.

Note also in these last two cases that the *rating1* element will have a winControl property that is the same as the newControl returned from the constructor.

To see this control in action, refer to the HTML Rating control sample.

The other simple controls—namely the DatePicker, TimePicker, and ToggleSwitch—are very similar to what we just saw with Ratings. All that changes are the specific properties and events involved; again, start with the Controls list page and navigate to any given control for all the specific details. For working samples refer to the HTML DatePicker and TimePicker controls and the HTML ToggleSwitch control samples.

Example: WinJS.UI.Tooltip

The WinJS.UI.Tooltip control works a little differently from the other simple controls. First, to attach a tooltip to a specific element, you can either add a data-win-control attribute to that element or place the element itself inside the control:

```
<!-- Directly attach the Tooltip to its target element -->
<targetElement data-win-control="WinJS.UI.Tooltip">
</targetElement>

<!-- Place the element inside the Tooltip -->
<span data-win-control="WinJS.UI.Tooltip">
    <!-- The element that gets the tooltip goes here -->
</span>

<div data-win-control="WinJS.UI.Tooltip">
    <!-- The element that gets the tooltip goes here -->
</div>
```

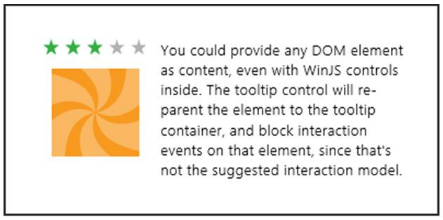
Second, the contentElement property of the tooltip control can name another element altogether, which will be displayed when the tooltip is invoked. For example, consider this piece of hidden HTML in our markup that contains other controls:

```
<div style="display: none;">
    <!--Here is the content element. It's put inside a hidden container
    so that it's invisible to the user until the tooltip takes it out.-->
    <div id="myContentElement">
        <div id="myContentElement_rating">
            <div data-win-control="WinJS.UI.Rating" class="win-small movieRating"
                data-win-options="{userRating: 3}">
            </div>
        </div>
        <div id="myContentElement_description">
            <p>You could provide any DOM element as content, even with WinJS controls inside. The
tooltip control will re-parent the element to the tooltip container, and block interaction
events on that element, since that's not the suggested interaction model.</p>
        </div>
        <div id="myContentElement_picture">
        </div>
    </div>
</div>
```

We can reference it like so:

```
<div data-win-control="WinJS.UI.Tooltip"
   data-win-options="{infotip: true, contentElement: myContentElement}">
   <span>My piece of data</span>
</div>
```

When you hover over the text (with a mouse or hover-enabled touch hardware), this tooltip will appear:



You could provide any DOM element as content, even with WinJS controls inside. The tooltip control will re-parent the element to the tooltip container, and block interaction events on that element, since that's not the suggested interaction model.

This example is taken directly from the HTML Tooltip control sample, so you can go there to see how all this works, including other options like `placement` and `infotip`. Do be aware, as the sample indicated, that controls within a `Tooltip` cannot themselves be interactive.

Example: WinJS.UI.ItemContainer

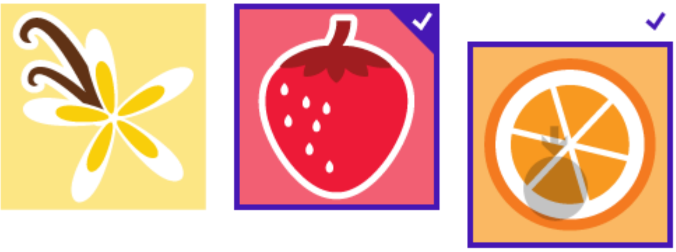
Like the `Tooltip`, the `ItemContainer` control wraps behaviors around some other piece of HTML, namely whatever is declared as children of the `ItemContainer`. For example, the following code from scenario 1 of the HTML ItemContainer sample wraps three `img` elements in separate containers (html/scenario1.html):

```
<div id="flavorSelector">
    <div class="scenario1Containers" id="scen1-item1" data-win-control="WinJS.UI.ItemContainer"
```

```
        data-win-options="{swipeBehavior: 'select', tapBehavior: 'toggleSelect'}">
        <img src="/images/110Vanilla.png" alt="Vanilla" draggable="false" />
    </div>
    <div class="scenario1Containers" id="scen1-item2" data-win-control="WinJS.UI.ItemContainer"
        data-win-options="{swipeBehavior: 'select', tapBehavior: 'toggleSelect'}">
        <img src="/images/110Strawberry.png" alt="Strawberry" draggable="false" />
    </div>
    <div class="scenario1Containers" id="scen1-item3" data-win-control="WinJS.UI.ItemContainer"
        data-win-options="{swipeBehavior: 'select', tapBehavior: 'toggleSelect'}">
        <img src="/images/110Orange.png" alt="Orange" draggable="false" />
    </div>
</div>
```

Note that each is marked with the swipeBehavior of select and the tapBehavior of toggleSelect. The other option for swipeBehavior is none (these come from the WinJS.UI.SwipeBehavior enumeration); the other options for tapBehavior are none, directSelect, and invokeOnly (from the WinJS.UI.TabBehavior enumeration). You can also set selectionDisabled to true to turn off selection completely.

In the graphic below, the left ItemContainer is unselected, the middle is selected, and the rightmost is in the process of being selected from a top-down swipe gesture (indicated by the arrow and circle). This is shown more dynamically in Video 5-1:

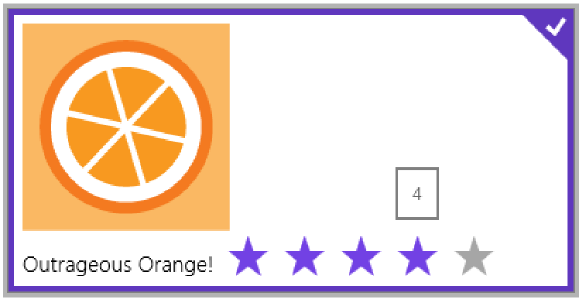


In all these cases the default swipeOrientation is vertical; the other option in WinJS.UI.Orientation is horizontal, which is demonstrated in scenario 3 of the sample.

Scenario 4 demonstrates using other controls within the ItemContainer, such as a Rating control (html/scenario4.html):

```
<div id="scen4Item1" data-win-control="WinJS.UI.ItemContainer"
    data-win-options="{swipeBehavior: 'select', tapBehavior:'toggleSelect'}">
    <div id="itemContent">
        <img src="/images/110Orange.png" />
        <div id="itemDetail">
            Outrageous Orange!
            <div id="myRatingControl" class="win-interactive"
                data-win-control="WinJS.UI.Rating"></div>
        </div>
    </div>
</div>
```

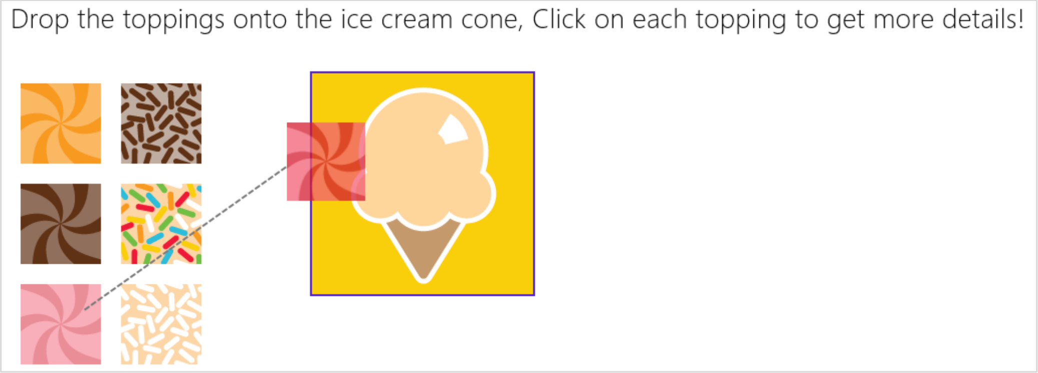
Take special note of the win-interactive class given to the Rating control—this is what tells the parent element (the ItemContainer) to pass interaction events down to the child control. This allows you to directly change the rating, where the rating's tooltip appears as it should for mouse hover:



By setting an ItemContainer.draggable option to true, you enable that item to be dragged away and received by other HTML5 drag-and-drop targets. Scenario 2 of the sample does this with six ItemContainer controls, making sure to disable default dragging on the img elements within them (html/scenario2.html):

```
<div class="scenario2Containers" id="scen2-item1" data-win-control="WinJS.UI.ItemContainer"
    data-win-options="{draggable: true, selectionDisabled: true,
        oninvoked:Scenario2.invokedHandler}">
    <img src="/images/60SauceCaramel.png" draggable="false" />
</div>

<!-- And so on -->
```

The drop target just registers for the standard HTML5 drag and drop events (dragenter, dragover, dragleave, and drop) to create an interactive ice cream cone builder:


Drop the toppings onto the ice cream cone, Click on each topping to get more details!

You can see that each item also has an oninvoked handler (marked for processing of course!) that displays some details for that topping. See Video 5-2 for a short demonstration.

If you've played with Windows 8 or Windows 8.1 at all, you'll probably recognize that the behaviors of the `ItemContainer` show up in a collection control like the `WinJS.UI.ListView`. And in fact, the `ListView` in Windows 8.1 uses the `ItemContainer` unabashedly! It was part of overhauling the ListView for Windows 8.1 that created the `ItemContainer` as a separate entity that can be used for single items outside of a collection.

Working with Controls in Blend

Before we move onto the subject of control styling, it's a good time to highlight a few additi onal features of Blend for Visual Studio where controls are concerned. As I mentioned in Video 2-2, the Assets tab in Blend gives you quick access to all the HTML elements and WinJS controls (among many other elements) that you can just drag and drop into whatever page is showing in the artboard. (See Figure 5-4.) This will create basic markup, such as a `div` with a `data-win-control` attribute for WinJS controls; then you can go to the HTML Attributes pane (on the right) to set options in the markup. (See Figure 5-5.)

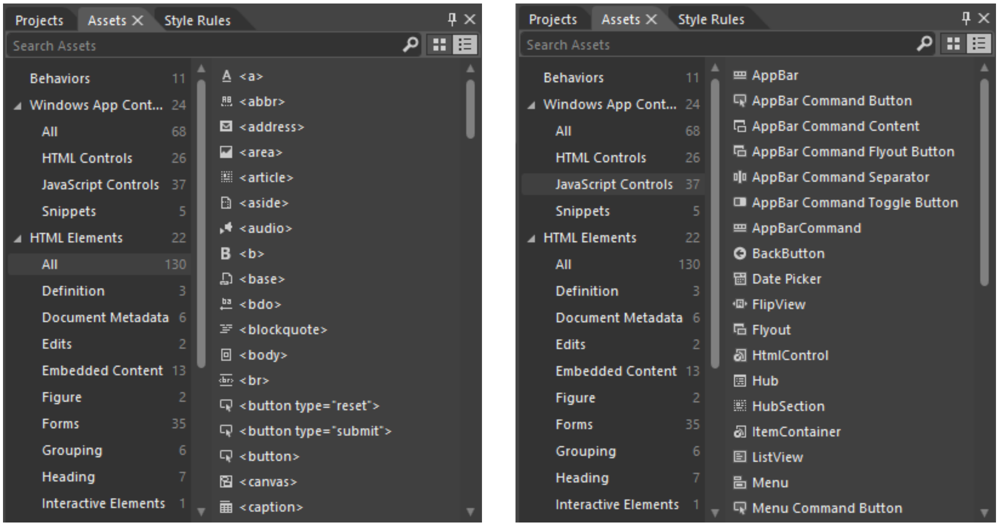
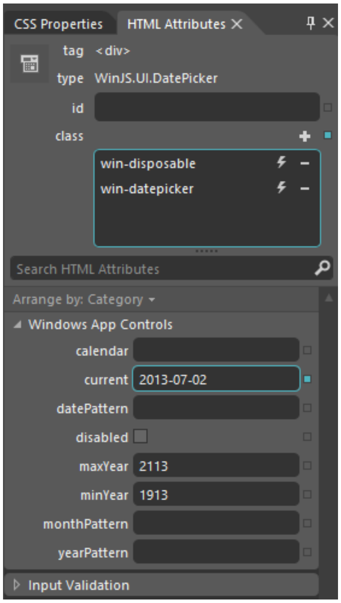


FIGURE 5-4 HTML elements (left) and WinJS control (right) as shown in Blend's Assets tab.

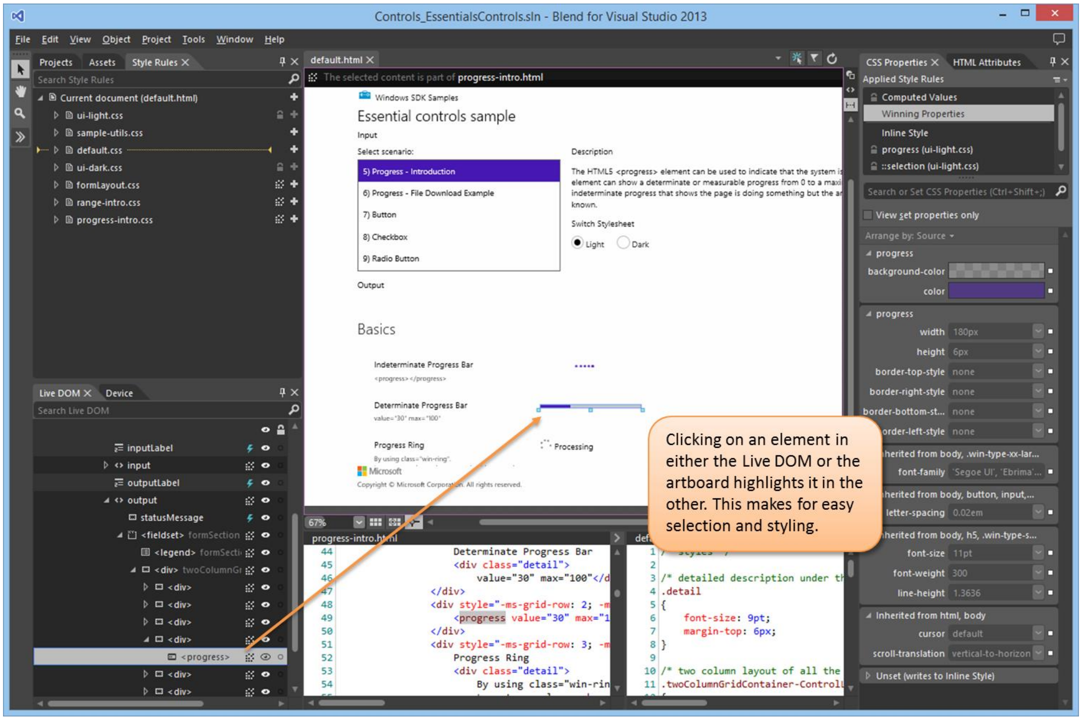


tag <div>
type WinJS.UI.DatePicker
id
class
win-disposable
win-datepicker

calendar
current 2013-07-02
datePattern
disabled □
maxYear 2113
minYear 1913
monthPattern
yearPattern

FIGURE 5-5 Blend's HTML Attributes tab shows WinJS control options, and editing them will affect the `data-win-options` attribute in markup.

Next, take a moment to load up the HTML essential controls sample into Blend. This is a great opportunity to try out Blend's Interactive Mode to navigate to a particular page and explore the interaction between the artboard and the Live DOM. (See Figure 5-6.) Once you open the project, go into interactive mode by selecting View -> Interactive Mode on the menu, pressing Ctrl+Shift+I, or clicking the small leftmost button on the upper right corner of the artboard. Then select scenario 5 (Progress introduction) in the listbox, which will take you to the page shown in Figure 5-6. Then exit interactive mode (same commands), and you'll be able to click around on that page. A short demonstration of using interactive mode in this way is given in Video 5-3.



Windows SDK Samples

Essential controls sample

Input

Select scenario:

5) Progress - Introduction
6) Progress - File Download Example
7) Button
8) Checkbox
9) Radio Button

Description

The HTML5 <progress> element can be used to indicate that the system is element can show a determinate or measurable progress from 0 to a maxi indeterminate progress that shows the page is doing something but the ar known.

Switch Stylesheet

● Light ○ Dark

Output

Basics

Indeterminate Progress Bar
<progress> </progress>

Determinate Progress Bar
value="30" max="100"

Progress Ring
By using class="win-ring".

Processing

Microsoft
Copyright © Microsoft Corporation. All rights reserved.

```
                    Determinate Progress Bar
                    <div class="detail">
                        value="30" max="100"</d
                </div>
                <div style="-ms-grid-row: 2; -m
                    <progress value="30" max="1
                </div>
                <div style="-ms-grid-row: 3; -m
                    Progress Ring
                    <div class="detail">
                        By using class="win-rin
```

```
/* styles */

/* detailed description under th
.detail
{
    font-size: 9pt;
    margin-top: 6px;
}

/* two column layout of all the
.twoColumnGridContainer-Controll
```

progress
background-color ▨
color 🟪

progress
width 180px
height 6px
border-top-style none
border-right-style none
border-bottom-st... none
order-left-style none

herited from body, .win-type-xx-lar...
font-family 'Segoe UI', 'Ebrima'...

herited from body, button, input,...
letter-spacing 0.02em

herited from body, h5, .win-type-s...
font-size 11pt
font-weight 300
line-height 1.3636

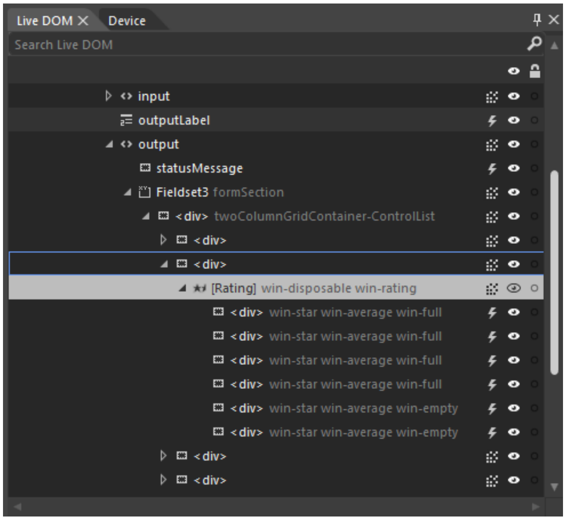
Inherited from html, body
cursor default
scroll-translation vertical-to-horizon

Clicking on an element in either the Live DOM or the artboard highlights it in the other. This makes for easy selection and styling.

FIGURE 5-6 Blend's interaction between the artboard and the Live DOM.

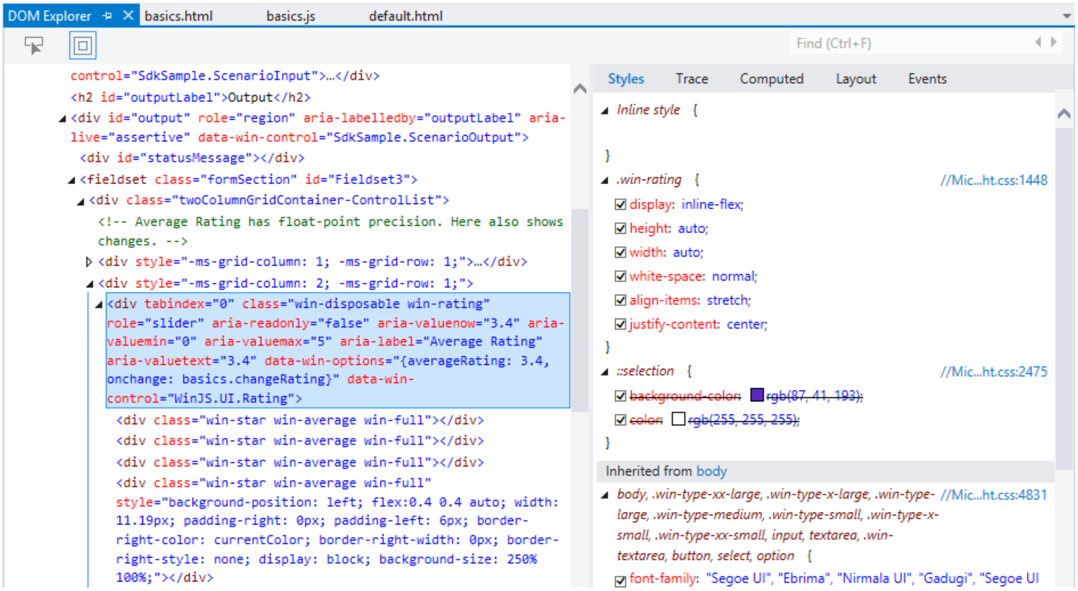
With the HTML essential controls sample, you'll see that there's just a single element in the Live DOM for intrinsic controls, as there should be, because all the internal details are part and parcel of the HTML/CSS rendering engine. On the other hand, load up the HTML Rating control sample instead and expand the div that contains one such control. There you'll see all the additional child elements that make up this control (shown in Figure 5-7), and you can refer to the right-hand pane for HTML attributes and CSS properties. You can see something similar (with even more detailed information), in the DOM Explorer of Visual Studio when the app is running. (See Figure 5-8.)

Tip To take a peek at what `win-*` and other classes are added to various WinJS controls, run a suitable app inside Visual Studio. In the DOM Explorer pane, navigate to the controls you're interested in and you'll see both their internal structure and the classes that are being applied. You can then also modify styles within the DOM Explorer to see their immediate effects, which shortcuts the usual trial-and-error experience with CSS (as you can also do in Blend). Once you know the styles you need, you can write them into your CSS files. For a short demonstration, see Video 5-4.



```
▷ <> input
   outputLabel
◢ <> output
     statusMessage
   ◢ Fieldset3 formSection
     ◢ <div> twoColumnGridContainer-ControlList
       ▷ <div>
       ◢ <div>
         ◢ [Rating] win-disposable win-rating
             <div> win-star win-average win-full
             <div> win-star win-average win-full
             <div> win-star win-average win-full
             <div> win-star win-average win-full
             <div> win-star win-average win-empty
             <div> win-star win-average win-empty
       ▷ <div>
       ▷ <div>
```

FIGURE 5-7 Expanding a WinJS control in Blend's Live DOM reveals the elements that are used to build it.



```
    control="SdkSample.ScenarioInput">…</div>
    <h2 id="outputLabel">Output</h2>
  ◢ <div id="output" role="region" aria-labelledby="outputLabel" aria-
    live="assertive" data-win-control="SdkSample.ScenarioOutput">
     <div id="statusMessage"></div>
   ◢ <fieldset class="formSection" id="Fieldset3">
     ◢ <div class="twoColumnGridContainer-ControlList">
        <!-- Average Rating has float-point precision. Here also shows
        changes. -->
      ▷ <div style="-ms-grid-column: 1; -ms-grid-row: 1;">…</div>
      ◢ <div style="-ms-grid-column: 2; -ms-grid-row: 1;">
        ◢ <div tabindex="0" class="win-disposable win-rating"
          role="slider" aria-readonly="false" aria-valuenow="3.4" aria-
          valuemin="0" aria-valuemax="5" aria-label="Average Rating"
          aria-valuetext="3.4" data-win-options="{averageRating: 3.4,
          onchange: basics.changeRating}" data-win-
          control="WinJS.UI.Rating">
           <div class="win-star win-average win-full"></div>
           <div class="win-star win-average win-full"></div>
           <div class="win-star win-average win-full"></div>
           <div class="win-star win-average win-full"
           style="background-position: left; flex:0.4 0.4 auto; width:
           11.19px; padding-right: 0px; padding-left: 6px; border-
           right-color: currentColor; border-right-width: 0px; border-
           right-style: none; display: block; background-size: 250%
           100%;"></div>
```

```
◢ Inline style {

}
◢ .win-rating {                                  //Mic...ht.css:1448
  ☑ display: inline-flex;
  ☑ height: auto;
  ☑ width: auto;
  ☑ white-space: normal;
  ☑ align-items: stretch;
  ☑ justify-content: center;
}
◢ ::selection {                                  //Mic...ht.css:2475
  ☑ ~~background-color:~~ ■ ~~rgb(87, 41, 193);~~
  ☑ ~~color:~~ □ ~~rgb(255, 255, 255);~~
}
Inherited from body
◢ body, .win-type-xx-large, .win-type-x-large, .win-type-  //Mic...ht.css:4831
  large, .win-type-medium, .win-type-small, .win-type-x-
  small, .win-type-xx-small, input, textarea, .win-
  textarea, button, select, option {
  ☑ font-family: "Segoe UI", "Ebrima", "Nirmala UI", "Gadugi", "Segoe UI
```

FIGURE 5-8 Expanding a WinJS control in Visual Studio's DOM Explorer also shows complete details for a control.

Control Styling

Now we come to a topic where we'll mostly get to look at lots of pretty pictures: the various ways in which HTML and WinJS controls can be styled. As we've discussed, this happens through CSS all the way, either in a stylesheet or by assigning `style.*` properties, meaning that apps have full control over the appearance of controls. In fact, absolutely *everything* that's visually different between HTML controls in a Windows Store app and the same controls on a web page is due to styling and styling alone.

> **Design help** Some higher-level design guidance for styling is available in the documentation: Branding your Windows Store apps.

For both HTML and WinJS controls, CSS standards apply including pseudo-selectors like `:hover`, `:active`, `:checked`, and so forth, along with `-ms-*` prefixed styles for emerging standards.

For HTML controls, there are also additional `-ms-*` styles—that aren't part of CSS3—to isolate specific parts of those controls. This is because the constituent parts of such controls don't exist separately in the DOM. So pseudo-selectors—like `::-ms-check` to isolate a checkbox mark and `::-ms-fill-lower` to isolate the left or bottom part of a slider—allow you to communicate styling to the depths of the rendering engine. In contrast, all parts of WinJS controls are addressable in the DOM, so they are just styled with specific `win-*` classes defined in the WinJS stylesheets and the controls are simply rendered with those style classes. Default styles are defined in the WinJS stylesheets, but apps can override any aspect of those to style the controls however you want.

In a few cases, as already pointed out, certain `win-*` classes define style packages for use with HTML controls, such as `win-backbutton`, `win-vertical` (for a slider) and `win-ring` (for a progress control). These are intended to style standard HTML controls to look like special system controls.

There are also a few general purpose `-ms-*` styles (not selectors) that can be applied to many controls (and elements in general), along with some general WinJS `win-*` style classes. These are summarized in the following table.

Style or Class	Description
`-ms-user-select: none | inherit | element | text | auto`	Enables or disables selection for an element. Setting to `none` is particularly useful to prevent selection in text elements.
`-ms-zoom: <percentage>`	Optical zoom (magnification).
`-ms-touch-action: auto | none` (and more)	Allows specific tailoring of a control's touch experience, enabling more advanced interaction models.
`-ms-touch-select: grippers | none`	Toggles "gripper" visual elements for touch text selection.
`win-interactive`	Prevents default behaviors for controls contained inside ItemContainer, FlipView, and ListView controls (see Chapter 7 for the latter two).
`win-swipeable`	Sets `-ms-touch-action` styles so a control within a ListView can be swiped (to select) in one direction without causing panning in the other.
`win-small`, `win-medium`, `win-large`	Size variations to some controls.
`win-textarea`	Sets typical text editing styles.

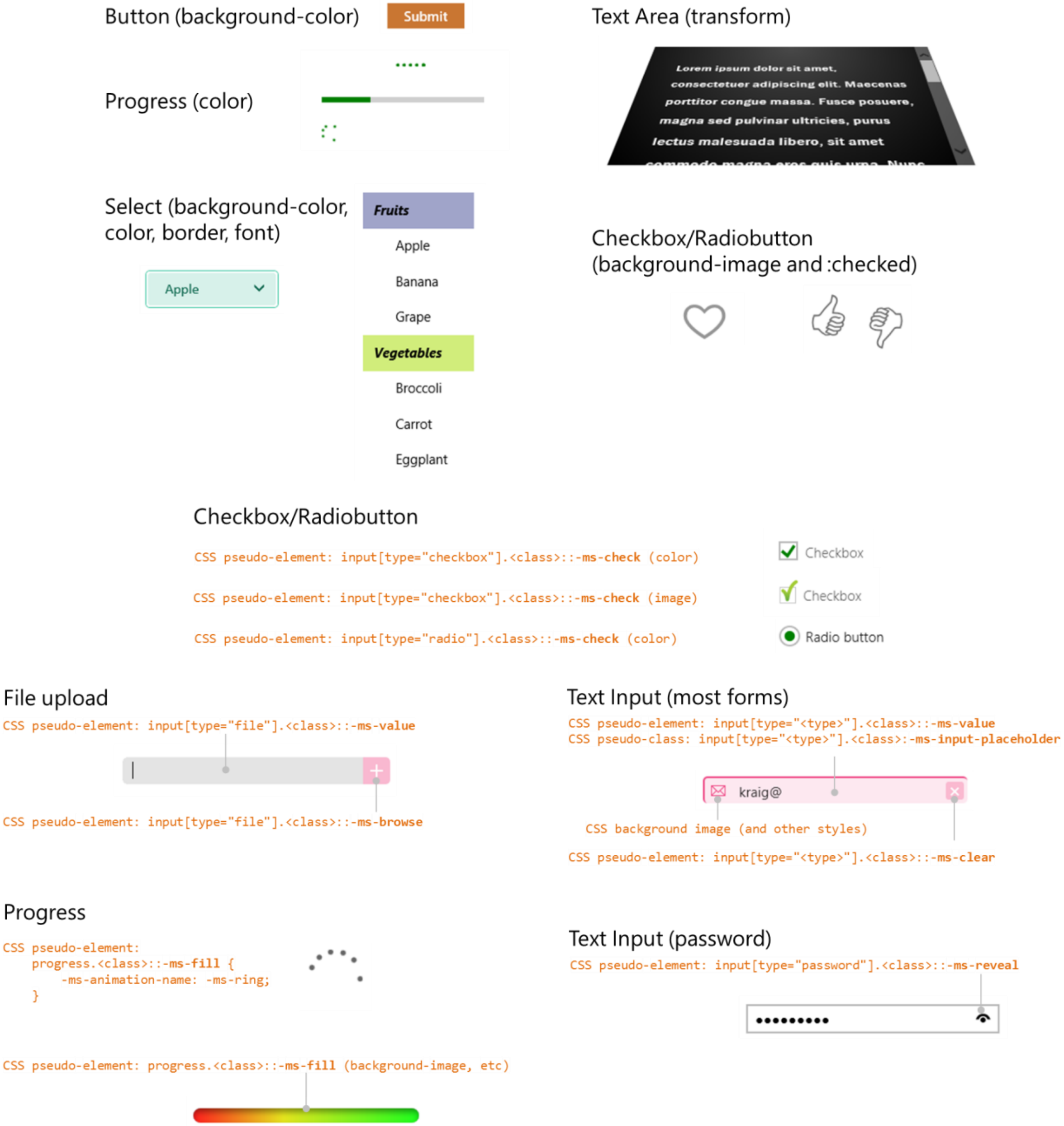
For all of these and more, spend some time with these three reference topics: WinJS CSS classes for typography, WinJS CSS classes for HTML controls, and CSS classes for WinJS controls. I also wanted to provide you with a summary of all the other vendor-prefixed styles (or selectors) that are supported within the CSS engine for Store apps; see the next table. Vendor-prefixed styles for animations, transforms, and transitions are still supported, though no longer necessary, because these standards have recently been finalized. I made this list because the documentation here can be hard to penetrate: you have to click through the individual pages under the Cascading Style Sheets topic in the docs to see what little bits have been added to the CSS you already know.

Area	Styles
Backgrounds and borders	-ms-background-position-[x | y]
Box model	-ms-overflow-[x | y]
Basic UI	-ms-text-overflow (for ellipses rendering) -ms-user-select (sets or retrieves where users are able to select text within an element) -ms-zoom (optical zoom)
Exclusions	-ms-wrap-[flow | margin | through]
Grid	-ms-grid and -ms-grid-[column | column-align | columns | column-span | grid-layer | row | row-align | rows | row-span]
High contrast	-ms-high-contrast-adjust
Regions	-ms-flow-[from | into] along with the MSRangeCollection method
Text	-ms-block-progression, -ms-hyphens and -ms-hypenate-limit-[chars | lines | zone], -ms-text-align-last, -ms-word-break, -ms-word-wrap, -ms-ime-mode, -ms-layout-grid and -ms-layout-grid-[char | line | mode | type], and -ms-text-[autospace | justify | overflow | underline-position]
Panning and Zooming	-ms-content-zoom-[chaining | limit | limit-max | limit-min | snap | snap-points | snap-type], -ms-content-zooming, -ms-overflow-style, -ms-scroll-[chaining | limit | limit-x-max | limit-x-min | limit-y-max | limit-y-min | rails | snap-points-x | snap-points-y | snap-type | snap-x | snap-y | translation]
Other	-ms-writing-mode

Styling Gallery: HTML Controls

Now we get to enjoy a visual tour of styling capabilities for Windows Store apps. Much can be done with standard styles, and then there are all the things you can do with special styles and pseudo-elements as shown in the graphics in this section. The specifics of all these examples can be seen in the HTML essential controls sample.

Also check out the very cool Applying app theme color (theme roller) sample. This beauty lets you configure the primary and secondary colors for an app, shows how those colors affect different controls, and produces about 200 lines of precise CSS that you can copy into your own stylesheet; you can also copy the appropriate color code into the branding fields of your manifest and use them in other branding graphics. This very much helps you create a color theme for your app, which we very much encourage to establish an app's own personality within the overall Windows design guidelines and not try to look like the system itself. (Do note that controls in system-provided UI, like the confirmation flyout when creating secondary tiles, system toast notifications, and message dialogs, will be styled with system colors. These cannot be controlled or duplicated by the app.)



Button (background-color)
Submit
Progress (color)
Text Area (transform)

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Maecenas porttitor congue massa. Fusce posuere, magna sed pulvinar ultricies, purus lectus malesuada libero, sit amet commodo magna eros quis urna. Nunc

Select (background-color, color, border, font)
Apple
Fruits
Apple
Banana
Grape
Vegetables
Broccoli
Carrot
Eggplant
Checkbox/Radiobutton (background-image and :checked)
Checkbox/Radiobutton

```
CSS pseudo-element: input[type="checkbox"].<class>::-ms-check (color)
CSS pseudo-element: input[type="checkbox"].<class>::-ms-check (image)
CSS pseudo-element: input[type="radio"].<class>::-ms-check (color)
```

```
CSS pseudo-element: input[type="file"].<class>::-ms-value
CSS pseudo-element: input[type="file"].<class>::-ms-browse
```

```
CSS pseudo-element: input[type="<type>"].<class>::-ms-value
CSS pseudo-class: input[type="<type>"].<class>:-ms-input-placeholder
```

kraig@

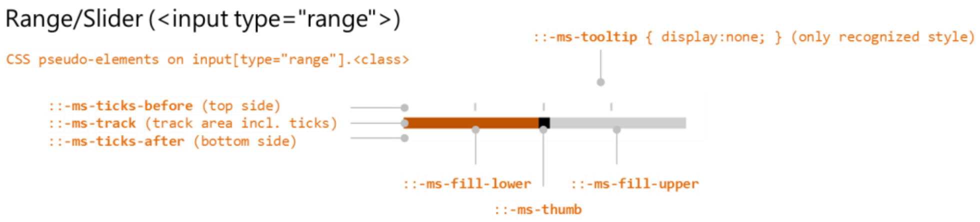
```
CSS background image (and other styles)
CSS pseudo-element: input[type="<type>"].<class>::-ms-clear
```

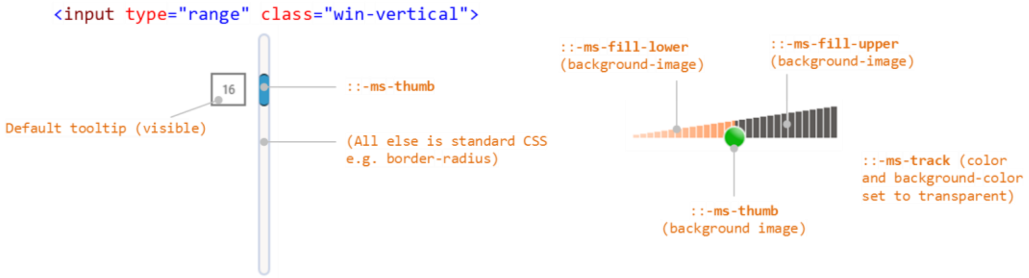
```
CSS pseudo-element:
    progress.<class>::-ms-fill {
        -ms-animation-name: -ms-ring;
    }
```

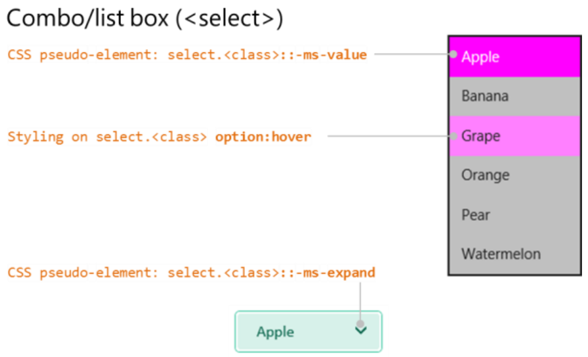
```
CSS pseudo-element: input[type="password"].<class>::-ms-reveal
```

••••••••••

```
CSS pseudo-element: progress.<class>::-ms-fill (background-image, etc)
```


Range/Slider (<input type="range">)
CSS pseudo-elements on input[type="range"].<class>
::-ms-tooltip { display:none; } (only recognized style)
::-ms-ticks-before (top side)
::-ms-track (track area incl. ticks)
::-ms-ticks-after (bottom side)
::-ms-fill-lower
::-ms-fill-upper
::-ms-thumb

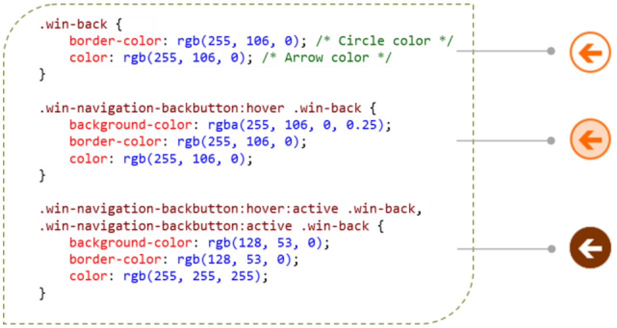



Combo/list box (<select>)
CSS pseudo-element: select.<class>::-ms-value
Styling on select.<class> option:hover
CSS pseudo-element: select.<class>::-ms-expand
Apple
Banana
Grape
Orange
Pear
Watermelon
Apple

Note Though not shown here, you can also use the `-ms-scrollbar-*` styles for scrollbars that appear on pannable content in your app.

Styling Gallery: WinJS Controls

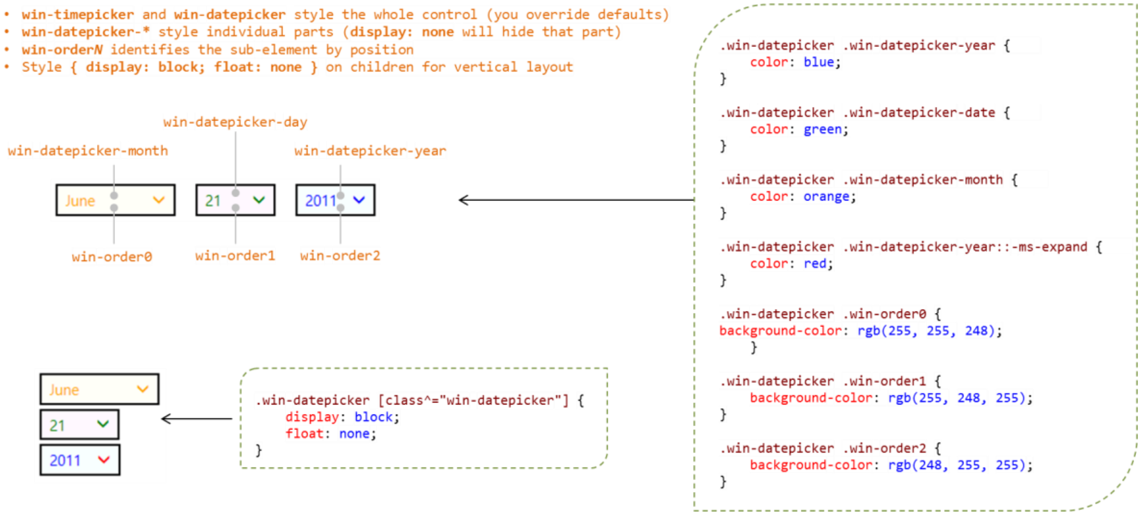
Similarly, here is a visual rundown of styling for WinJS controls, drawing again from the samples in the SDK: HTML DatePicker and TimePicker controls, HTML Rating control, HTML ToggleSwitch control, HTML Tooltip control, a modified version of the HTML Item Container sample (in the companion content), and the Navigation and navigation history sample. The latter specifically shows a little styling of the `BackButton` control. Scenario 4 (css/4_BackButton.css) overrides a few styles in the `win-navigation-backbutton` class to change the control's size. More generally, the back button control itself is made up of a `<button>` with the class `win-navigation-backbutton` (for the overall control, including pseudo-selectors) and a `<span>` with the class `win-back` (that isolates the ring and the arrow character). You can use these to change the coloration of the control:



```
.win-back {
    border-color: rgb(255, 106, 0); /* Circle color */
    color: rgb(255, 106, 0); /* Arrow color */
}

.win-navigation-backbutton:hover .win-back {
    background-color: rgba(255, 106, 0, 0.25);
    border-color: rgb(255, 106, 0);
    color: rgb(255, 106, 0);
}

.win-navigation-backbutton:hover:active .win-back,
.win-navigation-backbutton:active .win-back {
    background-color: rgb(128, 53, 0);
    border-color: rgb(128, 53, 0);
    color: rgb(255, 255, 255);
}
```

Note that if you set the `background-color` for `win-navigation-backbutton`, you'll set that color for the control's entire rectangle. By default that background is transparent, so you'll typically just style the inner part of the circle, as shown above.

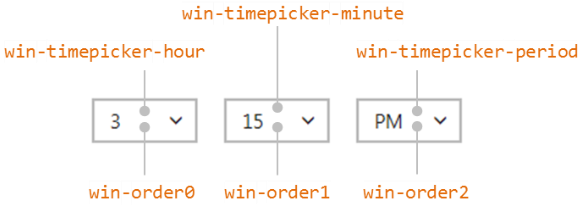
For the `DatePicker` and `TimePicker`, refer to styling for the HTML `select` element along with the `::-ms-value` and `::-ms-expand` pseudo-elements. I will note that the sample isn't totally comprehensive, so the visuals below highlight the finer points:



- **win-timepicker** and **win-datepicker** style the whole control (you override defaults)
- **win-datepicker-*** style individual parts (**display: none** will hide that part)
- **win-order*N*** identifies the sub-element by position
- Style **{ display: block; float: none }** on children for vertical layout

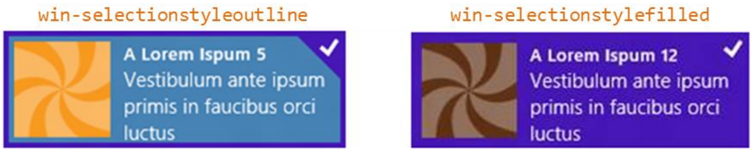
```
.win-datepicker .win-datepicker-year {
    color: blue;
}

.win-datepicker .win-datepicker-date {
    color: green;
}

.win-datepicker .win-datepicker-month {
    color: orange;
}

.win-datepicker .win-datepicker-year::-ms-expand {
    color: red;
}

.win-datepicker .win-order0 {
background-color: rgb(255, 255, 248);
    }

.win-datepicker .win-order1 {
    background-color: rgb(255, 248, 255);
}

.win-datepicker .win-order2 {
    background-color: rgb(248, 255, 255);
}
```

```
.win-datepicker [class^="win-datepicker"] {
    display: block;
    float: none;
}
```



The `ItemContainer` control has both constituent parts as well as two selection states, all of which have `win-*` classes to identify them.

First, the two selections classes are `win-selectionstylefilled` and (the default) `win-selectionstyleoutlined`. Adding these to the root element results in the standard WinJS styling below:[47]



A Lorem Ipsum 5
Vestibulum ante ipsum primis in faucibus orci luctus

A Lorem Ipsum 12
Vestibulum ante ipsum primis in faucibus orci luctus

Typically you'll use these selection style classes to create specific selectors for styling the control's individual parts, which are also identified with specific `win-*` classes:

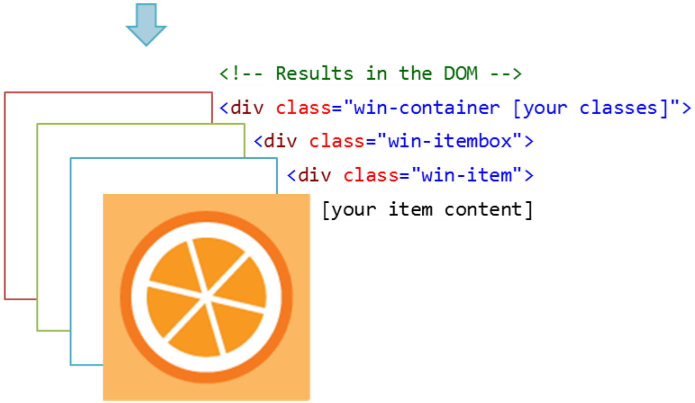
Style Class	Part
`win-container`	Styles the entire control, including areas around the control (margin and states like `:hover`).
`win-focusedoutline`	Styles the control's outline when it has the keyboard focus.
`win-itembox`	Styles the inner box containing the item; this is of limited use because the item generally overlays the item box.
`win-item`	Styles the item area; any children of the `ItemContainer` that define its contents can, of course, be styled separately through your own classes and selectors.
`win-selectioncheckmark`	Styles the checkmark character (applies to both selection styles).
`win-selectionborder`	For outline selection, styles the border lines; for filled selection, styles the color of the entire inner area (so you normally use a semitransparent fill color so that the item isn't obscured).
`win-selectioncheckmarkbackground`	Styles the triangular region around the checkmark, specifically through `border-*` styles.
`win-selectionhint`	Styles the checkmark when the item container is being swiped.

The `win-container`, `win-itembox`, and `win-item` classes identify successive layers of the control as it's built up by its constructor. That is, in your markup you'll have one root element for the

[47] Technically speaking, the WinJS stylesheets contain no references to `win-selectionstyleoutline` as all selectors simply use `:not(.win-selectionstylefilled)`.

ItemContainer control that contains the item contents. When that control is instantiated, the win-container class is added to that root element and two more div elements with win-itembox and win-item are inserted before the item contents. The classes let you target each layer for styling. Note that the layers shown below all overlap one another; the offset is added only for visualization purposes:

```
<!-- Markup in your HTML file -->
<div class="[your classes]" data-win-control="WinJS.UI.ItemContainer">
    [your item content]
</div>
```

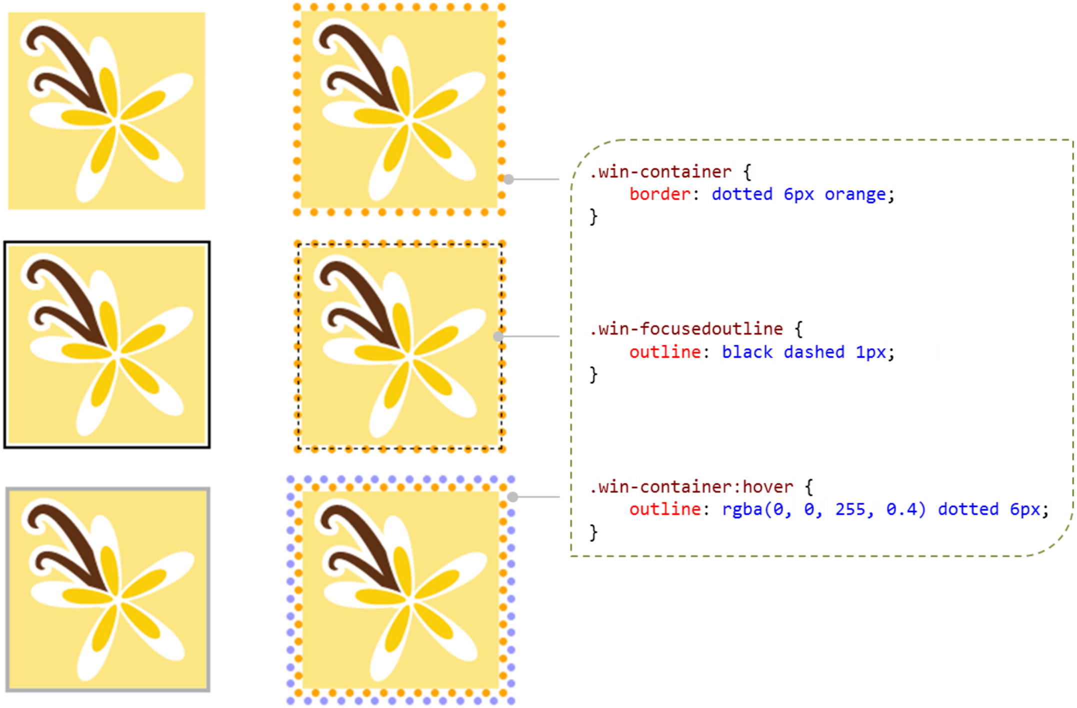


```
<!-- Results in the DOM -->
<div class="win-container [your classes]">
   <div class="win-itembox">
      <div class="win-item">
         [your item content]
```

In addition, a few more classes identify on the root element different control states or options:

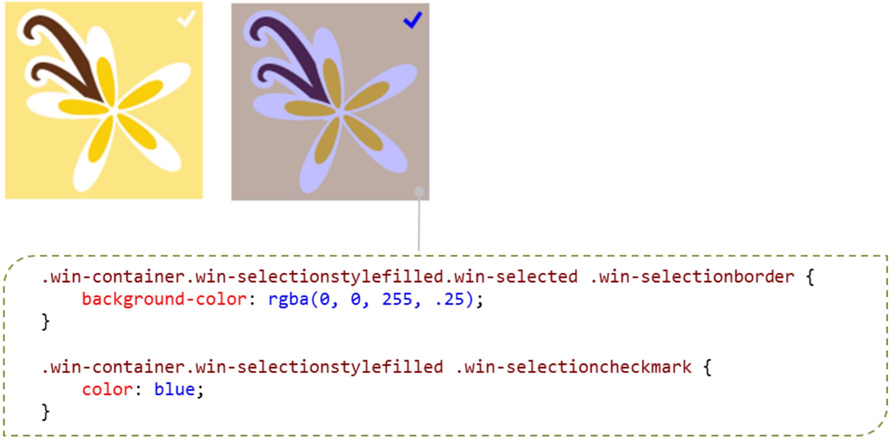
Style Class	State
win-swipeable	Added to controls with swipeBehavior set to select.
win-vertical	Added to controls with the vertical orientation.
win-horizontal	Added to controls with the vertical orientation.
win-selected	Added to controls that are in a selected state.

The following series of images are taken from the modified HTML ItemContainer sample found in this chapter’s companion content. First are some of the outline styles you can use. (Default styles are on the left, and somewhat ridiculous custom styles are shown on the right side with the applicable CSS.)



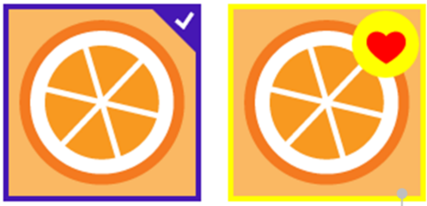
```
.win-container {
    border: dotted 6px orange;
}

.win-focusedoutline {
    outline: black dashed 1px;
}

.win-container:hover {
    outline: rgba(0, 0, 255, 0.4) dotted 6px;
}
```

With the `win-selectionstylefilled` option, the default styling is shown below left and some custom styling on the right. Notice that in the `win-selectionborder` class we style a semitransparent background-color as it overlays the whole item:



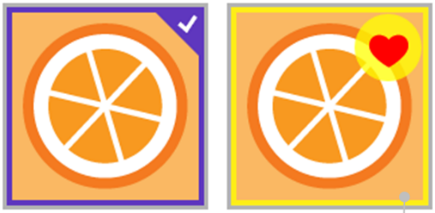
```
.win-container.win-selectionstylefilled.win-selected .win-selectionborder {
    background-color: rgba(0, 0, 255, .25);
}

.win-container.win-selectionstylefilled .win-selectioncheckmark {
    color: blue;
}
```

The following example shows default outline selection styling (left) and custom styling (right). To change the checkmark character itself, notice how we need to set the `font-size` of the default checkmark to `0px` and then use the `::before` pseudo-element to insert a different character. With the `win-selectioncheckmarkbackground` border, the default size is set to match the size of the checkmark box, and the left and bottom borders are colored with `transparent`: this is what produces the triangle. By setting the `border-width` larger, coloring all the borders, and adding a radius, we create the circle. The margin on the background separates the circle from the edge outline, and the margin on the `::before` character centers it in the circle.



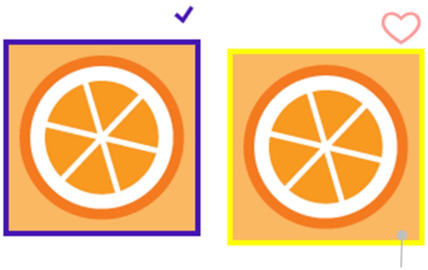
```
.win-container:not(.win-selectionstylefilled).win-selected .win-selectionborder {
    border-color: yellow;
}

.win-container:not(.win-selectionstylefilled).win-selected .win-selectioncheckmarkbackground {
    border-top-color: yellow;
    border-right-color: yellow;
    border-left-color: yellow;
    border-bottom-color: yellow;
    border-width: 25px;
    border-radius: 50px;
    margin: 5px;
}

.win-container:not(.win-selectionstylefilled) .win-selectioncheckmark {
    font-size: 0px; /* Hide the default checkmark */
    margin-right: 12px;
    margin-top: 10px;
}

.win-container:not(.win-selectionstylefilled) .win-selectioncheckmark::before {
    content: '\E0A5';
    font-size: 30px;
    color: red;
}
```

When changing the border styles, be sure to do it also for the selected *hover* state as well:



```
.win-container:not(.win-selectionstylefilled).win-selected:hover .win-selectionborder,
.win-container:not(.win-selectionstylefilled).win-selected:hover .win-selectioncheckmarkbackground {
    border-color: rgba(255, 255, 0, 0.75);
}
```

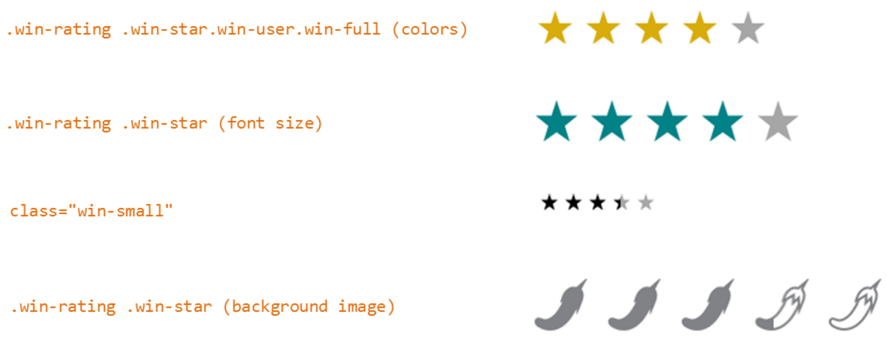
To finish up with ItemContainer, here's how we can style the selection hint (default again shown on the left)—in these cases the item is in the process of being swiped from top to bottom (see Video 5-5 for the full experience):



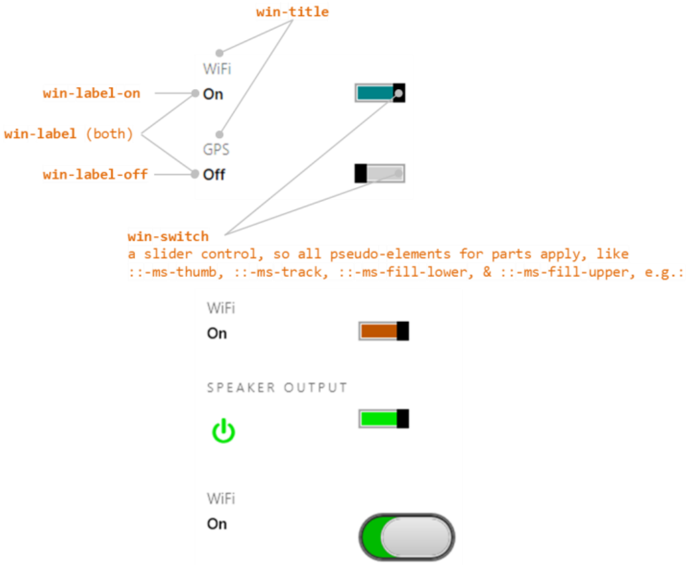
```
.win-container:not(.win-selectionstylefilled) .win-selectionhint {
    font-size: 0px; /* Hide the default checkmark */
}

.win-container:not(.win-selectionstylefilled) .win-selectionhint::before {
    content: '\E006';
    font-size: 30px;
    color: rgba(255, 0, 0, 0.4);
}
```

The Rating control similarly has states that can be styled in addition to its stars and the overall control. Again, win-* classes identify these individually and combinations style the variations:

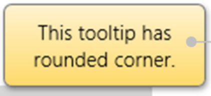
Style Class	Part
win-rating	Styles the entire control
win-star	Styles the control's stars generally
win-empty	Styles the control's empty stars
win-full	Styles the control's full stars
.win-star Classes	State
win-average	Control is displaying an average rating (user has not selected a rating and the averageRating property is non-zero)
win-disabled	Control is disabled
win-tentative	Control is displaying a tentative rating
win-user	Control is displaying user-chosen rating
Variation	Classes (selectors)
Average empty stars	.win-star.win-average.win-empty
Average full stars	.win-star.win-average.win-full
Disabled empty stars	.win-star.win-disabled.win-empty
Disabled full stars	.win-star.win-disabled.win-full
Tentative empty stars	.win-star.win-tentative.win-empty
Tentative full stars	.win-star.win-tentative.win-full
User empty stars	.win-star.win-user.win-empty
User full stars	.win-star.win-user.win-full


.win-rating .win-star.win-user.win-full (colors)
.win-rating .win-star (font size)
class="win-small"
.win-rating .win-star (background image)

For the ToggleSwitch, win-* classes identify parts of the control; states are implicit. Note that the win-switch part is just an HTML slider control (<input type="range">), so you can utilize all the pseudo-elements for its parts as shown in the "Styling Gallery: HTML Controls" section earlier.

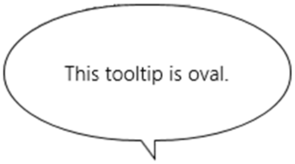

win-title
WiFi
win-label-on
On
win-label (both)
GPS
win-label-off
Off
win-switch
a slider control, so all pseudo-elements for parts apply, like
::-ms-thumb, ::-ms-track, ::-ms-fill-lower, & ::-ms-fill-upper, e.g.:
WiFi
On
SPEAKER OUTPUT
WiFi
On

And finally, for Tooltip, win-tooltip is a single class for the tooltip as a whole; the control can then contain any other HTML to which CSS applies using normal selectors. The tooltips shown here appear in relation to a gray button to which the tooltip applies:



Rounded Corner

```
#roundedCornerTooltip {
    border-radius: 6px;
    max-width: 140px;
    padding: 10px;
    border: 1px solid rgb(108, 108, 108);
    background-image:
        -ms-linear-gradient(-90deg, rgb(255, 242, 207),
        rgb(254, 218, 108));
    box-shadow: 1px 2px 6px rgba(0, 0, 0, 0.4);
    text-align: center;
}
```



Oval

```
#ovalTooltip {
    display: -ms-box;
    -ms-box-orient: vertical;
    -ms-box-align: middle;
}

#oval {
    border-radius: 100px 100px 100px 100px / 50px 50px 50px 50px;
    width: 212px;
    height: 100px;
    border: 1px solid black;
    background-color: white;
    text-align: center;
    line-height: 100px;
    color: black;
}

#beak {
    background-image: url("images/beak.svg");
    background-repeat: no-repeat;
    background-size: 100% 100%;
    width: 12px;
    height: 16px;
    margin-top: -1px;
}
```

Some Tips and Tricks

- The automatic tooltips on a slider (`<input type="range">`) are always numerical values; there isn't a means to display other forms of text, such as *Low*, *Medium*, and *High*. For something like this, you could consider a `Rating` control with three values, using the `tooltipStrings` property to customize the tooltips.

- The `::-ms-tooltip` pseudo-selector for the slider affects only visibility (with `display: none`); it cannot be used to style the tooltip generally. This is useful to hide the default tooltips if you want to implement custom UI of your own.

- There are additional types of `input` controls (different values for the `type` attribute) that I haven't mentioned. This is because those types have no special behaviors and just render as a text box. Those that have been specifically identified might also just render as a text box, but they can affect, for example, what on-screen keyboard configuration is displayed on a touch device (see Chapter 12) and also provide specific input validation (e.g., the number type only accepts digits).

- The WinJS attribute, data-win-selectable, when set to true, specifies that an element is selectable in the same way that all input and contenteditable elements are.
- If you don't find width and height properties working for a control, try using style.width and style.height instead.
- You'll notice that there are two kinds of button controls: <button> and <input type="button">. They're visually the same, but the former is a block tag and can display HTML inside itself, whereas the latter is an inline tag that displays only text. A button also defaults to <input type="submit">, which has its own semantics, so you generally want to use <button type="button"> to be sure.
- If a Tooltip is getting clipped, you can override the max-width style in the win-tooltip class, which is set to 30em in the WinJS stylesheets. Again, peeking at the style in Blend's Style Rules tab is a quick way to see the defaults.
- The HTML5 meter element is not supported for Store apps.
- There's a default dotted outline for a control when it has the focus (tabbing to it with the keyboard or calling the focus method in JavaScript). To turn off this default rectangle for a control, use <selector>:focus { outline: none; } in CSS.
- Store apps can use the window.getComputedStyle method to obtain a currentStyle object that contains the applied styles for an element, *or* for a pseudo-element. This is very helpful, especially for debugging, because pseudo-elements like ::-ms-thumb for an HTML slider control never appear in the DOM, so the styling is not accessible through the element's style property nor does it surface in tools like Blend. Here's an example of retrieving the background color style for a slider thumb:

```
var styles = window.getComputedStyle(document.getElementById("slider1"), "::-ms-thumb");
styles.getPropertyValue("background-color");
```

Custom Controls

As extensive as the HTML and WinJS controls are, there will always be something you wish the system provided but doesn't. "Is there a calendar control?" is a question I've often heard. "What about charting controls?" These clearly aren't included directly in the Windows SDK, and despite any wishing to the contrary, it means you or another third-party will need to create a custom control.

You can find a list of third-party control libraries on the Windows Partner Directory: visit http://services.windowsstore.com/ and click the "Control & Frameworks" filter on the left-hand side. Be sure to check for those that specifically offer HTML controls. Telerik, for example, provides calendar, chart, and gauge controls, among others.

> **Tip** One advantage of using built-in and professional third-party controls is that they implement Accessibility features like keyboarding and high contrast support (see Chapter 19). If you create a custom control that you might share with others, be sure to add similar support.

If none of those libraries meet your needs, you'll need to write a control of your own. Do consider using the `HtmlControl` or even `WinJS.UI.Pages` if what you need is mostly a reusable block of HTML/CSS/JavaScript without custom methods, properties, and events. Along similar lines, if what you need is a reusable block of HTML in which you want to do run-time data binding, check out `WinJS.Binding.Template`, which we'll see in Chapter 6. The `Template` isn't a control as we've been describing here—it doesn't support events, for instance—but it might be exactly what you need.

When you do need to implement a custom control, everything we've learned about WinJS controls applies. WinJS, in fact, uses the exact same control model, so you can look at the WinJS source code anytime you like for a bunch of reference implementations.

But let's spell out the pattern explicitly, recalling from our earlier definition that a control is just declarative markup (creating elements in the DOM) plus applicable CSS, plus methods, properties, and events accessible from JavaScript. Here are the steps:

1. Define a namespace for your control(s) by using `WinJS.Namespace.define` to both provide a naming scope and to keep excess identifiers out of the global namespace. (Do *not* add controls to the WinJS namespace.) Remember that you can call `WinJS.Namespace.define` many times for the same namespace to add new members, so typically an app will just have a single namespace for all its custom controls no matter where they're defined.

2. Within that namespace, define the control constructor by using `WinJS.Class.define` (or `derive`), assigning the return value to the name you want to use in `data-win-control` attributes. That fully qualified name will be `<namespace>.<constructor>`.

3. Within the constructor (of the form `<constructor>(element, options)`):
    a. You can recognize any set of options you want; these are arbitrary. Simply ignore any that you don't recognize.

    b. If `element` is `null` or `undefined`, create a `div` to use in its place.

    c. Assuming `element` is the root element containing the control, be sure to set `element.winControl=this` and `this.element=element` to match the WinJS pattern.

    d. Call `WinJS.Utilities.addClass(this.element, "win-disposable")` to indicate that the control implements the dispose pattern (see #5 below). Also set `this._disposed = false`. Alternately, use `WinJS.Utilities.markDisposable` (see the next section, "Implementing the Dispose Pattern"), which encapsulates these steps and parts of #5.

4. Within WinJS.Class.define, the second argument is an object containing your public methods and properties (those accessible through an instantiated control instance); the third argument is an object with static methods and properties (those accessible through the class name without needing to call new).

5. Implement a public method named dispose. As described in "Sidebar: The Ubiquitous dispose method" earlier in this chapter, this method is called when whatever is hosting your control is doing its cleanup. In response, the control should do the following (a generic structure is given in the next section):

    a. Mark itself as disposed by setting this._disposed = true (dispose should check if this is set to prevent reentrancy).

    b. Call removeEventListener for any event handlers added earlier.

    c. Call dispose on any child controls marked with the win-disposable class, if the method is available. This can be done with WinJS.Utilities.disposeSubTree.

    d. Cancel any outstanding async operations.

    e. Release object references, disconnect event listeners, release connections, and otherwise clean up any other allocations or resources (generally setting them to null so that the JavaScript garbage collector finds them).

6. For your events, mix (WinJS.Class.mix) your class with the results from WinJS.Utilities.createEventProperties(<events>) where <events> is an array of your event names (without on prefixes). This will create on<event> properties in your class for each name in the list.

7. Also mix your class with WinJS.UI.DOMEventMixin to add standard implementations of addEventListener, removeEventListener, dispatchEvent, and setOptions.[48]

8. In your implementation (markup and code), refer to classes that you define in a default stylesheet but that can be overridden by consumers of the control. Consider using existing win-* classes to align with general styling.

9. A typical best practice is to organize your custom controls in per-control folders that contain all the html, js, and css files for that control. Again, calls to WinJS.Namespace.define for the same namespace are additive, so you can populate a single namespace with controls that are defined in separate files.

[48] Note that there is also a WinJS.Utilities.eventMixin that is similar (without setOptions) that is useful for noncontrol objects that won't be in the DOM but still want to fire events. The implementations here don't participate in DOM event bubbling/tunneling.

Note Everything in WinJS—like WinJS.Class.define and WinJS.UI.DOMEventMixin—are just helpers for common patterns. You're not in any way required to use these, because in the end, custom controls are just elements in the DOM like any others and you can create and manage them however you like. The WinJS utilities just make most jobs cleaner, easier, and more consistent.

For more about WinJS.Class methods and mixins, see Appendix B, "WinJS Extras." Other sections of the appendix outline some obscure WinJS features that might be helpful when implementing custom controls.

Implementing the Dispose Pattern

The common pattern for the implementation of the dispose method looks like the following, which assumes that this._disposed was set to false in the constructor:

```
dispose: function () {
    if (this._disposed) { return; }
    this._disposed = true;

    // Call dispose on all children
    WinJS.Utilities.disposeSubTree(this.element);

    // Disconnect listeners. A simple button.onclick case is shown here.
    if (this._button && this._clickListener) {
        this._button.removeEventListener("click", this._clickListener, false);
    }

    // Cancel outstanding promises
    this._somePromise && this._somePromise.cancel();
    this._somePromise = null;

    //Null out other resources (however many there are)
    this._someOtherResource = null;
}
```

An example of this can be found in scenario 1 of the Dispose model sample.

A simpler way of implementing the pattern—and one that relieves you from having to remember certain details—is to use the WinJS.Utilities.markDisposable method. This adds the standard win-disposable class to an element and automatically provides a this._disposed flag. The part you provide is a function that contains your specific disposal code. Here's how markDisposable is implemented:

```
markDisposable: function (element, disposeImpl) {
    var disposed = false;
    WinJS.Utilities.addClass(element, "win-disposable");

    var disposable = element.winControl || element;
    disposable.dispose = function () {
        if (disposed) {
            return;
        }
        disposed = true;
        WinJS.Utilities.disposeSubTree(element);
        if (disposeImpl) {
            disposeImpl();
        }
    };
```

The benefit of `markDisposable` is that it gives you a way to put the `dispose` code right inside your constructor, where you can more easily match the construction and disposal steps. Scenario 2 of the Dispose model sample shows how to use this construct, and we'll see another example in the next section that gives us more context for discussion.

Sidebar: Using the WinJS Scheduler in Custom Controls

If you do any kind of async work in a custom control, be sure to employ the `WinJS.Utilities.Scheduler` API to appropriately mark each async task's relative priority. Typically, UI refresh work gets a higher priority, whereas secondary work like preloading additional content can be scheduled at low priority. If you search for "Scheduler" in the WinJS source file ui.js, you'll see many examples, and refer back to Chapter 3 for the general discussion of the Scheduler.

Custom Control Examples

To see this pattern in action, here are a couple of examples. First is what Chris Tavares, one of the WinJS engineers who has been a tremendous help with this book, described as the "dumbest control you can imagine." Yet it certainly shows the most basic structures (in this case `dispose` isn't needed):

```
WinJS.Namespace.define("AppControls", {
    HelloControl: WinJS.Class.define(function (element, options) {
        element.winControl = this;
        this.element = element;

        if (options.message) {
            element.innerText = options.message;
        }
    })
});
```

With this, you can then use the following markup so that `WinJS.UI.process/processAll` will instantiate an instance of the control (as an inline element because we're using `span` as the root):

```
<span data-win-control="AppControls.HelloControl"
    data-win-options="{ message: 'Hello, World'}">
</span>
```

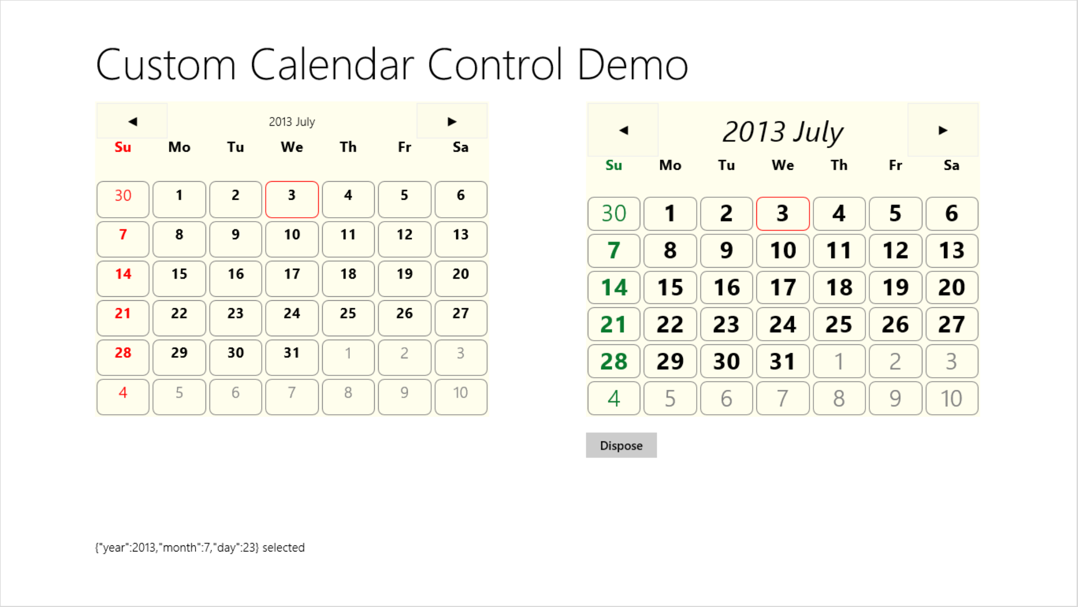
Note that the control definition code must be executed before `WinJS.UI.process/processAll` so that the constructor function named in `data-win-control` actually exists at that point.

For a more complete control, you can take a look at the HTML SemanticZoom for custom controls sample. There is also a post on the Windows Developer blog that takes a patterns-oriented approach to this subject. As for us here, let's work with the CalendarControl example in this chapter's companion content, which was created by my friend Kenichiro Tanaka of Microsoft Tokyo and is shown in Figure 5-9. (Note that this is example is only partly sensitive to localized calendar settings and doesn't support accessibility requirements; it is not meant to be full-featured.)

Following the steps given earlier, this control is defined using `WinJS.Class.define` within a Controls namespace (controls/calendar/calendar.js lines 4–12 shown here):

```
WinJS.Namespace.define("Controls", {
    Calendar : WinJS.Class.define(
        // constructor
        function (element, options) {
            this.element = element || document.createElement("div");
            this.element.className = "control-calendar";
            this.element.winControl = this;
```

The rest of the constructor (lines 14–72) builds up the child elements that define the control, making sure that each piece has a particular class name that, when scoped with the `control-calendar` class placed on the root element above, allows specific styling of the individual parts. The defaults for this are in controls/calendar/calendar.css; specific overrides that differentiate the two controls in Figure 5-9 are in css/default.css.



Custom Calendar Control Demo

◄			2013 July			►
Su	Mo	Tu	We	Th	Fr	Sa
30	1	2	3	4	5	6
7	8	9	10	11	12	13
14	15	16	17	18	19	20
21	22	23	24	25	26	27
28	29	30	31	1	2	3
4	5	6	7	8	9	10

◄			2013 July			►
Su	Mo	Tu	We	Th	Fr	Sa
30	1	2	3	4	5	6
7	8	9	10	11	12	13
14	15	16	17	18	19	20
21	22	23	24	25	26	27
28	29	30	31	1	2	3
4	5	6	7	8	9	10

{"year":2013,"month":7,"day":23} selected

FIGURE 5-9 Output of the Calendar Control example.

Within the constructor you can also see that the control wires up its own event handlers for its child elements, such as the previous/next buttons and each date cell. In the latter case, clicking a cell uses `dispatchEvent` to raise a `dateselected` event from the overall control itself. The event handler we assign to this generates the output along the bottom of the screen.

Lines 74–169 then define the members of the control. There are two internal methods, `_setClass` and `_update`, followed by three public methods, `dispose`, `nextMonth` and `prevMonth`, followed by three public properties, `year`, `month`, and `date`. Those properties can be set through the `data-win-options` string in markup or directly through the control object as we'll see in a moment.

To implement `dispose`, the Calendar Control uses `WinJS.Utilities.markDisposable` within the constructor:

```
WinJS.Utilities.markDisposable(this.element, disposeImpl.bind(this));
```

The `disposeImpl` function we pass to `markDisposable`, which is defined in the constructor itself, carefully reverses any allocations made in the constructor or elsewhere in the control's methods (look especially for uses of `new`). This includes an instance of `Windows.Globalization.calendar` (which could be replaced in `_update`), an array of cells, and quite a number of child elements. It also added event listeners to most of those elements.

When implementing your disposal process, first copy and paste your constructor code and then modify it to reverse each action. This makes it easier to see everything that was done in the constructor so that you won't forget anything. In the process, watch for any variables that you declared in the constructor with `var <name>` to access some part of the control, such as child elements. Because you'll probably need these again in `dispose`, change each one from `var <name>` to `this._<name>` such that you'll have it later on. For example, the calendar saves its header element like so:

```
this._header = document.createElement("div");
// Other initialization
this.element.appendChild(this._header);
```

Similarly, assign event handlers in `this._<handler>` properties. For example:

```
this._prevListener = function () {
    this.prevMonth();
};

this.element.querySelector(".prev").addEventListener("click", this._prevListener.bind(this));
```

Looking at the control's disposal code (the function passed to `markDisposable`), then, we can see how handy it is to have these instance properties around, especially for `removeEventListener`:[49]

```
function disposeImpl () {
    //Reverse the constructor's steps
    this._cal = null;
```

[49] If you're using `markDisposable` and write your disposal code within the constructor, you can depend on closures instead of instance variables. However, I prefer to keep instance variables clearly visible by referencing them through `this`.

```
    var prev = this.element.querySelector(".prev");
    prev && prev.removeEventListener("click", this._prevListener);

    var next = this.element.querySelector(".next");
    next && next.removeEventListener("click", this._nextListener);

    this.element.removeChild(this._header);

    var that = this;
    this._cells.forEach(function (cell) {
        cell.removeEventListener("click", that._cellClickListener);
        that._body.removeChild(cell);
    });

    this._cells = null;
    this.element.removeChild(this._body);
};
```

To test all this, you can click the Dispose button underneath the right-hand calendar in Figure 5-9. This calls the right-hand calendar's `dispose` method, after which the only thing left is the root `div` in which the control was created. The button's `click` handler then hides that `div`.

Anyway, at the very end of controls/calendar/calendar.js you'll see the two calls to `WinJS.Class.mix` to add properties for the events (there's only one here), and the standard DOM event methods like `addEventListener`, `removeEventListener`, and `dispatchEvent`, along with `setOptions`:

```
WinJS.Class.mix(Controls.Calendar, WinJS.Utilities.createEventProperties("dateselected"));
WinJS.Class.mix(Controls.Calendar, WinJS.UI.DOMEventMixin);
```

Very nice that adding all these details is so simple—thank you, WinJS![50]

Between controls/calendar/calendar.js and controls/calendar/calendar.css we thus have the complete definition of the control. In default.html and default.js we can then see how the control is used. In Figure 5-9, the control on the left is declared in markup and instantiated through the call to `WinJS.UI.processAll` in default.js.

```
<div id="calendar1" class="control-calendar" aria-label="Calendar 1"
    data-win-control="Controls.Calendar"
    data-win-options="{ year: 2012, month: 5, ondateselected: CalendarDemo.dateselected}">
</div>
```

You can see how we use the fully qualified name of the constructor as well as the event handler we're assigning to `ondataselected`. But remember that functions referenced in markup like this have to be marked for strict processing. The constructor is automatically marked through `WinJS.Class.define`, but the event handler needs extra treatment: we place the function in a

[50] Technically speaking, `WinJS.Class.mix` accepts a variable number of arguments, so you can actually combine the two calls into a single one. See Appendix B for more details.

namespace (to make it globally visible) and use WinJS.UI.eventHandler to do the marking:

```
WinJS.Namespace.define("CalendarDemo", {
    dateselected: WinJS.UI.eventHandler(function (e) {
        document.getElementById("message").innerText = JSON.stringify(e.detail) + " selected";
    })
});
```

Again, if you forget to mark the function in this way, the control won't be instantiated at all. (Remove the WinJS.UI.eventHandler wrapper to see this.)

To demonstrate creating a control outside of markup, the control on the right of Figure 5-9 is created as follows, within the *calendar2* div:

```
//Because we're creating this calendar in code, we're independent of WinJS.UI.processAll.
var element = document.getElementById("calendar2");

//Because we're providing an element, this will be automatically added to the DOM.
var calendar2 = new Controls.Calendar(element);

//Because this handler is not part of markup processing, it doesn't need to be marked.
calendar2.ondateselected = function (e) {
    document.getElementById("message").innerText = JSON.stringify(e.detail) + " selected";
}
```

There you have it!

Note For a control you really intend to share with others, you'll want to include the necessary comments that provide metadata for IntelliSense. See the "Sidebar: Helping Out IntelliSense" in Chapter 3 for more details. You'll also want to make sure that the control fully supports considerations for accessibility and localization, as discussed in Chapter 19.

Custom Controls in Blend

Blend is an excellent design tool for working with controls directly on the artboard, so you might be wondering how custom controls integrate into that story.

First, because custom controls are just elements in the DOM, Blend works with them like all other parts of the DOM. Try loading the Calendar Control example into Blend to see for yourself.

Next, a control can determine if it's running inside Blend's design mode if the Windows.ApplicationModel.DesignMode.designModeEnabled property is true. One place where this is very useful is when handling resource strings. We won't cover resources in full until Chapter 19, but it's important to know here that resource lookup through the data-win-res attribute works just fine in design mode but not through Windows.ApplicationModel.Resources.ResourceLoader (it throws exceptions). If you run into this, you can use the design-mode flag to just provide a suitable default instead of doing the lookup.

For example, one of the early partners I worked with had a method to retrieve a localized URI to their back-end services, which was failing in design mode. Using the design mode flag, then, we just had to change the code to look like this:

```
WinJS.Namespace.define("App.Localization", {
    getBaseUri: function () {
        if (Windows.ApplicationModel.DesignMode.designModeEnabled) {
            return "www.default-base-service.com";
        } else {
            var resources = new Windows.ApplicationModel.Resources.ResourceLoader();
            var baseUri = resources.getString("baseUrl");
            return baseUri;
        }
    }
});
```

Alternately, you could create a hidden element in the DOM that uses `data-win-res` to load the string into itself and then have a function like the below retrieve that string:

```
<!–In markup -->
<div id="baseUri" data-win-res="textContent: 'baseUrl'" style="display: none;"></div>

//In code
WinJS.Namespace.define("App.Localization", {
    getBaseUri: function () {
        return document.getElementById("baseUri").textContent;
    }
});
```

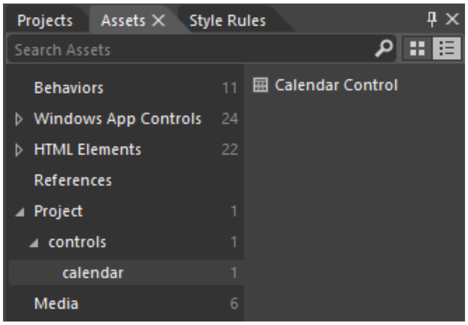
Finally, it is possible to have custom controls show up in the Assets tab alongside the HTML elements and the WinJS controls. For this you'll first need an OpenAjax Metadata XML (OAM) file that provides all the necessary information for the control, and you already have plenty of references to draw from. To find them, search for *_oam.xml files within *Program Files (x86)*. You should find some under the *Microsoft Visual Studio 12.0* folder and *deep* down within *Microsoft SDKs* where WinJS metadata lives. In both places you'll also find plenty of examples of the 12x12 and 16x16 icons you'll want for your control.

If you look in the controls/calendar folder of the CalendarControl example with this chapter, you'll find calendar_oam.xml and two icons alongside the .js and .css files. The OAM file, which must have a filename ending in *_oam.xml*, tells Blend how to display the control in its Assets panel and what code it should insert when you drag and drop the control into an HTML file. Here are the contents of that file:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- Use underscores or periods in the id and name, not spaces. -->
<widget version="2.0"
    spec="2.0"
    id="http://www.kraigbrockschmidt.com/schemas/ProgrammingWin_JS/Controls/Calendar"
    name="ProgWin_JS.Controls.Calendar"
    xmlns="http://openajax.org/metadata">

    <author name="Kenichiro Tanaka" />

    <!-- title provides the name that appears in Blend's Assets panel
```

```
        (otherwise it uses the widget.name). -->
    <title type="text/plain"><![CDATA[Calendar Control]]></title>

    <!-- description provides the tooltip fir Assets panel. -->
      <description type="text/plain"><![CDATA[A single month calendar]]></description>

    <!-- icons (12x12 and 16x16 provide the small icon next to the control
        in the Assets panel. -->
    <icons>
        <icon src="calendar.16x16.png" width="16" height="16" />
        <icon src="calendar.12x12.png" width="12" height="12" />
    </icons>

    <!-- This element describes what gets inserted into the .html file;
        comment out anything that's not needed -->
    <requires>
        <!-- The control's code -->
        <require type="javascript" src="calendar.js" />

        <!-- The control's stylesheet -->
        <require type="css" src="calendar.css" />

        <!-- Any inline script for the document head -->
        <require type="javascript"><![CDATA[WinJS.UI.processAll();]]></require>

        <!-- Inline CSS for the style block in the document head -->
        <!--<require type="css"><![CDATA[.control-calendar{}]]></require>-->
    </requires>

    <!-- What to insert in the body for the control; be sure this is valid HTML
        or Blend won't allow insertion -->
    <content>
        <![CDATA[
            <div class="control-calendar" data-win-control="Controls.Calendar"
                data-win-options="{ year: 2012, month: 6 }"></div>
        ]]>
    </content>
</widget>
```

When you add all five files to a project in Blend, you'll see the control's icon and title in the Assets tab (and hovering over the control shows the tooltip):



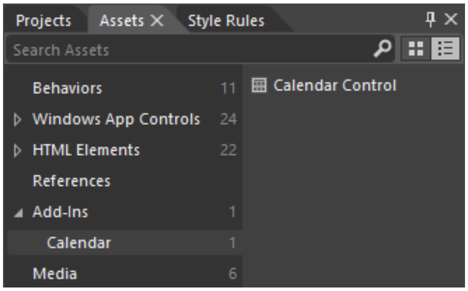
If you drag and drop that control onto an HTML page, you'll then see the different bits added in:

```
<!DOCTYPE html>
<html>
<head>
    <!-- ... -->
    <script src="calendar.js" type="text/javascript"></script>
    <link href="calendar.css" rel="stylesheet" type="text/css">
</head>
<body>
    <div class="control-calendar" data-win-control="Controls.Calendar"
        data-win-options="{month: 7, year: 2013}"></div>
</body>
</html>
```

But wait! What happened to the `WinJS.UI.processAll()` call that the XML indicated to include within a `script` tag in the header? It just so happens that Blend singles out this piece of code to check if it's already being called somewhere in the loaded script. If it is (as is typical with the project templates), Blend doesn't repeat it. If it does include it, or if you specify other code here, Blend will insert it in a `<script>` tag in the header. Of course, you'll probably want to move that call into a .js file.

Also, errors in your OAM file will convince Blend that it shouldn't insert the control at all, so be careful with the details. When making changes, Blend won't reload the metadata unless you reload the project or rename the OAM file (preserving the _oam.xml part). I found the latter is much easier, as Blend doesn't care what the rest of the filename looks like. In this renaming process too, if you find that the control disappeared from the Assets panel, it means you have an error in the OAM XML structure itself, such as attribute values containing invalid characters. For this you'll need to do some trial and error, and of course you can refer to all the OAM files already on your machine for details.

You can also make your control available to all projects in Blend. To do this, go to *Program Files (x86)\Microsoft Visual Studio 12.0\Blend*, create a folder called *Addins* if one doesn't exist, create a subfolder therein for your control (using a reasonably unique name), and copy all your control assets there (that is, the same stuff in the /controls/calendar folder of the example). When you restart Blend, you'll see the control listed under Addins in the Assets tab:



This would be appropriate if you create custom controls for other developers to use. In that case your desktop installation program would simply place your assets in the Addins folder. As for using such a control, when you drag and drop the control to an HTML file, its required assets (but not the icons nor the OAM file) are copied to the project into the root folder. You can then move them around however you like, patching up the file references as needed.

Sidebar: Custom Control Adorners

Controls that are built into Blend generally show small adorners that allow more interactivity with the control on the artboard. To do this, you implement an additional design-time DLL in C#/XAML using the Windows Presentation Foundation (WPF). More information can be found on Walkthrough: Creating a Design-time Adorner and especially Creating a Design-time adorner layer in Windows RT (Hungry Philosopher's blog).

What We've Just Learned

- The overall control model for HTML and WinJS controls, where every control consists of declarative markup, applicable CSS, and methods, properties, and events accessible through JavaScript.
- Standard HTML controls have dedicated markup; WinJS controls use `data-win-control` and `data-win-options` attributes, which are processed using `WinJS.UI.process` or `WinJS.UI.processAll`.
- Both types of controls can also be instantiated programmatically using `new` and the appropriate constructor, such as `button` or `WinJS.UI.Rating`.
- All controls have various options that can be used to initialize them. These are given as intrinsic attributes for HTML controls and within the `data-win-options` attribute for WinJS controls.
- All controls have standard styling as defined in the WinJS stylesheets: ui-light.css and ui-dark.css. Those styles can be overridden as desired, and some style classes, like `win-ring`, are used to style a standard HTML control to look like a Windows-specific control.
- Windows Store apps have rich styling capabilities for both HTML and WinJS controls alike. For HTML controls, `-ms-*`-prefixed pseudo-selectors allow you to target specific pieces of those controls. For WinJS controls, specific parts are styled using `win-*` classes that you can override.
- Custom controls are implemented in the same way WinJS controls are, and WinJS provides standard implementations of methods like `addEventListener`. It's important for custom controls to implement the WinJS `dispose` pattern and make use of the WinJS scheduler when asynchronous work is involved.
- Custom controls can also be shown in Blend's Assets panel either for a single project or for all projects.

Chapter 6

Data Binding, Templates, and Collections

Having just now in Chapter 5, "Controls and Control Styling," thoroughly teased you with plenty of UI elements to consider, I'm going to step away from UI and controls for a short time to talk about glue. You see, I have a young son and glue is a big part of his life! OK, I'm joking about the real glue, but it's an appropriate term when we talk about *data binding* because data binding is, in many ways, a kind of glue that keeps the UI we build with various controls appropriately attached to the data that the UI represents.

A while back I saw a question on one of the MSDN forums that asked, simply, "When do I use data binding at all?" It's a good question, because writers of both books and documentation often assume that their readers know the why's and when's already, so they just launch into discussions about models, views, and controllers, tossing out acronyms like MVC, MVVM, MVMMVMV, MMVMMVMVMVMVMCVVM, and so on until you think they're revving up an engine for some purpose or another or perhaps getting extra practice at writing confusing Roman numerals! Indeed, the whole subject can often be shrouded in some kind of impenetrable mystique. As I don't at all count myself among the initiates into such mysteries, I'll try to express the concepts in prosaic terms. I'll covers the basics of what data binding is and why you care about it, and then I'll demonstrate how data binding is expressed through WinJS. (Though you can implement it however you like, WinJS serves as a helpful utility here, as is true for most of the library. Other developers employ angular.js for this purpose, which is certainly another option but one that I don't address in this book.)

At first we'll be looking at data binding with simple data sources, such as single objects with interesting properties. Where data binding really starts to prove its worth, however, is with collections of such objects, which is exactly why we're talking about it here before going on to Chapter 7, "Collection Controls." To that end, we'll explore the different kinds of collections that you might encounter when writing apps, including those from WinRT, such as the vector, and the `WinJS.Binding.List` that is an essential building block for collection controls.

Speaking of building blocks, this chapter also includes a subject that flows naturally from the others: templates. A template is a way to define how to render some data structure such that you can give it any instance of that structure and have it produce UI. Again, this is clearly another building block for collection controls, but it's something you can use separately from them.

So let's play with the glue and get our hands sticky!

Data Binding

Put simply, data binding means to create relationships between properties of data objects and properties of UI elements (including styles). This way those UI elements automatically reflect what's happening in the data to which they are "bound," which is often exactly what you want to accomplish in your user experience. Data binding can also work in the other direction: data objects can also be bound to UI elements such that changes a user makes in the UI are reflected back to the data objects.

When small bits of UI and data are involved, you can easily set up all these relationships however you want, by directly initializing UI element values from their associated data object and updating those values when the data is modified. You've probably already written plenty of code like this. And to go the other direction, you can watch for change events on those UI elements and update the data objects in response. You've probably written that kind of code as well!

As the UI gets more involved, however, such as when you're using collections or you're setting up relationships to multiple properties of multiple elements, having more formalized structures to handle the data-to-UI wiring starts to make a lot of sense, especially when you start dealing with collections. It's especially helpful to have a *declarative* means to set up those relationships, rather than having to do it all through procedural code. In this section, then, we'll start off with the basics of how data binding works, and then we'll look at the features of the `WinJS.Binding` namespace and what it has to offer (including declarative structures). This gives us the basis for working with collections, which leads us naturally (in the next chapter) to how we bind collections to collection controls.

Let me note that we *won't* be talking about other formalized patterns and coding conventions for data binding, such as "model-view-controller" (MVC), "model-view-viewmodel" (MVVM), and others, primarily because this book's focus is on the features of the Windows platform more than higher-level software engineering practices. (Indeed, I've often found discussions of data binding to start right in on patterns and frameworks, leaving the basics of data binding in the dust!) If you like working with these patterns, you can of course design your app's architecture accordingly around the mechanisms that WinJS or other frameworks provide.

Data Binding Basics

The general idea of data binding is again to connect or "bind" properties of two different objects together, typically properties of a data object—or *context*—and properties of a UI object. We can generically refer these as *source* and *target*. A key here is that data binding generally happens between *properties* of the source and target objects, not the objects as a whole.

The binding can also involve converting values from one type into another, such as converting a set of separate source properties into a single string as suitable for the target. It's also possible to have multiple targets bound to the same source or one target bound to multiple sources. This flexibility is exactly why the subject of data binding can become somewhat nebulous, and why numerous conventions have evolved around it! Still, for most scenarios, we can keep the story simple.

A common data-binding scenario is shown in Figure 6-1, where we have specific properties of two UI elements, a `span` and an `img`, bound to properties of a data object. There are three bindings here: (1) the `span.innerText` property is bound to the `source.name` property; (2) the `img.src` property is bound to the `source.photoURL` property; and (3) the `span.style.color` property is bound to the output of a converter function that changes the `source.userType` property into a color.

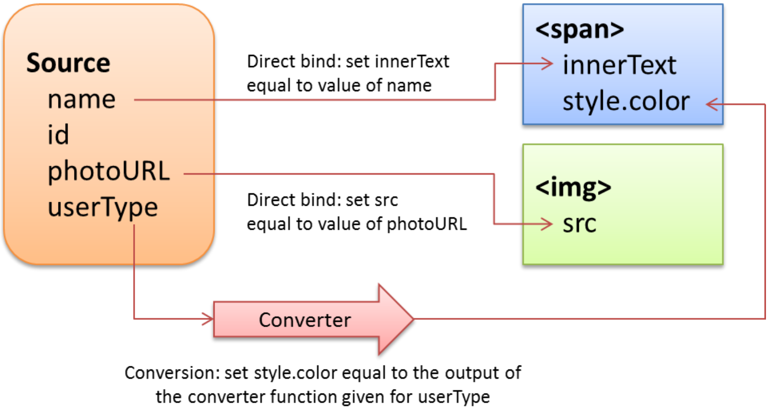
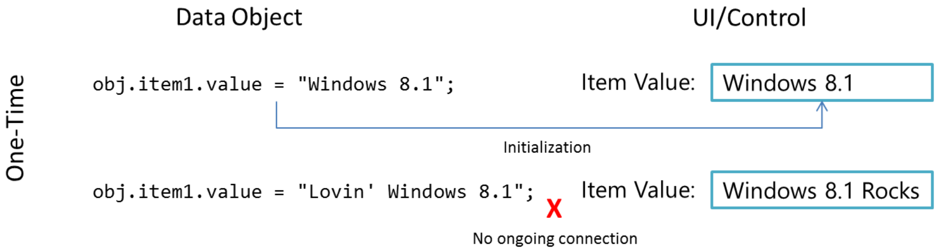

Source
name
id
photoURL
userType
Direct bind: set innerText
equal to value of name
<span>
innerText
style.color
Direct bind: set src
equal to value of photoURL
<img>
src
Converter
Conversion: set style.color equal to the output of
the converter function given for userType

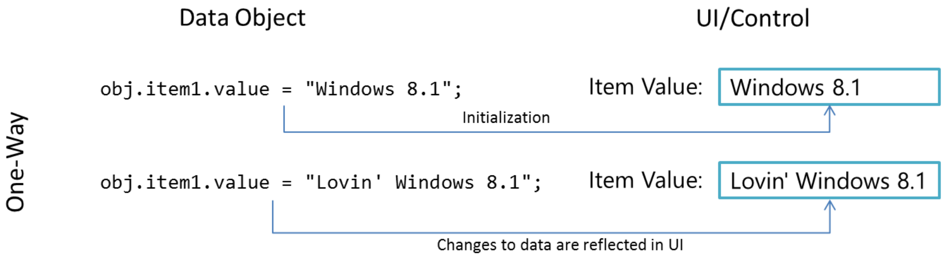
FIGURE 6-1 A common data-binding scenario between a source data object and two target UI elements, involving two direct bindings and one binding with a conversion function.

How these bindings actually behave at run time then depends on the particular *direction* of each binding, which can be one of the following (omitting any converters that might be involved):

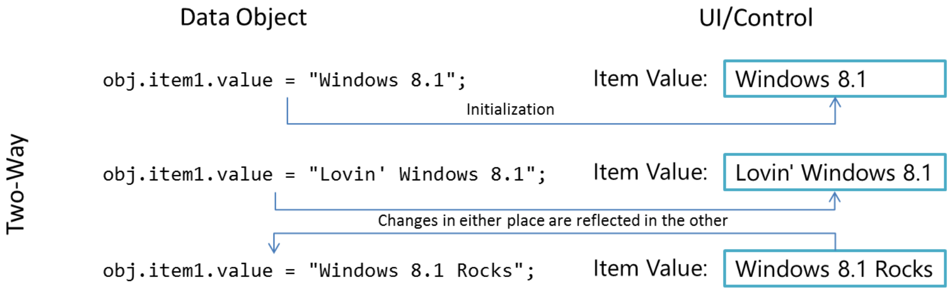
One-time: the value of the source property is copied to the target property at some point, after which there is no further relationship. This is what you automatically do when passing variables to control constructors, for instance, or simply initializing target property values with source properties. What's useful here is to have a declarative means to make such assignments directly in element attributes, as we'll see.



One-way: the target object listens for change events on bound source properties so that it can update itself with new values. This is typically used to update a UI element in response to underlying changes in the data. Changes within the target element (like a UI control), however, are not reflected back to the data itself (but can be sent elsewhere as with form submission, which could in turn update the data through another channel).


Data Object
UI/Control
One-Way
obj.item1.value = "Windows 8.1";
Item Value:
Windows 8.1
Initialization
obj.item1.value = "Lovin' Windows 8.1";
Item Value:
Lovin' Windows 8.1
Changes to data are reflected in UI

Two-way: essentially one-way binding in both directions, as the source object also listens to change events for target object properties. Changes made within a UI element like a text box are thus saved back in the bound source property, just as changes to the source property update the UI element. Obviously, there must be some means to not get stuck in an infinite loop; typically, both objects avoid firing another change event if the new value is the same as the existing one.


Data Object
UI/Control
Two-Way
obj.item1.value = "Windows 8.1";
Item Value:
Windows 8.1
Initialization
obj.item1.value = "Lovin' Windows 8.1";
Item Value:
Lovin' Windows 8.1
Changes in either place are reflected in the other
obj.item1.value = "Windows 8.1 Rocks";
Item Value:
Windows 8.1 Rocks

Data Binding in WinJS

Now that we've seen what data binding is all about, we can see how it can be implemented within a Windows Store app. Again, you can create whatever scheme you want for data binding or use a third-party JavaScript library for the job: it's just about connecting properties of source objects with properties of target objects.

If you're anything like a number of my paternal ancestors, who seemed to wholly despise relying on anyone to do anything they could do themselves (like drilling wells, mining coal, and manufacturing engine parts), you may very well be content with engineering your own data-binding solution. But if you have a more tempered nature like I do (thanks to my mother's side), I'm delighted when someone is thoughtful enough to create a solution for me. Thus my gratitude goes out to the WinJS team who, knowing of the common need for data binding, created the `WinJS.Binding` API. This supports one-time and one-way binding, both declaratively and procedurally, along with converter functions. At present, WinJS does not provide for two-way binding, but such structures aren't difficult to set up.

Within the WinJS structures, properties of multiple target elements can be bound to a single data source property. `WinJS.Binding`, in fact, provides for what are called *templates*, basically collections of target elements whose properties are all bound to the same data source, as we'll see later in this chapter. Though we don't recommend it, it's possible to bind a single target element to multiple sources, but this gets tricky to manage properly. A better approach in such cases is to wrap those separate sources into a single object and bind to its properties instead. This way the wrapper object can manage the process of combining multiple source properties into a single one to use in the binding relationship.

To understand core data binding with WinJS, let's look at how we'd write our own binding code and then see the solution that WinJS offers. We'll use the scenario shown in Figure 6-1, where we have a source object bound to two separate UI elements, with one converter that changes a source property into a color.

One-Time Binding

One-time binding, as mentioned before, is essentially what you do whenever you just assign values to properties of an element. Consider the following HTML, which is found in Test 1 of the BindingTests example in this chapter's companion content:

```
<!-- Markup: the UI elements we'll bind to a data object -->
<section id="loginDisplay1">
    <p>You are logged in as <span id="loginName1"></span></p>
    <img id="photo1"></img>
</section>
```

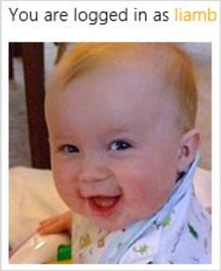
Given the following data source object:

```
var login1 = { name: "liam", id: "12345678",
    photoURL: "http://www.kraigbrockschmidt.com/images/Liam07.png", userType: "kid"};
```

we can bind as follows, where we include a converter function (`userTypeToColor`) in the process:

```
//"Binding" is done one property at a time, with converter functions just called directly
var name = document.getElementById("loginName1");
name.innerText = login1.name;
name.style.color = userTypeToColor(login1.userType);
document.getElementById("photo1").src = login1.photoURL;

function userTypeToColor(type) {
    return type == "kid" ? "Orange" : "Black";
}
```

This gives the following result, in which I shamelessly publish a picture of my kid as a baby:



You are logged in as liamb

With WinJS we can accomplish the same thing by using a declarative syntax and a processing function. In markup, we use the attribute `data-win-bind` to map target properties of the containing element to properties of the source object. The processing function, [`WinJS.Binding.processAll`](), then creates the necessary code to implement those binding relationships. This follows the same idea as using the declarative `data-win-control` and `data-win-options` attributes instruct `WinJS.UI.process[All]` how to instantiate controls, as we saw in Chapter 5.

The value of `data-win-bind` is a string of property pairs. Each pair's syntax is `<target property> : <source property> [<initializer>]` where the `<initializer>` is an optional function that determines how the binding relationship is set up.

Each property identifier can use dot notation as needed (see the sidebar coming up for additional syntax), and uses JavaScript property names. Property pairs are separated by a semicolon, as shown in the HTML for Test 2 in the example:

```
<section id="loginDisplay2">
    <p>You are logged in as
        <span id="loginName2"
            data-win-bind="innerText: name; style.color: userType Tests.typeColorInitializer">
        </span>
    </p>
    <img id="photo2" data-win-bind="src: photoURL"/>
</section>
```

Tip The syntax of `data-win-bind` is different than `data-win-options`! Whereas the options string is an object within braces `{ }` with options separated by a comma, a binding string has no braces and items are separated by *semicolons*. If you find exceptions being thrown from `Binding.processAll`, check that you have the right syntax.

Here we're saying that the `innerText` property of the *loginName2* target is bound to the source's `name` property and that the target's `style.color` property is bound to the source's `userType` property according to whatever relationship the `Tests.typeColorInitializer` establishes. As we'll see in a moment, that initializer is built directly around the `userTypeToColor` converter.

As you can see, the `data-win-bind` notation above does not specify the source object. That's the purpose of `Binding.processAll`. So, assuming we have a data source as before:

```
var login2 = { name: "liamb", id: "12345678",
    photoURL: "http://www.kraigbrockschmidt.com/images/Liam07.png", userType: "kid"};
```

we call processAll with the target element and the source object as follows:

```
//processAll scans the element's tree for data-win-bind, using given object as data context
WinJS.Binding.processAll(document.getElementById("loginDisplay2"), login2);
```

The data context The second argument to Binding.processAll is the data context (or source) with which to perform the binding. If you omit this, it defaults to the global JavaScript context. If, for example, you have a namespace called Data that contains a property bindSource, you can specify Data.bindSource in data-win-bind and omit a data context in processAll, or you can specify just bindSource in data-win-bind and pass Data as the second argument to processAll.

Other arguments Binding.processAll performs a deep traversal of all elements contained within the given root, so you need only call it once on that root for all data binding in that part of the DOM. If you want to process only that element's children and not the root element itself, pass true as the third parameter (called skipRoot). A fourth parameter called bindingCache also exists as an optimization for holding the results of parsing data-win-bind expressions. Both skipRoot and bindingCache are useful when working with binding templates that we'll talk about toward the end of this chapter.

The result of all this, in Test 2, is identical to what we did manually in Test 1. In fact, processAll basically takes the declarative data-win-bind syntax along with the source and target and dynamically executes the same code that we wrote out explicitly in Test 1. The one added bit is that the initializer function must be globally accessible and marked for processing, which is done as follows:

```
//Use a namespace to export function from the current module so WinJS.Binding can find it
WinJS.Namespace.define("Tests", {
    typeColorInitializer: WinJS.Binding.converter(userTypeToColor)
});
```

As with control constructors defined with WinJS.Class.define, WinJS.Binding.converter automatically marks the functions it returns as safe for processing. It does a few more things as well, but we'll return to that subject in a bit.

We could also put the data source object and applicable initializers within the same namespace.[51] For example, in Test 3 we place our login data object and the typeColorInitializer function in a LoginData namespace, and the markup and code now look like this:

```
<section id="loginDisplay3">
    <p>You are logged in as
        <span id="loginName3"
            data-win-bind="innerText: name; style.color: userType LoginData.typeColorInitializer">
        </span>
```

[51] More commonly, initializers and converters would be part of a namespace in which applicable UI elements are defined, because they're more specific to the UI than to a data source.

```
    </p>
    <img id="photo3" data-win-bind="src: photoURL"/>
</section>


WinJS.Binding.processAll(document.getElementById("loginDisplay3"), LoginData.login);

WinJS.Namespace.define("LoginData", {
        login : {
            name: "liamb", id: "12345678",
            photoURL: "http://www.kraigbrockschmidt.com/images/Liam07.png",
            userType: "kid"
        },

        typeColorInitializer: WinJS.Binding.converter(userTypeToColor)
    });
```

In summary, for one-time binding `WinJS.Binding` simply gives you a declarative syntax to do exactly what you'd do in code, with a lot less code. Because it's all just some custom markup and a processing function, there's no magic here, though such useful utilities are magical in their own way! In fact, the code here is really just one-way binding without having the source fire any change events. We'll see how to do that with `WinJS.Binding.as` in a moment after a couple more notes.

First, `Binding.processAll` is actually an async function that returns a promise. Any completed handler given to its `then`/`done` method will be called when the processing is finished, if you have additional code that's depending on that state.

Second, you can call `Binding.processAll` more than once on the same target element, specifying a different source object (data context) each time. This won't replace any existing bindings, mind you—it just adds new ones, meaning that you could end up binding the same target property to more than one source, which could become a big mess. So again, a better approach is to combine those sources into a single object and bind to that, using dot notation to identify nested properties.[52]

Third, the binding that we've created with WinJS here is actually *one-way* binding by default, not one-time, because one-way binding is the most common scenario. To get true one-time binding, use the built-in initializer function, `oneTime`, in your `data-win-bind` string. More on this under "Binding Initializers" later on.

Sidebar: Additional Property Syntax and Binding to WinJS Controls

Property identifiers in `data-win-bind` can be single identifiers like `name` or compound identifiers such as `style.color`. This applies to both source and target properties. This is important when binding to properties of a WinJS control, rather than its root element, as those properties must be referenced through `winControl`. For example, if we had a source object called `myData` and

[52] A final comment that only warrants a footnote is that in Windows 8, apps typically added a line of code `WinJS.Binding.optimizeBindingReferences = true` to avoid some memory leaks. This is done automatically in WinJS 2.0 (for Windows 8.1), so that line is no longer necessary.

wanted to bind some of its properties to those of a WinJS.UI.Rating control, we'd use the following syntax:

```
<div id="rating1" data-win-control="WinJS.UI.Rating"
    data-win-options="{onchange: changeRating}"
    data-win-bind="{winControl.averageRating: myData.average,
       winControl.userRating: myData.rating}">
</div>
```

Notice that data-win-control, data-win-options, and data-win-bind can be used together and UI.process[All] and Binding.processAll will pick up the appropriate attributes. *It's important, however, that* UI.process[All] *has completed its work before calling* Binding.processAll *as the latter depends on the control being instantiated.* Typically, then, you'll call Binding.processAll within the completed handler for UI.process[All]:

```
args.setPromise(WinJS.UI.processAll().then(function () {
    WinJS.Binding.processAll(document.getElementById("boundElement"));
}));
```

Within data-win-bind, array lookup for source properties using [ ] is not supported, though you can do that through an initializer. Array lookup is supported on the target side, as is dot notation. Also, if the target object has a property that you want to refer to using a hyphenated identifier (where dot notation won't work), you can use the following syntax:

```
<span data-win-bind="this[hyphenated-property']: source"></span>
```

That is, the target property string in data-win-bind basically carries through into the binding code that Binding.processAll generates; just as you can use this['hyphenated-property'] in code (this being the data context), you can use it in data-win-bind.

A similar syntax is necessary for binding *attributes* of the target element, such as the aria-* attributes for accessibility. As you probably know, attributes aren't directly accessible through JavaScript properties on an element: they're set through the element.setAttribute method instead. So you'd need to insert a converter into the data binding process to perform that particular step. Fortunately, WinJS.Binding includes initializers that do exactly this, initializer, setAttribute (for one-way binding) and setAttributeOneTime (for one-time binding). These are used as follows:

```
<label data-win-bind="this['aria-label']: title WinJS.Binding.setAttribute"></label>
<label data-win-bind="this['aria-label']: title
    WinJS.Binding.setAttributeOneTime"></label>
```

One-Way Binding

The goal for one-way binding is, again, to update a target property, typically in a UI control, when the bound source property changes. One-way binding means to effectively repeat the one-time binding process whenever the source property changes.

In the previous section's code, if we changed `login.name` after calling `Binding.processAll`, nothing will change in the output UI. So how can we automatically update the output?

Generally speaking, this requires that the data source maintains a list of *bindings*, where each binding describes a source property, a target property, and any associated converter function. The data source also needs to provide methods to manage that list, like *addBinding*, *removeBinding*, and so forth. Thirdly, whenever one of its bindable properties changes it goes through its list of bindings and updates any affected target property accordingly. All together, this makes what we call an *observable* source.

These requirements are quite generic; you can imagine that their implementation would pretty much join the ranks of classic boilerplate code. So, of course, `WinJS.Binding` provides just such an implementation with two functions at its core!

- `Binding.as` wraps any arbitrary object with an observable structure. (It's a no-op for nonobjects, and `Binding.unwrap` removes that structure if there's a need.)
- `Binding.define` creates a constructor for observable objects around a set of properties (described by a kind of empty object that just has property names). Such a constructor allows you to instantiate source objects dynamically, as when processing data retrieved from an online service.

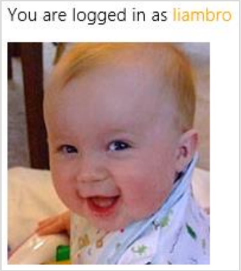
Let's see some code. Going back to the last BindingTests example above (Test 3), any time before or after `Binding.processAll` we can take the `LoginData.login` object and make it observable with just one line of code:

```
var loginObservable = WinJS.Binding.as(LoginData.login)
```

With everything else the same as before, we can now change a bound property within the `loginObservable` object:

```
loginObservable.name = "liambro";
```

This will update the target property (the mechanics of which we'll talk about under "Under the Covers: Binding mixins" a little later):


You are logged in as liambro

Here's how we'd then create and use a reusable class for an observable object (Test 4 in the BindingTests example). Notice the object we pass to `Binding.define` contains property names, but no values (they'll be ignored in any case):

```
WinJS.Namespace.define("LoginData", {
    //...

    //LoginClass is a constructor for observable objects with the specified properties
    LoginClass: WinJS.Binding.define({name: "", id: "", photoURL: "", userType: "" }),
});
```

We can now create instances of `LoginClass`, initializing desired properties with values. In this example, we're using a different picture and leaving `userType` uninitialized:

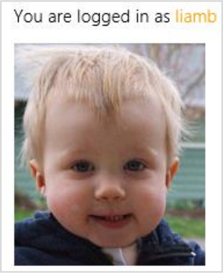
```
var login4 = new LoginData.LoginClass({ name: "liamb",
    photoURL: "http://www.kraigbrockschmidt.com/images/Liam08.png" });
```

Binding to this `login` object, the username initially comes out black.

```
//Do the binding (initial color of name would be black)
WinJS.Binding.processAll(document.getElementById("loginDisplay"), login4);
```

Updating the `userType` property will then cause an update the color of the target property, which happens through the converter automatically. In the example, this line of code is executed after a two-second delay so that you can see it change when you run the app:

```
login4.userType = "kid";
```



You are logged in as liamb

Sidebar: Binding to WinRT Objects

Although it is possible to declaratively bind directly to WinRT source objects, those objects support only one-time binding. The underlying reason for this is that each WinRT object is represented in JavaScript by a thin proxy that's linked to that object instance, but calling `WinJS.Binding.as` on this proxy does *not* make the WinRT observable within the rest of the WinJS binding mechanisms. For this reason, it's necessary to enforce one-time binding when using such sources directly. This can be done by specifying the WinJS `oneTime` initializer within each `data-win-bind` relationship, or specifying `oneTime` as the default initializer for `processAll` (the fifth argument). For example:

```
WinJS.Binding.processAll(element, winRTSourceObject, false, null, WinJS.Binding.oneTime);
```

As in the previous sidebar, if you're binding to element attributes, you'll need to use the setAttributeOneTime initializer instead.

To work around this limitation, you can create a custom proxy in JavaScript that watches changes in the WinRT source and copies those values to properties in the proxy. Because this proxy has its own set of properties, you can then make it observable with WinJS.Binding.as.

Implementing Two-Way Binding

As mentioned earlier, WinJS doesn't presently include any facilities for two-way binding, but it's straightforward to implement on your own:

1. Add listeners to the appropriate UI element events that relate to bound data source properties.
2. Within those handlers, update the data source properties.

The data source should be smart enough to know when the new value of the property is already the same as the target property, in which case it shouldn't try to update the target lest you get caught in a loop. The observable object code that WinJS provides does this type of check for you.

To see an example of this, refer scenario 1 of the Declarative binding sample in the SDK, which listens for the change event on text boxes and updates values in its source accordingly. The input controls are declared as follows (html/1_BasicBinding.html, omitting much of the surrounding HTML structure):

```
<input type="text" id="basicBindingInputText" />
<input type="text" id="basicBindingInputRed" />
<input type="text" id="basicBindingInputGreen" />
<input type="text" id="basicBindingInputBlue" />
```

and the output elements like so:

```
<p>The text you entered was <span data-win-bind="innerHTML: text"></span>.
The value for red is <span data-win-bind="innerHTML: color.red"></span>,
the value for green is <span data-win-bind="innerHTML: color['green']"></span>,
and blue is <span data-win-bind="innerHTML: color.blue"></span>.
</p>
<p data-win-bind="style.background: color BasicBinding.toCssColor">
And here's your color as a background.</p>
```

The source object is defined in js/1_BasicBinding.js as a member of the surrounding page control:

```
this.bindingSource = {
    text: "Initial text",
    color: {
        red: 128,
        green: 128,
        blue: 128
    }
};
```

and is made observable through this line of code:

```
var b = WinJS.Binding.as(this.bindingSource);
```

You can see how its members are referenced in the `data-win-bind` attributes above, so that when we call `processAll` (on the `div` that contains all the output element):

```
WinJS.Binding.processAll(element.querySelector("#basicBindingOutput"), this.bindingSource);
```

we've set up one-way binding between the data source and the output. To complete two-way binding, we set up event handlers for the `change` event of each `input` field that take the value from the control and copy it back to the observable source:

```
this.bindTextBox("#basicBindingInputText", b.text,
    function (value) { b.text = toStaticHTML(value); });

this.bindTextBox("#basicBindingInputRed", b.color.red,
    function (value) { b.color.red = toStaticHTML(value); });

this.bindTextBox("#basicBindingInputGreen", b.color.green,
    function (value) { b.color.green = toStaticHTML(value); });

this.bindTextBox("#basicBindingInputBlue", b.color.blue,
    function (value) { b.color.blue = toStaticHTML(value); });


bindTextBox: function (selector, initialValue, setterCallback) {
    var textBox = this.element.querySelector(selector);
    textBox.addEventListener("change", function (evt) {
        setterCallback(evt.target.value);
    }, false);
    textBox.value = initialValue;
}
```

In the HTML, you might have noticed an initializer called `BasicBinding.toCssColor`. This is defined in js/1_BasicBinding.js as follows:

```
var toCssColor = WinJS.Binding.initializer(
    function toCssColor(source, sourceProperty, dest, destProperty) {
        function setBackColor() {
            dest.style.backgroundColor =
                rgb(source.color.red, source.color.green, source.color.blue);
        }

        return WinJS.Binding.bind(source, {
            color: { red: setBackColor, green: setBackColor, blue: setBackColor, }
        });
    }
);

// A little helper function to convert from separate rgb values to a css color
function rgb(r, g, b) { return "rgb(" + [r, g, b].join(",") + ")"; }

WinJS.Namespace.define("BasicBinding", {
    toCssColor: toCssColor
});
```

Clearly, there's more going on here than just creating a simple converter as we did before, including use of the methods WinJS.Binding.initializer and WinJS.Binding.bind. Now is a good time, then, to peek under the covers to see how initializers work and what functions like bind are doing.

Under the Covers: Binding mixins

Earlier in the "One-Way Binding" section I spelled out three requirements for an observable object: it maintains a list of bindings, provides methods to manage that list, and iterates through that list to update any target properties when source properties change. And we saw that Binding.as and Binding.define let you easily add such structures to a source object or class definition (you can find the source for both in the WinJS base.js file).

Both as and define create an observable object from an existing one by adding the necessary methods that the rest of WinJS.Binding requires. (This is done through mixins; to review details on WinJS.Class.mix and mixins, refer to Appendix B, "WinJS Extras.")

The as method, for its part, assumes that the data source is just a plain object (and not a Date, Array, or nonobject) and creates a proxy object around it using an internal WinJS class called *ObservableProxy*. You wouldn't instantiate this class directly, of course, but it's instructive to look at its definition, where data is the object you give to as:

```
var ObservableProxy = WinJS.Class.mix(function (data) {
    this._initObservable(data);
    Object.defineProperties(this, expandProperties(data));
}, dynamicObservableMixin);
```

The call to _initObservable on the data source turns out to be very important. All it does is ensure that the class has a property called _backingData that's set to the original source. Without this you'd see a bunch of exceptions at run time.

Binding.define does pretty much the same thing except that its purpose is to create a constructor for an observable object class, rather than just wrapping an existing instance. It's how you define your own variant of the internal *ObservableProxy* class we see above.

You can also create an observable source manually, as shown in Scenario 3 of the Programmatic binding sample. Here it defines a RectangleSprite class with Class.define and then makes it observable (and adds a few more observable properties) as follows (js/3_CreatingBindableTypes.js):

```
WinJS.Class.mix(RectangleSprite,
    WinJS.Binding.mixin,
    WinJS.Binding.expandProperties({ position: 0, r: 0, g: 0, b: 0 })
);
```

Note that the RectangleSprite constructor in this case must call this._initObservable() or else none of the binding will work (that is, the app will crash and burn). This is the only case I know of where you need to explicitly call an underscore-named method in WinJS!

However you do it, though, the bottom line is that we're creating a new class that combines the members defined by each argument to mix. In the cases above we have three such arguments: the original class, WinJS.Binding.mixin, and whatever object is produced by WinJS.Binding.expandProperties.

expandProperties creates an object whose properties match those in the given object (the same names, but not the values), where each new property is wrapped in the proper structure for binding.[53] Clearly, this type of operation is useful only when doing a mix, and it's exactly why Binding.define can digest an oddball, no-values object like the one we gave to it in Test 4 of the BindingTest example.

By "proper structure for binding," I mean that each property has a get and set method (along with two properties names enumerable and configurable, both set to true). These methods rely on the class also having methods called getProperty and setProperty:

```
get: function () { return this.getProperty(propertyName); },
set: function (value) { this.setProperty(propertyName, value); },
```

It's unlikely that any arbitrary class will contain such methods already, which is where the Binding.mixin object comes in. It contains a standard implementation of the binding functions, like [get | set]Property, that the rest of WinJS.Binding expects.

The mixin object itself an extension of the more basic Binding.observableMixin, which just contains three methods to manage a list of bindings:

- bind Saves a binding (property name and a *listener* function to invoke on change) into an internal list (an array). This is the binding equivalent of addEventListener.
- unbind Removes a binding created by bind (removing it from the list), or removes all bindings if a specific one isn't given. This is the binding equivalent of removeEventListener.
- notify Goes through the bindings list for a property and asynchronously invokes its listeners with the new property value (at "normal" priority in the scheduler). notify will cancel any update that's already in progress for the same property. This is the binding equivalent of dispatchEvent.

The mixin object then adds five methods to manage properties through bind, unbind, and notify:

- setProperty Updates a property value and notifies listeners if the value changed.
- updateProperty Like setProperty, but returns a promise that completes when all listeners have been notified (the result in the promise is the new property value).

[53] expandProperties also includes properties of the object's prototype.

- getProperty Retrieves a property value as an observable object itself, which makes it possible to bind within nested object structures (obj1.obj2.prop3, etc.).
- addProperty Adds a new property to the object that is automatically enabled for binding (it adds the same properties and methods that that expandProperties does).
- removeProperty Removes a property altogether from the object.

The Binding.dynamicObservableMixin object, I should note, is exactly the same as mixin except for one small difference. If you take an observable class built with mixin and then derive a new class from it (see WinJS.Class.derive), the new class will not automatically be observable. If you build the base class with dynamicObservableMixin, on the other hand, its observability will apply to derived classes.

Programmatic Binding and WinJS.Binding.bind

Taking a step back from the details now, we can see that the eight mixin methods, along with the get and set implementations from expandProperties, fulfill the requirements of an observable data source. With such a source in hand you can do some interesting things programmatically. First, you can call the object's bind method directly to hook up any number of additional actions manually. A couple of scenarios come to mind where this would apply:

- You set up two-way binding between a local data source and the app's UI, and want to also sync changes to the source with a back-end service. A second handler attached through bind would be called whenever the source is modified through any other means.
- You need to create an intermediate source object that combines and consolidates property changes from other sources, simplifying binding to a final target UI element. Calling bind directly in such cases is necessary because Binding.processAll specifically works with UI elements declared in HTML rather than arbitrary JavaScript objects.

The notify method, for its part, is something you can call directly to trigger notifications. This is useful with additional bindings that don't necessarily depend on the values themselves, just the fact that they changed. The major use case here is to implement computed properties—ones that change in response to another property value changing.

The WinJS binding engine also has some intelligent handling of multiple changes to the same source property. After the initial binding, further change notifications are asynchronous and multiple pending changes to the same property are coalesced. So, if in our example we made several changes to the name property in quick succession:

```
login.name = "Kenichiro";
login.name = "Josh";
login.name = "Chris";
```

only one notification for the last value would be sent and that would be the value that shows up in bound targets.

A couple of additional demonstrations of calling these binding methods directly can also be found in scenarios 1 and 2 of the Programmatic binding sample, which doesn't use any declarative syntax at all. Scenario 1, for example, creates an observable source with `Binding.as` and wires it up with `bind` method to a couple of change handlers for two-way binding. (It employs the `WinJS.Utilities.query` and `WinJS.Utilities.QueryCollection.listen` methods, which are described in Appendix B.) Be mindful when looking at lines like this (js/1_BasicBinding.js):

```
this.bindSource.bind("x", this.onXChanged.bind(this));
```

that the first `bind` method on the source object comes from `WinJS.Binding.mixin`, whereas the `bind` method on the `onXChanged` function is the standard JavaScript method to manage the `this` variable!

Scenario 2 demonstrates a coding construct called a *binding descriptor* that works in conjunction with the *static* helper method `WinJS.Binding.bind`. Yes, be aware! `WinJS.Binding.bind` is yet another `bind` method that's separate from a source object's `bind` that's defined in the mixins and separate from a function's `bind` method. Fortunately, you must always call `WinJS.Binding.bind` with its full name, and I'll refer to it this way to avoid confusion.

I call `WinJS.Binding.bind` a helper function, because it's basically a way to make a bunch of `bind` calls for properties of a complex object. Consider the source object that's created in scenario 2 of the Programmatic binding sample (js/2_BindingDescriptors.js):

```
this.objectPosition = WinJS.Binding.as({
    position: { x: 10, y: 10},
    color: { r: 128, g: 128, b: 128 }
}
```

If we wanted to set up binding relationships to each of these properties, we'd have to make a bunch of calls to `this.objectPosition.bind` like this:

```
this.objectPosition.bind(this.objectPosition.position.x, <action>)
```

Such code gets ugly to write in a hurry. A binding descriptor, then, succinctly expresses the property-action mappings in a structure that matches the source object. The generic syntax is: `{ property: { sub-property: function(value) { ... } } }`. Here's a concrete example from scenario 2 (js/2_BindingDescriptors.js):

```
{
    position: {
        x: onPositionChange,
        y: onPositionChange
    },
    color: {
        r: onColorChange,
        g: onColorChange,
        b: onColorChange
    }
}
```

where onPositionChange and onColorChange both refresh the output in response to data changes.[54] Scenario 1 of the Declarative binding sample has another example (js/1_BasicBinding.js):

```
{
    color: {
        red: setBackColor,
        green: setBackColor,
        blue: setBackColor,
    }
}
```

When calling WinJS.Binding.bind, then, just pass the source object as the first argument and the binding descriptor as the second. As shown in the Declarative binding sample:

```
return WinJS.Binding.bind(source, {
    color: {
        red: setBackColor,
        green: setBackColor,
        blue: setBackColor,
    }
});
```

The return value of WinJS.Binding.bind is an object with a cancel method that will clear out all these binding relationships (basically iterating through the structure calling unbind). In the Declarative binding sample, it returns this object from the binding initializer where it appears, which is actually very important. So let's now turn to initializers.

Binding Initializers

When Binding.processAll encounters a data-win-bind attribute on an element, its main job is to take each <target property> : <source property> [<initializer>] string and turn it into a real binding relationship. Assuming that the data source is observable, this basically means calling the source's bind method with the source property name and some handler that will update the target property accordingly.

The purpose of the initializer function in this process is to define exactly *what* happens in that handler—that is, what happens to the target property in response to a source property update. In essence, an initializer provides the body of the handler given to the source's bind. In a simple binding relationship, that code might simply copy the source value to the target or it might involve a converter function. It could also consolidate multiple source properties—you can really do anything you want here. The key thing to remember is that the initializer itself will be called once and only once for each binding relationship, but the code it provides, such as a converter function, will be called every time the source property changes.

Now if you don't specify a custom initializer within data-win-bind, WinJS will always use a default,

[54] The actual code uses this.onPositionChange.bind(this) which I've simplified to avoid confusion between all the binds!

namely WinJS.Binding.defaultBind.[55] It simply sets up a binding relationship that copies the source property's value straight over to the target property, as you'd expect. In short, the following two binding declarations have identical behavior:

```
data-win-bind="innerText: name"
data-win-bind="innerText: name defaultBind"
```

WinJS.Binding provides a number of other built-in initializers that can come in handy, most of which we've already encountered:

- oneTime Performs a one-time copy of the source property to the target property without setting up any other binding relationship. This is necessary when the source is a WinRT object, as noted earlier in "Sidebar: Binding to WinRT Objects."
- setAttribute and setAttributeOneTime Similar to defaultBind and oneTime but injects a call to the target element's setAttribute method instead of just copying the source value to a target property. See "Sidebar: Additional Property Syntax and Binding to WinJS Controls" earlier for more details.
- addClassOneTime Like setAttributeOneTime except that it interprets the source property as a class name and thus calls the target element's classList.add method to apply that class. This is useful when working with templates, which are described later in this chapter, because you can assign static classes to elements with the class attribute and then apply additional data-bound classes with this initializer.

Beyond these, we enter into the realm of custom initializers. The most common and simplest case is when you need to inject a converter into the binding relationship. All this takes on your part is to pass your conversion function to WinJS.Binding.converter, which returns the appropriate initializer (that's also marked for declarative processing). We did this in Tests 2, 3 and 4 of the BindingTests example. For Tests 3 and 4, for example, the initializer LoginData.typeColorInitializer is created as follows:

```
WinJS.Namespace.define("LoginData", {
       //...
       typeColorInitializer: WinJS.Binding.converter(userTypeToColor)
   });
```

which we use in the HTML like so:

```
<span id="loginName3"
   data-win-bind="innerText: name; style.color: userType LoginData.typeColorInitializer">
</span>
```

Doing anything more requires that you implement the initializer function directly. This is necessary,

[55] As shown earlier in "Sidebar: Binding to WinRT Objects," you can override the default initializer by providing your own as the fifth argument to processAll (after skipRoot and bindingCache). Though this argument doesn't appear in the MSDN documentation, it is described within the WinJS source file and is safe to use.

for instance, if you need to apply a converter to a WinRT data source, where normally you're required to use the oneTime initializer already. A custom initializer can then do both steps at once.

Scenario 1 of the Declarative binding sample gives us an example of an initializer function (js/1_BasicBinding.js):

```
var toCssColor = WinJS.Binding.initializer(
    function toCssColor(source, sourceProperty, dest, destProperty) {
        function setBackColor() {
            dest.style.backgroundColor =
                rgb(source.color.red, source.color.green, source.color.blue);
        }

        return WinJS.Binding.bind(source, {
            color: { red: setBackColor, green: setBackColor, blue: setBackColor, }
        });
    }
);

// A little helper function to convert from separate rgb values to a css color
function rgb(r, g, b) { return "rgb(" + [r, g, b].join(",") + ")"; }

WinJS.Namespace.define("BasicBinding", {
    toCssColor: toCssColor
});
```

WinJS.Binding.initializer is just an alias for WinJS.Utilities.markSupportedForProcessing, as discussed in Chapter 4, "Web Content and Services." It makes sure you can reference this initializer from processAll and doesn't add anything else where binding is concerned.

The arguments passed to your initializer clearly tell you which properties of source and target (dest) are involved in this particular relationship. The source and dest arguments are the same as the dataContext and rootElement arguments given to processAll, respectively. The sourceProperty and destProperty arguments are both arrays that contain the "paths" to their respective properties, where each part of the identifier is an element in the array. That is, if you have an identifier like style.color in data-win-bind, the path array will be [style, color].

Tip If you find it tricky to set a breakpoint inside an initializer, just insert the debugger statement at the top and you'll always stop at that point.

With all this information in hand, you can then set up whatever binding relationships you want by calling the source's bind method with whatever properties and handlers you require. To duplicate the behavior of oneTime, as with WinRT sources, you wouldn't call bind but instead just assign the value of the source property to the destination property.

Although sourceProperty is what's present in the data-win-bind attribute, you're free to wire up to any other property you want to include in the relationship. The code above, for example, will be called with sourceProperty equal to [color], but it doesn't actually bind to that directly. It instead uses a binding descriptor with WinJS.Binding.bind to hook up the setBackColor handler to three separate color subproperties. Although setBackColor is used for all three, you could just as easily have separate handlers for each one if, for example, each required a unique conversion.

Ultimately, what's important is that the handler given to the source.bind method performs an appropriate update on the target object. In the code above, you can see that setBackColor sets dest.style.backgroundColor.

Hmmm. Do you see a problem there? Try changing the last data-win-bind attribute in html/1_BasicBinding.html to set the text color instead:

```
data-win-bind="style.color : color BasicBinding.toCssColor"
```

Oops! It still changes the background color! This is because the initializer isn't honoring destProperty, and that would be a difficult bug to track down when it didn't work as expected. Indeed, because the initializer pays no attention to destProperty you can use a nonexistent identifier and it will still change the background color:

```
data-win-bind="some.ridiculous.identifier : color BasicBinding.toCssColor"
```

Technically speaking, then, we could rewrite the code as follows:

```
dest.[destProperty[0]].[destProperty[1]] =
    rgb(source.color.red, source.color.green, source.color.blue);
```

Even this code makes the assumption that the target path has only two components—to be really proper about it, you need to iterate through the array and traverse each step of the path. Fortunately, WinJS.Utilities.getMember is a helper for just that purpose, though to do an assignment you need to pop the last property from the array so that you can use [ ] to reference it:

```
var lastProp = destProperty.pop();
var target = destProperty.length ?
    WinJS.Utilities.getMember(destProperty.join("."), dest) : dest;
destProperty.push(lastProp);

function setBackColor() {
    target[lastProp] = rgb(source.color.red, source.color.green, source.color.blue);
}
```

This code can be found in the modified Declarative binding sample in this chapter's companion content. Note that I'm building that reference outside of the setBackColor handler because there's no need to rebuild it every time a source property changes. Also, calling push to restore destProperty is important in scenario 3 when this initializer is used with a template that's rendered multiple times on the same page. In that case the same destProperty array is used with each call to the initializer, meaning that we don't want to make any permanent changes.

Remember also that sourceProperty can also contain multiple parts, so you may need to evaluate that path as well. The sample gets away with ignoring sourceProperty because it knows it's only using the toCssColor initializer with source.color to begin with. Still, if you're going to write an initializer, best to make it as robust as you can! Again, you can use getMember to assemble the reference you need, and because you're just retrieving the value, you can do it in one line:

```
var value = WinJS.Utilities.getMember(sourceProperty.join("."), source);
```

Binding Templates

Now that we understand how controls work from Chapter 5 and how data binding works to populate those controls from a data source, the next question to ask is how we might define reusable pieces of HTML that also integrate declarative data binding.

There are two ways to do this. First, you can certainly use data-win-bind syntax within an HtmlControl or a custom control (remembering to bind control properties through winControl rather than the root element). Once the control is instantiated with WinJS.UI.process[All], you can then call WinJS.Binding.processAll to bind each element to an appropriate source.

The second way is through the WinJS.Binding.Template control. It provides a way to define an HTML snippet, either inline or in a separate HTML file, and then render that snippet, however many times you want, with whatever data source you need for each rendering. The template makes the call to Binding.processAll automatically and provides some other options where binding is concerned.

To define a template control, first create a div with data-win-control = "WinJS.Binding.Template" and an optional data-win-options string. The contents of the template—which are copied into the DOM when you render the template—can then be defined either inline or in a separate file. In that markup you're free to use whatever controls you want along with data-win-bind attributes (as well as data-win-res attributes for localization).

Tip Blend for Visual Studio 2013 has some helpful features to easily create templates and data bindings from a data source for WinJS collection controls. We'll see this in Chapter 7.

Template designs For an extensive collection of premade templates (designed primarily for the ListView control, but a great reference nonetheless), see Item templates for grid layouts and Item templates for list layouts.

Here's an example from scenario 2 of the modified Declarative binding sample in this chapter's companion content (html/2_TemplateControl.html, where the toCssColor initializer is corrected as we did earlier for scenario 1):

```
<div id="templateControlTemplate" data-win-control="WinJS.Binding.Template">
    <!-- Bind both the background and the aria-label HTML attribute for the div -->
    <div class="templateControlRenderedItem"
        data-win-bind="style.background: color TemplateControl.toCssColor;
        this['aria-label']: text WinJS.Binding.setAttribute" role="article">
        <ol>
            <li data-win-bind="textContent: text"></li>
            <li><span class="templateControlTemplateLabel">r:
                </span><span data-win-bind="textContent: color.r"></span></li>
            <li><span class="templateControlTemplateLabel">g:
                </span><span data-win-bind="textContent: color.g"></span></li>
            <li><span class="templateControlTemplateLabel">b:
                </span><span data-win-bind="textContent: color.b"></span></li>
        </ol>
    </div>
</div>
```

Scenario 3 of the example, which is an addition I've made to the original sample, has the same template contents stored in html/3_TemplateContents.html (`<!DOCTYPE html>`, `<html>`, and `<body>` tags are optional and have no effect). We then refer to those contents with the control's `href` option in `data-win-options` (html/3_TemplateControlHref.html):

```
<div id="templateControlTemplate" data-win-control="WinJS.Binding.Template"
    data-win-options="{ href : '/html/3_TemplateContents.html' }">
</div>
```

Note Using the `href` option disables certain optimizations with templates that dramatically improve performance. You should only use `href`, then, when absolutely necessary.

What's unique about this control is that it automatically hides itself (setting `style.display = "none"`) inside its constructor so that its contents never appear in your layout. You can confirm this when you run scenarios 2 or 3 of the example—if you expand everything in Visual Studio's DOM Explorer, you won't be able to find the template control anywhere. The control instance, however, is still in memory: a quick `getElementById` or `querySelector` will get you to the hidden root element, and that element's `winControl` property is where you'll find the WinJS template object.

That object has a number of methods and properties that we'll return to in a bit. The most important of these is the asynchronous `render` method. Given whatever data source you want to bind to, `render` returns a promise that's fulfilled with a copy of the template's contents in a new element, where `Binding.processAll` has already been called with the data source. You can then attach that element to the DOM wherever you'd like.

Scenario 2 of the example, for instance, renders the template three times, binding each to a different source object that contains text and a color. First, in html/2_TemplateControl.html, the target div for the rendered templates is initially empty:

```
<div id="templateControlRenderTarget"></div>
```

In js/2_TemplateControl.js, we define the observable data sources as follows (showing `WinJS.Binding.define`):

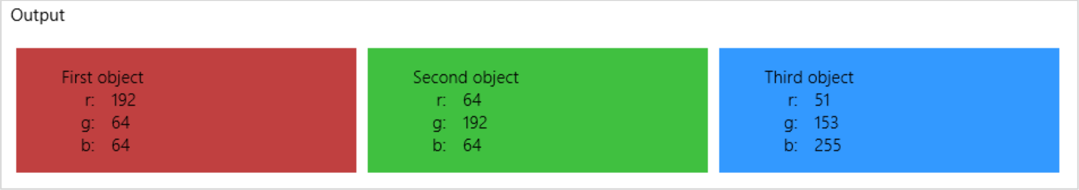
```
var DataSource = WinJS.Binding.define({
    text: "", color: { r: 0, g: 0, b: 0 }
});
```

```
// In the page control's init method
this.sourceObjects = [
    new DataSource({ text: "First object", color: { r: 192, g: 64, b: 64 } }),
    new DataSource({ text: "Second object", color: { r: 64, g: 192, b: 64 } }),
    new DataSource({ text: "Third object", color: { r: 51, g: 153, b: 255 } })
];
```

In the page's `ready` method, which is called after the page is rendered (meaning `UI.processAll` has instantiated the template), we do a bunch of work to wire up two-way binding and then render the template for each data source:

```
var templateControl = element.querySelector("#templateControlTemplate").winControl;
var renderHere = element.querySelector("#templateControlRenderTarget");

this.sourceObjects.forEach(function (o) {
    templateControl.render(o).then(function (e) {
        renderHere.appendChild(e);
    });
});
```

Again, the template control is hidden but still present in the DOM, so we can get to it with the `querySelector` call. We then get the target `div` and iterate through the data sources, rendering the template with each source in turn, and attaching the resulting element tree (in `e`) to the target. The result of all this is shown below:



```
Output

First object        Second object        Third object
  r:  192             r:  64               r:  51
  g:  64              g:  192              g:  153
  b:  64              b:  64               b:  255
```

Note that `render` will, by default, create an extra container `div` for the template contents. That is, the output above will have this structure:

```
<div id="templateControlRenderTarget">
    <div class="win-template win-disposable">
        <!-- Template contents, starting with <div class="templateControlRenderedItem"> -->
    </div>
    <div class="win-template win-disposable">
        <!-- ... -->
    </div>
    <div class="win-template win-disposable">
        <!-- ... -->
    </div>
</div>
```

You can eliminate that extra `div` by setting the template's `extractChild` option to `true`:

```
data-win-options="{ extractChild : true }"
```

Also keep in mind that `render` calls `Binding.processAll` as part of its process, using, of course, the source you gave to `render`. Because there's no point to process the template's root element—the one with the `data-win-control= "WinJS.Binding.Template"`—the call to `processAll` has the `skipRoot` argument set to `true`. Internally WinJS also uses the `bindingCache` argument to optimize repeated renderings. That's really what those arguments are there for.

As you can see, binding templates are quite straightforward to use. They will come in very handy when we talk about collection controls in the next chapter, because you can just point those controls to a template to use with each item in the collection. And with the `WinJS.UI.Repeater` control, we'll even see how you can use nested templates.

Sidebar: The Static WinJS.Binding.Template.render method

Within the `WinJS.Binding.Template` API is an odd duck of a method that's documented as `render.value`. This is actually just a static method whose actual name in code is `WinJS.Binding.Template.render` and whose implementation is simply the following:

```
value: function (href, dataContext, container) {
    return new WinJS.Binding.Template(null, { href: href }).render(dataContext, container);
}
```

This provides a programmatic shortcut for declaring a template with an `href` option, calling `WinJS.UI.processAll` to instantiate it, and then obtaining the control object and calling its `render` method. In short, `WinJS.Binding.Template.render`, which you'll always call with the full identifier, is a one-line wonder for quickly rendering a file-based template with some data source into a container element. Scenario 4 of the modified Declarative Binding sample in this chapter's companion content has a quick demonstration. The caveat is that the template will be loaded from disk each time you make this call, so it's really best for one-shot renderings. If you'll be using a template in multiple places, use an inline declaration so that the template object is present in the DOM for the lifetime of the page.

Template Options, Properties, and Compilation

Now that we've seen the basics of the `Template` control, let's see what other features it supports. First are the options for the constructor that you can include in `data-win-options`:[56]

- `href` Specifies the path of an in-package HTML file that contains the template definition, as we've seen. Using this implicitly sets `disableOptimizedProcessing` to `true`.
- `bindingInitializer` Specifies a binding initializer to apply as the default to all `data-win-bind` relationships. This does not override any explicit initializer given in `data-win-bind`.

[56] Two other options, `enableRecycling` and `processTimeout`, are deprecated and for internal use, respectively.

- debugBreakOnRender Executes a debugger statement whenever the template is first rendered and especially gives you a hook into a compiled template (see below). This is essential when a template's render call is being made implicitly from within another control like the ListView.
- disableOptimizedProcessing Turns off template compiling; see below.
- extractChild If set to true, suppresses creation of a containing div for the template contents, as discussed in the previous section. The default is false.

All of these except for href can be accessed at run time through properties with the same names on the control: bindingInitializer, debugBreakOnRender, disableOptimizedProcessing, and extractChild.[57] Note that changing any of these properties at run time will reset the template and cause it to be recompiled the next time it's used.

Related to the behavior of the extractChild option is a second argument to the render method. If you already have a container into which you want the template contents rendered, you can provide it through this argument. For example, with extractChild set to true, the following code in scenario 3 of the modified Declarative binding sample (in the companion content) produces the same result as the default behavior (js/3_TemplateControlHRef.js):

```
this.sourceObjects.forEach(function (o) {
    var container = document.createElement("div");
    WinJS.Utilities.addClass(container, "win-template win-disposable");
    renderHere.appendChild(container);
    templateControl.render(o, container);
});
```

Such code gives you complete control over the kind of container element that's created for the template, if you have need of it.

As for disableOptimizedProcessing and debugBreakOnRender, these relate to a significant change for WinJS 2.0 (in Windows 8.1): template compilation. In WinJS 1.0, template rendering was always done in an interpreted manner, meaning that each time you rendered a template, WinJS would parse your template's markup and create each element in turn. You could improve this process by creating a rendering function directly, but you then lost the advantages of declarative markup.

To improve performance of declarative templates, especially with the ListView control, WinJS 2.0 compiles each template upon first rendering. This step dynamically creates a rendering function that makes subsequent use of the template much faster. In fact, if you have a Windows 8 (WinJS 1.0) app that uses the ListView control and you simply migrate the project to Windows 8.1 (WinJS 2.0), you'll probably find it performing much better than before with no other changes.

Of course, with such extensive restructuring of the template engine there's always the possibility that it breaks some existing code somewhere—perhaps yours! If you find that's the case, or if for some

[57] One other property called isDeclarativeControlContainer is always set to true, and as with all other WinJS controls, the template has an element property that contains its root element in the DOM.

reason want your app to just run slower, set disableOptimizedProcessing to true to bypass compilation and use the WinJS 1.0 rendering behavior. And again, note that using href to refer to template content will implicitly disable compilation.

With compiled templates, the WinJS engineers found it very helpful to have a hook into the dynamically-generated code, so they included the debugBreakOnRender option. If set to true (try it in scenario 2) and you run an app in the debugger, you'll hit a debugger statement in code like this:

```
// generated template rendering function
return function render(data, container) {
    if (++sv23_debugCounter === 1) { debugger; }
    if (typeof data === "object" && typeof data.then === "function") {
        // async data + a container falls back to interpreted path
```

If you step through this code you can see the nature of the compiled template. The HTML contents are added through a single insertAdjacentHtml with a string, for instance, and the steps that would normally happen within the async Binding.processAll, including calls to initializers, are done inline, so there's no extra parsing of data-win-bind attributes. The same is true for WinJS controls in the template that would normally go through the async UI.processAll method: they are instead instantiated directly with new and a pre-parsed options object. Avoiding async calls and eliminating extra parsing clearly make compiled templates run much faster.

On the other hand, if you're using href or have disableOptimizedProcessing turned on (as in scenario 3 of the example), you'll instead hit a debugger statement inside this internal WinJS function:

```
function interpretedRender(template, dataContext, container) {
    if (++template._counter === 1
    && (template.debugBreakOnRender || WinJS.Binding.Template._debugBreakOnRender)) {
        debugger;
    }
```

Stepping through this code you'll see just how much more work is going on, such as a call to WinJS.UI.Fragments.renderCopy to asynchronously load up the template contents. Inside the completed handler for this (a variable called renderComplete), you'll also see calls to the async UI.processAll and Binding.processAll methods.

If you continue looking through both renderers (compiled or interpreted), you'll see they return an object with two properties called element and renderComplete. element is a promise that's fulfilled when the element tree has been built up in memory, and renderComplete is a function that's called once the promise is fulfilled (where data binding can then happen). This isn't at all important for using the Template object but becomes important with collection controls like the ListView when you want to do performance optimization through your own rendering functions. We'll see all that in Chapter 7.

Collection Data Types

No sooner had I sat down to start this section than my wife called me about the shopping list she'd left lying on our kitchen counter. It was a timely reminder that much of our reality where data is concerned is oriented not around single items or object instances, as we've dealt with thus far in this chapter, but around *collections* of such things. And as I said in the introduction to this chapter, dealing with collections and presenting them in an app's UI is where data binding becomes exceptional useful. Otherwise you'd become very proficient (though I imagine you are already!) at doing copy/paste with lots of redundant binding code.

I assume that you're already well familiar with the core collection type in JavaScript—the `Array`—whose various capabilities are all supported in Windows Store apps, as are the typed JavaScript collections like `Uint16Array`.

By itself, `Array` and typed arrays are not observable for data-binding purposes. Fortunately, WinJS provides the `WinJS.Binding.List` collection type that is easily built on an array. The `List` also supports creating grouped, sorted, and filtered *projections* of itself.

What also enters into the picture are specific language-neutral collection types that are used with WinRT APIs. In Chapter 4, for example, we encountered *vectors* in a number of places, such as network information, the credential locker, the content precacher, and background transfers. The other types are *iterators*, *key-value pairs*, *maps*, and *property sets*. As we're talking about collections in this section, we'll take the opportunity to familiarize ourselves with each of these constructs.

What's most important to understand about the WinRT collections—especially where data binding is concerned—is how they project into the JavaScript environment, because that determines whether you can easily bind to them. Earlier we learned that it's not possible to do one-way or two-way binding with WinRT objects, and these collections count as such. Fortunately, the JavaScript projection layer conveniently turns some of them into arrays, meaning that we can create a `List` and go from there.

Windows.Foundation.Collection Types

All of the WinRT collection types are found in the `Windows.Foundation.Collections` namespace. What you'll notice immediately upon perusing the namespace is that it actually contains only one concrete class, `PropertySet`, and otherwise it is chock full of "interfaces" with curious names like `IIterable<T>`, `IMapView<K, V>`, and `IVectorView<T>`.

If you've at least fiddled with .NET languages like C# in your development career, you probably already know what the `I` and `<T>` business is all about because you get to type them out all the time. For the rest of you, it probably makes you appreciate the simplicity of JavaScript! In any case, let me explain a little more.

An *interface*, first of all, is an abstract definition of a group of related methods and properties. Interfaces in and of themselves have no implementation, so they're sometimes referred to as *abstract* or *virtual types*. They simply describe *a way to interact* with some object that "implements" the interface, regardless of what else the object might do. Many objects, in fact, implement multiple interfaces. So anytime you see an `I` in front of some identifier, it's just a shorthand for a group of members with some well-defined behavior.[58]

With collections in particular, it's convenient to talk about the collection's behavior independently from the particular object type that the collection contains. That is, whether you have an array of strings, integers, floats, or other complex objects, you still use the same methods to manipulate the array and the same properties like `length` are always there. The `<T>` shorthand is thus a way of saying "of type T" or "of whatever type the collection contains." It certainly saves the documentation writers from having to copy and paste the same text into dozens of separate pages and do a search-and-replace on the data type! Of course, you'll never encounter an abstract collection interface directly—in every instance you'll see a concrete type in place of `<T>`. For instance, `IVector<String>` denotes a *vector* of *strings*: you navigate the collection through the methods of `IVector`, and the items in the collection are of type `String`.

In a few cases you'll see `<K>` and `<K, V>` where `K` means *key* and `V` means *value*, both of which can also be of some arbitrary type. Again, it's just a shorthand that enables us to talk about the behavior of the collection without getting caught up in the details of whatever object type it contains.

So that's the syntactical sugar—let's now look at the different WinRT collections.

Iterators

An *iterator* is the most basic form of a collection and one that maps directly to a simple array in JavaScript. The two iterator interfaces in WinRT are `IIterable<T>` and `IIterator<T>`. You'll never work with these directly in JavaScript, however. For one thing, a JavaScript array (containing objects of any type) can be used with any WinRT API that wants an `IIterable`. Second, when a WinRT API produces an iterator as a result, you treat it as an array. (There are, in fact, no WinRT APIs available from JavaScript that directly produce a plain iterator; they produce vectors and maps that derive from `IIterable` and thus have those methods.)

In short, iterators are transparent from the JavaScript point of view, so when you see `IIterable` in the documentation for an API, just think "array." For example, the first argument to `BackgroundDownloader.requestUnconstrainedDownloadsAsync` that we saw in Chapter 4 is documented as an `IIterable<DownloadOperation>`. In JavaScript this just means an array of `DownloadOperation` objects; the JavaScript projection layer will make sure that the array is translated into an `IIterable` suitable for WinRT.

Similarly, a number of APIs in the `Windows.Storage.FileIO` and `PathIO` classes, such as `appendLinesAsync` and `writeLinesAsync` all take arguments of type `IIterable<String>`, which to us just means an array of strings.

[58] There are actually all kinds of interfaces floating around the WinRT API. When working in JavaScript, however, you rarely bump into them because the language doesn't have such a formal construct. For example, a page control technically implements the `WinJS.UI.Pages.IPageControlMembers` interface, but you never see a direct reference to that name. Instead, you simply include its methods among the instance members of your page class. In contrast, when creating classes in other languages like C#, you explicitly list an interface as an abstract base class to inherit its methods.

Occasionally you'll run into an API like ImageProperties.savePropertiesAsync (in Windows.-Storage.FileProperties) that takes an argument of type IIterable<IKeyValuePair>, which forces us to pause and ask just what we're supposed to do with a collection interface of another interface! (IKeyValuePair<K, V> is also in Windows.Foundation.Collections.) Fortunately, the JavaScript projection layer translates IIterable<IKeyValuePair> into the concrete Windows.Foundation.-Collections.PropertySet class, which can be easily addressed as an array with named members, as we'll see shortly.

Vectors and Vector Views

By itself, an iterator has methods to go through the collection in only one direction (IIterable.first returns an IIterator whose only methods are getMany and moveNext). A *vector* is a more capable variant (IVector derives from IIterable) that adds random-access methods (getAt and indexOf) and methods to modify the collection (append, clear, insertAt, removeAt, removeAtEnd, replaceAll, and setAt). A vector can also report its size.[59]

Because a vector is a type of iterator, you can also just treat it as a JavaScript array—a vector that you obtain from some WinRT API, that is, will have methods like forEach and splice as you'd expect. The one caveat here is that if you inspect a vector in Visual Studio's debugger (as when you hover over a variable), you'll see only its IVector members. But trust me, the others are there!

Vectors basically exist to help marshal changes to the collection between the JavaScript environment and WinRT. This is why IIterable is used as arguments to WinRT APIs (the input array is effectively read-only), whereas IVector is used as an output type, either for the return value of a synchronous API or for the results of an asynchronous API. For example, the readLinesAsync methods of the FileIO and PathIO methods in Windows.Storage provide results as a vector.

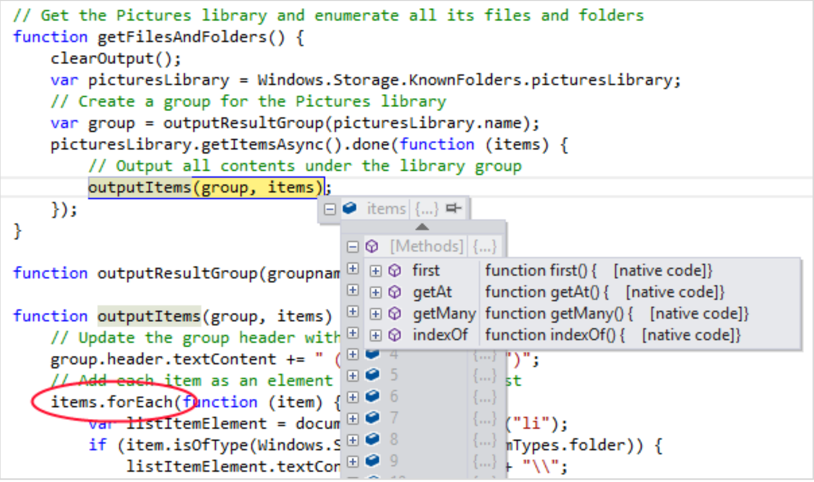
Within WinRT you'll most often encounter vectors with a read-write collection property on some object. For example, the Windows.UI.Popups.MessageDialog object, which we'll meet in Chapter 9, "Commanding UI," has a commands property of type IVector<IUICommand>, which translates into JavaScript simply as an array of UICommand objects. Because the collection can be modified by either the app or the WinRT API, a vector is used to marshal those changes across the boundary.

In quite a number of cases—properties and methods alike—the collection involved is read-only but still needs to support random-access characteristics. For this we have the *vector view* (IVectorView also derives from IIterable), which doesn't have the vector's modification methods. To give a few examples, a vector view of ConnectionProfile objects comes back from NetworkInformation.-getConnectionProfiles (which we saw in Chapter 4). Similarly, StorageFolder.getFilesAsync provides a vector view of StorageFile objects. The user's preferred languages in the Windows.-System.UserProfile.GlobalizationPreferences object is a vector view of strings. And you can

[59] The IObservableVector<T> interface in Windows.Foundations.Collections exists for other languages and is not ever seen in JavaScript.

always get a read-only view for any given read-write vector through the latter's `getView` method.

Now because a vector is just a more capable iterator, guess what? You can just treat a vector like an array. For example, the [Folder enumeration sample]() uses `StorageFolder.getFilesAsync` and `getItemsAsync` to list the contents of in your various media libraries, using `forEach` to iterate the array that those APIs produce. Using that example, here's what I meant earlier when I mentioned that Visual Studio shows only the `IVector` members—the items from `getItemsAsync` doesn't show array methods, but we can clearly call `forEach` (circled):



```
// Get the Pictures library and enumerate all its files and folders
function getFilesAndFolders() {
    clearOutput();
    var picturesLibrary = Windows.Storage.KnownFolders.picturesLibrary;
    // Create a group for the Pictures library
    var group = outputResultGroup(picturesLibrary.name);
    picturesLibrary.getItemsAsync().done(function (items) {
        // Output all contents under the library group
        outputItems(group, items);
    });
}

function outputResultGroupnam

function outputItems(group, items)
    // Update the group header with
    group.header.textContent += " (                ")";
    // Add each item as an element                st
    items.forEach(function (item) {
        var listItemElement = docum              ("li");
        if (item.isOfType(Windows.S              nTypes.folder)) {
            listItemElement.textCor              + "\\";
```

items {...}
[Methods] {...}
first function first() { [native code]}
getAt function getAt() { [native code]}
getMany function getMany() { [native code]}
indexOf function indexOf() { [native code]}
4 {...}
5 {...}
6 {...}
7 {...}
8 {...}
9 {...}

What all this means for data binding, to return to the context of this chapter, is that it's no problem to create a `WinJS.Binding.List` from the WinRT methods and properties that produce vectors and vector views, because those collections are projected as arrays.

Tip If you're enumerating folder contents to create a gallery experience—that is, displaying thumbnails of files in a control like the `WinJS.UI.ListView`—*always* use [`Windows.Storage.StorageFile.getThumbnailAsync`]() or [`StorageFile.getScaledImageAsThumbnailAsync`]() to retrieve a small image for your data source, which can be passed to `URL.createObjectURL` and the result assigned to an `img.src` attribute. This performs far better in both speed and memory efficiency, as the API draws from the thumbnails cache instead of loading the entire file contents (as happens if you call `URL.createObjectURL` on the full `StorageFile`). See the [File and folder thumbnail sample]() for demonstrations.

Maps and Property Sets

A *map* (IMap<K, V>) and its read-only *map view* companion (IMapView<K, V>) are additional derivations of the basic iterator. A map is composed of key-value pairs, where the keys and values can both be arbitrary types.[60] A closely related construct is the *property set* (IPropertySet), which relates to the map. The difference is that maps and map views are used (like vectors) to return collections from WinRT APIs and to handle certain properties of WinRT objects. Property sets, for their part, as used (like basic iterators) as input arguments to WinRT APIs.[61]

It's important to note right up front that maps and property sets are *not* projected as arrays. They *do* support value lookup through the [ ] operator, but otherwise do not have array methods. A map, instead, has methods (from IMap) called lookup (same as [ ]), insert, remove, clear, and hasKey, along with a size property and a getView method like the vector. A map view shares hasKey, lookup, and size, and adds a split method. You can also pass a map or map view to Object.keys to retrieve an array of keys.

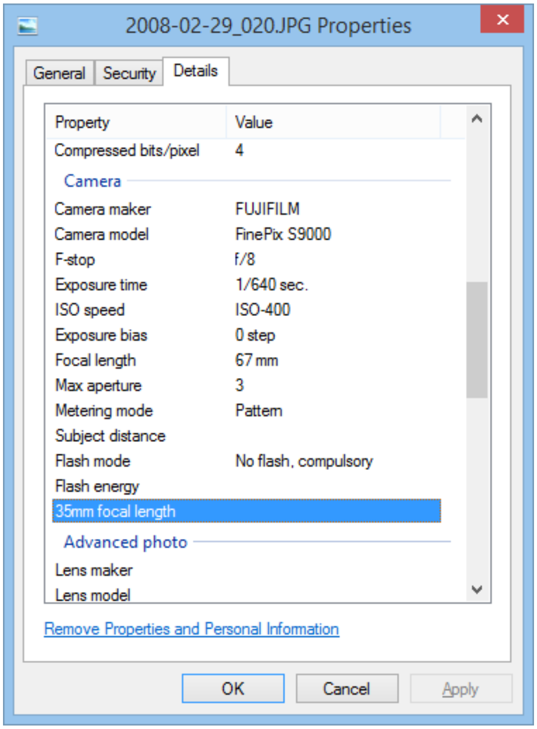
A property set has a more extensive interface, but let me come back to that in a bit.

The generic term for maps and property sets alike is a *dictionary*. A dictionary does just what the word means in colloquial language: it lets you look up an item (similar to a definition) in a collection (the dictionary) by a key (such as a word). This is very useful when the collection you're working with doesn't store items in a linear or indexed array, or doesn't have a need to do so. For example, when working with the Windows.Storage.Pickers.FileSavePicker class, you give the user a choice of file types through its fileTypeChoices property. This property is interesting because its type is IMap<String, IVector>, meaning that it's a map between a string key and a value that is itself an array. This makes sense, because you want to use something like "JPEG" for a key while yet mapping that single key to a group of values like ".jpg" and ".jpeg".

Maps are also used extensively when working with file properties, where you have access to a deeper hierarchy of metadata that's composed of key-value pairs. This is the same metadata that you can explore if you right-click a file in Windows Explorer and click the details tab, as shown here for a picture I took on a trip to Egypt where the camera recorded exposure details and so forth:

[60] Each pair is described by the IKeyValuePair<K, V> interface, but you don't encounter that construct explicitly in JavaScript. Also, the IObservableMap<K, V> interface is also meant for other languages and not used in JavaScript.

[61] In JavaScript, a property set is the only type of WinRT collection that you can create directly with the new operator. However, a few APIs that require a Map or IIterable argument have managed to slip through the API review process, thereby posing a problem for JavaScript developers. In these few cases, you have to find another method that returns the right type and then munge it to fit your needed input argument. See this forum post for an example of this workaround.


2008-02-29_020.JPG Properties
General
Security
Details
Remove Properties and Personal Information
OK
Cancel
Apply

Property	Value
Compressed bits/pixel	4
Camera	
Camera maker	FUJIFILM
Camera model	FinePix S9000
F-stop	f/8
Exposure time	1/640 sec.
ISO speed	ISO-400
Exposure bias	0 step
Focal length	67 mm
Max aperture	3
Metering mode	Pattern
Subject distance	
Flash mode	No flash, compulsory
Flash energy	
35mm focal length	
Advanced photo	
Lens maker	
Lens model	

As we'll see in Chapter 11, "The Story of State, Part 2," the `Windows.Storage.FileProperties` API is how you get to all this metadata. Images provide an `ImageProperties` object, whose `retrievePropertiesAsync` produces a map containing the metadata. You can play with this in the Simple Imaging sample if you'd like, where once you've obtained a map you can access individual properties by name (`retrievedProps` is the map):

```
retrievedProps["System.Photo.ExposureTime"]
```

On the flip side of this metadata scene is where we encounter property sets. Again, these are a form of dictionary like maps but one that an app needs to *create* for input to methods like `ImageProperties.savePropertiesAsync`. This is why `Windows.Foundation.Collections` has a concrete `PropertySet` class and not just an interface, because then we can `new` up an instance like so:

```
var properties = new Windows.Foundation.Collections.PropertySet();
```

and create key-value pairs with the `[ ]` operator:

```
properties["System.GPS.LatitudeNumerator"] = latNum;
```

or with the `insert` method:

```
properties.insert("System.GPS.LatitudeNumerator", latNum);
```

Be mindful that while the documentation for `PropertySet` lists quite a few methods and properties, only some of them are projected into JavaScript:

- Properties: size
- Lookup methods: [ ], lookup, hasKey, first (returns an iterator for the key-value pairs)
- Manipulation methods: clear, insert, remove
- Other: getView (retrieves a map view) and the mapChanged event

In summary, maps and property sets are distinct collection constructs that must be worked with through their own methods and properties. Vectors, on the other hand, are projected into JavaScript as an array and are thus suitable for data binding. If you want to bind to a map or property set, on the other hand, you'll need to maintain an array copy.

WinJS Binding Lists

The WinJS.Binding.List object is the core of collection-based data binding in WinJS and is what you use with both templates and collection controls, as discussed in this chapter and the next. That is, the name of the ListView control is quite apt: it's a *view* of a list but not the list itself. That list is a WinJS.Binding.List, which I'll just refer to as List or "list" for convenience.

A List takes a JavaScript array and makes it observable, because such a process means more than WinJS.Binding.as does with a single object. The array in question can contain any kind of objects, from simple values to complex objects and other arrays. All it takes is a new:

```
list = new WinJS.Binding.List(dataArray);
```

Note that you can build a List with a sparse array—that is, an array with noncontiguous indices (see Using Arrays).

Once created, the List provides a number of array-like methods and properties, namely length, concat, every, filter, forEach, indexOf, join (using a comma), lastIndexOf, map, push, pop, reduce, reduceRight, reverse, shift, slice, sort, and unshift. These operate exactly like they do with a typical array except that functions like concat produce another List rather than an array. The List also provides additional lookup and management methods: getAt, getItem, getItemFromKey, indexOfKey, move, setAt, and some. I'll leave you to check out the details of all these as needed, as they are all very simple and straightforward.

Hint You can create an empty List without an array, using new WinJS.Binding.List(). You then use methods like push to populate the list. Any data-binding relationships you set up with the empty list will remain in effect as you add items or remove and change existing items.

What's much more interesting are those List features that apply more to data binding in particular, which come under four groups:

- **Constructor options** The binding and proxy options affect whether items in the list are individually observable (binding) and whether the list uses the array directly for data storage (proxy).

- **Projections** The createFiltered, createGrouped, and createSorted methods generate new List objects whose contents are controlled by filtering, grouping, and sorting functions, respectively.
- **Events** As an observable object, the list can notify listeners when changes happen within its content. This is clearly important for any controls that are built on the List, such as the ListView. The supported events are itemchanged, iteminserted, itemmoved, itemmutated, itemremoved, and reload. (The List has the standard addEventListener, removeEventListener, and dispatchEvent methods, and provides on* properties for the events.) Three additional methods—notify, notifyMutated, and notifyReload—will trigger appropriate events; notifyReload is generally called within operations like reverse and sort.
- **Binding** The bind and unbind methods support binding to the list's own properties (rather than those of its contents). The dataSource property supplies an "adapter" that's specifically used by WinJS ListView and FlipView controls (see Chapter 7) or custom controls that want to use the same data model.

You'll find that many Windows SDK samples use a List, especially those that deal with collection controls; there's no shortage of examples. However, those samples are typically intertwined with the details of the control, so I've included the more fundamental BindingLists example in this chapter's companion content to demonstrate two aspects of the List class: how the list relates to its underlying array (if one exists), and the various projections. These are the subjects of the next two sections.

Sidebar: Async Data Sources and Populating from a Feed

The implementation of List is wholly synchronous, meaning that significant operations on large data sets can block the UI thread. To avoid this, consider executing list-modifying code at a lower than normal priority using the WinJS.Utilities.Scheduler discussed in Chapter 3, "App Anatomy and Performance Fundamentals." If you're creating a custom control around a potentially large collection, consider using data sources that implement the IListDataSource interface as used by WinJS controls like the ListView. This interface accommodates asynchronous behavior as well as virtualization. See "Collection Control Data Sources" in Chapter 7.

If your collection isn't so large that the synchronous nature of List will be a problem, you can still *populate* that list asynchronously such that items will automatically appear within data-bound controls. An excellent example of this (both virtualized and nonvirtualized cases) can be found in scenario 6 of the HTML FlipView control sample that we'll meet in Chapter 7 and discuss in that chapter's "Custom Data Sources and WinJS.UI.VirtualizedDataSource" section.

List-Array Relationships

By default, a List built on an array of simple values does not have any inherent relationship to the array. Scenario 1 of the BindingLists example in the companion content, for instance, creates a simple array of random numbers (this._array) and builds a list on it (this._list; js/scenario1.js):

```
this._list = new WinJS.Binding.List(this._array, this._options);
```

where this._options is optional and initially empty—more on this in a bit. Two buttons in this scenario then randomize the contents of the array and List separately. In both cases we iterate through the collection, using array[i] or List.setAt(i) to set the new numbers:

```
function randomizeArray(arr, num, lower, upper) {
    for (var i = 0; i < num; i++) {
        arr[i] = randInt(lower, upper);
    }
}

function randomizeList(list, num, lower, upper) {
    for (var i = 0; i < list.length; i++) {
        list.setAt(i, randInt(lower, upper));
    }
}
```

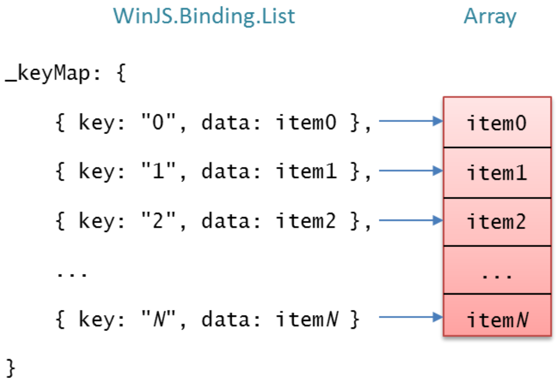
If you tap these buttons in scenario 1 (leaving the Proxy option unchecked for now), you'll see that modifying the array does *not* update the List, nor does modifying the List update the array. Now let's play with the checkboxes that recreate the list with its different constructor options:

Option	**Effect**
{binding: true} (default is false)	Calls WinJS.Binding.as for each item in the array, making each item individually observable if supported by as. This also affects any items inserted with setAt, push, unshift, etc. The binding option does not affect the relationship between the array and the list, and this option has no effect on arrays of nonobject values that are ignored by as.
{proxy: true} (default is false)	Instructs the list to use the underlying array for its own storage, meaning that changes to the list will directly update the array. Note that sparse arrays are not allowed with this option.

In scenario 1 you'll see that the Binding checkbox doesn't do anything, which is expected. Now check the Proxy option and you'll see that changes to the list now show up in the array because the list in this case is just a thin veneer over the array. Bear in mind, though, that if you then change the array, the list doesn't get updated. This means it won't fire any events and any other control that is built on the list will get badly out of sync with the real data. For this reason, it's best to avoid changing the array directly when using the proxy option.

It's very helpful to peek under the covers again and understand exactly what the List object is doing with the array, why the proxy works like it does, and what specific behaviors arise when object arrays are involved.

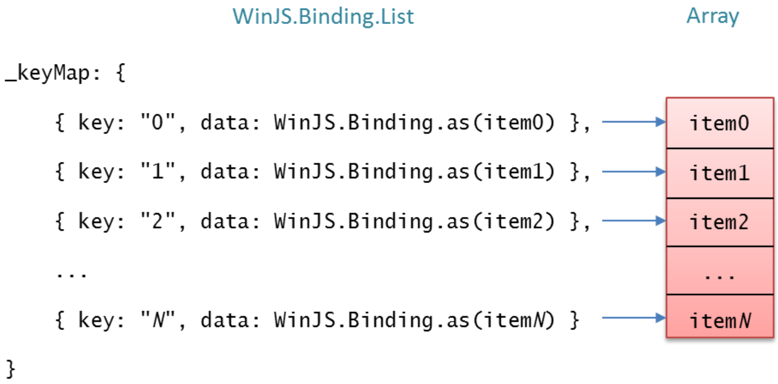
By default, with proxy set to false, the List creates an internal map of the array's contents (no such map exists if the List is created without an array). Each entry in the map holds a key for an array item (typically its positional index as a string) and the item's value, as shown in Figure 6-2.



```
_keyMap: {
    { key: "0", data: item0 },
    { key: "1", data: item1 },
    { key: "2", data: item2 },
    ...
    { key: "N", data: itemN }
}
```

FIGURE 6-2 The default relationship between a `WinJS.Binding.List` and its underlying array.

If the `binding` option is set to `true`, then the item in the map is the result of `WinJS.Binding.as(<array_item>)` instead, as shown in Figure 6-3.



```
_keyMap: {
    { key: "0", data: WinJS.Binding.as(item0) },
    { key: "1", data: WinJS.Binding.as(item1) },
    { key: "2", data: WinJS.Binding.as(item2) },
    ...
    { key: "N", data: WinJS.Binding.as(itemN) }
}
```

FIGURE 6-3 The relationship between a `WinJS.Binding.List` and its underlying array with the `binding` option set to `true`. If the array is sparse, the keys will not be contiguous, of course.

In both of these cases, all of the list manipulation methods like `concat`, `splice`, `reverse`, `sort`, and so forth *affect only the map*—the array itself is left untouched.

That changes when the `proxy` option is set to `true`, because here the `List` no longer maintains a separate map but just holds a direct reference to the array, as shown in Figure 6-4. In this case, the `binding` option is ignored and the manipulation methods *act directly on the array*—they simply call the array's methods of the same name. In this relationship it is obvious why changes to the `List` are reflected in the array, as scenario 1 of the example demonstrates.

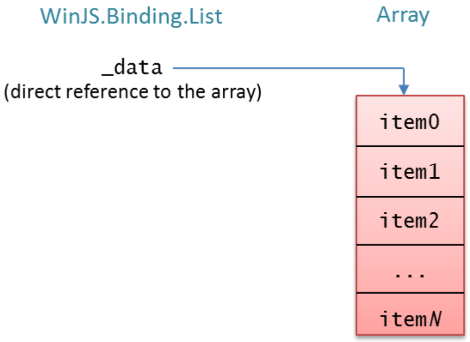


FIGURE 6-4 The relationship between a `WinJS.Binding.List` and its underlying array with the `proxy` option set to `true`. The `binding` option is ignored in this case.

With `proxy: true`, this relationship holds regardless of whether the array contains simple values or complex objects. Note also in this case that when you change the array, methods like `List.getAt` will clearly return the updated value.

In contrast, when the `List` is using a map (`proxy: false`), something a little more interesting happens with an array of objects by virtue of this little bit of (condensed) code in the list's constructor:

```
item = options.binding ? WinJS.Binding.as(inputArray[i]) : inputArray[i];
_keyMap[key] = { key: key, data: item }
```

If the item at `inputArray[i]` is a simple value, then the map will contain a *copy* of that value. However, if the item is an object, then the map will contain *the item object itself* (or an observable variant), rather than a copy. "So what?" you might ask. On the surface this doesn't seem to mean much at all, but in fact it has two important implications (still assuming `proxy: false`):

- If you replace a *whole* item object in the array, the corresponding `List` entry will be unchanged because the original item object is still intact.
- If you change *properties* on an item object in the array, the properties of the corresponding List entry *will also change*.

This effect is demonstrated in scenario 2 of the BindingLists example. This scenario repeats the UI as scenario 1 but uses an array of objects rather than simple values. I have added an additional output element that's data-bound to the first item in the list to show the effect of the `binding` option. The effect of the modification buttons and the Binding and Proxy checkboxes is the same as before. For example, with Proxy unchecked, the Modify Array Objects button replaces the array's content with new objects like so (js/scenario2.js):

```
arr[i] = { number: randInt(lower, upper), color: randColor() };
```

but because the list's map maintains references to the original items in the array, the `List` doesn't see any changes (see the left side of Figure 6-5). Similarly, changing the list's item through `setAt`:

```
list.setAt(i, { number: randInt(lower, upper), color: randColor() } );
```

replaces the whole object in the map with the new one, leaving the array unaffected (see the right side of Figure 6-5).

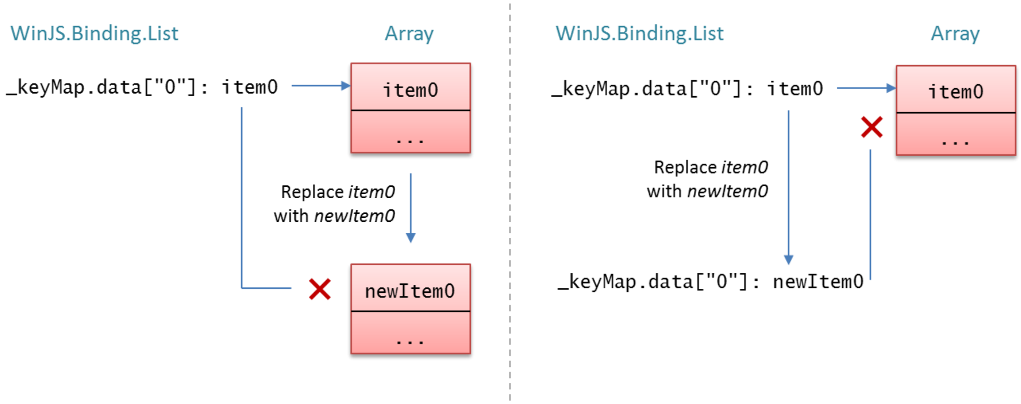


FIGURE 6-5 Replacing a whole object in either the array or the list will disconnect the related item in the other.

But now, with Proxy still unchecked, try the new button labeled Modify Array Object Properties, which instead modifies each array item this way:

```
arr[i].number = randInt(lower, upper);
arr[i].color = randColor();
```

Because we're now modifying the existing objects in the array instead of replacing them, and because the list's map contains references to those same objects, you'll see that those changes are reflected automatically in the `List` (as well as the one element bound to a list item). This is illustrated on the left side of Figure 6-6. On the flip side, the Modify List Object Properties button calls `List.getAt` to obtain the item in the map and then changes its properties:

```
var item = list.getAt(i);
item.number = randInt(lower, upper);
item.color = randColor();
```

Because that's still the same item as the one in the array, the array also sees those changes, as shown on the right side of Figure 6-6.

Note Tapping the Modify Array Objects or Modify List Objects buttons will replace all objects in the corresponding collection, disconnecting it from the other. The result is that modifying objects via properties will have no effect on the other collection. To restore the connection, re-create the List by changing one of the checkboxes.

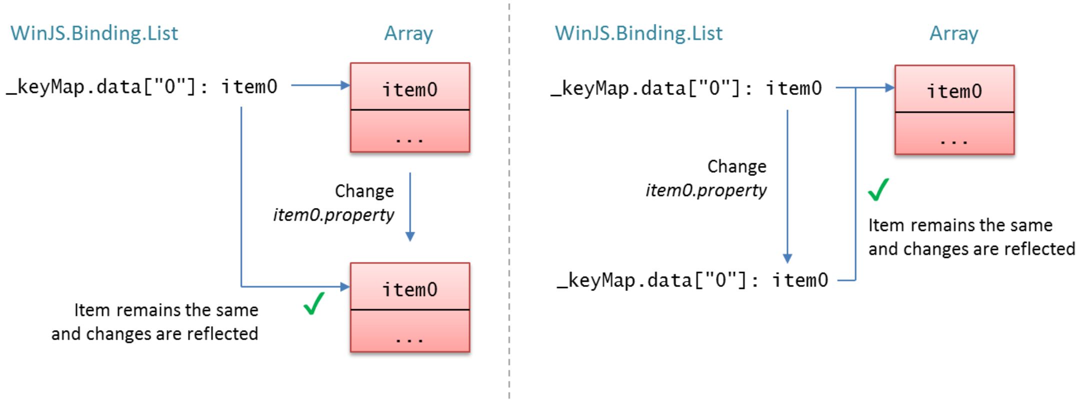


FIGURE 6-6 Changing properties of objects in either the array or the list will change the properties in the other, because both are still referencing the same object.

All of this is important to keep in mind as you're building more complex UI around a data source of some kind, whether using controls like the `ListView` or a custom control built on the `List`. That is, if you have code that modifies the array—or UI capabilities in the control to modify the `List`—you'll know how the list and the array will be correspondingly affected. Otherwise you might be left scratching your head wondering just why certain changes to the list or array (item replacement) don't affect the other whereas other changes (via properties) do!

Indeed, in scenario 2 you might notice that the `span` element that's data-bound to the item from `List.getAt(0)` automatically updates when changes come through the `List` but not when they come through the array. This is because the binding mechanics are watching the list's item, not the array item sitting beneath it. In such cases, be sure to make modifications through only the `List`.

It's also good to know how the array and `List` relate when you start using projections of the `List`, because changes to the projections will propagate to the original `List` as well, as we'll see next.

List Projections

The idea of a *projection* comes from the world of databases and data sources to mean a particular view of a data source that is still completely connected to that source. Projections are typically used to massage a raw data source into something more suitable for data binding to your UI, but without altering the original source.

For example, a data source for your personal contacts might have those contacts listed in any order and would of course contain the entire collection of those contacts. In an app's UI, however, you probably want to sort and re-sort those contacts by one or more fields in each record. Each time you change the sort order in the UI, you'd create another *sorted* projection and bind the UI to it. Again, creating a projection doesn't alter the data source. At the same time, changing properties of an item in the projection, inserting or removing items, or otherwise modifying the projection *does* propagate those changes back to the source. Similarly, if you add, remove, or change items in the underlying source—even through a projection—those changes propagate to all projections. In short, there is only ever *one* data source no matter how many projections are involved.

You might also want to present only a subset of your contacts in the UI based on certain filtering criteria. That's called a *filtered* projection. You might also want to *group* those contacts by the first letter of the last name or by location. Whatever the case, projections make all this easier by applying such operations at the level of the data source so that you can just use the same UI and the same data binding code across the board.

Sorting, filtering, and grouping are the most common operations applied to data projections, and thus `WinJS.Binding.List` supports these through its `createSorted`, `createFiltered`, and `createGrouped` methods:

Method	Return Type	Arguments and Description
`createSorted`	`SortedListProjection` (Note: The documentation for this and the other projection types shows only a few methods, but all `List` methods are available.)	Accepts a single *sorter* function argument that is called pairs of items in the list. The *sorter* returns a negative number if the first item is sorted before the second, zero if the two are sorted equally, and a positive number if the first is sorted after the second. Be aware that a *sorter* can be called up to N^2 times for a list of N items, so it should be very efficient. Note that a list's `sort` method does not create a projection. It applies the sorting in-place, just like the JavaScript array's `sort`.
`createFiltered`	`FilteredListProjection`	Accepts a single *predicate* function argument that is called for each item in the list, returning `true` if the item should be included in the filtered projection, `false` if it should not.
`createGrouped`	`GroupedSortedListProjection`	Accepts three function arguments. The first, *groupKey*, is called for each item in the list and returns the group key (a string) for that item. (Technically the `List` can handle any type of key usable with `[ ]` on an array; the `ListView` control, however, requires that keys are strings.) The second, *groupData*, is called once for each group with a representative item from that group. The *groupData* function returns an object that contains all the appropriate data for that group. Because groupData is called only once per group, you can do more computation here without affecting performance, such as calculating summary data for the group to include in the returned object. The third, *groupSorter*, is optional. It works the same as the argument to `createSorted` above but is applied to the data returned from the *groupData* function. Because the group data is typically a smaller set, this function doesn't need to be as efficient as a typical sorter.

Performance tip If deriving the group key from an item at run time requires an involved process, you'll improve overall performance by storing a prederived key in the item instead and just returning that from the group key function.

Globalization tip When sorting or otherwise comparing items, be sure to use globalization APIs like `localeCompare` to determine sort orders, or else you'll really confuse your customers in different world markets! For grouping of textual items, also use the `Windows.Globalization.Collation.CharacterGroupings` API. We'll see an example in Chapter 7 in "Quickstart #4: The ListView Grouping Sample."

Each projection generated by these methods is itself a special variant of the `List` class, meaning that each projection is individually bindable. You can also *compose* projections together. Calling `List.createFiltered().createSorted()`, for example, will first filter the original `List` and then apply sorting. `List.createFiltered().createGrouped()` will filter the `List` and then apply grouping (and sorting). It's quite common, in fact, to filter a list first and then apply sorting and/or grouping because filtering is typically a less expensive operation than sorting or grouping.

When you create a grouped projection, it won't necessarily call your grouping functions for every item right away. Instead, the projection will call those functions only when necessary, specifically when someone is asking the `List` or a projection for one or more items. This makes it possible to use projections with virtualized data sources where all the data isn't necessarily in memory but loaded only when a control like the `ListView` is preparing a page of items.

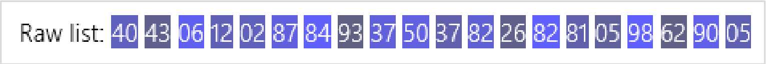
Debugging tip You can of course set breakpoints within sorter, predicate, and grouping functions and step through the code a few times. With large and even modest collections, however, breakpoints quickly become tedious as these functions will be called many, many times! Instead, try using `console.log` or `WinJS.log` to emit the parameters received by these functions as well as their return values, allowing you to review the overall results much more quickly. In fact, it's a great place to use `WinJS.log` with a tag specific to projections so that you can turn this specific logging output on and off independently.

Let's see some examples now, drawing from scenario 3 of the BindingLists example in the companion content for the fundamentals. Trust me, we'll see plenty more of projections in the next chapter when we work with collection controls!

Scenario 3 (js/scenario3.js) of the example creates an empty `List` without an underlying array and uses `push` to populate it with objects containing a random number and color (like scenario 2):

```
this._list = new WinJS.Binding.List();

for (var i = 0; i < num; i++) {
    this._list.push({ number: randInt(this._rangeLower, this._rangeUpper), color: randColor() });
}
```

The output of this list is as follows:

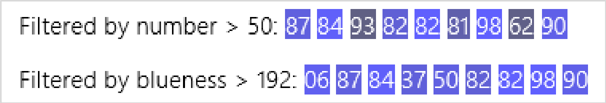


Raw list: 40 43 06 12 02 87 84 93 37 50 37 82 26 82 81 05 98 62 90 05

It then creates two filtered projections, one that contains only those items whose key is greater than

50 and another that contains only those items where the blue of the color is greater than 192. The first filtering is done with an inline predicate, the second with a separate function. Note that in this and the following code I've omitted calls to `WinJS.log` and intermediate variables used for logging, and I'll just show the output afterwards without additional comment:

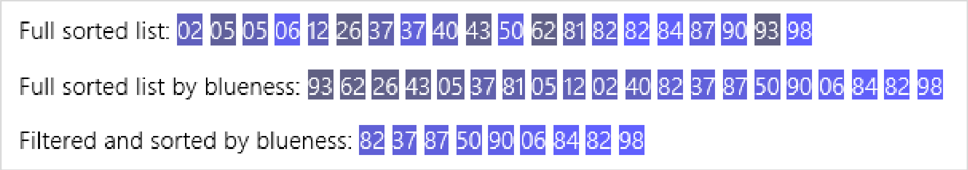
```
this._filteredByNumber = this._list.createFiltered(function (j) { return j.number > 50; });
this._filteredByBlueness = this._list.createFiltered(filterBluenessOver192);

function filterBluenessOver192(j) {
    return j.color.b > 192;
}
```



```
Filtered by number > 50: 87 84 93 82 82 81 98 62 90

Filtered by blueness > 192: 06 87 84 37 50 82 82 98 90
```

Then we have three sorted projections, one of the original list sorted by number (then blue ness in case of repeats), one sorted just by blueness, and one sorted from the list filtered by blueness:

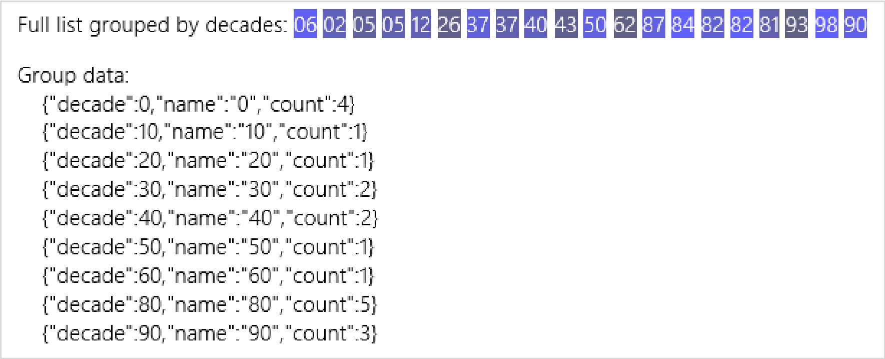
```
this._sorted = this._list.createSorted(sortByNumberThenBlueness);
this._sortedByBlueness = this._list.createSorted(sortByBlueness);
this._sortedByFilteredBlueness = this._filteredByBlueness.createSorted(sortByBlueness);

function sortByNumberThenBlueness (j, k) {
    var result = j.number - k.number;

    //If the items' numbers are the same, sort by blueness as the second tier
    if (result == 0) {
        result = sortByBlueness(j, k)
    }

    return result;
}

function sortByBlueness(j, k) {
    return j.color.b - k.color.b;
}
```



```
Full sorted list: 02 05 05 06 12 26 37 37 40 43 50 62 81 82 82 84 87 90 93 98

Full sorted list by blueness: 93 62 26 43 05 37 81 05 12 02 40 82 37 87 50 90 06 84 82 98

Filtered and sorted by blueness: 82 37 87 50 90 06 84 82 98
```

Then we can add a projection that's grouped by decades:

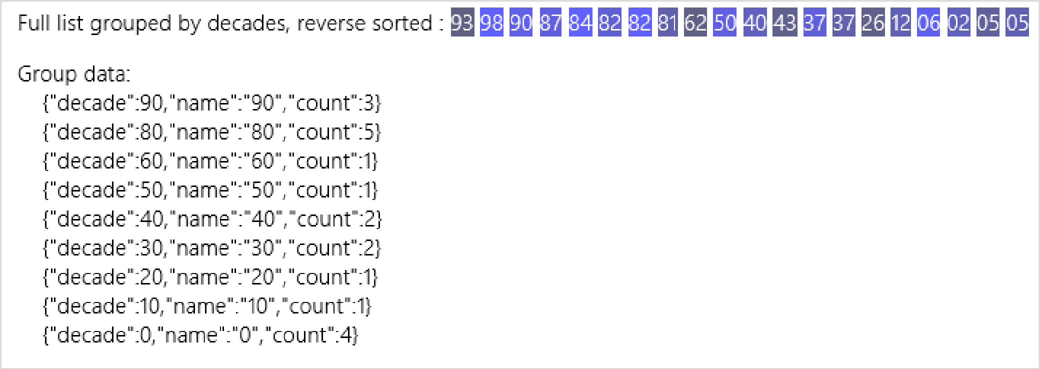
```
this._groupedByDecades = this._list.createGrouped(decadeKey, decadeGroupData.bind(this._list));

function decade(n) { return Math.floor(Math.floor(n / 10) * 10); }

function decadeKey(j) {
    return decade(j.number);
}

//"this" will be bound to the list that's being grouped
function decadeGroupData(j) {
    var dec = decade(j.number);

    //Do a quick in-place filtering of the list so we can get the population of the decade.
    var decArray = this.filter(function (v) { return (decade(v.number) == dec); })

    return {
        decade: dec,
        name: dec.toString(),
        count: decArray.length
    }
}
```



```
Full list grouped by decades: 06 02 05 05 12 26 37 37 40 43 50 62 87 84 82 82 81 93 98 90

Group data:
   {"decade":0,"name":"0","count":4}
   {"decade":10,"name":"10","count":1}
   {"decade":20,"name":"20","count":1}
   {"decade":30,"name":"30","count":2}
   {"decade":40,"name":"40","count":2}
   {"decade":50,"name":"50","count":1}
   {"decade":60,"name":"60","count":1}
   {"decade":80,"name":"80","count":5}
   {"decade":90,"name":"90","count":3}
```

The last one is grouped by decades with the groups sorted in descending order, using the same key and group data functions as above:

```
this._groupedByDecadesSortedByCount =
    this._list.createGrouped(decadeKey, decadeGroupData.bind(this._list),
        //j and k are keys as returned from decadeKey; k-j does reverse sort
        function (j, k) { return k - j; });
```



```
Full list grouped by decades, reverse sorted : 93 98 90 87 84 82 82 81 62 50 40 43 37 37 26 12 06 02 05 05

Group data:
   {"decade":90,"name":"90","count":3}
   {"decade":80,"name":"80","count":5}
   {"decade":60,"name":"60","count":1}
   {"decade":50,"name":"50","count":1}
   {"decade":40,"name":"40","count":2}
   {"decade":30,"name":"30","count":2}
   {"decade":20,"name":"20","count":1}
   {"decade":10,"name":"10","count":1}
   {"decade":0,"name":"0","count":4}
```

The Modify List button that's included with this scenario demonstrates how the projections are always tied to the underlying `List`: when the list is repopulated with new values, you'll see that all the projections get those new values as well. And if you look at the console output, you'll see the log entries from the different sorting, filtering, and grouping functions. (You can turn logging off in js/default.js by removing the `WinJS.Utilities.startLog` calls.)

> **Tip** When making many modifications to the root List, the group key, sorting, and filtering methods of the projections will be called repeatedly, but the *groupData* function won't get called unless the *groupKey* function returns a key that doesn't already exist. This means that the group data can get out of sync with the real list. For this reason I call the `List.notifyReload` method after repopulating the list to make sure that the projections are reset.

The Modify Projection button demonstrates that a change to a projection ripples through the other projections and the original list. In this case I just change the first item of the `groupedByDecades-SortedByCount` projection, which just goes to show that it doesn't matter how many projection layers you have—you're always talking to the same data source ultimately.

The code that generates the group data output answers a question you might have had earlier: what happens to the objects returned from the `createGrouped` method's *groupData* function? Did all that just vanish into the netherworld? Not at all! Those objects simply ended up in the `groups` property of the `GroupedSortedListProjection` object. This property is just another `List` (technically a `GroupedListProjection` object) that contains that group data. As a `List` you can bind UI to it, filter it, sort it, and so on. In the example, I just output its raw data along with its projection.

There's one last detail to point out with the *groupData* function and the contents of the `groups` list: because you can return whatever objects you want from *groupData*, and because that function is called only once for each group, you can calculate other information like sums, counts—and really anything else!—and include that in the returned object. The `List` and its projections won't care one way or the other. So if you want to communicate such information to the consumers of the grouped projection, the *groupData* function is the place to do it. This becomes very useful with collection controls like the `ListView` and especially the Semantic Zoom control that lets you switch between two lists with different levels of data. With semantic zoom, the zoomed-out view of a list typically shows group information and allows you to quickly navigate between groups. The objects you return from *groupData* are where you store any data you want to use in that view.

What We've Just Learned

- WinJS provides declarative data-binding capabilities for one-time and one-way binding, employing `data-win-bind` attributes and the `WinJS.Binding.processAll` method.
- `WinJS.Binding` mixins along with `WinJS.Binding.as` and `WinJS.Binding.define` simplify making arbitrary JavaScript objects observable and able to participate in data binding.

- With a little extra code to watch for changes in UI elements, an app can implement two-way binding.
- By default, data binding performs a simple copy between source and target properties. Through binding initializers, apps can customize the binding behavior, such as adding a conversion function or defining complex binding relationships between multiple properties.
- `WinJS.Binding.Template` controls provide a declarative means to easily define bits of HTML that can be repeatedly rendered with different source objects. Templates are heavily used in collection controls.
- In addition to the `Array` type of JavaScript, WinRT supports a number of other collection types such as the iterator, vector, map, and property set. Iterators and vectors project into JavaScript as an array and can be used with data binding through `WinJS.Binding.List`.
- `WinJS.Binding.List` provides an observable collection type that is a building block for collection controls. A List can be built from scratch or from an existing JavaScript array, as well as WinRT types that are projected as arrays.
- `List` projections provide filtering, sorting, and grouping capabilities on top of the original `List` without altering the list. As projections share the same data source as the original list, changes to items in the list or a projection will propagate to the list and all other projections.

Chapter 7

Collection Controls

It's a safe bet to say that wherever you are, right now, you're probably surrounded by quite a number of visible collections. This book you're reading is a collection of chapters, and chapters are a collection of pages. Those pages are collections of paragraphs, which are collections of words, which are collections of letters, which are (assuming you're reading this electronically) collections of pixels. On and on....

Your body, too, has collections on many levels, which is what one studies in college-level anatomy courses. Looking around my office and home, I see even more collections: a book shelf with books; scrapbooks with pages and pages of pictures; cabinets with cans, boxes, and bins of food; my son's innumerable toys; the case of DVDs. Even the forest outside is a collection of trees and bushes, which then have branches, which then have leaves. On and on....

We look at these things *as* collections because we've learned how to generalize specific instances of unique things—like leaves or pages or my son's innumerable toys—into categories or groups. This gives us powerful ways to organize and manage those things (except for the clothes in my closet, as my wife will attest). And just as the physical world around us is made of collections, the digital world that we use to represent the physical is naturally full of collections as well.

In Chapter 6, "Data Binding, Templates, and Collections," we learned about the data side of this story: the features of the `WinJS.Binding` namespace, including binding templates and the observable `List` class. Now we turn our attention to collection controls through which we visualize and manipulate that data.

In this chapter we'll explore the three collection controls provided by WinJS—available on both Windows and Windows Phone—that can handle items of arbitrary complexity both in terms of data and presentation (unlike the HTML controls). These are the `FlipView`, which shows one item from a collection at a time; the `Repeater`, which when combined with the `ItemContainer` we saw in Chapter 5, "Controls and Control Styling," provides a lightweight means to display a collection of multiple items; and the `ListView`, which displays a collection of multiple items with provisions for layouts, interactivity, drag and drop, keyboarding, cell spanning, and more. As you might expect, the ListView is the richest of the three. Because it's the centerpiece of many app designs, we'll be spending the bulk of this chapter exploring its depths.

In this mix we'll also encounter how to work with some additional data sources, such as files and online feeds, and we'll cross paths with the concept of *semantic zoom*, which is implemented through the WinJS `SemanticZoom` control.

"But hey," you might be asking, "what about the intrinsic HTML collection controls like `<select>` and `<table>`, as well as other list-related elements like `<ul>`, `<ol>`, and `<datalist>`? Don't these have a place in this discussion?" Indeed they do! Not so much with static content, of course—you already know how to write such HTML. What's instead really interesting is asking how we can bind such elements to a List so that they, like other controls, automatically reflect the contents of that collection. This turns out to be one of the primary uses of the Win JS Repeater—and our very first topic!

Get a Bing Search API account Three of the SDK samples that we'll be working with require a Bing Search API account, which is free for under 5000 transactions a month. Visit the Windows Azure Marketplace page for this API to get started. Once you've signed up, go to the My Account page. The Primary Account Key listed there is what you'll need in the samples.

ListView 1.0 to 2.0 changes The ListView control got quite an overhaul between WinJS 1.0 (Windows 8) and WinJS 2.0 (Windows 8.1), resulting in many performance improvements and API changes. This chapter focuses on only the WinJS 2.0 ListView and its features and does not point out the specific changes from WinJS 1.0. For those details, please refer to API changes for Windows 8.1.

Collection Control Basics

In previous chapters we've built our understanding of collections, templates, data binding, and simple controls that can be used within a template. Collection controls—the Repeater, the FlipView, and the ListView—bring all these fundamentals together to bring those collections to life in your app's UI.

Note Technically the WinJS.UI.NavBarContainer control, which we'll see in Chapter 9, "Commanding UI," is also a collection control and can, in fact, be used outside of a nav bar. Its utility outside that context is limited, however. You might also come across the WinJS TabContainer control, but this is for internal use by the ListView.

Quickstart #1: The WinJS Repeater Control with HTML controls

Here's a quick quiz question for a quickstart: given all that you know about data binding, the WinJS.Binding.List, data-win-bind, and WinJS.Binding.processAll, how would you take an empty HTML <select> element like this:

```
<select id="select1"> <!-- Options to be created at runtime --></select>
```

and populate it with data from a dynamically-generated array, perhaps from a web API?

```
var animals = [ { id: 1, description: "Hamster" },
                { id: 2, description: "Koala" },
                { id: 3, description: "Llama" } ];
```

A quick, brute-force method, which you've probably employed at some time in your career, would be to just iterate the array and create <option> elements within the <select>:

```
var e = document.getElementById("select1");
```

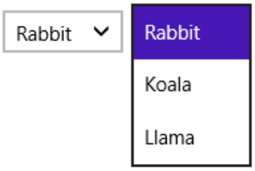
```
animals.forEach(function (item) {
    var o = document.createElement("option");
    o.value = item.id;
    o.textContent = item.description;
    e.appendChild(o);
});
```

And you'd generally repeat this process whenever the array contents changed, clearing out the `<select>` and creating each `<option>` anew.

Now to detect such changes automatically, we'd want to turn that array into a `Binding.List`, then drop in a `data-win-bind` attribute to each `<option>` element and call `Binding.processAll` for it:

```
//Make each item in the List individually bindable
var bindingList = new WinJS.Binding.List(animals, { binding: true });

bindingList.forEach(function (item) {
    var o = document.createElement("option");
    o.setAttribute("data-win-bind", "value: id; textContent: description");
    e.appendChild(o);
    WinJS.Binding.processAll(o, item);
});

//Change one item in the list to show that binding is set up.
var item = bindingList.getAt(0);
item.description = "Rabbit";
```

This would work quite well, producing a `<select>` element as follows:



Now you might be thinking, "We could encapsulate this process into a custom control, declared with `data-win-control` on the `<select>` element, yes?" After all, we know that when `WinJS.UI.process[All]` sees a `data-win-control` attribute, it simply calls the given constructor (with options) and lets that constructor do whatever it wants. This means we can really use WinJS controls or a custom control with any container element we'd like, not just `div` and `span`: we could put all the above code into control constructor, specify the `List` through one of its options, and even turn the child elements declared in HTML into a `WinJS.Binding.Template` that gets rendered for each item in the collection. Then our markup would become very simple (assuming appropriate namespaces):

```
<select data-win-control="Controls.ListMaker" data-win-options="{data: Data.bindingList}">
    <option data-win-bind="value: id; textContent: description"></option>
</select>
```

If that's how you're thinking, you're well attuned to some folks on the WinJS team who created a little beauty that does exactly this: the WinJS.UI.Repeater control. The Repeater is useful anywhere you need to create multiple copies of the same set of elements where each copy is bound to an item in a collection. It neither adds nor imposes any other functionality, though of course you can have it render whatever interactive content you want, including other WinJS controls and nested Repeaters.

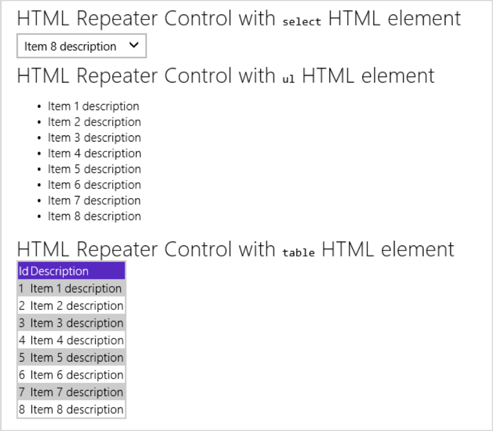
Here's how it's used in scenario 1 of the HTML Repeater control sample with <select>, <ul>, and <tbody> elements; note that each Repeater has only one immediate child element (html/scenario1.html):

```
<select data-win-control="WinJS.UI.Repeater" data-win-options="{data: Data.items}">
    <option data-win-bind="value: id; textContent: description"></option>
</select>

<ul data-win-control="WinJS.UI.Repeater" data-win-options="{data: Data.items}">
    <li data-win-bind="textContent: description"></li>
</ul>

<table class="table">
    <thead class="table-header"><tr><td>Id</td><td>Description</td></tr></thead>
    <tbody class="table-body"
        data-win-control="WinJS.UI.Repeater" data-win-options="{data: Data.items}">
        <tr class="table-body-row">
            <td data-win-bind="textContent: id"></td>
            <td data-win-bind="textContent: description"></td>
        </tr>
    </tbody>
</table>
```

The data option here points to the repeater's data source, Data.items, which is a WinJS.Binding.List defined in js/scenario1.js with some thoroughly uninspiring items:

```
WinJS.Namespace.define("Data", {
    items: new WinJS.Binding.List([
        { id: 1, description: "Item 1 description" },
        { id: 2, description: "Item 2 description" },
        { id: 3, description: "Item 3 description" },
        //And so on...
    ])
});
```

The output for scenario 1 is shown below.



HTML Repeater Control with select HTML element

Item 8 description

HTML Repeater Control with ul HTML element

- Item 1 description
- Item 2 description
- Item 3 description
- Item 4 description
- Item 5 description
- Item 6 description
- Item 7 description
- Item 8 description

HTML Repeater Control with table HTML element

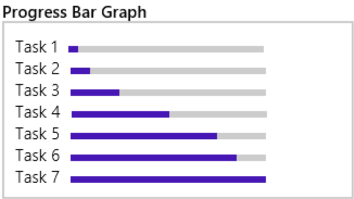
Id	Description
1	Item 1 description
2	Item 2 description
3	Item 3 description
4	Item 4 description
5	Item 5 description
6	Item 6 description
7	Item 7 description
8	Item 8 description

Because the Repeater turns its child element (and there must be only one) into a `Template` using the `extractChild` option, those elements are removed from the DOM. Rendering the template for each item in the collection will then create individual copies bound to those items. And because the Repeater just works with a template, you can just as easily declare the template elsewhere and perhaps use it with multiple `Repeater` controls. In this case you just point to it in the `template` option, as shown in scenario 2 of the sample where we see both `<label>` and `<progress>` elements in the `Template` control (html/scenario2.html):

```
<div class="template" data-win-control="WinJS.Binding.Template">
    <div class="bar">
        <label class="label" data-win-bind="textContent: description"></label>
        <progress data-win-bind="value: value" max="100"></progress>
    </div>
</div>
<h3>Progress Bar Graph</h3>
<div class="graph" data-win-control="WinJS.UI.Repeater"
    data-win-options="{data: Data.samples2, template: select('.template')}">
</div>
```

The recommended practice for naming templates, that's shown here, is to use a class rather than an id (which also works, but we'll discuss the caveats in "Referring to Templates" later in the chapter). You then use `select('<selector>')` to refer to the template. Personally, I wouldn't use a generic name like *template*; something like *barGraphTemplate* would be better.

Anyway, the result of this Repeater is as follows, which shows that the Repeater is perfect at creating things like graphs and charts where repeated elements are involved:



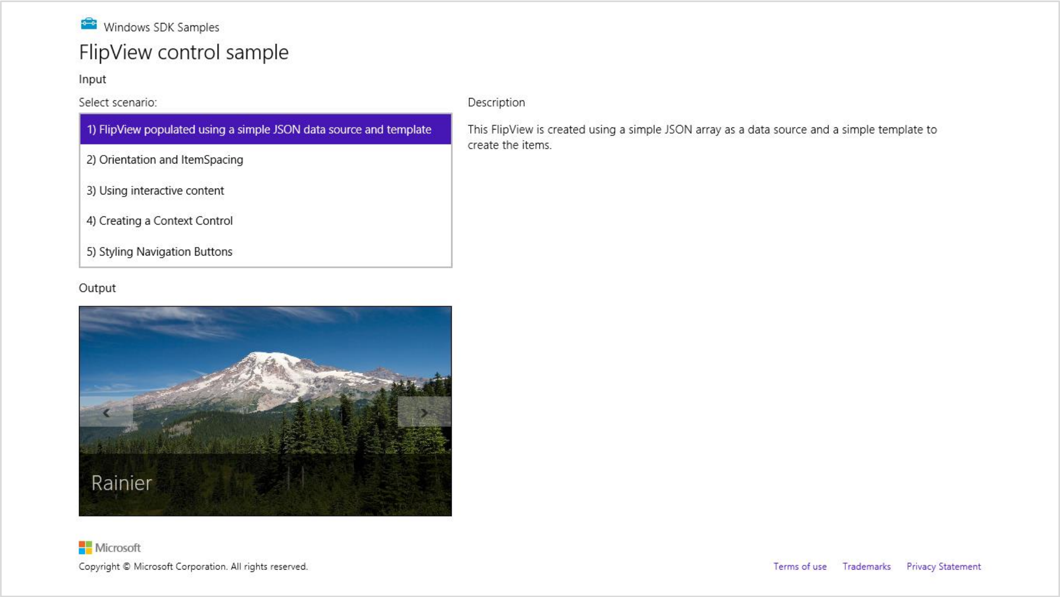
With the Repeater and others control that can declaratively reference a template (like the FlipView and ListView), note that it's important to always declare the template *before* any references. This is so `WinJS.UI.processAll` will instantiate the template first; otherwise references to it will not be valid.

It's also possible to specify an item rendering function for the `template` option (see scenario 3 in the sample), because that's ultimately what gets inserted there when you use a declarative template. We'll come back to this in "How Templates Work with Collection Controls," and we'll see more of the Repeater in "Repeater Features and Styling."

Quickstart #2: The FlipView Control Sample

As shown in Figure 7-1, the HTML FlipView control sample is both a great piece of reference code for the FlipView and a great visual tool through which to explore the control itself. For the purposes of this Quickstart, let's just look at the first scenario of populating the control from a simple data source and using a template for rendering the items, as we're already familiar with these mechanisms and will become even more so! We'll come back to the other FlipView scenarios later in this chapter in "FlipView Features and Styling."

It's worth mentioning that although this sample demonstrates the control's capabilities in a relatively small area, a FlipView can be any size, even occupying most of the screen. A common use for the control, in fact, is to let users flip through full-sized images in a photo gallery. See Guidelines for FlipView controls for more.



Windows SDK Samples

FlipView control sample

Input

Select scenario:

1) FlipView populated using a simple JSON data source and template

2) Orientation and ItemSpacing

3) Using interactive content

4) Creating a Context Control

5) Styling Navigation Buttons

Description

This FlipView is created using a simple JSON array as a data source and a simple template to create the items.

Output

Microsoft

Copyright © Microsoft Corporation. All rights reserved.

Terms of use Trademarks Privacy Statement

FIGURE 7-1 The HTML FlipView control sample; the FlipView is the control displaying the picture.

The FlipView's constructor is WinJS.UI.FlipView, and its primary options are itemDataSource and itemTemplate (html/simpleFlipview.html):

```
<div id="simple_FlipView" class="flipView" data-win-control="WinJS.UI.FlipView"
    data-win-options="{ itemDataSource: DefaultData.bindingList.dataSource,
        itemTemplate: simple_ItemTemplate }">
</div>
```

The Template control (also in html/simpleFlipview.html) is just like those we've seen before:[62]

```
<div id="simple_ItemTemplate" data-win-control="WinJS.Binding.Template">
    <div class="overlaidItemTemplate">
        <img class="image" data-win-bind="src: picture; alt: title" />
        <div class="overlay">
            <h2 class="ItemTitle" data-win-bind="innerText: title"></h2>
        </div>
    </div>
</div>
```

Note again that a template must be declared in markup before any controls that reference them (or you can use a function, see "How Templates Work with Collection Controls"). Anyway, the prosaically named simple_ItemTemplate here is made of img and h2 elements, the latter being contained in a div whose background color is partially transparent (see css/default.css for the .overlaidItemTemplate .overlay selector). As usual, we're also binding these elements to the picture and title properties of the data source.

Tip Within both FlipView and ListView controls, as with the ItemContainer, you need to add the win-interactive class to any nested controls for them to be directly interactive rather than being treated as static content in the overall item. win-interactive specifically tells the outer item container to pass input events to the inner controls.

There's one important distinction with the FlipView's itemDataSource option—did you see it? Instead of directly referring to the WinJS.Binding.List of DefaultData.bindingList (which is created in js/DefaultData.js as we've seen many times), we're binding to the list's dataSource property:

```
data-win-options="{ itemDataSource: DefaultData.bindingList.dataSource }"
```

The dataSource property is an object that provides the methods of the WinJS.UI.IListDataSource interface, and it exists specifically to adapt a List to the needs of the FlipView and ListView controls. (It exists for no other purpose, in fact.) If you forget and attempt to just bind to the List directly, you'll see an exception that says, "Object doesn't support property or method 'createListBinding'." In other words, both FlipView and ListView don't work directly with a List; they work with an IListDataSource. As we'll see later in "Collection Control Data Sources," this allows the control to work with other kinds of sources like the file system or online feeds.

[62] In the sample you might notice the inline style="display:none" on the template. This is unnecessary as templates hide themselves automatically.

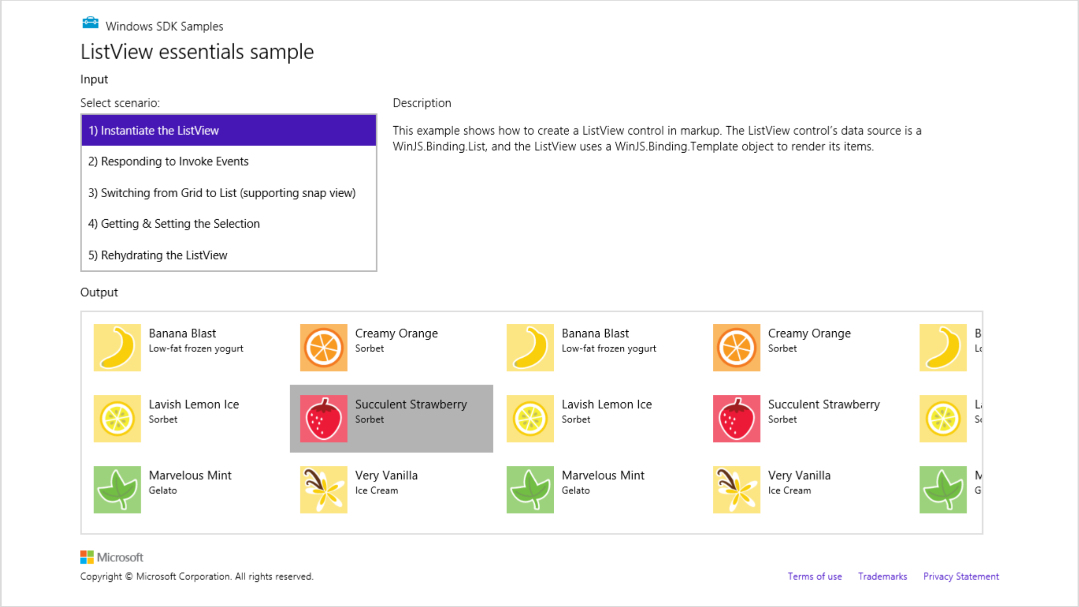
Whatever the case, note that `itemDataSource` sets up one-way binding by default, but you can use other binding initializers to change that behavior.

Quickstart #3: The ListView Essentials Sample

The basic mechanisms for data sources and templates apply to the ListView control exactly as they do to FlipView, Repeater, and any other control. We can see these in the HTML ListView essentials sample (shown in Figure 7-2); scenarios 1 and 2 specifically create a ListView and respond to item events.

The key thing that distinguishes a ListView from other collection controls is that it applies a *layout* to its presentation of that collection. That is, in addition to the data source and the template, the ListView also needs something to describe how those items visually relate to one another. This is the ListView's `layout` property, which we see in the markup for scenario 1 of the sample along with a few other behavioral options (html/scenario1.html):

```
<div id="listView" data-win-control="WinJS.UI.ListView"
    data-win-options="{ itemDataSource: myData.dataSource,
      itemTemplate: smallListIconTextTemplate, selectionMode: 'none',
      tapBehavior: 'none', swipeBehavior: 'none', layout: { type: WinJS.UI.GridLayout } }">
</div>
```


Windows SDK Samples
ListView essentials sample
Input
Select scenario:
1) Instantiate the ListView
2) Responding to Invoke Events
3) Switching from Grid to List (supporting snap view)
4) Getting & Setting the Selection
5) Rehydrating the ListView
Description
Output

This example shows how to create a ListView control in markup. The ListView control's data source is a WinJS.Binding.List, and the ListView uses a WinJS.Binding.Template object to render its items.

FIGURE 7-2 The HTML ListView essentials sample.

As with the FlipView, the ListView's itemDataSource property must be an object with the IListDataSource interface, conveniently provided by a Binding.List.dataSource property. Again, we can place other kinds of data sources behind this interface, as we'll see in the "Collection Control Data Sources" section.

The control's item template is defined earlier in scenario1.html with the id of *smallListIconTextTemplate* and is essentially the same sort of thing we saw with the FlipView (an img and some text elements), so I won't list it here. And as with the other collection controls you can use a rendering function instead. See "How Templates Work with Collection Controls" later on.

In the control options we see three behavioral properties: selectionMode, tapBehavior, and swipeBehavior. These are all set to 'none' in this sample to disable selection and click behaviors entirely, making the ListView a passive display. It can still be panned, but the items don't respond to input. (Also see "Sidebar: Item Hover Styling.")

As for the layout property, this is an object of its own, whose type property indicates which layout to use. WinJS.UI.GridLayout, as we're using here, is a two-dimensional top-to-bottom then left-to-right algorithm, suitable for horizontal panning (but which can also be rearranged for vertical panning). WinJS provides another layout type called WinJS.UI.ListLayout, a one-dimensional top-to-bottom organization that's suitable for vertical panning, especially in narrow views. (We'll see this with the Grid App project template shortly; the ListView essentials sample doesn't handle narrow widths.) The other layout in WinJS.UI is CellSpanningLayout for variable-sized items, and it's also a relatively simple matter to create custom layouts. We'll see all of these in "ListView Features and Styling" except for custom layouts, which are discussed in Appendix B, "WinJS Extras."

Tip A number of errors will cause the ListView constructor to fail. First, check that your data source is constructed properly and field names match between it and the template. Second, if you're using a WinJS.Binding.List, be sure to assign its dataSource property to the ListView's itemDataSource. Third, the ListView will crash if the data source can't be found or isn't instantiated yet, so move that earlier in your code. Similarly, the template must always be present before creating the ListView, so its markup should come before the ListView's. And finally, make sure the reference to the template in the ListView's options is correct.

Now while the ListView control in scenario 1 displays only passive items, we often want those items to respond to a click or tap. Scenario 2 shows this, where the tapBehavior property is set to invoke (see html/scenario2.html). Technically this should be invokeOnly because invoke isn't a real option and we're getting invokeOnly by default. Other options come from the WinJS.UI.TapBehavior enumeration. Other variations are toggleSelect, which will select or deselect an item, depending on its state, and then invoke it; and directSelect, where an item is always selected and then invoked. You can also set the behavior to none so that clicks and taps are ignored, as we saw in scenario 1.

When an item is invoked, the ListView control fires an itemInvoked event. You can wire up a handler by using either addEventListener or the ListView's oniteminvoked property. Here's how scenario 2 does it (slightly rearranged from js/scenario2.js):

```
var listView = element.querySelector('#listView').winControl;
listView.addEventListener("iteminvoked", itemInvokedHandler, false);

function itemInvokedHandler(eventObject) {
    eventObject.detail.itemPromise.done(function (invokedItem) {
        // Act on the item
    });
}
```

Note that we're listening for the event on the WinJS *control*, but it also works to listen for the event on the containing element thanks to bubbling. This can be helpful if you need to add listeners to a control before its instantiated, because the containing element will already be there in the DOM.

In the code above, you could also assign a handler by using the listView.oniteminvoked property directly, or you can specify the handler in the iteminvoked property data-win-options (in which case it must be marked safe for processing). The event object you then receive in the handler contains a *promise* for the invoked item, not the item itself, because the underlying data source might deliver the full item asynchronously. So you need to call it's done or then method to obtain the actual item data. It's also good to know that you should never change the ListView's data source properties directly within an iteminvoked handler, because you'll probably cause an exception. If you have the need, wrap the change code inside a call to setImmediate so that you can yield the UI thread first.

Sidebar: Item Hover Styling

Although disabling selection and tap behaviors on a ListView creates a passive control, hovering over items with the mouse (or suitable touch hardware) still highlights each item; refer back to Figure 7-2. You can control this by styling the .win-container:hover pseudo-selector for the desired control. For example, the following style rule removes the hover effect entirely:

```
#myListView .win-container:hover {
    background-color: transparent;
    outline: 0px;
}
```

Quickstart #4: The ListView Grouping Sample

Displaying a list of items is great, but more often than not, a collection needs another level of organization—such as filtering, sorting, and especially grouping. This is readily apparent when I open the file drawer next to my desk, which contains a collection of various important and not so important papers. Right away, on the file folder tabs, I see my groups: Taxes, Financials, Community, Insurance, Cars, Writing Projects, and Miscellany (among others). Clearly, then, we need a grouping facility within a collection control and ListView is happy to oblige.

There are two parts to this. One is grouping of the data source itself, which we know happens through the List.createGrouped method (along with createFiltered and createSorted), as we saw in Chapter 6. The WinJS.Binding.GroupedSortedListProjection that we get back in that case supplies both its grouped items (through its dataSource property) and a GroupedListProjection of the groups themselves through its `groups` property. Note that when we refer to `groups` in a ListView we'll also use its `groups.dataSource` property.

The second part is representing the grouped data visually. This is demonstrated in the HTML ListView grouping and Semantic Zoom sample (the output for scenario 1 is shown in Figure 7-3). As with the Essentials sample, the code in js/groupedData.js contains a lengthy in-memory array around which we create a `List`. Here's a condensation to show the item structure (I'd show the whole array, but this is making me hungry for some dessert!):

```
var myList = new WinJS.Binding.List([
    { title: "Banana Blast", text: "Low-fat frozen yogurt", picture: "images/60Banana.png" },
    { title: "Lavish Lemon Ice", text: "Sorbet", picture: "images/60Lemon.png" },
    { title: "Creamy Orange", text: "Sorbet", picture: "images/60Orange.png" },
    ...
```

Here we have a bunch of items with `title`, `text`, and `picture` properties. We can group them any way we like and even change the groupings on the fly. As Figure 7-3 shows, the sample groups these by the first letter of the title using both a `GridLayout` and a `ListLayout`.

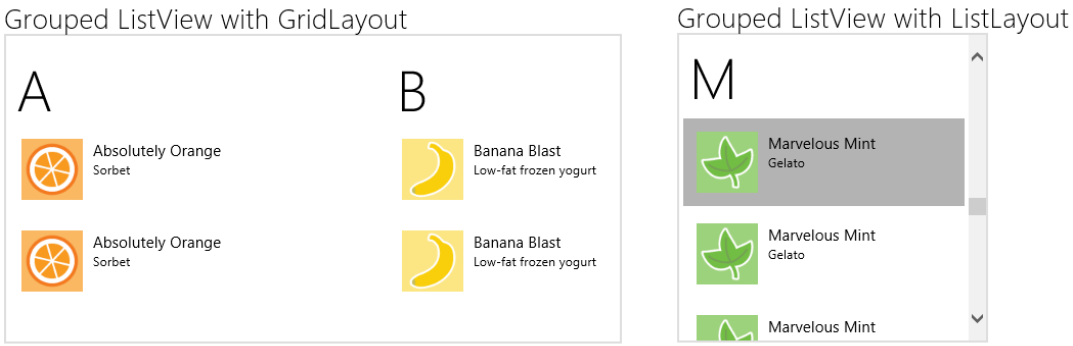


FIGURE 7-3 The output of scenario 1 of the HTML ListView grouping and Semantic Zoom sample.

If you take a peek at the ListView reference, you'll see that the control works with two templates and two collections: that is, alongside its `itemTemplate` and `itemDataSource` properties are ones called `groupHeaderTemplate` and `groupDataSource`. The group-capable layouts use these to organize the groups and create the headers above the items.

The header template in html/scenario1.html is very simple:

```
<div id="headerTemplate" data-win-control="WinJS.Binding.Template">
    <div class="simpleHeaderItem">
        <h1 data-win-bind="innerText: groupTitle"></h1>
    </div>
</div>
```

This is referenced in the control declaration along with the appropriate grouped projection's `groups.dataSource` (other options omitted):

```
<div id="listView" data-win-control="WinJS.UI.ListView"
    data-win-options="{ groupDataSource: myGroupedList.groups.dataSource,
        groupHeaderTemplate: headerTemplate }">
</div>
```

myGroupedList is, of course, created with the original list's createGrouped method:

```
var myGroupedList = myList.createGrouped(getGroupKey, getGroupData);
```

The getGroupKey function returns a single character to use for the grouping. With textual data, you should always use the Windows.Globalization.Collation.CharacterGroupings class and its lookup method to determine groupings—never assume that something like the first character in a string is the right one! The sample shows how simple this is:

```
var charGroups = Windows.Globalization.Collation.CharacterGroupings();

function getGroupKey(dataItem) {
    return charGroups.lookup(dataItem.title);
}
```

Remember that this group key function determines *only* the association between the item and a group, nothing more. It also gets called for every item in the collection when createGrouped is called, so it should be a quick operation. This is why we call CharacterGroupings outside of the function.

Performance tip As noted in Chapter 6, if deriving the group key from an item at run time requires an involved process, you'll improve overall performance by storing a prederived key in the item instead and just returning that from the group key function.

The sample's group data function, getGroupData, is called with a representative item for each group to obtain the data that ends up in the groups collection. It simply returns an object with a single groupTitle property that's the same as the group key, but of course you can make that value anything you want. Note that by using our world-ready getGroupKey function, we're handling globalization concerns appropriately:

```
function getGroupData(dataItem) {
    var key = getGroupKey(dataItem);

    return {
        groupTitle: key
    };
}
```

You might be asking, "Why do we have the group data function separated out at all? Why not just create that collection automatically from the group keys?" It's because you often want to include additional properties within the group data for use in the header template or in a zoomed-out view (with semantic zoom). Think of your group data function as providing summary information for each group. (The header text is only the most basic such summary.) Because this function is called only once per group, rather than once per item, it's the proper time to calculate or otherwise retrieve summary-level data. For example, to show an item count in the group headers, we just need to include that property in the objects returned by the group data function, then data-bind an element in the header template to that property.

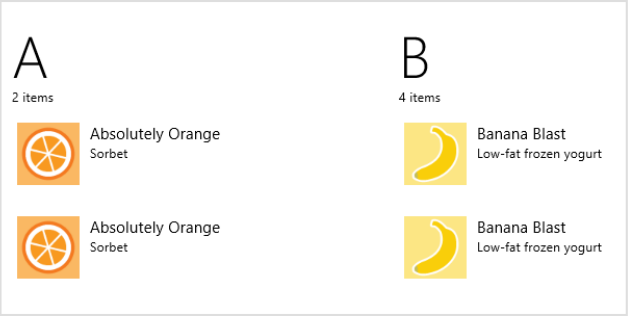
For example, in a slightly modified version of the sample in this chapter's companion code I use `createFiltered` to obtain a projection of the list filtered by the current key.[63] The `length` property of this projection is then the number of items in the group:

```
function getGroupData(dataItem) {
    var key = getGroupKey(dataItem);

    //Obtain a filtered projection of our list, checking for matching keys
    var filteredList = myList.createFiltered(function (item) {
        return key == getGroupKey(item);
    });

    return {
        groupTitle: key,
        count: filteredList.length
    };
}
```

With this `count` property in the collection, we can use it in the header template:

```
<div id="headerTemplate" data-win-control="WinJS.Binding.Template">
    <div class="simpleHeaderItem">
        <h1 data-win-bind="innerText: groupTitle"></h1>
        <h6><span data-win-bind="innerText: count"></span> items</h6>
    </div>
</div>
```

After a small tweak in css/scenario1.css—changing the `simpleHeaderItem` class height to 65px to make a little more room—the list will now appears as follows:



[63] Creating a filtered projection is also useful to intentionally limit the number of items you want to display in a control, where your predicate function returns `true` for only that number.

One other note for scenario 1 is that although it doesn't use a group sorter function with createGrouped. It actually does an initial (globalized) sort of the raw data before creating the List:

```
var sortedData = rawData.sort(function (left, right) {
    return right.title.localeCompare(left.title);
});

var myList = new WinJS.Binding.List(sortedData);
```

Although this results in sorted groups, adding new items to the list or a projection would not sort them properly nor sort the groups (especially if a new group is created as a result). It would be better, then, to create a sorted projection first (through createSorted), then the grouped projection from that using a locale-aware group sorter function. The modified sample shows this, but I'll leave you to examine the code.

The other little bit demonstrated in this sample—in scenario 3—is the ability to create headers that can be invoked. This is done by setting the ListView's groupHeaderTapBehavior property to invoke (html/scenario3.html; other options omitted):

```
<div id="listView" data-win-control="WinJS.UI.ListView"
    data-win-options="{groupHeaderTapBehavior: WinJS.UI.GroupHeaderTapBehavior.invoke }">
</div>
```

A header is invoked with a click or tap, obviously, and if it has the keyboard focus the Enter key will also do the job. When invoked, the ListView fires a groupheaderinvoked event where the eventArgs.detail object contains groupHeaderPromise and groupHeaderIndex properties.

ListView in the Grid App Project Template

Now that we've covered the details of the ListView control and in-memory data sources, we can finally understand the rest of the Grid App project template in Visual Studio and Blend. As we covered in "The Navigation Process and Navigation Styles" section of Chapter 3, "App Anatomy and Performance Fundamentals," this project template provides an app structure built around page navigation: the home page (pages/groupedItems) displays a collection of sample data (see js/data.js) in a ListView control, where each item's presentation and the group headings are described by templates. Figure 7-4 shows the layout of the home page and identifies the relevant ListView elements. As we discussed before, tapping an item navigates to the pages/itemDetail page and tapping a heading navigates to the pages/groupDetail page, and now we can see how that all works with the ListView control.

The ListView in Figure 7-4 occupies the lower portion of the app's contents. Because it can pan horizontally, it actually extends all the way across; various CSS margins are used to align the first items with the layout silhouette while allowing them to bleed to the left when the ListView is panned.

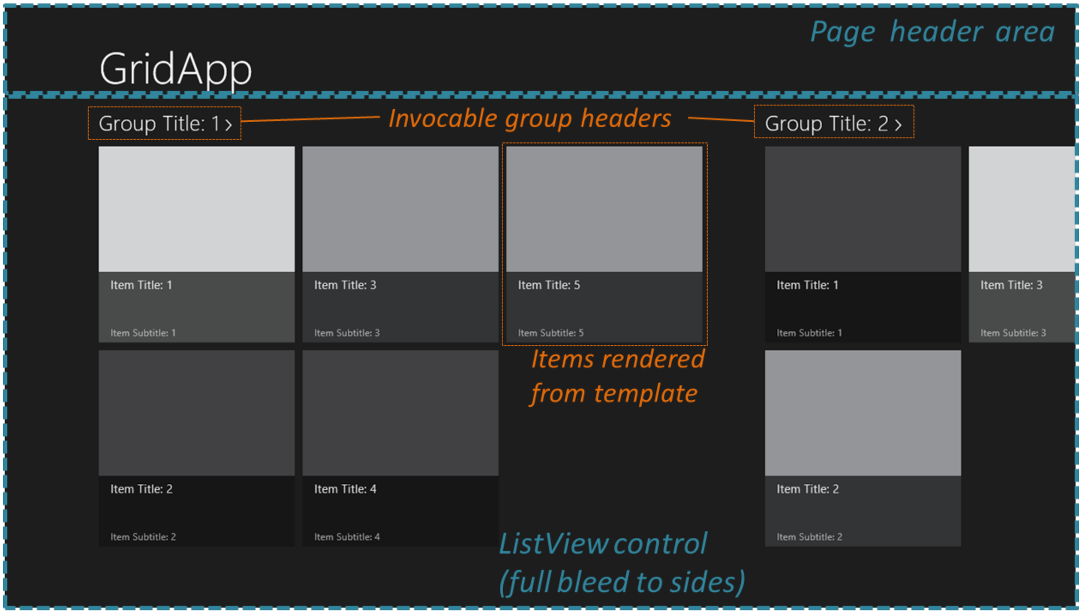


FIGURE 7-4 ListView elements as shown in the Grid App template home page. (All colored items are added labels and lines.)

There's quite a bit going on with the ListView in this project, so let's take one part at a time. For starters, the control's markup in pages/groupedItems/groupedItems.html is very basic, where the only option is to indicate that the items have no selection behavior:

```
<div class="groupeditemslist win-selectionstylefilled" aria-label="List of groups"
    data-win-control="WinJS.UI.ListView"
    data-win-options="{ selectionMode: 'none' }"
        layout: {type: WinJS.UI.GridLayout, groupHeaderPosition: 'top'} >
</div>
```

Switching over to pages/groupedItems/groupedItems.js, the page's `ready` method handles initialization:

```
ready: function (element, options) {
    var listView = element.querySelector(".groupeditemslist").winControl;
    listView.groupHeaderTemplate = element.querySelector(".headerTemplate");
    listView.itemTemplate = element.querySelector(".itemtemplate");
    listView.addEventListener("groupheaderinvoked", this._groupHeaderInvoked.bind(this));
    listView.oniteminvoked = this._itemInvoked.bind(this);
    listView.itemDataSource = Data.items.dataSource;
    listView.groupDataSource = Data.groups.dataSource;
    listView.element.focus();
}
```

Here you can see that the control's templates can be set in code just as easily as from markup, and in this case we're using a class to locate the template element instead of an id. Why does this work? It's because we've actually been referring to elements the whole time: the app host automatically creates a variable for an element that's named the same as its id. It's the same thing. Plus, references to templates ultimately resolve into a rendering function, which we'll again cover later.

You can also see how this page assigns handlers to the `iteminvoked` and `groupheaderinvoked` events. Those handlers call `WinJS.Navigation.navigate` to go to the itemDetail or groupDetail pages as we saw in Chapter 3:

```
_itemInvoked: function (args) {
    var item = Data.items.getAt(args.detail.itemIndex);
    nav.navigate("/pages/itemDetail/itemDetail.html", { item: Data.getItemReference(item) });
    }
},

_groupHeaderInvoked: function (args) {
    var group = Data.groups.getAt(args.detail.groupHeaderIndex);
    nav.navigate("/pages/groupDetail/groupDetail.html", { groupKey: group.key });
},
```

Here now are the templates for the home page (pages/groupedItems/groupedItems.html):

```
<div class="headertemplate" data-win-control="WinJS.Binding.Template">
    <button class="group-header win-type-x-large win-type-interactive"
        role="link" tabindex="-1" type="button" >
        <span class="group-title win-type-ellipsis" data-win-bind="textContent: title"></span>
        <span class="group-chevron"></span>
    </button>
</div>

<div class="itemtemplate" data-win-control="WinJS.Binding.Template">
    <div class="item">
        <img class="item-image" src="#" data-win-bind="src: backgroundImage; alt: title" />
        <div class="item-overlay">
            <h4 class="item-title" data-win-bind="textContent: title"></h4>
            <h6 class="item-subtitle win-type-ellipsis"
                data-win-bind="textContent: subtitle"></h6>
        </div>
    </div>
</div>
```

Nothing new here, just `Template` controls with sprinkling of data-binding syntax.

As for the data itself (which you'll likely replace), this is defined in js/data.js as an in-memory array that feeds into a `Binding.List`. In the `sampleItems` array each item is populated with inline data or other variable values. Each item also has a `group` property that comes from the `sampleGroups` array. Unfortunately, this latter array has almost identical properties as the items array, which can be confusing. To help clarify that a bit, here's the complete property structure of an item:

```
{
    group : {
        key,
        title,
        subtitle,
        backgroundImage,
        description
    },
    title,
    subtitle,
    description,
    content,
    backgroundImage
}
```

As we saw with the ListView grouping sample earlier, the Grid App project template uses `createGrouped` to set up the data source. What's interesting to see here is that it sets up an initially empty list, creates the grouped projection (omitting the group sorter function), and then adds the items by using the list's `push` method:

```
var list = new WinJS.Binding.List();
var groupedItems = list.createGrouped(
    function groupKeySelector(item) { return item.group.key; },
    function groupDataSelector(item) { return item.group; }
);

generateSampleData().forEach(function (item) {
    list.push(item);
});
```

This clearly shows the dynamic nature of lists and ListView: you can add and remove items from the data source, and one-way binding will make sure the ListView is updated accordingly. In such cases you do *not* need to refresh the ListView's layout—that happens automatically. I say this because there's occasional confusion with the ListView's `forceLayout` method, which you only need to call, as the documentation states, "when making the ListView visible again after its `style.display` property had been set to 'none'." You'll find, in fact, that the Grid App code doesn't use this method at all.

In js/data.js there are also a number of other utility functions, such as `getItemsFromGroup`, which uses `List.createFiltered`. Other functions provide for cross-referencing between groups and items, as is needed to navigate between the items list, group details (where that page shows only items in that group), and item details. All of these functions are wrapped up in a namespace called `Data` at the bottom of js/data.js, so references to anything from this file are prefixed elsewhere with `Data.`.

And with that, I think you'll be able to understand everything that's going on in the Grid App project template to adapt it to your own needs. Just remember that all the sample data, like the default logo and splash screen images, are intended to be wholly replaced with real data that you obtain from other sources, like a file or some web API, and wrapped in a `List`. Some further guidance on this can be found in the Create a blog reader tutorial on the Windows Dev Center, and although the tutorial uses the Split App project template, there's enough in common with the Grid App project template that the discussion is applicable to both.

The Semantic Zoom Control

Because we've already loaded up the HTML ListView grouping and Semantic Zoom sample, and have completed our first look at the collection controls, now is a good time to check out another very interesting WinJS control: Semantic Zoom.

Semantic zoom lets users easily switch between two views of the same data: a zoomed-in view that provides details and a zoomed-out view that provides more summary-level information. The primary use case for semantic zoom is a long list of items that a user will likely get bored of panning all the way from one end to the other, no matter how fun it is to swipe the screen with a finger. With semantic zoom, you can zoom out to see headers, categories, or some other condensation of the data, and then tap on one of those items to zoom back into its section or group. The design guidance recommends having the zoomed-out view fit on one to three screenfuls at most, making it very easy to see and comprehend the whole data set.

Go ahead and try semantic zoom through scenario 2 of the ListView grouping and Semantic Zoom sample. To switch between the views, use pinch-zoom touch gestures, Ctrl+/Ctrl- keystrokes, Ctrl+mouse wheel, and/or the small zoom button that automatically appears in the lower-right corner of the control, as shown in Figure 7-5. When you zoom out, you'll see a display of the group headers, as also shown in the figure. For the dynamic experience, see Video 7-1 in the companion content, where I show the effects both at normal and slow speeds.

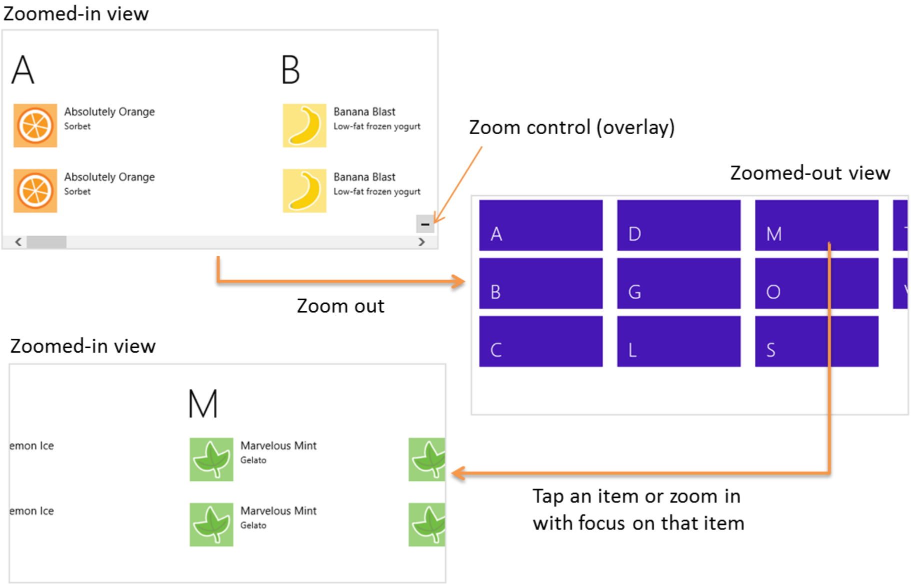


FIGURE 7-5 Semantic zoom between the two views in the ListView grouping and Semantic Zoom sample. The zoom control overlay appears only for the mouse (as does the scrollbar). See Video 7-1 for the dynamic effect.

The control itself is quite straightforward to use. In markup, declare a WinJS control using the WinJS.UI.SemanticZoom constructor. Within that element you then declare two (and only two) child elements: the first defining the zoomed-in view, and the second defining the zoomed-out view—always in that order. Here's how the sample does it with two ListView controls (plus the template used for the zoomed-out view; I'm showing the code in the modified sample included with this chapter's companion content):

```
<div id="semanticZoomTemplate" data-win-control="WinJS.Binding.Template" >
    <div class="semanticZoomItem">
        <h2 class="semanticZoomItem-Text" data-win-bind="innerText: groupTitle"></h2>
    </div>
</div>

<div id="semanticZoomDiv" data-win-control="WinJS.UI.SemanticZoom">
    <div id="zoomedInListView" class="win-selectionstylefilled"
        data-win-control="WinJS.UI.ListView"
        data-win-options="{ itemDataSource: myGroupedList.dataSource,
            itemTemplate: mediumListIconTextTemplate,
            groupDataSource: myGroupedList.groups.dataSource,
            groupHeaderTemplate: headerTemplate,
            selectionMode: 'none', tapBehavior: 'none', swipeBehavior: 'none'
            layout: { type: WinJS.UI.GridLayout } }">
    </div>

    <div id="zoomedOutListView" data-win-control="WinJS.UI.ListView"
        data-win-options="{ itemDataSource: myGroupedList.groups.dataSource,
            itemTemplate: semanticZoomTemplate,
            selectionMode: 'none', tapBehavior: 'invoke', swipeBehavior: 'none' }" >
    </div>
</div>
```

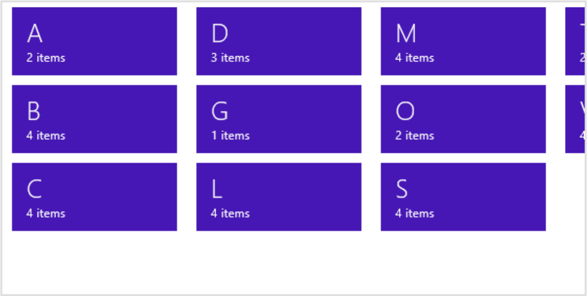
The first child, *zoomedInListView*, is just like the ListView for scenario 1 with group headers and items; the second, *zoomedOutListView*, uses the groups as items and renders them with a different template. The semantic zoom control *simply switches between the two views* in response to the appropriate input gestures. When the zoom changes, the semantic zoom control fires a zoomchanged event where the args.detail value in the handler is true when zoomed out, false when zoomed in. You might use this event to make certain app bar commands available for the different views, such as commands in the zoomed-out view to change sorting or filtering, which would then affect how the zoomed-in view is displayed. We'll see the app bar in Chapter 9.

The control has a few other properties, such as enableButton (a Boolean to control the visibility of the overlay button; default is true), locked (a Boolean that disables zooming in either direction and can be set dynamically to lock the current zoom state; default is false), and zoomedOut (a Boolean indicating if the control is zoomed out, so you can initialize it this way; default is false). There is also a forceLayout method that's used in the same case as the ListView's forceLayout: namely, when you remove a display: none style.

The zoomFactor property is an interesting one that determines how the control animates between the two views, something you can see more easily in the slowed-down segment of Video 7-1. The animation is a combination of scaling and cross-fading that makes the zoomed-out view appear to drop down from or rise above the plane of the control, depending on the direction of the switch, while the zoomed-in view appears to sink below or come up to that plane. To be specific, the zoomed-in view scales between 1 and zoomFactor while transparency goes between 1 and 0, and the zoomed-out view scales between 1/zoomFactor and 1 while transparency goes between 0 and 1. The default value for zoomFactor is 0.65, which creates a moderate effect. Lower values (minimum is 0.2) emphasize the effect, and higher values (maximum is 0.8) minimize it.

Where styling is concerned, you do most of what you need directly to the Semantic Zoom's children. However, to style the Semantic Zoom control itself you can override styles in win-semanticzoom (for the whole control) and win-semanticzoomactive (for the active view). The win-semanticzoombutton style also lets you style the zoom control button if needed.

It's important to understand that semantic zoom is intended to switch between two views of the same data and *not* to switch between completely different data sets (again see Guidelines and checklist for the Semantic Zoom control). Also, the control does not support nesting (that is, zooming out multiple times to different levels). Yet this doesn't mean you have to use the same kind of control for both views: the zoomed-in view might be a list, and the zoomed-out view could be a chart, a calendar, or any other visualization that makes sense. The zoomed-out view, in other words, is a great place to show summary data that would be otherwise difficult to derive from the zoomed-in view. For example, using the same changes we made to include the item count with the group data for scenario 1 (see "Quickstart #4" above), we can just add a little more to the zoomed-out item template (as done in the modified sample in this chapter's companion content):



A
2 items

B
4 items

C
4 items

D
3 items

G
1 items

L
4 items

M
4 items

O
2 items

S
4 items

The other thing you need to know is that the semantic zoom control does not work with arbitrary child elements. An exception about a missing zoomableView property will tell you this! Each child control must provide an implementation of the WinJS.UI.IZoomableView interface through a property called zoomableView. Of all built-in HTML and WinJS controls, only the ListView and Hub do this (see Chapter 8, "Layout and Views"), which is why you typically see semantic zoom in those contexts. However, you can certainly provide this interface on a custom control, where the object returned by the constructor should contain a zoomableView property, which is an object containing the IZoomableView methods. Among these methods are beginZoom and endZoom for obvious purposes, and getCurrentItem and setCurrentItem that enable the semantic zoom control to zoom in to the right group when it's tapped in the zoomed-out view.

For more details, check out the HTML SemanticZoom for custom controls sample, which also serves as another example of a custom control. The documentation also has a topic called SemanticZoom templates where you'll find a few additional template designs for zoomed-out views.

How Templates Work with Collection Controls

As we've looked over the collection controls, I've mentioned that you can use a function instead of a declarative template for properties like `template` (Repeater), `itemTemplate` (FlipView and ListView), and `groupHeaderTemplate` (ListView). This is an important capability because it allows you to dynamically render items in a collection individually, using its particular contents to customize its view, in contrast to a declarative template that will render each item identically. A rendering function also allows you to initialize item elements in ways that can't be done in the declarative form, such as building them up in asynchronous stages with delay-loaded images. This level of control provides many opportunities for performance optimization, a subject we'll return to at the end of this chapter after we've explored ListView thoroughly.

For the time being, it's helpful to understand exactly what's going on with declarative templates and how that relates to custom template functions. Once you see how they work, you will probably start dreaming up many uses for them!

Struggling for a template design? The documentation has two pages that contain a variety of pre-defined templates (both HTML and CSS). These are oriented around the ListView control but can be helpful anywhere a template is needed. The two pages are Item templates for grid layouts and Item templates for list layouts.

Referring to Templates

When you refer to a declarative template in the Repeater, FlipView, or ListView controls, what you're actually referring to is an *element*. You can use an element id as a shortcut because the app host creates variables with those names for the elements they identify. However, I don't recommend this approach, especially within page controls (which you'll probably use often). The first concern is that only one element can have a particular id, which means you'll get really strange behavior if you happen to render the page control twice in the same DOM.

The second concern is a timing issue. The element id variable that the app host provides isn't created until the chunk of HTML containing the element is added to the DOM. With page controls, `WinJS.UI.processAll` is called before this time, which means that element id variables for templates in that page won't yet be available. As a result, any controls that use an id for a template will either throw an exception or just show up blank. Both conditions are guaranteed to be terribly, terribly confusing.

To avoid this issue with a declarative template, place the template's name in its class attribute (and be sure to make that name unique and descriptive):

```
<div data-win-control="WinJS.Binding.Template"
   class="recipeItemTemplaterecipeItemTemplate" ...></div>
```

Then in your control declaration, use the select('<selector>') syntax in the options record, where <selector> is anything supported by element.querySelector:

```
<div data-win-control="WinJS.UI.ListView"
      data-win-options="{ itemTemplate: select('.recipemyItemTemplate') }"></div>
```

There's more to this, actually, than just a querySelector call. The select function within the options searches from the root of its containing page control. If no match is found, it looks for another page control higher in the DOM, then looks in there, continuing the process until a match is found. This lets you safely use two page controls at once that both contain the same class name for different templates, and each page will use the template that's most local.

You can also retrieve the template element using querySelector directly in code and assign the result to the appropriate property. This would typically be done in a page's ready function, as demonstrated in the Grid App project, and doing so avoids both concerns identified here because querySelector will be scoped to the page contents and will happen after UI.processAll.

Tip If you're uncertain about whether your data source is providing the right information to the template, just remove the template reference from the control's options. Without a template, the control will just output the text of the data source, allowing you to easily examine its contents.

Template Functions (Part 1): The Basics

Whenever you assign a Template object to one of the collection controls' template properties, those controls detect that it's an object and uses its render method when needed. However, the collection controls *also* detect if you instead assign a rendering function to those properties, which can be done both programmatically or declaratively. In other words, if you provide a function directly—which I will refer to simply as a *renderer*—it will be called in place of Template.render. This gives you complete control over what elements are generated for each individual data item as well as *how* and *when* they're created. (Warning! There be promises in your future!)

Again, we'll talk about rendering stages at the end of this chapter. For now, let's look at the core structure of a renderer that applies to the Repeater, FlipView, and ListView controls, examples of which you can find in the HTML ListView item templates, HTML ListView optimizing performance samples, and scenario 6 of the HTML FlipView control sample.

For starters, you can specify a renderer by name in data-win-options in all three controls for their respective template properties. That function must be marked for processing as discussed in Chapter 5 because it definitely participates in UI.processAll. Assigning a function in JavaScript, on the other hand, doesn't need the mark.

In its basic form, a renderer receives an item promise as its first argument and returns a promise that's fulfilled with the root element of the rendered template. Here's what that looks like in practice:

```
function basicRenderer(itemPromise) {
    return itemPromise.then(buildElement);
};

function buildElement (item) {
    var result = document.createElement("div");

    //Build up the item, typically using innerHTML
    return result;
}
```

The item comes as a promise because it might be delivered asynchronously from the data source; thus, we need to attach a completed handler to it. That completed handler, `buildElement` in the code above, then receives the item data and returns the rendered item's root element as a result.

The critical piece here is that the renderer is returning the promise from `itemPromise.then` (which is why we're *not* using `done`). Remember from Chapter 3 that `then` returns a separate promise that's fulfilled when the completed handler given to `then` itself returns. And the return value from that completed handler is what this second promise delivers as its own result. The simple structure shown here, then, very succinctly returns a promise that's fulfilled with the rendered item.

Why not just have the renderer return the element directly? Well, for one, it's possible that you might need to call other async APIs in the process of building the element—this especially comes into play when building up the element in stages by delay-loading images, as we'll see in "Template Functions (Part 2): Optimizing Item Rendering." Second, returning a promise allows the collection control that's using this renderer to chain the item promise and the element-building promise together. This is especially helpful when the item data is coming from a service or other potentially slow feed, and with page loading because it allows the control to cancel the promise chain if the page is scrolled away before those operations complete. In short, it's a good idea!

Just to show it, here's how we'd make a renderer directly usable from markup, as in `data-win-options = "{itemTemplate: Renderers.basic }"`:

```
WinJS.Namespace.define("Renderers", {
    basic: WinJS.Utilities.markSupportedForProcessing(function (itemPromise) {
        return itemPromise.then(buildElement);
    })
}
```

It's also common to just place the contents of a function like `buildElement` directly within the renderer itself, resulting in a more concise expression of the exact same structure:

```
function basicRenderer(itemPromise) {
    return itemPromise.then(function (item) {
        var result = document.createElement("div");

        //Build up the item, typically using innerHTML
        return result;
    })
};
```

Inside the element creation function (whether named or anonymous) you then build up the elements of the item along with the classes to style with CSS. As an example, here's the declarative template from scenario 1 of the HTML ListView item templates sample (html/scenario1.html):

```
<div id="regularListIconTextTemplate" data-win-control="WinJS.Binding.Template">
    <div class="regularListIconTextItem">
        <img src="#" class="regularListIconTextItem-Image" data-win-bind="src: picture" />
        <div class="regularListIconTextItem-Detail">
            <h4 data-win-bind="innerText: title"></h4>
            <h6 data-win-bind="innerText: text"></h6>
        </div>
    </div>
</div>
```

And here's the equivalent renderer, found in scenario 2 (js/scenario2.js):

```
var MyJSItemTemplate = WinJS.Utilities.markSupportedForProcessing(
    function MyJSItemTemplate(itemPromise) {
        return itemPromise.then(function (currentItem) {
            // Build ListView Item Container div
            var result = document.createElement("div");
            result.className = "regularListIconTextItem";
            result.style.overflow = "hidden";

            // Build icon div and insert into ListView Item
            var image = document.createElement("img");
            image.className = "regularListIconTextItem-Image";
            image.src = currentItem.data.picture;
            result.appendChild(image);

            // Build content body
            var body = document.createElement("div");
            body.className = "regularListIconTextItem-Detail";
            body.style.overflow = "hidden";

            // Display title
            var title = document.createElement("h4");
            title.innerText = currentItem.data.title;
            body.appendChild(title);

            // Display text
            var fulltext = document.createElement("h6");
            fulltext.innerText = currentItem.data.text;
            body.appendChild(fulltext);

            //put the body into the ListView Item
            result.appendChild(body);

            return result;
        });
    });
```

Note that within a renderer you always have the data item in hand, so you don't need to quibble over the details of declarative data binding and initializers: you can just directly use the needed properties from `item.data` and apply whatever conversions you require.

You might also notice that there are a lot of DOM API calls in this renderer for what is a fairly simple template. If you took a look at one of the compiled templates discussed in Chapter 6, you will have seen that it does most of its work by assigning a string to the root element's `innerHTML` property. Generally speaking, once you get to about four elements in your item rendering, setting `innerHTML` becomes faster than the equivalent `createElement` and `appendChild` calls. This is because the parser that's applied to `innerHTML` assignments is a highly optimized piece of C++ code in the app host and doesn't need to go through any other layers to get to the DOM API.

Such an optimization doesn't matter so much for a FlipView control whose items are rendered one at a time, or even a Repeater with a small or moderate number of items, but it becomes *very* important for a ListView with potentially thousands of items.

Taking this approach, the renderer above could also be written as follows with the same results:

```
var MyJSItemTemplate = WinJS.Utilities.markSupportedForProcessing(
    function MyJSItemTemplate(itemPromise) {
        return itemPromise.then(function (currentItem) {
            // Build ListView Item Container div
            var result = document.createElement("div");
            result.className = "regularListIconTextItem";
            result.style.overflow = "hidden";

            var data = currentItem.data;
            var str = "<img class='regularListIconTextItem-Image' src='" + data.picture + "'/>"
            str += "<div class='regularListIconTextItem-Detail' style='overflow:hidden'>";
            str += "<h4>" + data.title + "</h4>";
            str += "<h6>" + data.text + "</h6>";
            str += "</div>";

            result.innerHTML = str;
            return result;
        });
    });
```

Creating Templates from Data Sources in Blend

Blend for Visual Studio 2013 offers some shortcuts for creating templates for WinJS controls directly from a data source, where it inserts markup into your HTML file so you can go right into styling. The process is described here, and Video 7-2 provides a walk through.

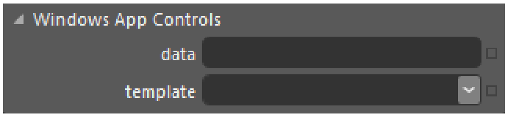
First create your data source in code as you normally would, making sure it's accessible from markup. In early stages of development you can use placeholder data, of course. As in the video, here's one that lives in a data.js file and is accessible via *Data.seasonalItems*:

```
var s1Title = "Item Title";
var s1Subtitle = "Item Sub Title";
var s1Subtext = "Quisque in porta lorem dolor amet sed consectetuer ising elit, ...";

var seasonalList = [
    { title: s1Title, subtitle: s1Subtitle, description: s1Subtext,
        image: "/images/assets/section2_1a.jpg" },
    { title: s1Title, subtitle: s1Subtitle, description: s1Subtext,
        image: "/images/assets/section2_1b.jpg" },
    { title: s1Title, subtitle: s1Subtitle, description: s1Subtext,
        image: "/images/assets/section2_1c.jpg" },
    { title: s1Title, subtitle: s1Subtitle, description: s1Subtext,
        image: "/images/assets/section2_1d.jpg" },
];

var seasonalItems = new WinJS.Binding.List(seasonalList);

WinJS.Namespace.define("Data", {
    seasonalItems: seasonalItems
});
```

In markup, or directly in Blend, insert a control wherever you need either through markup or by dragging a control from the Assets pane to the artboard. In the video I use a `Repeater` control, whose default markup is very simple:

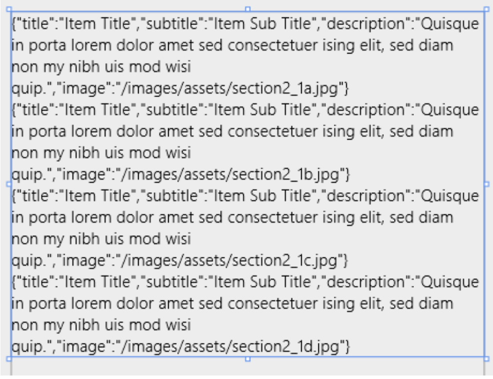
```
<div data-win-control="WinJS.UI.Repeater"></div>
```

With that control selected, the HTML Attributes pane (on the right) will have a section for Windows App Controls, which lists the relevant properties of the control. For the `Repeater` we have just *data* and *template*:



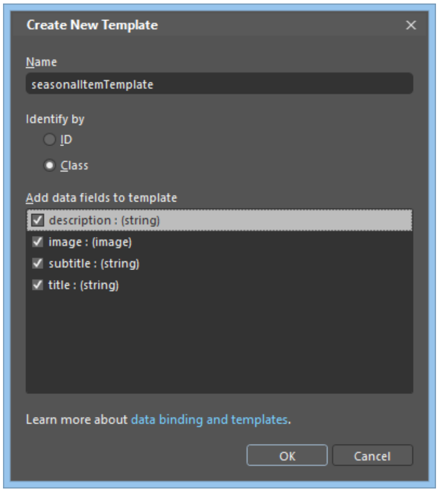
In the data property, enter the identifier for your data source (*Data.seasonalItems* in the example). This will create a `data-win-options` string in your markup, and you should see the untemplated results in the control:

```
<div data-win-control="WinJS.UI.Repeater" data-win-options="{data:Data.seasonalItems}" ></div>
```



{"title":"Item Title","subtitle":"Item Sub Title","description":"Quisque in porta lorem dolor amet sed consectetuer ising elit, sed diam non my nibh uis mod wisi quip.","image":"/images/assets/section2_1a.jpg"}
{"title":"Item Title","subtitle":"Item Sub Title","description":"Quisque in porta lorem dolor amet sed consectetuer ising elit, sed diam non my nibh uis mod wisi quip.","image":"/images/assets/section2_1b.jpg"}
{"title":"Item Title","subtitle":"Item Sub Title","description":"Quisque in porta lorem dolor amet sed consectetuer ising elit, sed diam non my nibh uis mod wisi quip.","image":"/images/assets/section2_1c.jpg"}
{"title":"Item Title","subtitle":"Item Sub Title","description":"Quisque in porta lorem dolor amet sed consectetuer ising elit, sed diam non my nibh uis mod wisi quip.","image":"/images/assets/section2_1d.jpg"}

Next, click the drop-down next to the *template* property and select <Create New Template…>, which brings up the dialog below wherein you'll conveniently see the members of your data source. Check those you need and give your template a name:



Create New Template

Name: seasonalItemTemplate

Identify by
○ ID
● Class

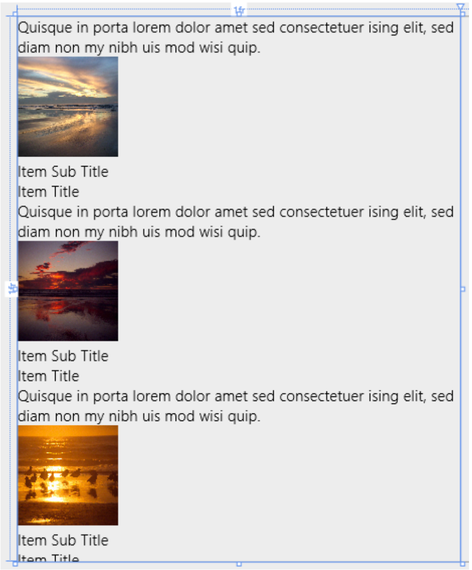
Add data fields to template
☑ description : (string)
☑ image : (image)
☑ subtitle : (string)
☑ title : (string)

Learn more about data binding and templates.

When you press OK, Blend will create unstyled markup in your HTML file and insert the appropriate reference in the control's options. Because I selected to identify the template with a class, the reference uses the `select` syntax:

```
<div class="seasonalItemTemplate" data-win-control="WinJS.Binding.Template">
    <div>
        <div data-win-bind="textContent:description"></div>
        <img data-win-bind="src:image" height="100" width="100">
        <div data-win-bind="textContent:subtitle"></div>
        <div data-win-bind="textContent:title"></div>
    </div>
</div>

<div data-win-control="WinJS.UI.Repeater" data-win-options="{data:Data.seasonalItems,
    template:select('.seasonalItemTemplate')}" ></div>
```

With this, you'll see the template being rendered (below left), at which point we can just reorder and style the elements as usual, resulting in much better output (below right):

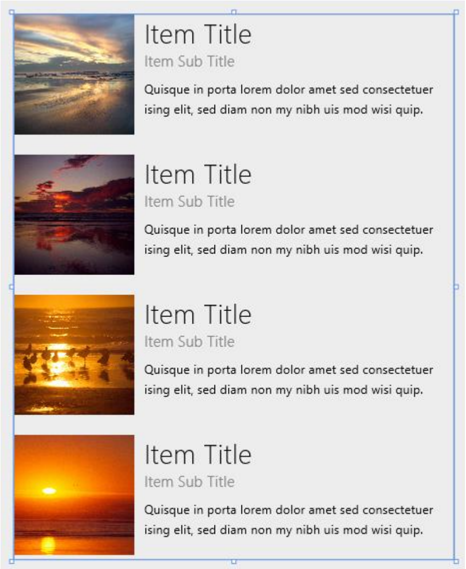


Quisque in porta lorem dolor amet sed consectetuer ising elit, sed diam non my nibh uis mod wisi quip.

Item Sub Title
Item Title
Quisque in porta lorem dolor amet sed consectetuer ising elit, sed diam non my nibh uis mod wisi quip.

Item Sub Title
Item Title
Quisque in porta lorem dolor amet sed consectetuer ising elit, sed diam non my nibh uis mod wisi quip.

Item Sub Title
Item Title



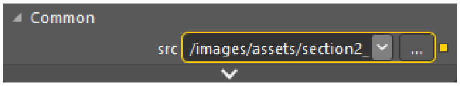
Item Title
Item Sub Title
Quisque in porta lorem dolor amet sed consectetuer ising elit, sed diam non my nibh uis mod wisi quip.

Item Title
Item Sub Title
Quisque in porta lorem dolor amet sed consectetuer ising elit, sed diam non my nibh uis mod wisi quip.

Item Title
Item Sub Title
Quisque in porta lorem dolor amet sed consectetuer ising elit, sed diam non my nibh uis mod wisi quip.

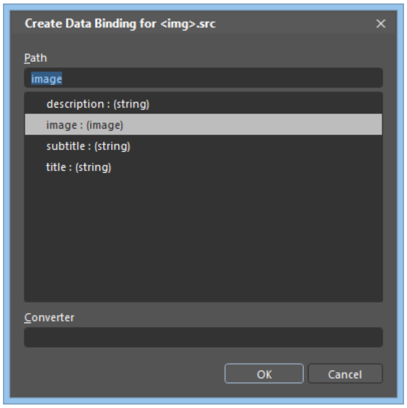
Item Title
Item Sub Title
Quisque in porta lorem dolor amet sed consectetuer ising elit, sed diam non my nibh uis mod wisi quip.

Blend also provides a shortcut to create and edit `data-win-bind` entries for individual properties, which works especially well inside a template. With a specific control selected (whether inside a template or anywhere else), click the little square to the right of a property in the HTML attributes pane, as shown below for the `img` element in the template we just created:



src /images/assets/section2_

The yellow highlight here means that the property is data-bound already. When you click the square, select Edit Data Binding... on the menu and you'll see the dialog below, where you can edit the `data-win-bind` entry or add one if none exists:



Create Data Binding for <img>.src

Path
image

description : (string)
image : (image)
subtitle : (string)
title : (string)

Converter

For a data context to appear here, note that WinJS.Binding.processAll must be called somewhere in your code for the element in question, or a suitable parent element. Otherwise no context will appear.

Repeater Features and Styling

In Quickstart #1 we've already covered the full extent of the Repeater control's options. Its data option refers to a Binding.List data source, and template can be used to refer to a template control declared elsewhere in your markup or a rendering function, if the template is not otherwise is declared as a child of the Repeater. Both of these options are, of course, available as read-write properties of the Repeater object at runtime, and changing either one will fire an itemsloaded event.[64]

Like all other WinJS controls, the read-only Repeater.element property contains the element in which the Repeater was declared; remember that the element also has a winControl property that will container the Repeater object. The only other property of the Repeater is length (read-only), which holds the number of items in the control.

Where methods are concerned, the Repeater has the usual roster of addEventListener, removeEventListener, dispatchEvent, and dispose. Its only custom method is elementFromIndex through which you can easily retrieve the root HTML element for one of the rendered children.

As a collection control, the Repeater is clearly affected by changes to its data source. As its children are bound to the source, they update automatically; when items are added to or removed from the source, the Repeater automatically adds or removes children. In all these cases, the Repeater fires various events:

Event trigger	Before DOM is updated	After DOM is updated
The List in the data property fires its reload event	itemsreloading	itemsreloaded
Individual item in the data source changed	itemchanging	itemchanged
Item added to the data source	iteminserting	iteminserted
Item removed from the data source	itemremoving	itemremoved
Item moved in the data source	itemmoving	itemmoved
data or template property changes	n/a	itemsloaded

To see the effect of adding and removing items, scenarios 4-6 of the HTML Repeater control sample all have Add Item and Remove Item buttons that just add and remove an item from the data source.

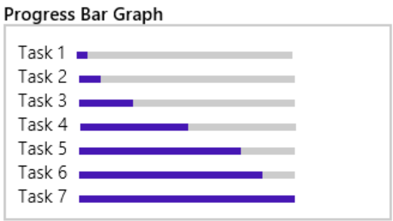
What's more interesting in these scenarios is their demonstration of basic styling (scenario 4), using the iteminserted and itemremoved events to trigger animations (scenario 5), and using nested

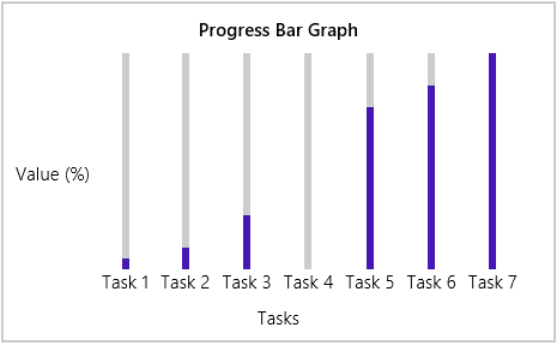
[64] A bug in the WinJS 2.0 repeater control throws an exception when you change the data or template properties from code when the template is not declared inline. For details and workaround, see my blog WinJS.UI.Repeater bug and workarounds (WinJS 2.0).

`WinJS.Binding.Template` controls (scenario 6).

With styling, the `Repeater` has no default styles because it has no visuals of its own: the repeater's element will have a `win-repeater` class added to it, but there are no styles for this class in the WinJS stylesheets. It's completely for your styling needs.

The repeater's children won't receive any styling classes of their own either, so it's up to you to define styles for the appropriate selectors. Scenarios 4, 5, and 6, for example, all have buttons to switch between Horizontal Layout and Vertical Layout, as shown below for scenarios 4 and 5.





In these cases the whole graph is a `div` (horizontal by default), where the `Repeater` is used inside for the list of tasks to create each bar (html/scenario4.html):

```
<div class="template" data-win-control="WinJS.Binding.Template">
    <div class="bar">
        <label class="label" data-win-bind="textContent: description"></label>
        <div class="barClip">
            <progress class="progress" data-win-bind="value: value" max="100"></progress>
        </div>
    </div>
</div>

<div class="graphArea horizontal">
    <div class="graphTitle win-type-large">Progress Bar Graph</div>
    <div class="graphData" data-win-control="WinJS.UI.Repeater"
        data-win-options="{data: Data.samples4, template: select('.template')}">
    </div>
    <div class="graphTaskAxis">Tasks</div>
    <div class="graphValueAxis">
        Value (%)
    </div>
</div>
```

The Vertical Layout button will remove the *horizontal* class and add a *vertical* class to the *graphArea* element, and the Horizontal Layout button does the opposite. In css/scenario4.css, you can see that the *graphArea* element is laid out with a CSS grid and all other elements like the bars with CSS flexboxes. The styling simply controls the placement of elements in the grid and the direction of the flexbox, all of which is specific to the elements that end up in the DOM and isn't affected by the Repeater itself.

Scenario 5 is the same as scenario 4 but adds small animation effects when items are added or removed, because the Repeater doesn't include any on its own (unlike the ListView). The effects are created using the WinJS Animations Library that we'll meet in Chapter 14, "Purposeful Animations." Here's a simplified version of the iteminserted handler (js/scenario5.js):

```
repeater.addEventListener("iteminserted", function (ev) {
    var a = WinJS.UI.Animation.createAddToListAnimation(ev.affectedElement);
    a.execute();
};
```

Scenario 6, finally, is very interesting because it shows the ability to nest Template controls, which in this case even nests the same template inside itself! Here the data source also has a nested structure (js/scenario6.js):

```
WinJS.Namespace.define("Data", {
    samples6: new WinJS.Binding.List([
        { value: 5, description: "Task 1" },
        {
            value: 50,
            description: "Task 2",
            subTasks: new WinJS.Binding.List([
                { value: 50, description: "Task 2: Part 1" },
                { value: 50, description: "Task 2: Part 2" }
            ])
        },
        { value: 25, description: "Task 3" },
        // ... (remaining data omitted)
    ])
});
```

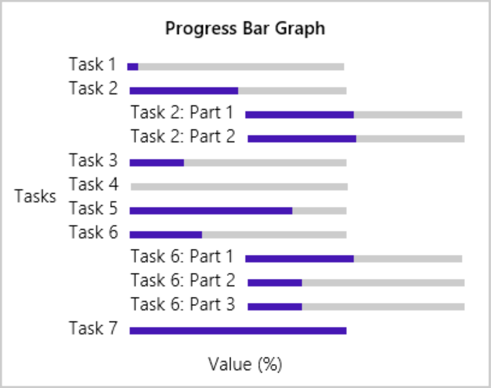
Take a close look now at the template in html/scenario6.html:

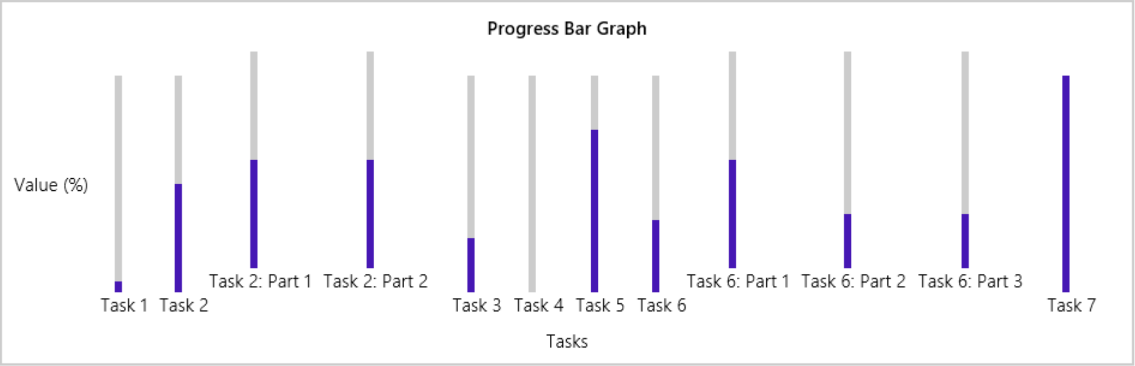
```
<div class="template" data-win-control="WinJS.Binding.Template">
    <div class="bar">
        <label class="label" data-win-bind="textContent: description"></label>
        <div class="barClip">
            <progress class="progress" data-win-bind="value: value" max="100"></progress>
        </div>
        <div class="subTasks" data-win-control="WinJS.UI.Repeater"
            data-win-options="{template: select('.template')}"
            data-win-bind="winControl.data: subTasks">
        </div>
    </div>
</div>
<div class="graphArea horizontal">
    <div class="graphTitle win-type-large">Progress Bar Graph</div>
    <div class="graphData" data-win-control="WinJS.UI.Repeater"
        data-win-options="{data: Data.samples6, template: select('.template')}">
    </div>
    <div class="graphTaskAxis">Tasks</div>
    <div class="graphValueAxis">
```

```
        Value (%)
    </div>
</div>
```

Notice how the first `Repeater` (at the bottom) refers to the full data source, so it generates the first level of the graph. For this repeater, each rendering of the template is bound to a top-level item in the data source. Within the template, then, the second-level `Repeater` (in the *subTasks* element) has its `data` option bound to a `subTasks` property of that first-level item, if it exists. Otherwise the `Repeater` will create an empty `WinJS.Binding.List` to work with so you can still add and remove items, but initially that repeater will be empty.

The initial output of scenario 6 is as follows, shown for both horizontal and vertical layouts:

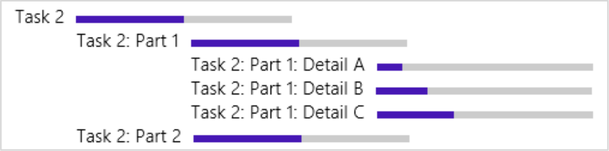




Nesting the second-level `Repeater` that refers to the same template is perfectly legal—a template control just renders its child elements when asked, and if that happens to contain a copy of itself, then you'll have some recursive rendering, but nothing that's going to confuse WinJS! In fact, the nested template structure will easily accommodate additional levels of data. If you modify "Task 2" in the data (js/scenario6.js) to add details to "Part 1":

```
{
    value: 50,
    description: "Task 2",
   subTasks: new WinJS.Binding.List([
      {
         value: 50,
         description: "Task 2: Part 1",
         subTasks: new WinJS.Binding.List([
            { value: 12, description: "Task 2: Part 1: Detail A" },
            { value: 24, description: "Task 2: Part 1: Detail B" },
            { value: 36, description: "Task 2: Part 1: Detail C" },
         ])
      },

      { value: 50, description: "Task 2: Part 2" }
   ])
},
```

you'll see this output for Task2 *without* any changes to the code or markup:



```
Task 2
      Task 2: Part 1
                 Task 2: Part 1: Detail A
                 Task 2: Part 1: Detail B
                 Task 2: Part 1: Detail C
      Task 2: Part 2
```

So very cool! Indeed, you can start to see that nested templates and repeaters can work very well to render highly structured data like an XML document or a piece of complex JSON. Of course, in many cases you'll want the rendered items to be interactive rather than static as we've seen here. In that case you can use a `WinJS.UI.ItemContainer` within a repeater (see sidebar), a `ListView` control, or possibly nested `ListView` controls.

Sidebar: Repeater + ItemContainer = Lightweight ListView

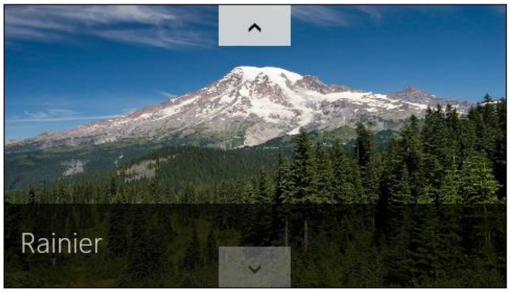
One reason that the `ItemContainer` and `Repeater` elements were created for WinJS 2.0 was that many developers wanted a UI that worked a lot like a `ListView`, but without all the `ListView` features. Thus instead of trying to make a `ListView` in which all those features could be disabled, the WinJS team instead pulled the per-item `ListView` behaviors into its own control, the `ItemContainer` (see Chapter 5), and then created the simple `Repeater` to make it easy to create such controls bound to items in a data source.

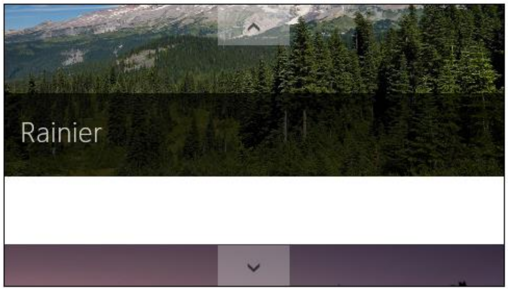
As a result, it's very straightforward in WinJS 2.0 to create a lightweight type of `ListView` where you have fully interactive items (select, swipe, drag, and invoke behaviors) but without any other layout policy or list-level interactivity (panning, incremental loading, keyboard navigation, reordering, etc.) In other words, the `Repeater` and `ItemContainer` controls are excellent building blocks for creating your own collection controls, which is typically a better choice than trying to bludgeon the ListView into something it wasn't made to do!

FlipView Features and Styling

The WinJS.UI.FlipView is a very capable and flexible control for any situation where you want to page through items in a data source one at a time. We saw the basics earlier, in "Quickstart #2," so let's now see the rest of its features through the other scenarios of the HTML FlipView control sample. (It's worth repeating that although this sample demonstrates the control's capabilities in a relatively small area, a FlipView can be any size. Refer again to Guidelines for FlipView controls for more.)

Scenario 2 of the sample ("Orientation and Item Spacing") demonstrates the control's orientation property. This determines the placement of the arrow controls: left and right (horizontal) or top and bottom (vertical) as shown below. It also determines the enter and exit animations of the items and whether the control uses the left/right or up/down arrow keys for keyboard navigation. This scenario also let you set the itemSpacing property (an integer), which determines the number of pixels between items when you swipe items using touch (below right). Its effect is not visible when using the keyboard or mouse to flip; to see it on nontouch devices, use touch emulation in the Visual Studio simulator to drag items partway between page stops.





Scenario 3 ("Using interactive content") shows the use of a renderer function instead of a declarative template, as we learned about in "How Templates Work with Collection Controls." Scenario 3 uses a renderer (a function called mytemplate in js/interactiveContent.js) to create a "table of contents" for the item in the data source marked with a "contentArray" type:

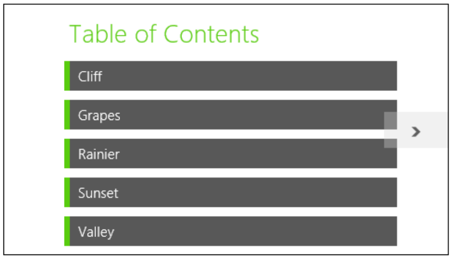


Table of Contents

Cliff

Grapes

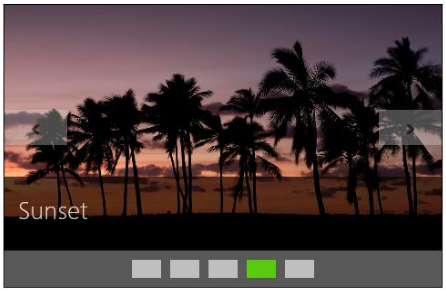
Rainier

Sunset

Valley

Scenario 3 also sets up a listener for click events on the TOC entries, the handler for which flips to the appropriate item by setting the FlipView's currentPage property. The picture items then have a back link to the TOC. See the clickHandler function in the code for both of these actions.

Scenario 4 ("Creating a context control") demonstrates adding a navigation control to each item:



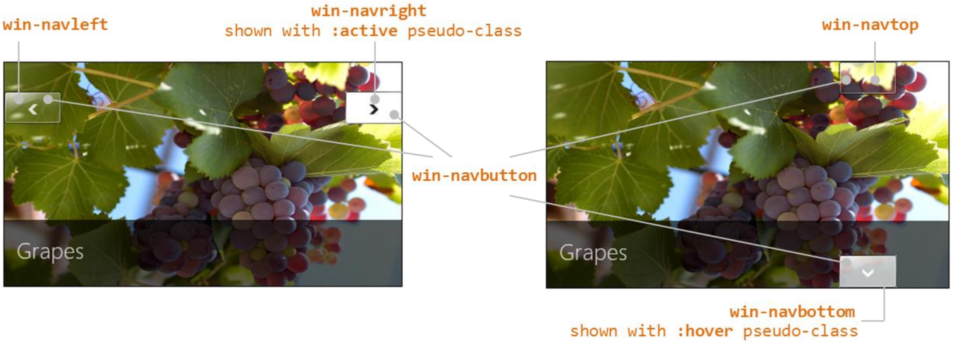
The items themselves are rendered using a declarative template, which in this case just contains a placeholder `div` called *ContextContainer* for the navigation control (html/context- Control.html):

```
<div>
    <div id="contextControl_FlipView" class="flipView" data-win-control="WinJS.UI.FlipView"
        data-win-options="{ itemDataSource: DefaultData.bindingList.dataSource,
            itemTemplate: contextControl_ItemTemplate }">
    </div>
    <div id="ContextContainer"></div>
</div>
```

When the control is initialized in the `processed` method of js/contextControl.js, the sample calls the FlipView's async `count` method. The completed handler, `countRetrieved`, then creates the navigation control using a row of styled radiobuttons. The `onpropertychange` handler for each radiobutton then sets the FlipView's `currentPage` property.

Scenario 4 also sets up listeners for the FlipView's `pageselected` and `pagevisibilitychanged` events. The first is used to update the navigation radiobuttons when the user flips between pages. The other is used to prevent clicks on the navigation control during a flip. (The event occurs when an item changes visibility and is fired twice per flip, once for the previous item, and again for the new one.)

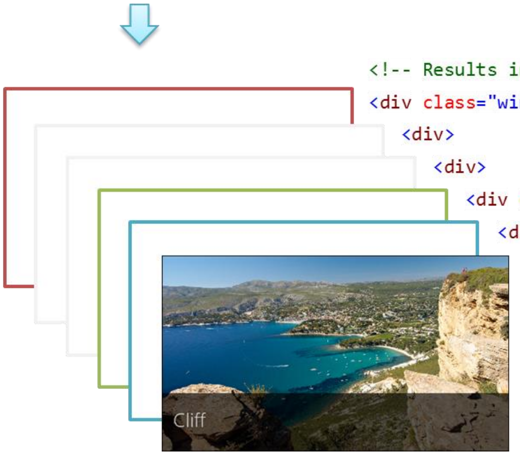
Scenario 5 ("Styling Navigation Buttons") demonstrates the styling features of the FlipView, which involves various `win-*` styles and pseudo-classes as shown here (also documented on [Styling the FlipView and its items]()):



If you were to provide your own navigation buttons in the template (wired to the next and previous methods), hide the default by adding display: none to the <control selector> .win-navbutton style rule.

As we saw with the ItemContainer in Chapter 5, the FlipView creates some intermediate div elements between the root where you declare the FlipView and where the template gets rendered. These layers are classed with win-flipview (the root element), win-surface (the panning region), and win-item (where a template is rendered); you'll typically use these to style margins, padding, etc.:

```
<!-- Markup in your HTML file -->
<div data-win-control="WinJS.Binding.Template">
    [your template content]
</div>

<div class="[your classes]" data-win-control="WinJS.UI.FlipView">
</div>
```



```
<!-- Results in the DOM -->
<div class="win-flipview [your classes]">
    <div>
        <div>
            <div class="win-surface">
                <div class="win-item">
                    [your template content]
```

The win-surface class is where you style a different background color for the gap created with the itemSpacing property. To demonstrate this, scenario 5 of the modified sample in this chapter's companion content sets itemSpacing to 50 (html/stylingButtons.html) and adds the following CSS (css/stylingButtons.css):

```
#stylingButtons_FlipView .win-surface {
    background-color: #FFE0E0;
}
```

Finally, there are a few other methods and events for the FlipView that aren't used in the sample, so here's a quick rundown:

- pageCompleted   An event raised when flipping to a new item is fully complete (that is, the new item has been rendered). In contrast, the aforementioned pageselected event will fire when a *placeholder* item (not fully rendered) has been animated in. See "Template Functions (Part 2): Optimizing Item Rendering" at the end of this chapter.

- `datasourcecountchanged` An event raised for obvious purpose, which something like scenario 4 would use to refresh the navigation control if items could be added or removed from the data source.
- `next` and `previous` Methods to flip between items (like `currentPage`), which would be useful if you provided your own navigation buttons.
- `forceLayout` A method to call specifically when you make a FlipView visible by removing a `display: none` style.
- `setCustomAnimations` A method that allows you to control the animations used when flipping forward, flipping backward, and jumping to a random item.

For details on all of these, refer to the `WinJS.UI.FlipView` documentation.

Collection Control Data Sources

Before we get into the details of the ListView, it's appropriate to take a little detour into the subject of data sources as they pertain specifically to collection controls. In all the examples we've seen thus far, we've been using synchronous, in-memory data sources built with `WinJS.Binding.List`, which works well up to about 2000–3000 total items. But what if you have a different kind of source, perhaps one that works asynchronously (doing data retrieval off the UI thread)? It certainly doesn't make sense to pull everything into memory first, and especially not with data sources that have tens or hundreds of thousands of items. For such sources we need a solution that's scalable and can be virtualized.

For these reasons, collection controls like the FlipView and ListView don't work directly against a `List` (like the Repeater)—they instead work against an abstract data source defined by a set of interfaces. That abstraction allows you to implement any kind of data source you want and plug it into the controls. Those sources could be built on top of an in-memory object, like the `List`, data from a service, object structures from other WinRT APIs or WinRT components, and really anything else. The abstraction is simply a way to shape any kind of data into something that the controls can use.

In this section, we'll first look at those interfaces and their relationships. Then we'll see two helpers that WinJS provides: the `WinJS.UI.StorageDataSource` object, which works with the file system, and the `WinJS.UI.VirtualizedDataSource`, which serves as a base class for custom data sources.

Tip If you define a data source (like a `Binding.List`) within a page control, the declarative syntax `select('.pagecontrol').winControl.myData.dataSource` can be used to assign that source to an `itemDataSource` or `groupDataSource` property. Here, *pageControl* class is a class you'd add to your page control's root element. That element's `winControl` property gets you to the object defined with `WinJS.UI.Pages.define`, wherein the `myData` property would return the `Binding.List`.

Super performance tip I said this in Chapter 6, but it's very much worth saying again. If you're enumerating folder contents to display images in a collection control (what is generally called a *gallery experience*), use metadata to represent file content instead of using file I/O to read that file content. *Always* use Windows.Storage.StorageFile.getThumbnailAsync or getScaledImageAsThumbnailAsync (or the same methods on the StorageFolder object) to retrieve a small image for your data source, which can be passed to URL.createObjectURL and the result assigned to an img element. (The StorageDataSource.getThumbnail method is a helper for this too, as we'll see later.) This is much faster and uses less memory, as the API draws from the thumbnails cache instead of touching the file contents (as happens if you call URL.createObjectURL on a StorageFile). See the File and folder thumbnail sample for demonstrations. You might also be interested in watching Marc Wautier's What's new for working with Files session from //build 2013, where he talks about the improved performance with thumbnails, especially where working with OneDrive is concerned.

The Structure of Data Sources (Interfaces Aplenty!)

When you assign a data source to the itemDataSource property of the FlipView and ListView controls, or the ListView's groupDataSource property, that object is expected to implement the methods of the interface called IListDataSource.[65] Everything these controls do with their data sources happens exclusively through these methods and those of several companion interfaces: IListBinding, IItemPromise, and IItem. The fact that the data source is abstracted behind these interfaces, and that most of the methods involved are asynchronous, means that the data source can work with any kind of data, whether local or remote. On the other side of the picture, collection controls that want to receive change notifications create a handler object with the IListNotificationHandler methods. Together, all these interfaces provide the necessary means to asynchronously manipulate items in the data source and to bind to those items (that is, enumerate them and register for change notifications).

Tip It is entirely expected and recommended that custom collection controls work with these same interfaces to support a variety of data sources. There's little point in reinventing it all! Also take note of the WinJS.UI.computeDataSourceGroups helper that adds grouping information to a data source give an IListDataSource.

For in-memory source, the WinJS.Binding.List conveniently supplies everything that's needed through its dataSource property, as we've seen. This is why you'll always see a reference to List.dataSource with the FlipView and ListView: they know *nothing* about the List itself. As mentioned earlier in this chapter, if you see an exception about a missing method called *createListBinding*, it's probably telling you that you're assigning a List instead of a List.dataSource to one of the FlipView or ListView source properties.

The createListBinding method, in fact, is the one through which a source object (with IListDataSource) provides the binding capabilities expressed through IListBinding, and how a

[65] See "Windows.Foundation.Collection Types" in Chapter 6 for an overview of what interfaces are in JavaScript. For custom sources, you typically get the methods from the VirualizedDataSource class.

collection control registers an object, with the ILIstNotificationHandler methods, to listen for data changes. The general relationships between these and the source are illustrated in Figure 7-6.

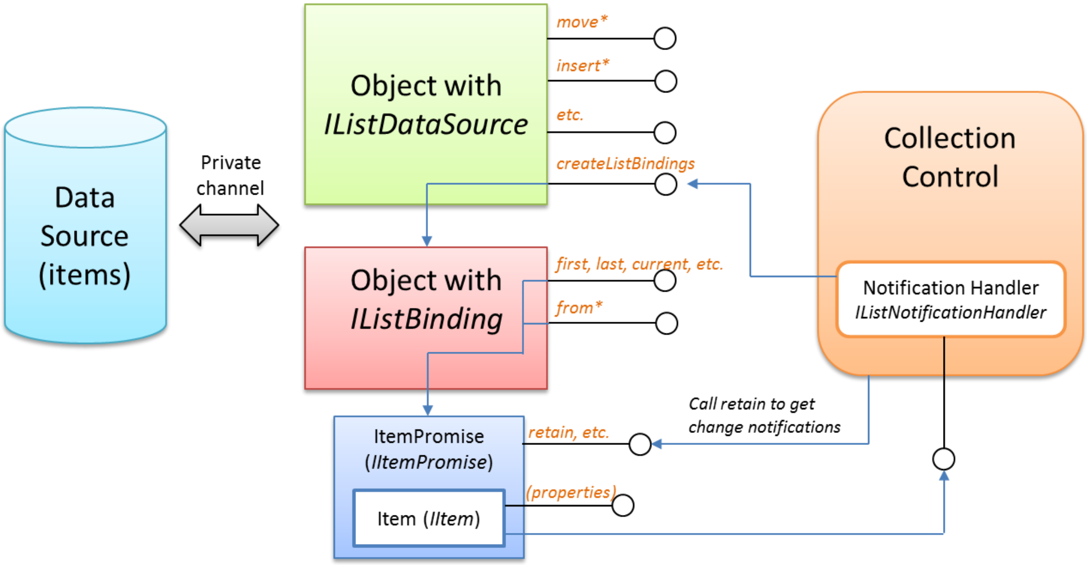


FIGURE 7-6 The general relationships between a data source, the IListDataSource and IListBinding interfaces, and a collection control like the ListView that uses that data source through those interfaces and provides a notification handler with IListNotificationHandler methods.

Looking at Figure 7-6, the relationship between the data source and the objects that represent it is private and very much depends on the nature of the data, but all that's hidden from the controls on the right. You can also see that items are represented by promises that have the usual then/done methods, of course, but also sport a few others as found in IItemPromise. What these promises deliver are then item objects with the IItem methods. It's a lot to keep track of, so let's break it down.

The members of IListDataSource, in the table below, primarily deal with manipulations of items in the source. If you've worked with databases, you'll recognize that this interface expressed the typical set of create, update, and delete operations. The object behind the interface can implement these however it wants. The List.dataSource object, for example, mostly uses List methods like move, splice, push, setAt, etc. But be mindful again that all the IListDataSource methods that affect item content (change, getCount, item*, insert*, move*, and remove) are asynchronous and return item promises (IItemPromise) for the results in question.

Member (methods unless noted)	Description
createListBindings	Returns an object that implements IListBinding methods that allow for enumeration of data items and change notifications as needed for data binding.
beginEdits, endEdits	Changes made between a call to beginEdit and endEdit will defer notification (through the statusChanged event) until endEdit is called.
itemFromKey, itemFromIndex, itemFromDescription	Item lookup methods using a key (string), index (number), or a description (object).
change	Updates an item with new data.

insertAfter, insertAtEnd, insertAtStart, insertBefore	Inserts a new item in the source relative to an existing one or at the beginning or end of the source.
moveAfter, moveBefore, moveToEnd, moveToStart	Moves an existing item elsewhere in the source, if supported (read-only and non-orderable collections would not, for instance).
remove	Deletes an existing item.
statusChanged (event) plus addEventListener, et. al.	Raised when the data source has changed in some way; a data source is not required to implement this (the List.dataSource does not).
getCount	Returns a promise for the total number of items in the data source. This can be an estimated size as it's used by collection controls to do virtualization.
invalidateAll	Instructs a data source to clear and reset any caches it might be maintaining.

You'll also find that all operations that reference existing items, like change, move*, insertAfter, and so on, reference items by keys rather than indicies. This was a conscious design decision because indices can be quite volatile in different data sources. Keys, on the other hand, which a source typically assigns to uniquely identify an item, are very stable.

The members of IListBinding, for their part, deal with enumeration of items in the source. All of its methods are asynchronous and return promises with IItemPromise methods:

Member (methods unless noted)	**Description**
current	Returns a promise the current item in the enumeration.
jumpToItem	Makes a given item the current one and returns a promise for it.
first, last, previous, next	Returns a promise for an item in the enumeration, relative to the whole or to the current item.
fromKey, fromIndex, fromDescription	Returns a promise for an item from a key (string), index (number), or a description (object). Each of these methods make the returned item the current one.
releaseItem, release	Stops change requests for an item or all items if a notification handler was provided to IListDataSource.createListBinding.

To listen to change notifications on any particular item, the control that's using the data source must first provide an object with IListNotificationHandler to the createListBinding method, as shown earlier in Figure 7-6. This step provides the handler to the source, but doesn't actually start notifications. That step happens when the control requests an item (through whatever other method) to get an item promise and calls IItemPromise.retain. When the control is done listening, it retrieves the item from the promise (IItemPromise.then or done) and calls that item's release.

This per-item notification is done to support virtualization. If you have a data source with potentially thousands of items, but the control that's using that source only displays a few dozen at a time, then there's no reason for the source to keep every item in memory: the control would instead call retain for those items that are visible (or about to be visible), and later call release when they're well out of view. This allows the data source to manage its own memory efficiently.

That's the whole relationship in a nutshell. For the most part, you won't have to deal with all these details. For custom data sources, the VirtualizedDataSource provides a core implementation, as we'll see later. And when your data source is a List, you can just manipulate that source through the List methods. At the same time, you can also make changes through the IListDataSource and IListBinding methods directly. This is necessary when the data source isn't a List, and becomes important if you're creating a custom collection control that supports arbitrary sources.

An example of this can be found in the HTML ListView working with data sources sample. Scenarios 2 and 3 use a ListView to displays letter tiles like those found in many word games:

A series of buttons lets you shuffle the list, add a tile, remove a tile, or swap tiles. In scenario 2, these manipulations are done through the `Binding.List` to which the ListView is bound. Adding a tile, for example, just happens with `List.push` (js/scenario2.js):

```
function addTile() {
     var tile = generateTile();
    lettersList.push(tile);
}
```

Scenario 3, on the other hand, does all the same stuff, still using a `List`, but performs manipulations through `IListDataSource` methods. Adding a tile, for example, happens through `insertAtEnd` (js/scenario3.js):[66]

```
function addTile() {
    var ds = document.getElementById("listView3").winControl.itemDataSource;
    var tile = generateTile();
    ds.insertAtEnd(null, tile);
}
```

The shuffling operation is more involved as it must first call `createListBinding` to get the `IListBinding` methods through which it can enumerate the collection. This results in an array of item promises, which can be fulfilled with `WinJS.Promise.join`. It then calls the source's `beginEdits` to batch changes, randomly moves items to the top of the list with `moveToStart`, and then calls `endEdits` to wrap up. The code is a bit long, so you can look at it in js/scenario3.js.

A FlipView Using the Pictures Library

For everything we've seen in the FlipView sample already, it really begs for the ability to do something completely obvious: flip through pictures in a folder. How might we implement something like that? We already have an item template containing an `img` tag, so perhaps we just need some URIs for those files. We could make an array of these using the `Windows.Storage.KnownFolders.picturesLibrary` folder and that library's `StorageFolder.getFilesAsync` method (declaring the *Pictures Library* capability in the manifest, of course!). This would give us a bunch of `StorageFile` objects for which we could call `URL.createObjectURL`. We could store those URIs in an array and create a `List`, whose `dataSource` property we can assign to the FlipView's `itemDataSource`:

```
var myFlipView = document.getElementById("pictures_FlipView").winControl;
```

[66] The sample calls `beginEdits` and `endEdits` here, which is not necessary for a single addition or deletion. The methods *are* effective when swapping items or shuffling the whole set where multiple changes are involved.

```
Windows.Storage.KnownFolders.picturesLibrary.getFilesAsync()
    .done(function (files) {
        var pixURLs = [];

        files.forEach(function (item) {
            var url = URL.createObjectURL(item, {oneTimeOnly: true });

            pixURLs.push({type: "item", title: item.name, picture: url });
        });

        var pixList = new WinJS.Binding.List(pixURLs);
        myFlipView.itemDataSource = pixList.dataSource;
    });
```

Although this approach works, it can consume quite a bit of memory with a larger number of high-resolution pictures because each picture has to be fully loaded into memory. We can alleviate the memory requirements by loading thumbnails instead of the full image, but there's still the significant drawback that the images are stretched or compressed to fit into the FlipView without any concern for aspect ratio, and this produces lousy results unless every image is the same size (highly unlikely!).

A better approach is to use the WinJS.UI.StorageDataSource object, a demonstration of which is found in scenario 8 of the modified HTML FlipView sample in this chapter's companion content. Another example can be found in the StorageDataSource and GetVirtualizedFilesVector sample, which creates a ListView over the Pictures folder as well.

StorageDataSource provides all the necessary interfaces, of course, and works directly with the file system metadata as a data source instead of an in-memory array. It provides automatic change detection, so if you add files to a folder or modify existing ones, it will fire off the necessary change notifications so that bound controls can update themselves. Furthermore, it provides automatic scaling and cropping services for thumbnails—pictures, for example, are auto-cropped with a 0.7 height to width ratio, using the upper part of a picture where the subject is usually found. Videos automatically receive an adornment to distinguish them from pictures, and also use a 0.7 height to width ratio. Music and documents, on the other hand, automatically use a square representation, as befits album a rt and icons. You can also customize this behavior through the ThumbnailMode options that you pass to the StorageDataSource constructor.

The StorageDataSource works with something called a *query*, which we'll learn about in Chapter 11, "The Story of State, Part 2."[67] Fortunately, WinJS lets you shortcut the process for media libraries and just pass in a string like "Pictures" (js/scenario8.js):

```
myFlipView.itemDataSource = new WinJS.UI.StorageDataSource("Pictures");
```

[67] To be specific, the first argument to the StorageDataSource constructor is a query object that comes from the Windows.Storage.Search API. Queries feed into the powerful StorageFolder.createFileQueryWithOptions function and are ways to enumerate files in a folder along with metadata like album covers, track details, and thumbnails that are cropped to maintain the aspect ratio. Shortcuts like "Pictures", "Music", and "Videos" (which require the associated capability in the manifest) just create typical queries for those libraries.

This will create a StorageDataSource on top of the contents of the Pictures library (assuming you've declared the capability, of course).

The caveat with StorageDataSource is that it doesn't directly support one-way binding: you'll get an exception if you try to refer to item properties directly in a declarative template. To work around this, you have to explicitly use WinJS.Binding.oneTime as the initializer function for each property (or set it as the default in Binding.processAll). This template is in html/scenario8.html:

```
<div id="pictures_ItemTemplate" data-win-control="WinJS.Binding.Template">
    <div class="overlaidItemTemplate">
        <img class="image" data-win-bind="src: thumbnail InitFunctions.thumbURL;
            alt: name WinJS.Binding.oneTime" />
        <div class="overlay">
            <h2 class="ItemTitle" data-win-bind="innerText: name WinJS.Binding.oneTime"></h2>
        </div>
    </div>
</div>
```

In the case of the img.src property, the query gives us items of type Windows.Storage.BulkAccess.FileInformation (the s variable in the code below), which contains a thumbnail image, not a URI. To convert that image data into a URI, we need to use our own binding initializer:

```
WinJS.Namespace.define("InitFunctions", {
    thumbURL: WinJS.Binding.initializer(function (s, sp, d, dp) {
        var thumb = WinJS.Utilities.getMember(sp.join("."), s);

        if (thumb) {
            var lastProp = dp.pop();
            var target = dp.length ? WinJS.Utilities.getMember(dp.join("."), d) : d;
            dp.push(lastProp);
            target[lastProp] = URL.createObjectURL(thumb, { oneTimeOnly: true });
        }
    })
});
```

Note that thumbnails aren't always immediately available in the FileInformation object, which is why we have to verify that we actually have one before creating a URI for it. This means that quickly flipping through the images might show some blanks. To solve this particular issue, we can listen for the FileInformation.onthumbnailupdated event and update the item at that time. The best way to accomplish this is to use the StorageDataSource.loadThumbnail helper, which makes sure to call removeEventListener for this WinRT event. (See "WinRT Events and removeEventListener" in Chapter 3.) You can use this method within a binding initializer, as demonstrated in scenario 1 of the aforementioned StorageDataSource and GetVirtualizedFilesVector sample, or within a rendering function that takes the place of the declarative template. Scenario 9 of the modified FlipView sample does this.

Custom Data Sources and WinJS.UI.VirtualizedDataSource

Creating a custom data source means, when all is said and done, to provide the implementations of all the interfaces we saw earlier in "The Structure of Data Sources (Interfaces Aplenty!)." Clearly, that would be a lot of coding work and would involve change detection and lots of optimization work like caching and virtualization. To make the whole job easier, WinJS provides the `WinJS.UI.VirtualizedDataSource` class that does most of the heavy-lifting and provides all the `IListDataSource` and `IListBinding` methods. The piece you provide is a stateless *adapter*—an object that implements some or all of the methods of the `IListDataAdapter` interface—to customize or adapt the `VirtualizedDataSource` to your particular data store.

Tip The Using ListView topic in the documentation contains a number of performance tips for custom data sources that aren't covered here. As it notes, the `IListDataAdapter` interface and the `VirtualizedDataSource` object are the best means for creating an asynchronous data source.

Walkthrough For more on the concepts around custom data sources and a walkthrough of using an adapter, watch Sam Spencer's talk from //build 2011, Build data-driven collection and list apps using ListView, specifically between 31:16 and 48:48. It's interesting to note that he builds a data source on top of WCF data services (OData) talking to a SQL Server database hosted on Windows Azure. An adapter is also applicable to other cloud services, data-related WinRT APIs, and local databases.

If you take a look at `IListDataAdapter`, you'll see that it has many of the same methods as `IListDataSource`, because it handles many of the same functions. Be sure not to confuse the two, however, because there are differences and even methods with the same names might have different arguments.

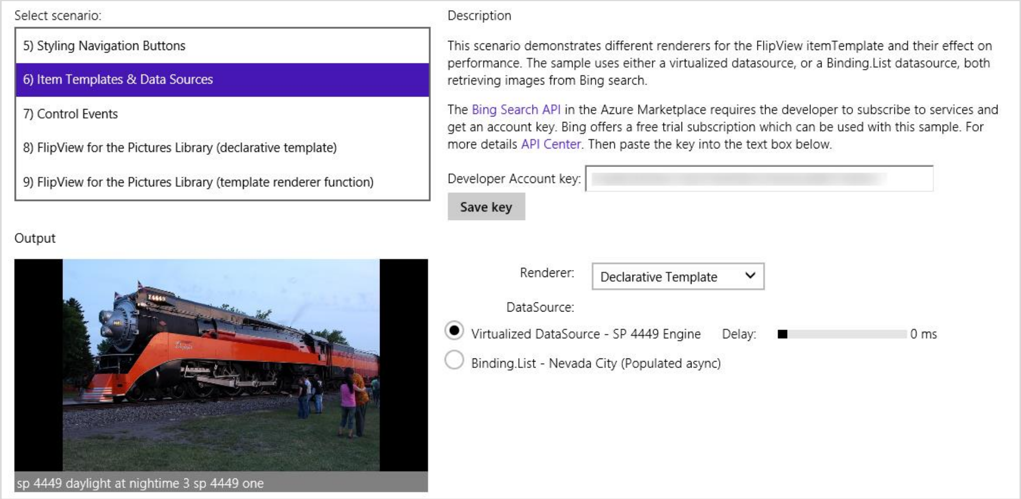
Member (methods unless noted)	Description
`change`	Asynchronously modifies an item.
`getCount`	Returns a promise that's fulfilled with an estimated count of the items in the data source. As with `IListDataSource`, the count does not have to be accurate: it's primarily used to help a collection control manage its paging or virtualization.
`insertAfter`, `insertAtEnd`, `insertAtStart`, `insertBefore`	Adds items to the source. All the methods return a promise for the item added.
`itemsFromDescription`, `itemsFromEnd`, `itemsFromIndex`, `itemsFromKey`, `itemsFromStart`	Return a promise that's fulfilled with one or more items depending on location (start, end), index (number), key (a string), or description (object). All these methods can be asked to retrieve more than one item (e.g., some number before and after the primary item) to efficiently support paging and virtualization. The group of items delivered by the promise is specifically an object with the `IFetchResult` interface.
`moveAfter`, `moveBefore`, `moveToEnd`, `moveToStart`	Moves an item in the source, returning a promise for the item.
`remove`	Deletes an item from the source, returning a promise for the item.
`setNotificationHandler`	Registers a notification handler. In this case the handler should implement the `IListDataNotificationHandler` interface methods.
`compareByIdentity` (property)	Indicates whether the object's identity is used to detect changes as opposed to its value.
`itemSignature`	Returns a string representation of an item to use for comparisons.

When you look at these methods and think about them in relation to different kinds of data sources, it should be clear that certain methods won't be needed. A read-only source, for example, has no need for the `insert*` or `move*` methods. Fortunately, the adapter interface and the `VirtualizedDataSource` are designed to be flexible, allowing you to implement only those adapter methods as described in the following table.

Source capability	**Applicable methods**
read-only, **index**-based	`getCount`, `itemsFromIndex`
read-only, **key**-based	`getCount`, `itemsFromKey`, `itemsFromStart`, `itemsFromEnd`
read-write **without** ordering	Above methods plus `change`, `remove`, `insertAtEnd`
read-write **with** ordering	Above methods plus `insertAtStart`, `insertBefore`, `insertAfter`, `insertAtEnd`, `moveAfter`, `moveBefore`, `moveToEnd`, and `moveToStart`
change notifications	Add `setNotificationHandler`

Now for some examples! First, if you've been keeping score, you might have noticed that we've talked about every scenario in the HTML FlipView sample (the modified one) except for scenario 6. That's because this particular scenario implements a custom data source to work with images from Bing Search. This is where you'll need the Bing Search API key that I mentioned at the beginning of this chapter, so if you didn't get a key yet, do that now.

Scenario 6 is shown in Figure 7-7, using the modified sample in which I've changed the default search strings from Seattle and New York to a few things closer to my heart (my son's favorite train and the community where I live). You can do a lot in this scenario. The controls let you select from different forms of item rendering, which we'll be talking about later in this chapter, and to select either a virtualized online data source with a configurable delay time—as we'll be talking about here—or an online source that's incrementally loaded into a `Binding.List`.



Select scenario:

5) Styling Navigation Buttons
6) Item Templates & Data Sources
7) Control Events
8) FlipView for the Pictures Library (declarative template)
9) FlipView for the Pictures Library (template renderer function)

Description

This scenario demonstrates different renderers for the FlipView itemTemplate and their effect on performance. The sample uses either a virtualized datasource, or a Binding.List datasource, both retrieving images from Bing search.

The Bing Search API in the Azure Marketplace requires the developer to subscribe to services and get an account key. Bing offers a free trial subscription which can be used with this sample. For more details API Center. Then paste the key into the text box below.

Developer Account key: [redacted]

Save key

Output

sp 4449 daylight at nightime 3 sp 4449 one

Renderer: Declarative Template

DataSource:

● Virtualized DataSource - SP 4449 Engine Delay: 0 ms

○ Binding.List - Nevada City (Populated async)

FIGURE 7-7 Scenario 6 of the modified HTML FlipView sample in the companion content (cropped).

When you select the Virtualized DataSource option, as shown in the figure, the code assigns the FlipView and instance of a custom class called `bingImageSearchDataSource`:

```
dataSource = new bingImageSearchDataSource.datasource(devkey, "SP4449", delay);
```

where *devkey* is the Bing Search API account key, "SP4449" is the search string, and *delay* is from the Delay range control (to simulate network latency). The class itself is implemented in js/bingImageSearchDataSource.js, deriving from WinJS.UI.VirtualizedDataSource:

```
WinJS.Namespace.define("bingImageSearchDataSource", {
    datasource: WinJS.Class.derive(WinJS.UI.VirtualizedDataSource,
        function (devkey, query, delay) {
            this._baseDataSourceConstructor(new bingImageSearchDataAdapter(devkey, query, delay));
        })
});
```

The call to this._baseDataSourceConstructor is the same as calling new on the Virtiualized-DataSource. However you do it, the argument to this constructor is your adapter object. In this case it's an instance of the bingImageSearchDataAdapter class, which is the bulk of the code in js/bingImageSearchDataSource.js. Because we have a read-only, index-based, nonorderable source without change notification, the adapter implements only the getCount and itemFromIndex methods. Both of these make HTTP requests to Bing to retrieve the necessary data.

In the earlier table, I mentioned that getCount doesn't have to be accurate—it just helps a collection control plan for virtualization; the sample, in fact, always returns 100 if it successfully pings the server with a request for 10. Typically, once itemFromIndex has been called and you start pulling down real data, you can make the count increasingly accurate.

Speaking of counts, the itemFromIndex method is interesting because it's not just asked to retrieve one item: it's might also be asked for some number of items on either side. This specifically supports pre-caching of items near the current point at which a control is displaying a collection because those are the most likely items that the user will navigate to next. By asking the data source for more than one at a time—especially when making an HTTP request—we cut down network traffic and deliver a smoother user experience.

As an optimization, these extra item requests are not hard rules. Depending on the service, the adapter can decide to deliver however many items is best for the service. For example, if itemsFromIndex is asked for 20 items before and after the primary item, but the service works best with requests of 32 items at a time, the adapter can deliver 15 before and 16 after; or if the service is best with 64 items, the adapter can deliver 31 before and 32 after.

To show this aspect of the adapter, here's the beginning of itemsFromIndex, after which is just the code making the HTTP request and processing the results into an array. The _minPageSize and _maxPageSize properties represent the optimal request range (js/bingImageSearchDataSource.js):

```
itemsFromIndex: function (requestIndex, countBefore, countAfter) {
    // Some error checking omitted

    var fetchSize, fetchIndex;
    // See which side of the requestIndex is the overlap
    if (countBefore > countAfter) {
        countAfter = Math.min(countAfter, 10);    //Limit the overlap
        //Bound the request size based on the minimum and maximum sizes
        var fetchBefore = Math.max(Math.min(countBefore,
            this._maxPageSize - (countAfter + 1)), this._minPageSize - (countAfter + 1));
        fetchSize = fetchBefore + countAfter + 1;
        fetchIndex = requestIndex - fetchBefore;
    } else {
        countBefore = Math.min(countBefore, 10);
        var fetchAfter = Math.max(Math.min(countAfter,
            this._maxPageSize - (countBefore + 1)), this._minPageSize - (countBefore + 1));
        fetchSize = countBefore + fetchAfter + 1;
        fetchIndex = requestIndex - countBefore;
    }

    // Build up a URL for the request
    var requestStr = "https://api.datamarket.azure.com/Data.ashx/Bing/Search/Image"

    // Common request fields (required)
    + "?Query='" + that._query + "'" + "&$format=json" + "&Market='en-us'" + "&Adult='Strict'"
    + "&$top=" + fetchSize + "&$skip=" + fetchIndex;
```

The items, as noted in the table of adapter methods, are returned in the form of an object with the IFetchResult interface. Unlike the others we've seen, this one contains only properties, where items is the most important as it contains the item data (objects with IItem). Here's how the sample builds the fetch result: for each item in the HTTP response, it creates a data object (that can contain whatever information you want) and pushes it into an array called results:

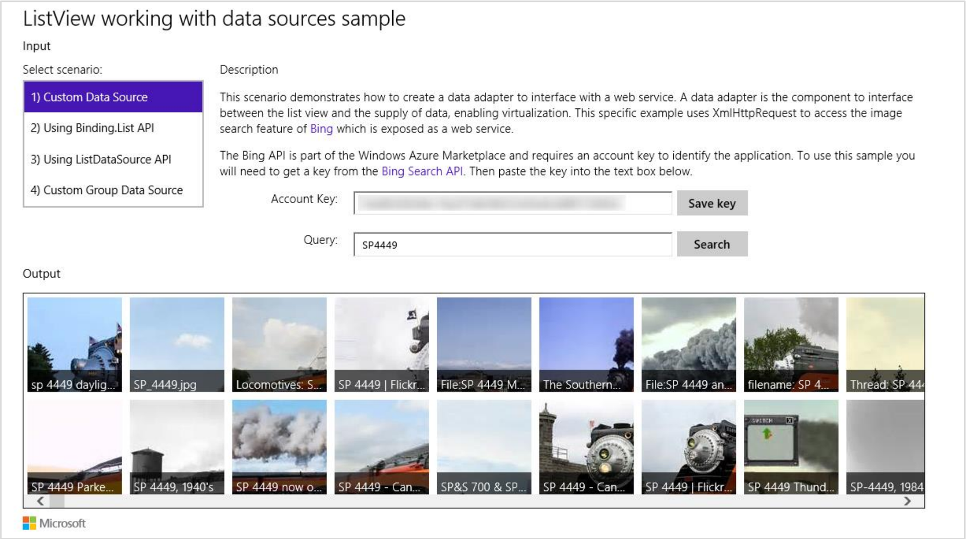
```
for (var i = 0, itemsLength = items.length; i < itemsLength; i++) {
    var dataItem = items[i];
    results.push({
        key: (fetchIndex + i).toString(),
        data: {
            title: dataItem.Title,
            thumbnail: dataItem.Thumbnail.Url,
            width: dataItem.Width,
            height: dataItem.Height,
            linkurl: dataItem.Url,
            url: dataItem.MediaUrl
        }
    });
}
```

The return value, that's ultimately delivered through a promise, then has the results array as the item's data property, along with two other properties:

```
return {
    items: results, // The array of items
    offset: requestIndex - fetchIndex, // The offset into the array for the requested item
    totalCount: that._maxCount,
};
```

The prefetching nature of the `item*` methods is helpful for the FlipView but essential for the ListView. This is demonstrated more clearly in the HTML ListView working with data sources sample that we saw earlier. Scenario 1 (see Figure 7-8) uses the same Bing Search data source as the FlipView sample with one small change. In the case of the FlipView, though, the page size is set between 1 and 10 and the largest number that the source will pull down at once is 100 items. In the ListView sample, the page size is set to 50 and the max number to 1000, as befits a collection control that will show more items at one time. Otherwise everything about the data source and the adapter is the same.

> **Another example** The PDF viewer showcase sample contains another implementation of a custom data source for reference, which you can find in its js/pdfDataSource.js file.



ListView working with data sources sample

Input

Select scenario:

- 1) Custom Data Source
- 2) Using Binding.List API
- 3) Using ListDataSource API
- 4) Custom Group Data Source

Description

This scenario demonstrates how to create a data adapter to interface with a web service. A data adapter is the component to interface between the list view and the supply of data, enabling virtualization. This specific example uses XmlHttpRequest to access the image search feature of Bing which is exposed as a web service.

The Bing API is part of the Windows Azure Marketplace and requires an account key to identify the application. To use this sample you will need to get a key from the Bing Search API. Then paste the key into the text box below.

Account Key: [redacted] Save key

Query: SP4449 Search

Output

FIGURE 7-8 Scenario 1 of the modified HTML ListView working with data sources sample, which allows you to enter a search term of your choice.

You can see the effect of the ListView's precaching by adding some console output to the top of the `itemsFromIndex` method in js/bingImageSearchDataSource.js:

```
console.log("itemsFromIndex: requestIndex = " + requestIndex + ", countBefore = " +
    countBefore + ", countAfter =" + countAfter);
```

> **Tip** A collection control can call `itemFromIndex` many times, so console output is often a more efficient way to watch what's happening than breakpoints.

With this in place, run the sample and page around quickly in the ListView, even dragging the scroll thumb far down in the list. You'll see console output that shows what the ListView is asking for:

```
itemsFromIndex: requestIndex = 0, countBefore = 0, countAfter =1
itemsFromIndex: requestIndex = 50, countBefore = 0, countAfter =30
```

```
itemsFromIndex: requestIndex = 150, countBefore = 0, countAfter =14
itemsFromIndex: requestIndex = 322, countBefore = 1, countAfter =80
itemsFromIndex: requestIndex = 508, countBefore = 1, countAfter =82
itemsFromIndex: requestIndex = 210, countBefore = 0, countAfter =0
itemsFromIndex: requestIndex = 607, countBefore = 0, countAfter =253
itemsFromIndex: requestIndex = 907, countBefore = 0, countAfter =41
itemsFromIndex: requestIndex = 1000, countBefore = 0, countAfter =0
itemsFromIndex: requestIndex = 543, countBefore = 13, countAfter =148
```

It's good to note, even though we haven't seen it yet, that the fulfillment of item promises—that is, the delivery of items—has a direct effect on the ListView control's `loadingState` property and its `loadingStateChanged` event. The ListView tracks whether its items have been rendered through completion of the item promises. The rest of your UI code, on the other hand, can just watch the ListView's `loadingStateChanged` event to track its state: there's no need to watch the data source directly from that other code.

To wrap up this sample and this section, now, scenario 4 demonstrates a custom data source implemented on top of two JavaScript arrays (items and groups) using two adapters and `VirtualizedDataSource`. We're still using a ListView here, but a `Binding.List` is nowhere in sight.

What's important in this scenario is that the items returned from the adapter's `itemsFromIndex` contain both the `key` and a `groupKey` properties of `IItem` (js/scenario4.js):

```
// Iterate and form the collection of items. results is returned in
// the IFetchResult.items property.
for (var i = fetchIndex; i <= lastFetchIndex; i++) {
    var item = that._itemData[i];
    results.push({
        key: i.toString(), // the key for the item itself
        groupKey: item.kind, // the key for the group for the item
        data: item // the data fields for the item
    });
}
```

where the `groupKey` values in the items match the `key` values in the `itemFromIndex` results returned by the group's data source and are also used in the group's `itemFromKey` implementation. Take a look through js/scenario4.js for more—the code is well commented.

Sidebar: Custom Data Sources in C++

Even though the `IListDataSource` and other interfaces are documented as part of WinJS, there's nothing that says they have to be implemented on JavaScript objects. The beauty of interfaces is that it doesn't matter how they're implemented, so long as they do what they're supposed to. Thus, if working with a data source will perform better when written in a language like C++, you can implement it as a class in a WinRT Component. This allows you to instantiate the object from JavaScript and still take advantage of the performance of compiled C++. The basics of writing a component are discusssed in Chapter 18, "WinRT Components."

ListView Features and Styling

Having already covered data sources and templates along with a number of ListView examples, we can now explore the additional features of the ListView control, such as styling, loading state transitions, drag and drop, and layouts. Optimizing item rendering then follows in the last section of this chapter (and applies to FlipView and ListView). First, however, let me answer a very important question.

When Is ListView the Right Choice?

ListView is the hands-down richest control in all of Windows. It's very powerful, very flexible, and, as we're already learning, very deep and intricate. But for all that, sometimes it's also just the wrong choice! Depending on the design, it might be easier to just use basic HTML/CSS layout or the `WinJS.UI.Hub` control, as we'll see in Chapter 8.

Conceptually, a ListView is defined by the relationship between three parts: a data source, templates, and layout. That is, items in a data source, which can be grouped, sorted, and filtered, are rendered using templates and organized with a layout (typically with groups and group headers). In such a definition, the ListView is intended to help visualize *a single collection of similar and/or related items*, where their groupings also have a relationship of some kind.

With this in mind, the following factors strongly suggest that a ListView is a *good* choice to display a particular collection:

- The collection can contain a variable number of items to display, possibly a very large number, showing more when the app runs on a larger display.
- It makes sense to organize and reorganize the items into various groups.
- Group headers help to clarify the common properties of the items in those groups, and they can be used to navigate to a group-specific page.
- It makes sense to sort and/or filter the items according to different criteria.
- Different groupings of items and information about those groups suggest ways in which semantic zoom would be a valuable user experience.
- The groups themselves are all similar in some way, meaning that they each refer to a similar kind of thing. Different place names, for example, are similar; a news feed, a list of friends, and a calendar of holidays are not similar.
- Items might be selectable individually or in groups so that you can apply commands on them.

On the flip side, opposite factors suggest that a ListView is *not* the right choice:

- The collection contains a limited or fixed number of items, or it isn't really a collection of related items at all.

- It doesn't make sense to reorganize the groupings or to filter or sort the items.
- You don't want group headers at all.
- You don't see how semantic zoom would apply.
- The groups are highly dissimilar—that is, it wouldn't make sense for the groups to sit side-by-side if the headers weren't there.
- Some items might be selectable or interactive, while others are not.

Let me be clear that I'm not talking about *design* choices here—your designers can hand you any sort of layout they want and as a developer it's *your* job to implement it! What I'm speaking to is how you choose to approach that implementation, whether with controls like ListView and Hub or just with HTML/CSS layout.

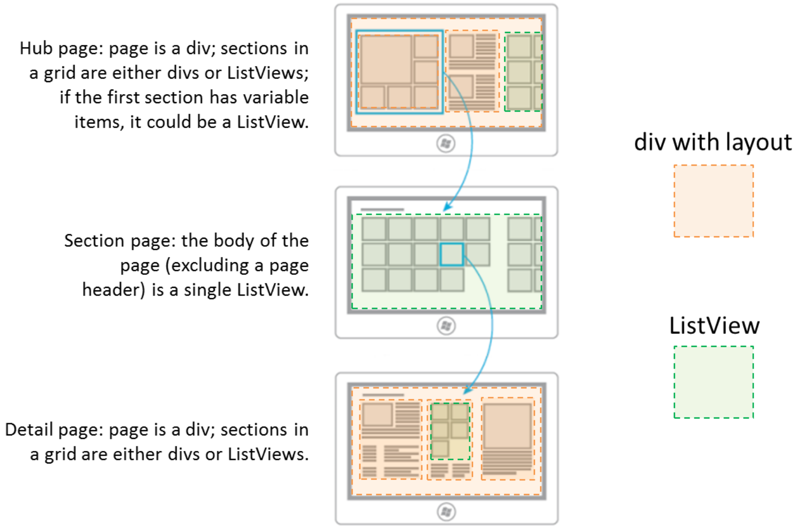
I say this because in working with the developers who created the very first apps for the Windows Store (especially before the Hub control was available in Windows 8.1), we frequently saw them trying to use ListView in situations where it just wasn't appropriate. An app's hub page, for example, might combine a news feed, a list of friends, and a calendar. An item details page might display a picture, a textual description, and a media gallery. In both cases, the page contains a limited number of sections and the sections contain very different content with very dissimilar items. Because of this, using a ListView gets complicated. It's better to use a single pannable `div` with a CSS grid in which you can lay out whatever sections you need, or the Hub control that was created for these scenarios.

Within those sections, of course, you might use ListView controls to display an item collection. I've illustrated these choices in Figure 7-9 (on the next page) using an image from the Navigation patterns topic, because you'll probably receive similar images from your designers. Ignoring the navigation arrows, the hub and details pages typically use a `div` at the root, whereas a section page is often a ListView. Within the hub and details pages there might be some ListView controls, but where the content is essentially fixed (like a single item), the best choice is a `div`.

A clue that you're going down the wrong path, by the way, is if you find yourself trying to combine multiple collections of unrelated data into a single source, binding that source to a ListView, and implementing a renderer to tease all the data apart again so that everything renders properly! All that extra work could be avoided simply by using the Hub or straight HTML/CSS layout.

For more on ListView design, see Guidelines for ListView controls, which includes details on the interaction patterns created with combinations of selection, tap, and swipe behaviors.

Tip I'll say it again: if you're creating a gallery experience with thumbnails in a ListView, avoid loading whole image files for that purpose. See the "Super performance tip" in the "Collection Control Data Sources" section earlier in this chapter.



Hub page: page is a div; sections in a grid are either divs or ListViews; if the first section has variable items, it could be a ListView.

Section page: the body of the page (excluding a page header) is a single ListView.

Detail page: page is a div; sections in a grid are either divs or ListViews.

FIGURE 7-9 Breaking down typical hub-section-detail page designs into `div` elements and ListView controls. The Hub control is typically useful for pannable regions that contain different types of sections.

Options, Selections, and Item Methods

In previous sections we've already seen some of the options you can use when creating a ListView, options that correspond to the control's properties. Let's look now at the complete set of properties, methods, and events, which I've organized into a few groups— after all, those properties and methods form quite a collection in themselves! Because the details for the individual properties are found on the `WinJS.UI.ListView` reference page, what's most useful here is to understand how the members of these groups relate (enumerations noted here are also in the `WinJS.UI` namespace):[68]

- **Data sources and templates** We've already seen the `groupDataSource`, `groupHeaderTemplate`, `itemDataSource`, and `itemTemplate` properties many times, so little more needs to be said on the technical details. For specific item template designs, the documentation provides two galleries of examples that you'll find in Item templates for grid layouts (11 designs) and Item templates for list layouts (6 designs).

- **Addressing items** The `currentItem` property gets or sets the item with the focus, and the `elementFromIndex` and `indexOfElement` methods let you cross-reference between an item index and the DOM element for that item. The latter could be useful if you have other controls in your item template and need to determine the surrounding item in an event handler.

- **Item visibility** The `indexOfFirstVisible` and `indexOfLastVisible` properties let you know

[68] I'll remind you that the ListView was overhauled for WinJS 2.0, as I describe on ListView Changes between WinJS 1.0 and WinJS 2.0. Here I describe the WinJS 2.0 control; anything you see in the docs that I omit is probably deprecated.

what indices are visible, and they can be used to scroll the ListView appropriate for a given item. The ensureVisible method brings the specified item into view, if it's been loaded. Also, the scrollPosition property contains the distance in pixels between the first item in the list and the current viewable area. Though you can set the scroll position of the ListView with this property, it's reliable only if the control's loadingState (see "Loading state" group below) is ready, otherwise the ListView may not yet know its actual dimensions. It's thus better to instead use ensureVisible or indexOfFirstVisible to control scroll position.

- **Item invocation** The itemInvoked event, as we've seen, fires when an item is tapped, unless the tapBehavior property is not set to none, in which case no invocation happens. Other tapBehavior values from the TapBehavior enumeration will always fire this event but determine how the item selection is affected by the tap: invokeOnly, directSelect, and toggleSelect. You can override the selection behavior on a per-item basis using the selectionchanging event and suppress the animation if needed. See the "Item Tap/Click Behavior" sidebar after this list. Also note that the args.details.itemIndex value that you receive with this event is a zero-based index relative to the current *visible* items in the ListView.
- **Item selection** The selectionMode property contains a value from the SelectionMode, enumeration, indicating single-, multi-, or no selection. At all times the selection property contains a ListViewItems object whose methods let you enumerate and manipulate the selected items (such as setting selected items through its set method). Changes to the selection fire the selectionchanging and selectionchanged events; with selectionchanging, its args.detail.newSelection property contains the newly selected items. For more on this, refer to scenario 4 of the HTML ListView essentials sample and the whole of the HTML ListView customizing interactivity sample, which among other things demonstrates a using the ListView in a master-detail layout (scenario 2). Note also that support for keyboard selection is built in.
- **Header invocation The groupHeaderTapBehavior is set to a value from the GroupHeaderTapBehavior enumeration, which can be either invoke or none** (the default)**. When set to invoke, group headers will fire the groupheaderInvoked event in response to taps, clicks, or the Enter key (when the header has the focus). As noted earlier with "Quickstart #4," a demonstration is found in scenario 3 of the** HTML ListView grouping and Semantic Zoom sample. With the keyboard, the Tab key will navigate from a group of items to its header, after which the arrow keys navigate between headers. It's also recommended that apps also handle the Ctrl+Alt+G keystroke to navigate from items in the current group to the header.
- **Swiping** Related to item selection is the swipeBehavior property that contains a value from the SwipeBehavior enumeration. This determines the response to swiping or cross-slide gestures on an item where the gesture moves perpendicular to the panning direction of the list. If swipeBehavior is set to none, swiping has no effect on the item and the gesture is bubbled up to the parent element. If this is set to select, the gesture selects the item.

- **Layout** As we've also seen, the `layout` property (an object) describes how items are arranged in the ListView, which we'll talk about more in "Layouts" below. Note that orientation (vertical or horizontal) is a property of the layout and not of the ListView itself. We've also seen the `forceLayout` function that's specifically used when a `display: none` style is removed from a ListView and it needs to re-render itself. One other method, `recalculateItemPosition`, repositions all items in the ListView and is meant specifically for UI designer apps or when changing items within a cell-spanning layout.

- **Loading behavior** A ListView is set up to provide random access to all the items it contains but keeps only five total pages of items in memory at a time. The total number of items is limited to the `maxDeferredItemCleanup` property. More on this under "Loading State Transitions" below.

- **Loading state** The read-only `loadingState` property contains either `"itemsLoading"` (the list is requesting items and headers from the data source), `"viewportLoaded"` (all items and headers that are visible have been loaded from the source), `"itemsLoaded"` (all remaining nonvisible buffered items have been loaded), or `"complete"` (all items are loaded, content in the templates is rendered, and animations have finished). Whenever this property changes while the ListView is updating its layout due to panning, the `loadingStateChanged` event fires. Again, see "Loading State Transitions."

- **Drag and drop** If the `itemsDraggable` property is `true`, the ListView can act as an HTML5 drag and drop source, such that items can be dragged to other controls. If `itemsReorderable` is `true`, the ListView allows items within it to be moved around via drag and drop. The events that occur during drag and drop are `itemDragStart`, `itemDragLeave`, `itemDragEnter`, `itemDragBetween`, `itemDragChanged`, `itemDragDrop`, and `itemDragEnd`. See "Drag and Drop" below for more.

- **Events** `addEventListener, removeEventListener, and dispatchEvent are the standard DOM methods for handling and raising events. These can be used with any event that the ListView supports. To round out the event list, there are two more to mention. First, contentanimating` fires when the control is about to run an item entrance or transition animation, allowing you to either prevent or delay those animations. Second, the `keyboardnavigating` event indicates that the user has tabbed to a new item or a header.

- **Semantic zoom** The `zoomableView` property contains the `IZoomableView` implementation as required by semantic zoom (apps will never manipulate this property).

- **Dispose pattern** The ListView implements the `dispose` method and also has a `triggerDispose` method to run the process manually.

Sidebar: Referring to Enumerations in data-win-options

When specifying values from enumerations in `data-win-options`, you can use two forms. The most explicit way is to use the full name of the value, as shown here (other options omitted):

```
<div id="listView" data-win-control="WinJS.UI.ListView"
    data-win-options="{
        selectionMode:  WinJS.UI.SelectionMode.none,
        tapBehavior: WinJS.UI.TapBehavior.invokeOnly,
        groupHeaderTapBehavior: WinJS.UI.GroupHeaderTapBehavior.invoke,
        swipeBehavior: WinJS.UI.SwipeBehavior.none }">
</div>
```

Because all those values resolve to strings, you can just use the string values directly:

```
<div id="listView" data-win-control="WinJS.UI.ListView"
    data-win-options="{
        selectionMode:  'none',
        tapBehavior: 'invokeOnly',
        groupHeaderTapBehavior: 'invoke',
        swipeBehavior: 'none' }">
</div>
```

Either way, always be careful about using exact spellings here. The forgiving nature of JavaScript is such that if you specify a bogus option, you won't necessarily see any exception: you'll just see default behavior for that option. This can be a hard bug to find, so if something isn't working quite right, really scrutinize your `data-win-options` string.

Sidebar: Item Tap/Click Behavior

When you tap or click an item in a ListView with the `tapBehavior` property set to something other than `none`, there's a little ~97% scaling animation to acknowledge the tap (this does not happen for headers). If you have some items in a list that can't be invoked (like those in a particular group or ones that you show as disabled because backing data isn't available), they'll still show the animation because the `tapBehavior` setting applies to the whole control. To remove the animation for any specific item, you can add the `win-interactive` class to its element within a renderer function, which is a way of saying that the item internally handles tap/click events, even if it does nothing but eat them. If at some later time the item becomes invocable, you can, of course, remove that class.

If you need to suppress selection for an item, add a handler for the ListView's `selectionchanging` event and call its `args.detail.preventTapBehavior` method. This works for all selection methods, including swipe, mouse click, and the Enter key.

Styling

Following the precedent of Chapter 5 and the earlier sections on Repeater and FlipView, styling is best understood visually as in Figure 7-10, where I've applied some garish CSS to some of the `win-*` styles so that they stand out. I also highly recommend that you look at the Styling the ListView and its items and How to brand your ListView topics in the documentation.

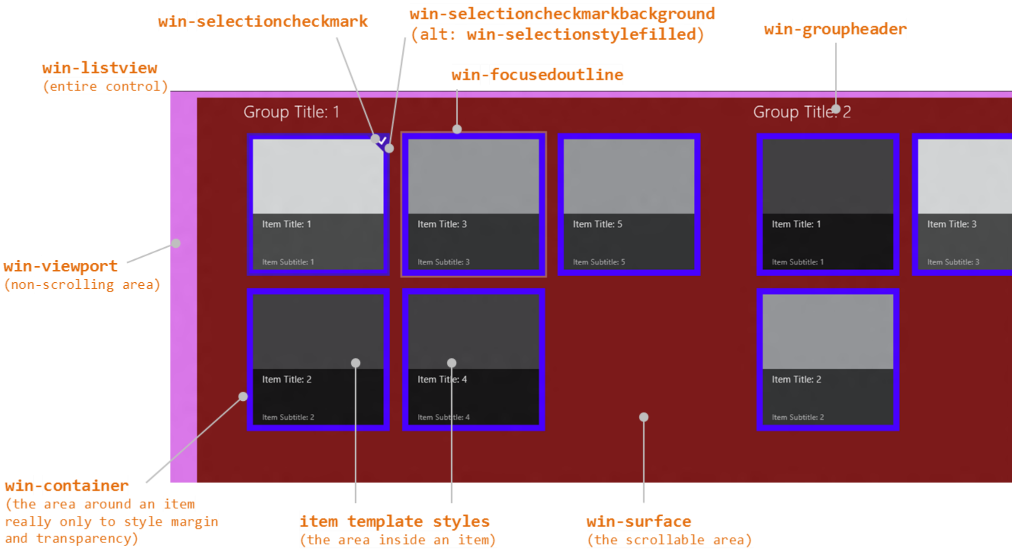


FIGURE 7-10 Most of the style classes as utilized by the ListView control.

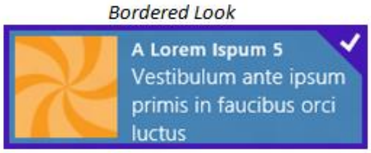
Some short notes about styling:

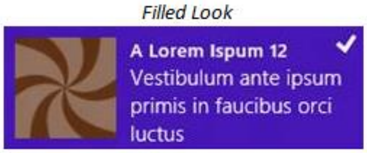
- Remember that Blend is your best friend here!
- As with styling the FlipView, a class like `win-listview` is most useful with styles like borders around the control. You can also style background colors and nonscrolling images here if the viewport, surface, and items have a transparent background as well.
- To style a background that pans with the items, set `background-image` for the `win-surface` selector.
- The ListView will automatically display a `<progress>` control while loading items, and you can style this with the `.win-listview .win-progress` selector.
- `win-viewport` styles the nonscrolling background of the ListView and is the best place to style margins. As the container for the scrollable area, the `win-viewport` element will also have a `win-horizontal` or `win-vertical` class depending on the layout's orientation.
- `win-container` primarily exists for two things. One is to create space between items using `margin` styles, and the other is to override the default background color, often making its

background transparent so that the `win-surface` or `win-listview` background shows through. Note that if you set a `padding` style here instead of `margin`, you'll create areas *around* what the user will perceive as the item but that are *still invoked as the item*. Not good. So always use `margin` to create space between items.

- Though `win-item` is listed as a style, it's deprecated and may be removed in the future: just style the item template directly.
- The documentation points out that styles like `win-container` and `win-surface` are used by multiple WinJS controls. (FlipView uses a few of them.) If you want to override styles for a ListView, be sure to scope your selectors with other classes like `.win-listview` or a particular control's id or class.
- The default ListView height is 400px, and the control does *not* automatically adjust itself to its content. You'll almost always want to override the `height` style in CSS or set it from JavaScript when you know the space that the ListView should occupy in your overall layout.
- Not shown in the figure is the `win-backdrop` style that's used as part of the `win-container` element. The "backdrop" is a blank item shape that can appear when the user very quickly pans a ListView to a new page and before items are rendered. This is gray by default, but you can add styles in the `.win-container .win-backdrop` selector to override it.

Selections and selection state take a little more explaining than one bullet item. First, the default selection styling is a "bordered" look (below left). If you want the filled look (below right), add the `win-selectionstylefilled` class to the ListView's root element.





The following styles then apply to the different parts of the selection:

Style class	**Part identified**
`win-selectionborder`	The border around a selected item.
`win-selectionbackground`	The background of selected items.
`win-selectionhint`	The selection hint that appears behind a selected item during swiping.
`win-selectioncheckmark`	The selection checkmark.
`win-selectioncheckmarkbackground`	The checkmark background.

If you've read Chapter 5, you'll recognize these as the same ones that apply to the `ItemContainer` control, and, in fact, they have exactly the same meaning. For examples on using these classes, refer to the "Styling Gallery: WinJS Controls" in that chapter.

Loading State Transitions

If you're like myself and others in my family, you probably have an ever-increasing stockpile of digital photographs that make you glad that 1TB+ hard drives keep dropping in price. In other words, it's not uncommon for many consumers to have ready access to collections of tens of thousands of items that they will at some point want to pan through in a ListView. But just imagine the overhead of trying to load thumbnails for every one of those items into memory to display in a list. On low-level and low-power hardware, you'd probably be causing every suspended app to be quickly terminated, and the result will probably be anything but "fast and fluid"! The user might end up waiting a *really* long time for the control to become interactive and will certainly tire of watching a progress ring.

With this in mind, the ListView always reflects the total extent of the list in its scrollbar. This gives the user has some idea of the size of the list and allows scrolling to any point in the list (random access). At the same time, the ListView keeps a total of only five pages or screenfuls of items in memory at any given time, limiting the number of items to `maxDeferredItemCleanup` if you set that property. This generally means that the visible page (in the viewport) plus two buffer pages ahead and behind will be loaded in memory at any given time. If you're viewing the first page, the buffer extends four pages ahead; if you're on the last page, the buffer extends four pages behind—you get the idea. (The priority at which these different pages are loaded also changes with the panning direction via the scheduler—those pages that are next in the panning direction are scheduled at a higher priority than those in the opposite direction. This is all transparent to your app.)

When the pages first start loading, the ListView's `loadingState` property will be set to `itemsLoading`. When all the visible items are loaded, the state changes to `viewportLoaded`. Once all the buffered pages are loaded, the state changes to `itemsLoaded`. When all animations are done, the state becomes `complete`. The `loadingstatechanged` event will of course fire on each transition.

Whenever the user pans to a location in the list, any pages that fall out of the viewport or buffer zone are discarded, if necessary, to stay under `maxDeferredItemCleanup` and loading of the new viewport page and its buffer pages begins. Thus, the ListView's `loadingState` property will start again at `itemsLoading` and then transition through the other states as before. If the user pans some more during this time, the `loadingState` is again reset and the process begins anew.

Sidebar: Incremental Loading

Apart from potentially very large but known collections, other collections are, for all intents and purposes, essentially unbounded, like a news feed that might have millions of items stretching back to the Cenozoic Era (at least by Internet reckoning!). With such collections, you probably won't know just how many items there are at all; the best you can do is just load another chunk when the user wants them.

Although the ListView itself doesn't provide support for automatically loading another batch of items at the appropriate time, it's relatively straightforward to do within either a data source or an item rendering function. Just watch for item requests near the end of the list (however far you want to make it), and use that as a trigger to load more items. Within an item renderer, check the position of the items being rendered, which tells you where the ListView's viewport is relative to the collection. In a data source, watch the index or key in IListDataAdapter methods like itemsFromIndex, especially when the countAfter argument exceeds the end of the current list. Either way, you then load more items into the collection, changes that should generate change notifications to the control. The control will call the source's getCount method in response and update its scrollbar accordingly.

A small demonstration of this can be found in scenarios 2 and 3 of the HTML ListView incremental loading behavior sample, which adds more items to a Binding.List when needed from within the item renderer.

Drag and Drop

It's very natural when one is looking at a collection of neat stuff to want to copy or move some of that neat stuff to some other location in the app. HTML5 drag and drop provides a standard for how this works between elements, and the ListView is capable of participating in such operations with both mouse and touch.

To briefly review the standard, a draggable element has the draggable="true" attribute. It becomes a source of a *dataTransfer* object (that carries the data) and sees dragstart, drag, dragenter, dragleave, dragover, and dragend events at appropriate times (for a concise reference, see DragEvent on MSDN). A target element, for its part, will see dragenter, dragleave, dragover, and drop events and have access to the *dataTransfer* object. There's more to it, such as drop effects you set within various events in response to the state of the Ctrl and Alt keys, but those are the basics.

The ListView implements these parts of the HTML5 spec on your behalf, surfacing similar events and giving you access to the *dataTransfer* object, whose setData and getData methods are what you use to populate and retrieve the data involved.

The ListView can participate in drag and drop in four ways. First, it can be made reorderable within itself, independent of exchanging items with other sources or targets. Second, it can be made a drag source, so you can drag items from the ListView to other HTML5 drop targets. Third, the ListView can be a drop target and accept data dragged from other HTML5 sources. And fourth, items within a ListView can themselves be individual drop targets. (Note that ListView and HTML5 drag and drop is not presently enabled between apps, just within an app.)

Let's go through each of these possibilities using examples from the HTML ListView reorder and drag and drop sample. Reordering in a ListView is perhaps the simplest: it requires nothing more than setting the itemsReorderable option to true, as demonstrated in scenario 1 (html/scenario1.html):

```
<div id="listView" data-win-control="WinJS.UI.ListView" data-win-options="{
    itemDataSource: myData.dataSource, itemTemplate: smallListIconTextTemplate,
    itemsReorderable: true, layout: { type: WinJS.UI.GridLayout } }">
</div>
```

That's it—with this one option and no other code (you can see that js/scenario1.js does nothing else), you get the behavior shown in Video 7-3, for both single and multiple items (shown with the mouse as it's more efficient). Under the covers, reordering of the ListView fundamentally means moving items in the data source, in response to which the ListView updates its display. To be precise, the ListView uses the moveBefore and moveAfter methods of IListDataSource to do the reordering. This implies, of course, that the data source itself is reorderable. If it isn't, you'll get an exception complaining about moveAfter. (If you like throwing exceptions for fun, try adding itemsReorderable: true to the ListView in scenario 1 of the HTML ListView working with data sources sample.)

In short, setting itemsReorderable turns on all the code to reorder items in the data source in response to user action in the ListView. Quite convenient!

Although scenario 1 doesn't show it, the ListView's various itemdrag* events will fire when reordering takes place, as they do for all drag and drop activities in the control:

- itemdragstart when dragging begins,
- itemdragbetween as the item is moved around in the list,
- itemdragleave and itemdragenter if the item moves out of and into the ListView,
- itemdragdrop if and when the item is released, making it the one you'd use to detect a reordering, and
- itemdragend when it's all over (including when the item is dragged out and released, or the ESC key is pressed).

The only event not represented here is itemdragchanged, which specifically signals that items currently being dragged have been updated in the source. Note also that loadingstatechanged will be fired after itemdragdrop and itemdragend as the control re-renders itself.

To serve as a drag source, independent of reordering, set the ListView's itemsDraggable property to true, as in scenario 2 (html/scenario2.html):

```
<div id="listView" data-win-control="WinJS.UI.ListView" data-win-options="{
    itemDataSource: myData.dataSource, selectionMode: 'none',
    itemTemplate: smallListIconTextTemplate,
    itemsDraggable: true, layout: { type: WinJS.UI.GridLayout } }">
</div>
```

When dragging starts in the ListView, it will fire an itemdragstart event, whose eventArgs.detail contains two objects. The first is the HTML5 dataTransfer object, which you populate with your source data through its setData method. The second is a dragInfo object that specifically contains a ListView Selection object for the items being dragged (selected or not). The sample uses dragInfo.getIndices to source the indicies of those items (js/scenario2.js):

```
listView.addEventListener("itemdragstart", function (eventArgs) {
    eventArgs.detail.dataTransfer.setData("Text",
        JSON.stringify(eventArgs.detail.dragInfo.getIndices()));
});
```

The target in this scenario is another div named *myDropTarget* that simply handles the HTML5 drop event (and a few others to give visual feedback). In the drop handler, the eventArgs.dataTransfer property contains the HTML5 dataTransfer object again, whose getData method returns the goods (an array of indices in this sample):

```
dropTarget.addEventListener("drop", function (eventArgs) {
    var indexSelected = JSON.parse(eventArgs.dataTransfer.getData("Text"));
    var listview = document.querySelector("#listView").winControl;
    var ds = listview.itemDataSource;

    ds.itemFromIndex(indexSelected[0]).then(function (item) {
        WinJS.log && WinJS.log("You dropped the item at index " + item.index + ", "
        + item.data.title, "sample", "status");
    });
});
```

You can, of course, do whatever you want with the dropped data. As you can see, the sample simply looks up the item in the ListView's data source. If you wanted to move the item out of the source list, you would use that index to call the data source's remove method.

Moving on to scenarios 3 and 4, these make the ListView a drop target. Scenario 3 allows dropping the data at a specific location in the control by setting itemsReorderable is true. Scenario 4, on the other hand, leaves itemsReorderable set to false but then implements a handler for the HTML5 drop event. This is a key difference: you use the ListView's itemdragdrop event only if the ListView is reorderable, meaning that you'll have an index for the specific insertion point; otherwi se you can use the HTML5 drop event and insert the data where it makes sense. For example, if the data source is sorted, you'd just append the new item to the collection and let its sorted projection figure out the location.

In both cases, of course, data source must support insertions; otherwise you'll see an exception.

In scenario 3 now, with itemsReorderable: true, dropping something on the ListView will fire an itemdragdrop event. The eventArgs.detail object here will contain the index of the drop location, the insertAfterIndex for the insertion point, and the HTML5 dataTransfer object (js/scenario3.js):

```
listView.addEventListener("itemdragdrop", function (eventArgs) {
    var dragData = eventArgs.detail.dataTransfer &&
        JSON.parse(eventArgs.detail.dataTransfer.getData(“Text”));

    if (dragData && dragData.sourceId === myDragContent.id) {
        var newItemData = { title: dragData.data,
            text: ("Source id: " + dragData.sourceId), picture: dragData.imgSrc };
        // insertAfterIndex tells us where in the list to add the new item. If we're
        // inserting at the start, insertAfterIndex is -1. Adding 1 to insertAfterIndex
        // gives us the nominal index in the array to insert the new item.
        myData.splice(eventObject.detail.insertAfterIndex + 1, 0, newItemData);
    }
});
```

Scenario 4 with `itemsReorderable: false` just implements a drop handler with pretty much the same code, only it always inserts the dropped item at the beginning of the list (js/scenario4.js):

```
listView.addEventListener("drop", function (eventArgs) {
   var dragData = JSON.parse(eventArgs.dataTransfer.getData("Text"));

   if (dragData && dragData.sourceElement === myDragContent.id) {
      var newItemData = { title: dragData.data,
         text: ("id: " + dragData.sourceElement), picture: dragData.imgSrc };
      var dropIndex = 0;
      myData.splice(dropIndex, 0, newItemData);
   }
});
```

Be aware that the `myData` object in both these cases is a `Binding.List`; if you use a different data source behind the ListView, use the `IListDataSource` methods to insert the item or items instead.

The last scenario in the sample shows how to drop data on a specific *item* in the ListView, rather than into the control as a whole, which simply means adding HTML5 drag and drop event handlers to the items themselves. Scenario 5 does this within the item template (html/scenario5.html):

```
<div id="smallListIconTextTemplate" data-win-control="WinJS.Binding.Template">
   <div class="smallListIconTextItem" ondragover="Scenario5.listItemDragOverHandler(event)"
      ondrop="Scenario5.listItemDropHandler(event)"
      ondragleave="Scenario5.listItemDragLeaveHandler(event)">
      <!-- Other content omitted -->
   </div>
</div>
```

In other words, handling drag and drop on an item in a ListView—or any other collection control or custom control for that matter—is simply a matter of handling the HTML5 events on the items and has nothing to do with the collection control itself. Where the ListView gets involved is just to act as a thin proxy on the HTML5 events so that it can add a little more information to support reordering and selection information.

Layouts

The ListView's `layout` property contains an object that's used to visually organize the list's items. Whatever layout you provide as an option to the ListView constructor determines the control's initial appearance. Changing the `layout` at run time tells the ListView to re-render itself with the new structure, which is how a ListView can easily switch between one- and two-dimensional layouts, between horizontal and vertical orientations, and so on. An example can be found in scenario 3 of the HTML ListView essentials sample.

`WinJS.UI` contains several prebuilt layouts, each of which is an object class in its own right that follows the recommended design guidelines for presenting collections:

- `GridLayout` A two-dimensional layout that can pan horizontally or vertically based on the CSS grid. This is the ListView's default if you don't specify a layout.

- ListLayout A one-dimensional layout that can pan horizontally or vertically, based on the CSS flexbox.
- CellSpanningLayout A derivative of the GridLayout that supports items of different sizes—that is, items that can span rows and columns.

You can also create custom layouts, which are covered in Appendix B and the HTML ListView custom layout sample.

How you specify a layout depends on whether you're doing it in markup or code. In markup, the layout option within the data-win-options string has this syntax:

```
layout: { type: <layout> [, <options>] }
```

<layout> is the full name of the layout constructor, such as WinJS.UI.GridLayout, WinJS.UI.-ListLayout, WinJS.UI.CellSpanningLayout, or a custom class; <options> then provides options for that constructor. For example, the following configures a GridLayout with headers on the left and a maximum of four rows:

```
layout: { type: WinJS.UI.GridLayout, groupHeaderPosition: 'left', maximumRowsOrColumns: 4 }
```

If you create the layout object in JavaScript by using new to call the constructor directly and assigning the result to the layout property, you provide such options directly to the constructor:

```
listView.layout = new WinJS.UI.GridLayout({ groupHeaderPosition: "left",
    maximumRowsOrColumns: 4 });
```

You can also set properties on the ListView's layout object in JavaScript once it's been created, if you want to take that approach. Changing properties will generally update the layout.

In any case, each layout has its own unique options, as described in the following tables, which are also accessible at run time as properties on the layout object. As with the enumerations for ListView options, you can use string values, such as 'horizontal', or the full identifier from the enumeration, such as WinJS.UI.Orientation.horizontal.[69]

GridLayout option/property	Description
groupHeaderPosition	Controls the placement of headers in relation to their groups using a value from the HeaderPosition enumeration: top (the default) or left (which becomes right in right-to-left languages).
maximumRowsOrColumns	Controls the number of items the layout will place vertically before starting another column (with orientation set to horizontal) or place horizontally before starting another row (orientation set to vertical).
orientation	Controls the panning direction of the layout with a value from the Orientation enumeration: horizontal (the default) or vertical.

[69] Other properties you see in the documentation are either deprecated from WinJS 1.0 or for internal use. You'll also see many methods on these objects that are how the ListView talks to the layout; apps don't call those methods directly.

ListLayout option/property	Description
groupHeaderPosition	Controls the placement of headers in relation to their groups using a value from the HeaderPosition enumeration: top (the default) or left (which becomes right in right-to-left languages).
orientation	Controls the panning direction of the layout with a value from the Orientation enumeration: horizontal or vertical (the default)

CellSpanningLayout option/property	Description
groupHeaderPosition	Controls the placement of headers in relation to their groups using a value from the HeaderPosition enumeration: top (the default) or left (which becomes right in right-to-left languages).
maximumRowsOrColumns	Controls the number of items the layout will place vertically before starting another column (with orientation set to horizontal) or place horizontally before starting another row (orientation set to vertical).
orientation	Always horizontal; the vertical orientation is not supported.
groupInfo	Identifies a function that returns an object whose properties indicate whether cell spanning should be used and the size of the cell. This is called only once within a layout process.
itemInfo	Identifies a function that returns an object whose properties describe the exact size for each item and whether the item should be placed in a new column or row (depending on orientation).

Let's see these options in action. The horizontal and vertical variations for GridLayout and ListLayout are demonstrated in scenario 6 of the HTML ListView essentials sample. The four layout properties are declared as follows (for different controls, of course; html/scenario6.html), with the output shown in Figures 7-11 and 7-12:

```
layout: { type: WinJS.UI.GridLayout, orientation: WinJS.UI.Orientation.horizontal}
layout: { type: WinJS.UI.GridLayout, orientation: WinJS.UI.Orientation.vertical}

layout: { type: WinJS.UI.ListLayout, orientation: WinJS.UI.Orientation.vertical}
layout: { type: WinJS.UI.ListLayout, orientation: WinJS.UI.Orientation.horizontal}
```

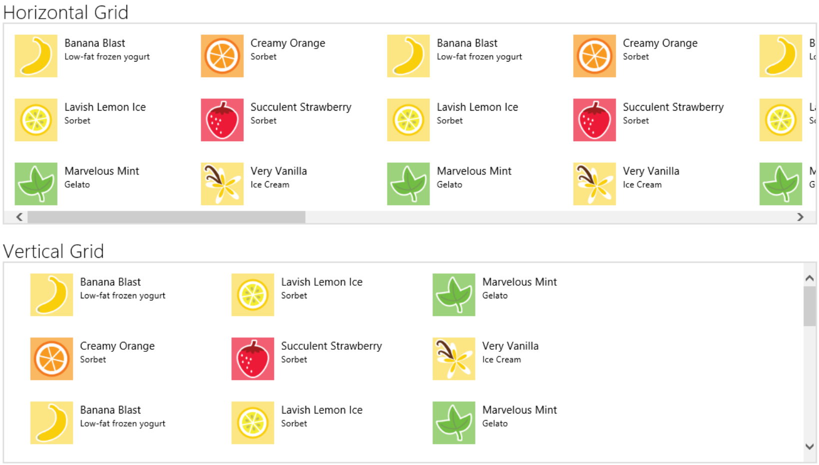


FIGURE 7-11 Horizontal (default) and vertical orientations for the GridLayout; notice the directions in which the items are laid out: top to bottom in horizontal, right to left in vertical (which is reversed with right-to-left languages). I've included the scrollbars here to show the panning direction more clearly.

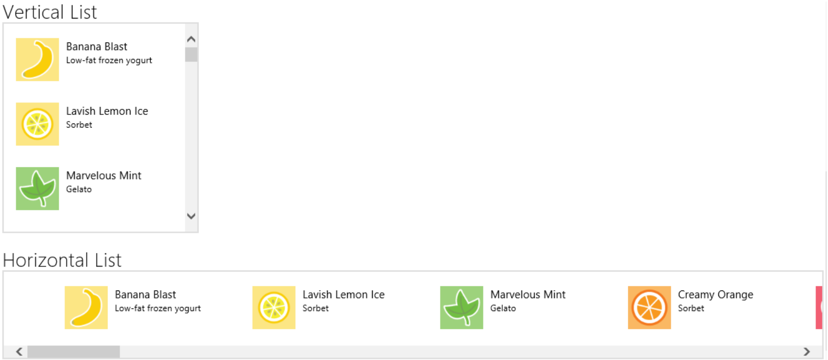


FIGURE 7-12 Vertical (default) and horizontal orientations for the `ListLayout`. I've again included the scrollbars here to show the panning direction more clearly.

In the Vertical Grid of Figure 7-11, the layout was able to fit only three items horizontally before starting a new row on a 1366x768 display, where I took the screenshot. On a larger monitor where the control is wider, I get more horizontal items. If, however, I wanted to always limit each row to three items, I could use the `maximumRowsOrColumns` options like so:

```
layout: { type: WinJS.UI.GridLayout, orientation: WinJS.UI.Orientation.vertical,
    maximumRowsOrColumns: 3}
```

With the `groupHeaderPosition` option, the easiest way to see its effect is to go to scenario 1 of the HTML ListView grouping and Semantic Zoom sample and set the option to `left` in the ListView declarations of html/scenario1.html. The results are shown in Figure 7-13.

```
layout: { type: WinJS.UI.GridLayout, groupHeaderPosition: 'left' }
layout: { type: WinJS.UI.ListLayout, groupHeaderPosition: 'left' }
```

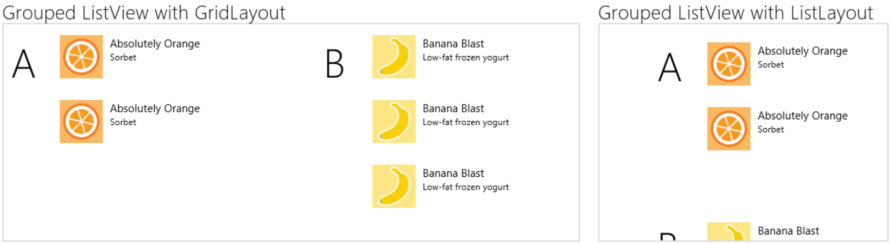


FIGURE 7-13 Using `groupHeaderPosition: 'left'` with `GridLayout` and `ListLayout`. With a right to left language, the left position will shift to the right. Compare this to Figure 7-3, and note that I made the controls a little bigger in CSS so that the items would show fully.

For the `CellSpanningLayout` now, we can turn to Scenarios 4 and 5 of the HTML ListView item templates sample, the output of which is shown in Figure 7-14. (The only difference between the scenarios is that 4 uses a rendering function and 5 uses a declarative template.)

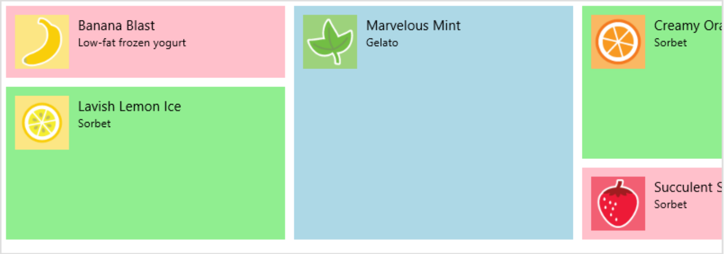


FIGURE 7-14 The HTML ListView item templates sample showing multisize items through cell spanning.

> **Tip #1** If you start playing with this sample and make changes to the CSS in any given scenario, be aware that it does not scope the CSS for each scenario to the associated page, which means that styles from the most recently loaded scenario might be used by the others. To correct this, I've added such scoping to the modified sample in this chapter's companion content.

> **Tip #2** If you alter items in a cell spanning layout, call the `ListView.recalculateItemPosition` method after the change is made. If you're using a data source other than the `Binding.List`, also call the `IListDataSource.beginEdits` before making changes and `endEdits` afterwards.

The basic idea of cell spanning is to define a layout grid based on the size of the smallest item. For best performance, make the grid as coarse as possible, where every other element in the ListView is a multiple of that size.

You define the cell grid through the `CellSpanningLayout.groupInfo` property. This is a function that returns an object with three properties: `enableCellSpanning`, which is set to `true` (unless you want `GridLayout` behavior!), and `cellWidth` and `cellHeight`, which contain the pixel dimensions of your minimum cell. In the sample (see js/data.js), this function is named *groupInfo* like the layout's property. I've given it a different name here (and omitted some other bits) for clarity:

```
function cellSpanningInfo() {
    return {
        enableCellSpanning: true,
        cellWidth: 310,
        cellHeight: 80
    };
}
```

> **Tip** In thinking about the layout of your ListView, know that three styles in the WinJS stylesheets set default item spacing. You can override these with the same selectors depending on your `orientation`, and they're important to know for calculations that we'll see shortly:
>
> ```
> .win-horizontal .win-gridlayout .win-container {
>     margin: 5px;
> }
> .win-vertical .win-gridlayout .win-container {
>     margin: 5px 24px 5px 7px;
> }
> .win-rtl > .win-vertical .win-gridlayout .win-container {
> ```

```
        margin: 5px 7px 5px 24px;
    }
```

The second required piece is a function you specify in the itemInfo property, which is called for every item in the list and should thus execute quickly. It receives an item index and returns an object with the item's width and height properties, along with an optional newColumn property that lets you control whether the layout should start a new column for this item. Here's the general idea:

```
function itemInfo(itemIndex) {
    //determine values for itemWidth and itemHeight given itemIndex
    return {
        newColumn: false,
        itemWidth: itemWidth,
        itemHeight: itemHeight
    };
}
```

In the sample, itemInfo is implemented by performing a quick lookup for the item in a size map (again in js/data.js):

```
var sizeMap = {
    smallListIconTextItem: { width: 310, height: 80 },
    mediumListIconTextItem: { width: 310, height: 170 },
    largeListIconTextItem: { width: 310, height: 260 },
    defaultSize: { width: 310, height: 80 }
};

var itemInfo = WinJS.Utilities.markSupportedForProcessing(function itemInfo(itemIndex) {
    var size = sizeMap.defaultSize;

    var item = myCellSpanningData.getAt(itemIndex);
    if (item) {
        size = sizeMap[item.type];
    }

    return size;
});
```

With both of these functions in place (and both are required), you then specify them in the ListView's layout property in code or in markup. Here's how it's done declaratively in a data-win-options string (html/scenario4.html and html/scenario5.html):

```
layout: { groupInfo: groupInfo, itemInfo: itemInfo, type: WinJS.UI.CellSpanningLayout }
```

Now notice that the heights returned from itemInfo are not exact multiples of the cell height of 80 in groupInfo. The same would be true if we spanned columns, as we'll see later. This is because we have to take into account the margins between items as determined by the win-container styles shown earlier. You do this according to either of the following formulae (which are the same, just rearranged):

itemSize = ((cellSize + containerMargin) x span) – containerMargin

cellSize = ((itemSize + containerMargin) / span) - containerMargin

where *Size* here is either width of height, depending on the dimension you're calculating, *span* is the number of rows or columns an item is spanning, and *containerMargin* is the top+bottom margin when calculating height or left+right when calculating width.

You use the first formula if you want to start with the cell dimension as defined by the `groupInfo` function and calculate the size of the item as you'll report in `itemInfo` and as you'll style in CSS. If you want to start from the item size in CSS or `itemInfo` and get the cell dimension for `groupInfo`, use the second formula.

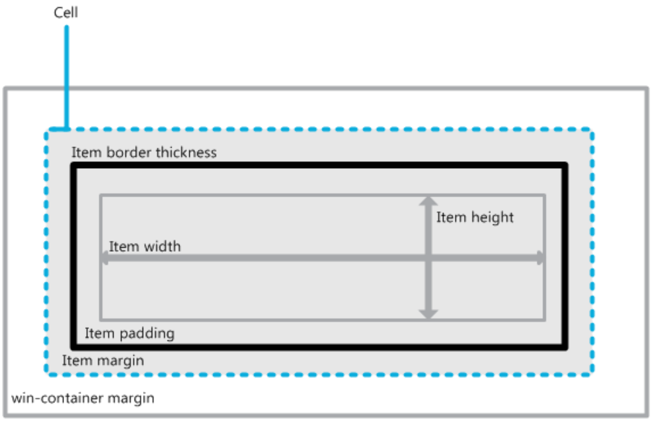
In the sample, we have items with the same widths but spanning one, two, or three rows, so we want to do the calculations for height. Using the first formula, *cellSize* is 80 (the height from `groupInfo`); and the top and bottom margins from `win-container` for a horizontal grid are both 5px, so *containerMargin* is 10. Plugging in the spans, we get the following *itemSize* results:

((80 + 10) * 1) - 10 = 80

((80 + 10) * 2) - 10 = 170

((80 + 10) * 3) - 10 = 260

The `itemInfo` function, then, should return 310x80, 310x170, and 310x260, as we see it does. In CSS the *width+padding+margin+border* and *height+padding+margin+border* styles for each item must match these dimensions exactly. This is illustrated nicely in the [How to display items that are different sizes]() topic in the documentation:



Our styles in in css/scenario4.css and css/scenario5.css are thus the following:

```
.smallListIconTextItem {
    width: 300px;
    height: 70px;
    padding: 5px;
}

.mediumListIconTextItem {
    width: 300px;
    height: 160px;
    padding: 5px;
}

.largeListIconTextItem {
    width: 300px;
    height: 250px;
    padding: 5px;
}
```

Now let's play with the width instead, using a scenario 7 I've added to the modified sample in this chapter's companion content. This gets interesting because of how the `CellSpanningLayout` fills in gaps when laying out items. That is, although it generally places items in columns from top to bottom, then left to right (or right to left for some languages), it can backfill empty spaces with suitable items it comes across later on.

In this added scenario 7, I've create new `groupInfo2` method in js/data.js that sets the cell size to 155x80. The small items will be occupy one cell, the medium items will span two columns, and the large items will span two rows and two columns. To make the calculations more interesting, I've also set some custom margins on win-container (css/scenario7.css):

```
.page7 .win-horizontal .win-gridlayout .win-container {
    margin: 9px 7px 3px 2px;  /* top, right, bottom, left*/
}
```

Applying the first formula again we get these item dimensions:

Item size	Width (cellWidth = 155)	Height (cellHeight = 80)
small (1x1)	((155 + (7+2)) * 1 - (7+2) = 155	((80 + (9+3)) * 1 - (9+3) = 80
medium (2x1)	((155 + (7+2)) * 2 - (7+2) = 319	((80 + (9+3)) * 1 - (9+3) = 80
large (2x2)	((155 + (7+2)) * 2 - (7+2) = 319	((80 + (9+3)) * 2 - (9+3) = 172

These sizes are what's returned by the `itemInfo2` functions in js/data.js (via `sizeMap2`), and they are accounted for in css/scenario7.css. The results (with a taller ListView as well) are shown in Figure 7-15, where I've also added numbers in the data item titles to reveal the order of item layout (and apologies for clipping the text...experiments must make sacrifices at times!). Take special note of how the layout placed items sequentially (as with 1–5) the backfilled a gap (as with 6).

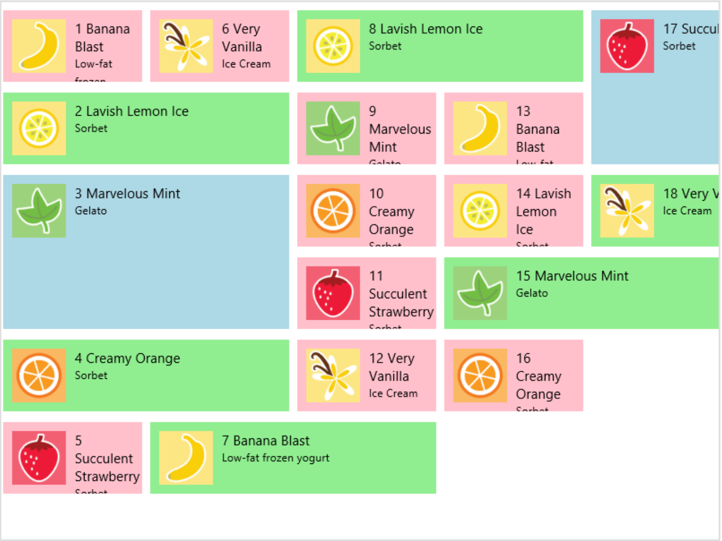

1 Banana Blast Low-fat frozen
6 Very Vanilla Ice Cream
8 Lavish Lemon Ice Sorbet
17 Succul Sorbet
2 Lavish Lemon Ice Sorbet
9 Marvelous Mint Gelato
13 Banana Blast Low-fat
3 Marvelous Mint Gelato
10 Creamy Orange Sorbet
14 Lavish Lemon Ice Sorbet
18 Very V Ice Cream
11 Succulent Strawberry Sorbet
15 Marvelous Mint Gelato
4 Creamy Orange Sorbet
12 Very Vanilla Ice Cream
16 Creamy Orange Sorbet
5 Succulent Strawberry Sorbet
7 Banana Blast Low-fat frozen yogurt

FIGURE 7-15 A horizontal `CellSpanningLayout` with varying item widths, showing infill of gaps.

To play a little with the `newColumn` property in `itemInfo`, as follows, try forcing a column break before items #7 and #15 because they span odd columns (this code is in a comment in `itemInfo2`):

```
newColumn: (index == 6 || index == 14),   //Break on items 7 and 15 (index is 6 and 14)
```

The result of this change is shown in Figure 7-16.

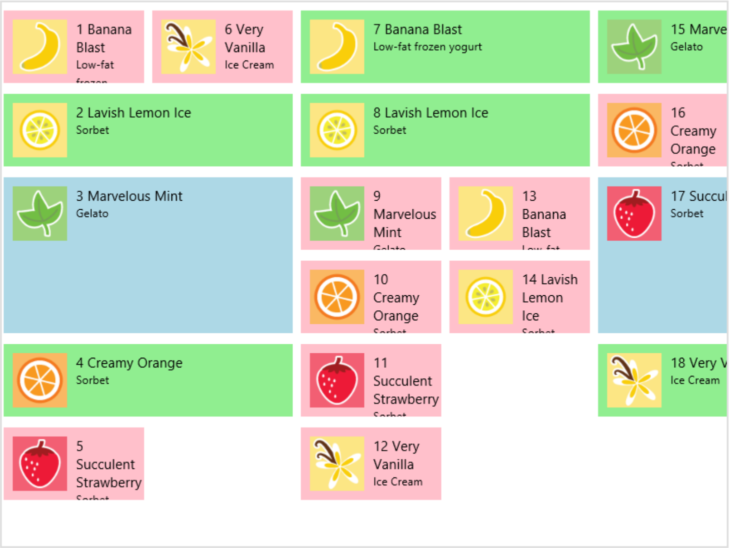

1 Banana Blast Low-fat frozen
6 Very Vanilla Ice Cream
7 Banana Blast Low-fat frozen yogurt
15 Marve Gelato
2 Lavish Lemon Ice Sorbet
8 Lavish Lemon Ice Sorbet
16 Creamy Orange Sorbet
3 Marvelous Mint Gelato
9 Marvelous Mint Gelato
13 Banana Blast Low-fat
17 Succul Sorbet
10 Creamy Orange Sorbet
14 Lavish Lemon Ice Sorbet
4 Creamy Orange Sorbet
11 Succulent Strawberry Sorbet
18 Very V Ice Cream
5 Succulent Strawberry Sorbet
12 Very Vanilla Ice Cream

FIGURE 7-16 Using new columns in cell spanning on items 7 and 15.

Three last notes: First, if you're working with cell spanning and the layout gets all wonky, double-check your match for the item sizes; even making a one-pixel mistake will throw it off. Second, if the item size in a style rule like `smallListIconTextItem` ends up being smaller than the size of a child element, such as `regularListIconTextItem` (which includes margin and padding), the larger size wins in the layout, and this can also throw things off. And third, remember that the `CellSpanningLayout` supports only a horizontal orientation; if you want a vertical experience, you'll need a custom layout. For details, refer again to Appendix B in the section "Custom Layouts for the ListView Control."

Template Functions (Part 2): Optimizing Item Rendering

Where managing, displaying, and interacting with large collections is concerned, performance is super-critical. The WinJS team invests heavily in the performance of the ListView control as a whole, but one part is always the app's responsibility: item rendering. Simply said, it's the app's sacred duty to render items as quickly as possible, which keeps the ListView itself fluid and responsive.

When we first looked at template functions or *renderers* (see "How Templates Work with Collection Controls"), I noted that they give us control over both how and *when* items are constructed. That *when* part is very important, because it's possible to render items in stages, doing the most essential work quickly and synchronously while deferring other asynchronous work like loading thumbnails. Within a renderer, then, you can implement progressive levels of optimization for ListView (and also FlipView, though this is less common). Just using a renderer, as we already saw, is the first level; now we're ready to see the others. This is a fascinating subject, because it shows the kind of sophistication that the ListView enables for us!

Note Beyond what's discussed here, refer also to the Using ListView topic in the documentation, where you'll find a variety of other performance tips.

Our context for this discussion is the HTML ListView optimizing performance sample that demonstrates all these levels and allows you to see their respective effects (also using the Bing Search API data source that we've seen elsewhere, so you'll need your API key again). Here's an overview:

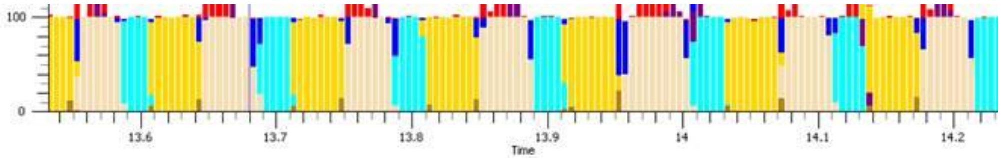
- A *simple* or basic renderer allows control over the rendering on a per-item basis and allows asynchronous delivery of item data and the rendered element.
- A *placeholder* renderer separates creation of the item element into two stages. The first stage returns only those elements that define the shape of the item. This allows the ListView to quickly do its overall layout before all the details are filled in, especially when the data is coming from a potentially slow source. When item data is available, the second stage is then invoked to copy that data into the item elements and create additional elements that don't contribute to the shape.
- A *multistage* renderer extends the placeholder renderer to defer expensive visual operations,

like loading thumbnails and running animations, until the item is visible and the ListView isn't being rapidly panned.

- Finally, a multistage *batching* renderer batches image loading to minimize re-rendering of the DOM as images become available.

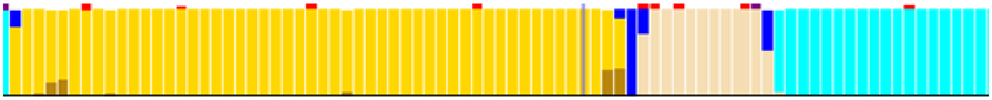
With all of these renderers, *always strive to make them execute as fast as possible*, as described earlier in "Template Functions (Part 1)." Especially minimize the use of DOM API calls, which includes setting individual properties. Use an `innerHTML` string where you can to create elements rather than discrete calls, and minimize your use of `getElementById`, `querySelector`, and other DOM-traversal calls by caching the elements you refer to most often. This will make a big difference.

To visualize the effect of these improvements, the following graphic shows an example of how unoptimized ListView rendering might happen (this is output from some of the performance tools discussed in Chapter 3):



The yellow bars indicate execution of the app's JavaScript—that is, time spent inside the renderer. The beige bars indicate the time spent in DOM layout, and aqua bars indicate actual rendering to the screen. As you can see, when elements are added one by one, there's quite a bit of breakup in what code is executing when, and the kicker here is that most display hardware refreshes only every 10–20 milliseconds (50–100Hz). As a result, there's lots of choppiness in the visual rendering.

After making improvements, the chart can look like the one below, where the app's work is combined in one block, thereby significantly reducing the DOM layout process (the beige):



Let's see what steps we can take to make this happen. As a baseline for our discussion, here is a *simple renderer*—this and all the following code is taken from js/scenario1.js:

```
function simpleRenderer(itemPromise) {
    return itemPromise.then(function (item) {
        var element = document.createElement("div");
        element.className = "itemTempl";
        element.innerHTML = "<img src='" + item.data.thumbnail +
            "' alt='Databound image' /><div class='content'>" + item.data.title + "</div>";
        return element;
    });
}
```

Again, this structure waits for the item data to become available, and it returns a promise for the element. That is, the return value from itemPromise.then is a promise that's fulfilled with element, if and when the ListView needs it. If the ListView pans the item out of view before this promise is fulfilled, it will cancel the promise.

A *placeholder renderer* separates building the element into two stages. The return value is an object that contains a minimal placeholder in the element property and a renderComplete promise that is fulfilled with the remainder of the element:

```
function placeholderRenderer(itemPromise) {
    // create a basic template for the item which doesn't depend on the data
    var element = document.createElement("div");
    element.className = "itemTempl";
    element.innerHTML = "<div class='content'>...</div>";

    // return the element as the placeholder, and a callback to update it when data is available
    return {
        element: element,

        // specifies a promise that will be completed when rendering is complete
        // itemPromise will complete when the data is available
        renderComplete: itemPromise.then(function (item) {
            // mutate the element to include the data
            element.querySelector(".content").innerText = item.data.title;
            element.insertAdjacentHTML("afterBegin", "<img src='" +
                item.data.thumbnail + "' alt='Databound image' />");
        })
    };
}
```

Note that the element.innerHTML assignment could be moved inside the function in renderComplete because the itemTempl class in css/scenario1.css specifies the width and height of the item directly. The reason why it's in the placeholder is because it provides the default "…" text, and you could just as easily provide a default in-package image here instead (which would render quickly).

In any case, the element property defines the item's shape and is returned synchronously from the renderer. This lets the ListView (or other control) do its layout, after which it will fulfill the renderComplete promise. You can see that renderComplete essentially contains the same sort of thing that a simple renderer returns, minus the already created placeholder elements. (For another example, the added scenario 9 of the FlipView example in this chapter's companion content implements this approach.)

Of course, now that we've separated the time-critical and synchronous part of element creation from the rest, we can complete the rendering in asynchronous stages, taking however long is necessary. The *multistage renderer* uses this capability to delay-load images and other media until the rest of the item is wholly present in the DOM, and to further delay effects like animations until the item is truly on-screen. This recognizes that users often pan around within a ListView quite rapidly, so it makes sense to asynchronously defer the more expensive operations until the ListView has settled into a stable position.

The hooks for this are a property called ready (a promise) and two methods, loadImage and isOnScreen, that are attached to the item provided by the itemPromise:

```
renderComplete: itemPromise.then(function (item) {
   // item.ready, item.loadImage, and item.isOnScreen available
})
```

Here's how you use them:

- ready Return this promise from the first completed handler in any async chain you might use in the renderer. This promise is fulfilled when the full structure of the element has been rendered and is visible. This means you can chain another then with a completed handler in which you do post-visibility work like loading images.
- isOnScreen Returns a promise whose fulfillment value is a Boolean indicating whether the item is visible or not. In present implementations, this is a known value, so the promise is fulfilled synchronously. By wrapping it in a promise, though, it can be used in a longer chain.
- loadImage Downloads an image from a URI and displays it in the given img element, returning a promise that's fulfilled with that same element. You attach a completed handler to this promise, which itself returns the promise from isOnScreen. Note that loadImage will create an img element if one isn't provided and deliver it to your completed handler.

The following code shows how these are used (where element.querySelector traverses only a small bit of the DOM and is a highly optimized method in the first place, so it's not a concern):

```
renderComplete: itemPromise.then(function (item) {
   // mutate the element to update only the title
   if (!label) { label = element.querySelector(".content"); }
   label.innerText = item.data.title;

   // use the item.ready promise to delay the more expensive work
   return item.ready;
   // use the ability to chain promises, to enable work to be cancelled
}).then(function (item) {
   //use the image loader to queue the loading of the image
   if (!img) { img = element.querySelector("img"); }
   return item.loadImage(item.data.thumbnail, img).then(function () {
      //once loaded check if the item is visible
      return item.isOnScreen();
   });
}).then(function (onscreen) {
   if (!onscreen) {
      //if the item is not visible, then don't animate its opacity
      img.style.opacity = 1;
   } else {
      //if the item is visible then animate the opacity of the image
      WinJS.UI.Animation.fadeIn(img);
   }
})
```

I warned you that there would be promises aplenty in these performance optimizations! But all we have here is the basic structure of chained promises. The first async operation in the renderer updates simple parts of the item element, such as text. It then returns the promise in item.ready. When that promise is fulfilled—or, more accurately, *if* that promise is fulfilled—you can use the item's async loadImage method to kick off an image download, returning the item.isOnScreen promise from that completed handler. When and if that isOnScreen promise is fulfilled, you can perform those operations that are relevant only to a visible item.

I again emphasize the *if* part of all this because it's very likely that the user will be panning around within the ListView while all this is happening. Having all these promises chained together makes it possible for the ListView to cancel the async operations any time these items are scrolled out of view and/or off any buffered pages. Suffice it to say that the ListView control has gone through a *lot* of performance testing!

Which brings us to the final multistage *batching* renderer, which combines the insertion of images in the DOM to minimize layout and repaint work. It does this by letting loadImage create the img elements for us, so they're not initially in the DOM, and then inserting batches of them until there's a 64-millisecond gap between images coming in.

The sample does this inside a renderComplete that now has this structure (most code omitted):

```
renderComplete: itemPromise.then(function (i) {
    item = i;
    // ...
    return item.ready;
}).then(function () {
    return item.loadImage(item.data.thumbnail);
}).then(thumbnailBatch()
).then(function (newimg) {
    img = newimg;
    element.insertBefore(img, element.firstElementChild);
    return item.isOnScreen();
}).then(function (onscreen) {
    //...
})
```

This is almost the same as the multistage renderer except for the insertion of a call to this mysterious thumbnailBatch function between the item.loadImage call and the item.isOnScreen check. The placement of thumbnailBatch() in the chain indicates that its return value is a completed handler that itself returns another promise, and somewhere in there it takes care of the batching.

The thumbnailBatch function itself is created by another function called createBatch, which is one of the finest examples of promise magic you'll see:

```
//During initialization (outside the renderer)
thumbnailBatch = createBatch();

//The implementation of createBatch
function createBatch(waitPeriod) {
    var batchTimeout = WinJS.Promise.as();
    var batchedItems = [];

    function completeBatch() {
        var callbacks = batchedItems;
        batchedItems = [];

        for (var i = 0; i < callbacks.length; i++) {
            callbacks[i]();
        }
    }

    return function () {
        batchTimeout.cancel();
        batchTimeout = WinJS.Promise.timeout(waitPeriod || 64).then(completeBatch);

        var delayedPromise = new WinJS.Promise(function (c) {
            batchedItems.push(c);
        });
        return function (v) { return delayedPromise.then(function () { return v; }); };
    };
}
```

This code is designed to be something you can just drop into your own apps with complete blind faith that it will work as advertised: you don't have to understand (or even pretend to understand) how it works. Still, I know some readers will be curious, so I've deconstructed it all at the end of Appendix A, "Demystifying Promises."

What We've Just Learned

- The WinJS.UI.Repeater provides a simple and lightweight means to iterate over a Binding.List and render a data-bound template for each item into a container.
- The WinJS.UI.FlipView control displays one item of a collection at a time; WinJS.UI.ListView displays multiple items according to a specific layout. Both support different options, different behaviors, and rich styling capabilities.
- Central to all collection controls is the idea that a data source exists and an item template defines how each item in that source is rendered. Templates can be either declarative or procedural.
- ListView works with the added notion of layout. WinJS provides three built-in layouts. GridLayout is a two-dimensional, horizontally or vertically panning list; CellSpanningLayout is a horizontal GridLayout that allows items to span rows or columns; ListLayout is for a one-dimensional vertically or horizontally panning list. It is also possible to implement custom layouts, which are explained in Appendix B.
- ListView provides the capability to display items in groups, organizing items into groups according to a group data source. WinJS.Binding.List provides methods to created grouped,

sorted, and filtered projections of items from a data source.

- The Semantic Zoom control (`WinJS.UI.SemanticZoom`) provides an interface through which you can switch between two different views of a data source, a zoomed-in (details) view and a zoomed-out (summary) view. The two views can be very different in presentation but should display related data. The `IZoomableView` interface is required on each of the views so that the Semantic Zoom control can switch between them and scroll to the correct item. The ListView implements this interface.
- The FlipView and ListView controls work with their data sources through the `IListDataSource` interface, allowing any kind of source (synchronous or asynchronous) to operate behind that interface. WinJS provides a `StorageDataSource` to create a collection over `StorageFile` items, and it also provides the `VirtualizedDataSource` where an object with the `IListDataAdapter` interface is used to customize its behavior.
- Procedural templates are implemented as template functions, or renderers. These functions can implement progressive levels of optimization for delay-loading images and adding items to the DOM in batches.

Chapter 8

Layout and Views

Compared to other members of my family, I seem to need the least amount of sleep and am often up late at night or up before dawn. To avoid waking the others, I generally avoid turning on lights and just move about in the darkness (and here in the rural Sierra Nevada foothills, it can get *really* dark!). Yet because I know the layout of the house and the furniture, I don't need to see much. I need only a few reference points, such as a door frame, a corner on the walls, or the edge of the bed, to know exactly where I am. What's more, my body has developed a muscle memory for where doorknobs are located, how many stairs there are, how many steps it takes to get around the bed, and so on. It's helped me understand how visually impaired people "see" their own world.

If you observe your own movements in your home and your workplace—probably when the spaces are lit!—you'll probably find that you move in fairly regular patterns. This is one of the most important considerations in home design: a skilled architect looks carefully at how people in the home might move between primary spaces like the kitchen, dining room, and living room, and even within a single workspace like the kitchen. Then the architect designs the home's layout so that the most common patterns of movement are easy and free from obstructions. If you've ever lived in a home where it wasn't designed this way, you can very much appreciate what I'm talking about!

There are two key points here: first, good layout makes a huge difference in the usability of any space, and second, human beings quickly form habits around how they move about within a spa ce, habits that hopefully make their movement more efficient and productive.

Good app design follows the same principles, which is exactly why Microsoft recommends following consistent patterns within your apps, as described on the Design Principles page and UX Patterns. Those recommendations are not in any way whimsical or haphazard: they are the result o f many years of research and investigation into what works best for apps and for Windows as a whole. The placement of the charms, for instance, as well as commands on an app bar (as we'll see in Chapter 9, Commanding UI"), arise from the reality of human a natomy, namely how far we can move our thumbs around the edges of the screen when holding a tablet device.

With page layout, in particular, the recommendations on Laying out an app page—describing where headers and body content are ideally placed, the spacing between items, and so forth—can seem rather limiting, if not draconian. The silhouette, however, is meant to be a good starting point, not a hard-and-fast rule. What's most important is that the shape of an app's layout helps users develop a visual and physical muscle memory that can be applied across many apps. Research has shown that users develop such habits very quickly, even within a matter of minutes, but of course those habits are not exact to specific pixels! In other words, the silhouette represents a general shape that helps users immediately understand how an app works and where to go for certain functions, just like you can easily recognize the letter "S" in many different fonts. This is very efficient and productive. On the other hand, when presented with an app that uses a completely different layout (or, worse, a layout that is similar to the silhouette but behaves differently), users must expend much more energy j ust figuring out where to look and where to tap, just as I would have to be much more careful late at night if you moved my furniture around!

The bottom line is that very good reasons support *all* the design principles for Windows Store apps, layout included. As I've said before, if you're fulfilling the designer role for your app, study the principles and patterns referred to above. If someone else is fulfilling that role, make sure *they* study those principles!

Either way, we'll be reviewing the key ones early in this chapter. After that, our focus will be on how we implement layout designs, not creating the designs themselves. (Although I apparently got the mix of my parent's genes that bestowed an aptitude for technical communication, my brother got most of the genes for artistry.)

With layout, make sure to remember that *the user is always in control of where each view of your app appears on the screen and how much space it has to work with*. Users can move app views around on their displays, narrowing the view down to 500px (or even 320px if the app allows it) to share display space with other apps. On a large monitor, this means having as many as four app views running side-by-side—and the same is true with each additional monitor on the system.

The display space for any view also depends on the display hardware itself—different monitors have different sizes, of course, as well as different pixel densities for which Windows might apply automatic scaling. Most displays can also be rotated, so orientation plays an important role in layout. We'll be exploring all of these factors in the first half of this chapter.

You might have noticed in the last few paragraphs that I switched from using the word *app* in this context to the word *view*. This was intentional. Windows 8.1 introduces the ability for apps to create multiple independent views that operate on independent UI threads. Each view is an additional space in which the app can display content and even maintain a separate navigation stack, and the user retains control over each view's size, placement, and lifetime (that is, the user can close a view at any time). With multiple views, then, apps can extend themselves across multiple monitors or have multiple side-by-side views on the same monitor.

That said, each view is essentially just a page container, so layout principles for any one page applies across views. In the latter half of this chapter, we'll talk about the additional layout tools you have to handle whatever conditions you encounter. This includes the WinJS hub control and various CSS capabilities, such as pannable sections, snap points, flexbox, grid, and multicolumn text.

I'll remind you again that some UI elements like the app bar and flyouts don't participate in layout; I'll cover these in other chapters. Also, auxiliary app pages that service contracts (such as Settings and Search) will exist outside your main navigation flow. These should employ the same layout principles covered in this chapter, but how and when they should appear will be covered in later chapters.

Principles of Page Layout

Page layout (in whatever view) is truly one of the most important considerations in Windows app design. The principle of "do more with less" (also referred to as "content before chrome") means that most of what you display on any given page is meaningful content, with little in the way of commanding surfaces, persistent navigation tabs, and passive graphical elements like separators, blurs, and gradients that don't in themselves mean anything. Another way of putting this is that content itself should be directly interactive rather than composed of passive elements that are acted upon when the user invokes some other command (the chrome). Semantic zoom is a good example of such interactive content—instead of needing buttons or menus elsewhere in the app to switch between views, the capability is inherent in the control itself, with the small zoom button appearing only when needed for mouse users. Other app commands, for the most part, are similarly placed on UI surfaces that appear when needed through app bars, nav bars, and other flyouts, as we'll see in Chapter 9.

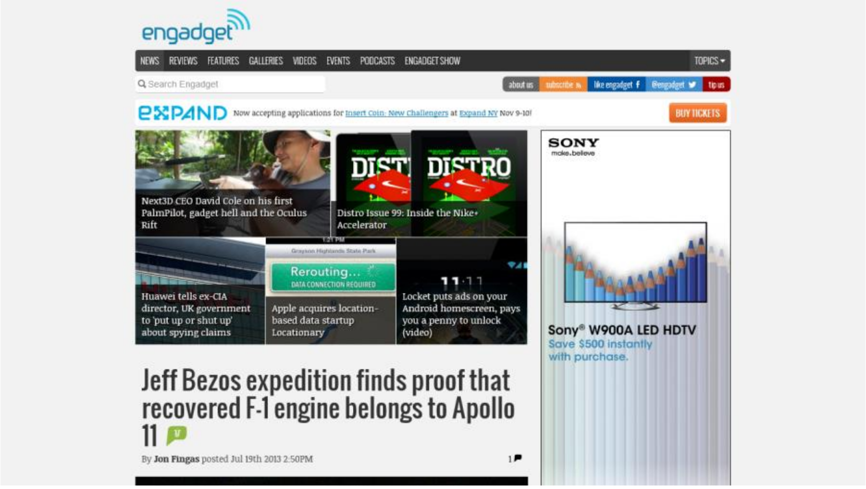
In short, "do more with less" means immersing the user in the experience of the content rather than distracting them with nonessentials. In Windows app design, then, emphasis is given to the *space* around and between content, which serves to organize and group content without the need for lines and boxes. These essentially transparent "space frames" help consumer's eyes focus on the content that really matters. Windows app design also uses typography (font size, weight, color, etc.) to convey a sense of structure, hierarchy, and relative importance of different content. That is, because the text on a page is already content, why not use its characteristics—the typography—to communicate what is often communicated with extraneous chrome? (As with the layout silhouette, the general use of the Segoe UI font within app design is not a hard-and-fast requirement but a starting point. Having a consistent *type ramp* for different headings is more important than the font.)

Figure 8-1 shows a typical web application design for a blog. Notice the persistent chrome along the top and right side: search commands, navigation tabs, navigation controls, and so forth. Perhaps only 50% of the screen space is left for content.

Figure 8-2 shows a Windows Store app design for the same content (with a single view)—in this case using the Feed reader sample in the Windows SDK. All the ancillary commands have been moved offscreen. Search is accomplished through the Search charm (or on the app bar); Settings through the Settings charm; adding feeds, refresh, and navigation through commands on the app bar; and switching views through semantic zoom. Typography is used to convey the hierarchy instead of a menu or folder structure. All this reduces distractions, leaves much more room for content, and creates a much more immersive and engaging experience, don't you think? (Note that a Search box is one control that often does appear on-canvas because of its central role in many app experiences.)

Design ideas a-plenty The Windows Developer Center has a great Inspiration section that contains a few case studies and quite a number of "idea books" for different categories of apps, from games, entertainment, and news apps to medical, enterprise administration, and educational apps. Each idea book discusses not only design but also key features that such apps would want to use. The case studies, for their part, show how to convert websites, iOS apps, and enterprise line-of-business apps to

Windows Store apps. Definitely worth a few minutes of your time to at least see what's available!



Now accepting applications for Insert Coin: New Challengers at Expand NY Nov 9-10!

Next3D CEO David Cole on his first PalmPilot, gadget hell and the Oculus Rift

Distro Issue 99: Inside the Nike+ Accelerator

Huawei tells ex-CIA director, UK government to 'put up or shut up' about spying claims

Apple acquires location-based data startup Locationary

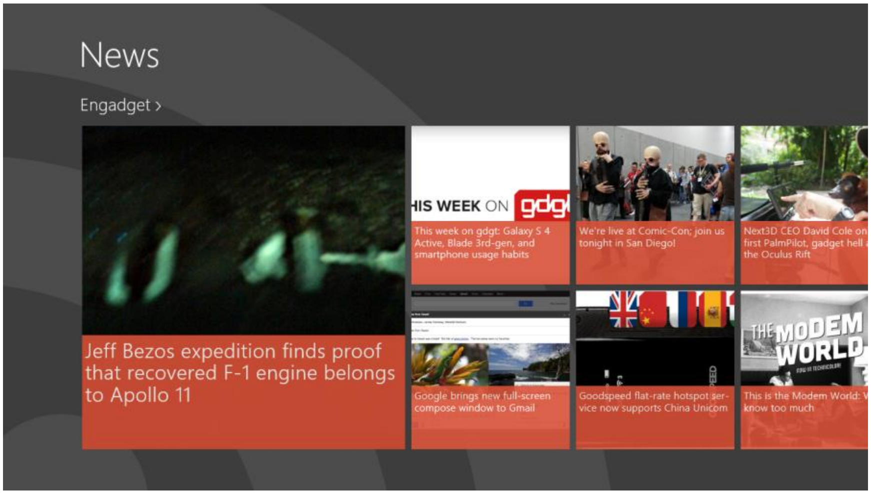
Locket puts ads on your Android homescreen, pays you a penny to unlock (video)

Jeff Bezos expedition finds proof that recovered F-1 engine belongs to Apollo 11

By Jon Fingas posted Jul 19th 2013 2:50PM

Sony® W900A LED HDTV
Save $500 instantly with purchase.

FIGURE 8-1 A typical desktop or web application design that emphasizes chrome at the expense of content.



Jeff Bezos expedition finds proof that recovered F-1 engine belongs to Apollo 11

This week on gdgt: Galaxy S 4 Active, Blade 3rd-gen, and smartphone usage habits

We're live at Comic-Con; join us tonight in San Diego!

Next3D CEO David Cole on first PalmPilot, gadget hell the Oculus Rift

Google brings new full-screen compose window to Gmail

Goodspeed flat-rate hotspot service now supports China Unicom

This is the Modem World: know too much

FIGURE 8-2 The Feed reader sample drawing on the same feed as the web app in Figure 8-1. Most of the chrome has disappeared, leaving much more space for content or visually-relaxing space.

Even where typography is concerned, Windows app design encourages the use of distinct font sizes, again called the typographic ramp, to establish a sense of hierarchy. The default WinJS stylesheets—ui-light.css and ui-dark.css—provide four fixed sizes where each level is proportionally larger than the previous (42pt = 80px, 20pt = 40px, etc.), as shown in Figure 8-3 and described more fully on Guidelines for fonts (and see also Guidelines for typography). These proportions allow users to easily establish an understanding of content structure with just a glance. Again, it's a matter of encouraging habit and muscle memory, and Microsoft's research has shown that beyond this size granularity, users are generally unable to differentiate where a piece of content fits in a hierarchy.

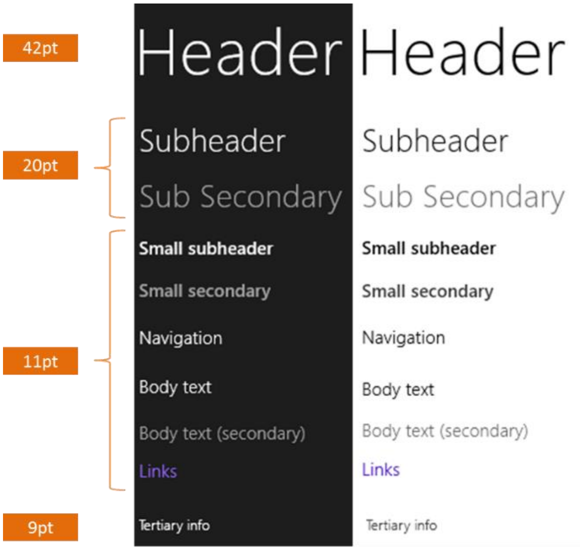


Figure 8-3 The typographic ramp of Windows Store app design, shown in both the ui-dark.css (left) and ui-light.css (right) stylesheets.

Within the body of content, Windows app design encourages these layout principles:

- Let content flow from edge to edge (full bleed).
- Keep ergonomics in mind: pan the page along the long edge of the view (primarily horizontal in landscape aspect ratios, vertical in portrait aspect ratios).
- Pan on a single axis only to create a sense of stability and to support swiping to select (as with the ListView control), or employ rails to limit panning directions to a single axis.
- Create visual alignment, structure, and clarity with the page silhouette, aligning elements on a grid for consistency. Refer again to Laying out an app page. This shape is what allows a consumer's eyes to recognize something as a Store app without having to think about it, which provides a feeling of familiarity and confidence.

As I've mentioned before, the project templates in Visual Studio and Blend have these principles baked right in and thus provide a convenient starting point for apps. Even if you start with the Blank App template, the others like the Grid App and Hub App will serve as a reference point. This is exactly what we did with the Here My Am! app in Chapter 2, "Quickstart."

Another important guiding principle that's relevant to layout is "scaling beautifully on any size screen." What we've talked about in this section has been the design principles for laying out a page. Scaling beautifully, on the other hand, means understanding the conditions that affect and drive the layout: view sizes, display resolutions, and pixel densities. This means making sure you design every page in your app to appropriately handle all these concerns and then using the tools you have at your disposal to implement those designs, as we'll see next.

The last piece to consider with layout in general is that it's often not a static concern only: the ease of creating animations means that a dynamic relationship can exist between a page and the content on that page. Keep this in mind with your designs; animation can certainly affect the personality of your app and the user's enjoyment of your app.

Sizing, Scaling, and Views: The Many Faces of Your App

If there's one certainty about layout for a Windows Store app, it's that the display space for each of its views (whether one or multiple) will likely change during a single app session and change frequently. For one, auto-rotation—especially on tablet devices—makes it quick and simple for the user to switch between landscape and portrait orientations. Second, a device might be connected to an external display or a user might move an app view from one display to another, meaning that the view needs to adjust itself to different resolutions and pixel densities on the fly. Third, users can change display settings through PC Settings > PC & Devices > Display, including orientation and effective scaling. Fourth, users have the ability to arrange app views to share space on the same monitor, meaning that any given view can be sized down to 500 horizontal pixels or even 320px if the app indicates such support in its manifest. (Resizing is accomplished using touch/mouse gestures or the Windows+. [period] and Windows+> [shift+period] keystrokes.) And finally, an app can create multiple independent views that the user can place on separate monitors, where each monitor can again have different display characteristics.

You definitely want to test each view of your app with all of these variances: view sizes, orientations, display resolutions, and pixel densities. Different app sizes can be tested directly on any given machine, but for the others you'll need access to a variety of hardware or you can use the Visual Studio simulator and the Device tab of Blend to simulate the most canonical conditions. But however you test these variations, the big question is how to write an app that can handle them.

Layout performance tips The Managing layout efficiently topic in the documentation has some helpful tips to improve layout performance in your app. One is to recognize that accessing and setting certain properties and styles—specifically those that affect placement and visibility of an element—will trigger a layout pass. As such, it helps to write code such that these changes are batched together. The second recommendation is to create and initialize elements before adding them to the DOM, or, if that's unavoidable, to hide the element by setting the `display: none` style rather than setting visibility or opacity. Only `display: none` will remove an element from layout passes.

Variable View Sizing and Orientations

We already got an introduction to view sizing in Chapter 1, "The Life Story of a Windows Store App" (see Figure 1-6), and we encountered the basics in Chapter 2. To review what we've learned and fill out the story, here are the conditions that affect view size and the basic guidelines for responding to them:

- Regardless of width, views always span the full height of the display, in either orientation. The minimum height is always 768px.
- When the *device* is in portrait mode, meaning the physical display's *aspect ratio* is taller than it is wide, only one view can appear on the screen at a time (implying that multiple views is for landscape and multimonitor scenarios). Portrait mode is very common with small tablet devices, which are often called *portrait first* devices.
- When the device is in landscape mode, multiple views from multiple apps can share the horizontal screen space. The number is determined by the total width and the minimum size of the views involved.
- By default, views can be sized down to 500px wide. This value allows two 500px-wide views to run side-by-side on a 1024x768 display (the smallest on which Windows will run), with a 24px gutter between them. Views should always be fully functional down to 500px—that is, all of its features are still accessible, though an app can collapse command structures, remove labels, or otherwise tighten up the UI as needed.
- With a minimum 768px height, resizing a view to a width narrower than 768px will change the effective aspect ratio from landscape to portrait. The app decides, though, at what width it might change its layout from a landscape-oriented model (horizontally panning) to a portrait-oriented model (vertically panning).
- If an app has the Minimum Width in its manifest set to 320px, the user can size views down to that narrower size. Use this optional setting only if a narrow width makes sense for your app as a whole (and it's appropriate to display reduced functionality at this width). At the same time, supporting the narrow width increases the likelihood that a user will keep the view visible alongside others, especially on larger displays. You can also consider using extra vertical space to display other content when the primary content does not fill the space. For example, a video app could show additional information and recommendations that wouldn't be shown when playing the video in full screen landscape.
- Views can be sized *up* from these minimums to the full extent of the current display, which can be very large. Views need to adapt themselves to more space by reflowing content, showing more content, and/or scaling content to larger sizes.
- Apps can be launched directly into different widths at the request of other apps (affecting the initial size of the primary view).
- An app does *not* have programmatic control over view width. The user always controls resizing.[70]

[70] The `tryUnsnap` API that existed in Windows 8 is deprecated in Windows 8.1 and does nothing.

- An app can specify Supported Rotations in the manifest—these specifically affect whether an app can be launched without affecting the device rotation and how the app is notified when rotations happen. Apps can also lock the orientation at run time to prevent device orientation changes while that app is in the foreground.

As a result of these conditions, *every page of an app, in each view*—including an extended splash screen—*must be prepared to handle arbitrary sizes and aspect ratios down to the app's supported minimum*. Repeat this like a mantra because it's easy to forget when you're just developing and testing an app on a single device. And make sure your designers are thinking about it too, because when your app gets out to thousands of customers, they will certainly be exercising all the possibilities! (For more design details, especially about reducing functionality in narrow views, see Guidelines for resizing windows to tall and narrow layouts.)

It should also be obvious that resizing a view never changes the *mode* of the app in that view or causes it to navigate to another page. That is, *always* maintain the state of the view across size boundaries. Otherwise users will become very confused about where they are in your app experience!

Let's now go into the details of a few areas. First we'll explore how to handle size changes, we'll compare adaptive and fixed layout strategies, and then we'll see how we work with device orientations. Later on we'll talk about scaling considerations, but as those factors come back to the app as an effective view size, they aren't directly important for layout decisions. The same goes for creating and managing views, which are again just additional page containers where each page has the same layout concerns as we're discussing here.

Sidebar: View Properties in the ApplicationView Object

The `ApplicationView` object in the `Windows.UI.ViewManagement` namespace provides a number of interesting properties for any given view. You obtain this object by calling its static `getForCurrentView` method:

```
var view = Windows.UI.ViewManagement.ApplicationView.getForCurrentView();
```

Each view as an `id` used to identify it when an app uses multiple views (see "Multiple Views" later on), as well as a `title` that Windows will show when switching between views. The `isFullScreen`, `isOnLockScreen`, and `isScreenCaptureEnabled` flags serve obvious purposes, the latter being important when an app displays rights-protected content. Two other Boolean flags, `adjacentToLeftDisplayEdge` and `adjacentToRightDisplayEdge` tell you where the view is specifically located on the screen, allowing you to make specific layout decisions for those cases if needed. The `orientation` property (a value from the `ApplicationViewOrientation` enumeration) can be `landscape` and `portrait`. This along with the full screen and edge properties are demonstrated in the Application Views sample, if you're interested.

The view object has one other member, the `consolidated` event, that is fired when the view is closed, as when the user executes a close gesture (a swipe down or Alt+F4).

Handling Size Changes

Designing an app for different sizes, as noted previously, is a matter of thinking through the user experience for full screen landscape and portrait views, partial landscape views (landscape aspect ratio), partial portrait views (narrow views down to 500px with a portrait aspect ratio), and possibly narrow views below 500px. We did this in Chapter 2 with the Here My Am! app. Once you get to the implementation details, however, handling size changes in each view of an app is very much the same story as *responsive design* for web pages and generally doesn't need to be more complicated than that.

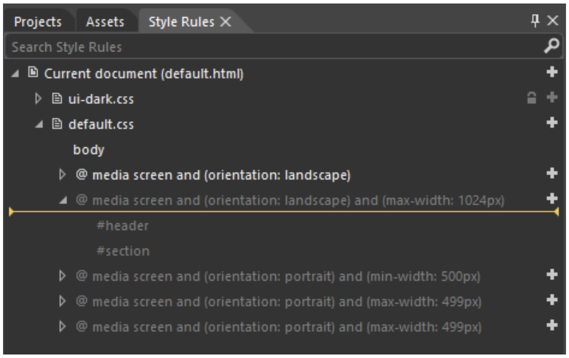
Responsive design has two parts. First are those things you can handle declaratively in CSS:

- Place all size-independent and default styles outside of any media queries. Apps typically specify styles for their preferred full screen orientation, such as landscape, as the default.
- For size-specific styles, use media queries with appropriate combinations of `orientation`, `min-width`, `max-width`, `min-height`, and `max-height` media features to isolate different layout cases. For example:

```
/* Default styles here */

@media screen and (orientation: landscape) and (max-width: 1024px) {
    /* Styling for smaller landscape layouts */
}

@media screen and (orientation: portrait) and (min-width: 500px) {
    /* Styling for portrait aspect ratios (width/height < 1)*/
}

@media screen and (orientation: portrait) and (max-width: 499px) {
    /* Styling for narrow portrait aspect ratios (width/height < 1)*/
}

@media screen and (orientation: landscape)
and (min-width: 1600px) and (min-height: 1200px) {
    /* Styling for landscape layouts on larger displays */
}

@media screen and (orientation: portrait)
and (min-width: 1200px) and (min-height: 1600px) {
    /* Styling for portrait layouts on larger displays */
}
```

- The CSS for each size typically changes placement of elements within CSS grids, the flow directions within a CSS flexbox, and element `display` styles (to show or hide those elements).
- In CSS there are also variables for the viewport height and viewport width: `vh` and `vw`. You can prefix these with a percentage number—for example, `100vh` is 100% of the viewport height, and `3.5vw` is 3.5% of the viewport width. These variables can also be used in CSS `calc` expressions.

Remember when styling your app in Blend that there's a visual affordance in the Style Rules pane that lets you control the exact insertion point of any new CSS styles in the given stylesheet. This is very helpful when working with the specific media queries:



Current document (default.html)
ui-dark.css
default.css
body
@ media screen and (orientation: landscape)
@ media screen and (orientation: landscape) and (max-width: 1024px)
#header
#section
@ media screen and (orientation: portrait) and (min-width: 500px)
@ media screen and (orientation: portrait) and (max-width: 499px)
@ media screen and (orientation: portrait) and (max-width: 499px)

Tip With media queries you might be tempted to use `<link media=>` tags in page headers to keep styles separate. Except for printing, this doesn't typically work because you're not reloading the page when a size change happens and the media-specific stylesheets won't be loaded.

The second part of responsive design involves those things that you can change only from JavaScript. For example, to change the panning direction of a ListView or Hub control, you need to change its appropriate `orientation` property (or switch a ListView from a `GridLayout` to a `ListLayout`). You might also change a list of buttons to a single drop-down `select` element to offer the same functionality through a more compact UI. These things can't be done through CSS. (Note that some controls like the `WinJS.UI.AppBar` handle size changes automatically, as we'll see in Chapter 9.)

There are two main events you can use to trigger such layout changes. First is `window.onresize`, of course (as well as `resize` events that controls like the Hub and ListView raise), where you can obtain exact dimensions of the current view through the `window.innerWidth` and `window.innerHeight` properties. The `document.body.clientWidth` and `document.body.clientHeight` properties will be the same, as will be the `clientWidth` and `clientHeight` properties of any element (like a `div`) that occupies 100% of the document body. Within the `window.onresize` event, the `args.view.outerWidth` and `args.view.outerHeight` properties are also available.

Tip When the user initiates resizing, the active app will receive the `window.onblur` event. Games often use this event to pause themselves while resizing takes place.

The other event comes from the standard Media Query Listener API in JavaScript. This interface (part of the W3C CSSOM View Module, see http://dev.w3.org/csswg/cssom-view/) allows you to add event handlers for media query state changes, such as:

```
var mql = window.matchMedia("(orientation: portrait)");
mql.addListener(styleForPortrait);

function styleForPortrait() {
    if (mql.matches) {
        //...
    }
}
```

You can see that the media query strings you pass to `window.matchMedia` are the same as used in CSS directly, and in the handler you can, of course, perform whatever actions you need from JavaScript.

> **And yet another tip** You might find cases where you want to check your layout on the `resuming` event, as display characteristics might have changed while an app was suspended. You'll get a `resize` event when a view is resumed to a different size, of course, but if the app is resumed to the same size as before, you won't see the event.

Adaptive and Fixed Layouts

As just described, app design has to consider sizes from 500px wide (or 320px wide) by 768px high, all the way up to 3200px by 1800px QHD displays. How does one approach this wide range of possibilities? A design typically starts with a baseline experience for the common 1366x768 size. Here each view should be fully functional, of course, and display enough content to be really engaging. Going down from there to smaller sizes, a view need to retain its functionality to the 500px width, collapsing and/or changing some of the UX controls along the way. If it goes down to a 320px width, it can design a more focused experience on the assumption that the view will occupy a narrow space alongside a number of other apps. In such cases you might elect to hide some functionality.

Going up from the baseline is a different story. Of course the view will continue to be fully functional, but now the question becomes, "What do you do with more space?" On this subject, I recommend you read the Guidelines for scaling to screens, which has good information on the kinds of display sizes your app might encounter as well as guidance on what to do with additional real estate when you have it.[71]

The first part of the answer is "Fill the view!" Nothing looks more silly than an app running on a 27" monitor that was designed and implemented with only 1366x768 in mind, because it will occupy only a quarter to half of the screen at best. As I've said a number of times, imagine the kinds of reviews and ratings your app might be given in the Windows Store if you don't pay attention to details like this!

The second part of the answer depends on your app's content for the view in question. If you have only fixed content, which is common with games, you'll want to use a fixed layout that scales up (and down) to the window size. If you have variable content, meaning that you should show more when

[71] Another good resource is session 2-150 from the //build 2013 conference, entitled "Beautiful apps at any size screen," and the Mail app, used as a demo in that session, provides a strong example of dynamic adaptation.

there's more screen space, you want to use an adaptive layout.

To implement a fixed layout, start by designing around a minimum size and a fixed aspect ratio, such as 1024x768 (a 4:3 aspect ratio) or 1366x768 (a 16:9 aspect ratio).[72] You then code against these fixed coordinate systems regardless of actual view size. That is, wrap your content into a `div` that's styled to your fixed dimensions. I've also added a `body` background color here to see the letterboxing:

```
<!-- In markup -->
<div id="viewbox" class="fixedlayout">
    <p>Content goes here</p>
</div>

/* In CSS */
body {
    background-color: #8e643c;
}

.fixedlayout {
    height: 768px;
    width: 1024px;
}
```

Then add a listener for `window.onresize` events and apply a CSS 2D scaling transform to the root element based on the difference between the reference size and the actual size:

```
window.onresize = resizeLayout;

function resizeLayout(e) {
    var viewbox = document.getElementById("viewbox");

    var w = window.innerWidth;
    var h = window.innerHeight;
    var bw = viewbox.clientWidth;
    var bh = viewbox.clientHeight;
    var wRatio = w / bw;
    var hRatio = h / bh;
    var mRatio = Math.min(wRatio, hRatio);
    var transX = Math.abs(w - (bw * mRatio)) / 2;
    var transY = Math.abs(h - (bh * mRatio)) / 2;
    viewbox.style["transform"] =
        "translate(" + transX + "px," + transY + "px) scale(" + mRatio + ")";
    viewbox.style["transform-origin"] = "top left";
}
```

This preserves the aspect ratio and works to scale the contents up as well as down. The translation included with the transform makes sure the body background shows around any part that the fixed content doesn't occupy.

[72] Some apps might be able to adjust their aspect ratio and fill the window, but more commonly the ratio remains fixed and extra space is filled with letterboxing. Note also that WinJS 1.0 provided a ViewBox control that does what I'm showing here, but it was removed in WinJS 2.0 because it was seldom used and is very simple to replicate.

You can find an example of this in the FixedLayout project in the companion content. This app draws a 4x3 grid of circles on a canvas, as shown in Figure 8-4, to match the 1024x768 layout size. On a 1366x768 display or larger with a 16:9 aspect ratio, we get letterboxing on the sides; in the 320px narrow view (as set in the manifest), we get letterboxing on the top.

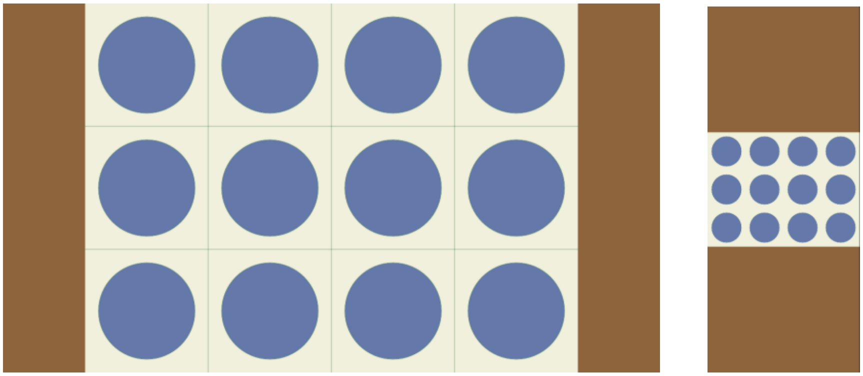


FIGURE 8-4 Fixed layout scaling with a simple CSS transform, showing letterboxing on a full-screen 16:9 ratio display (left) and in a narrow 320px view (right).

Sidebar: Raster Graphics and Fixed Layouts

If you use raster graphics within a fixed layout, size them according to the maximum 2560x1440 resolution so that they'll look good on the largest screens and they'll still scale down to smaller ones (rather than being stretched up). Alternately, you can load different graphics (through different `img.src` URIs) that are better suited for the most common screen size.

Note that resolution scaling (discussed later in this chapter) will still be applicable. If you're running on a high-density 10.6" 2560x1440 display (180% scale), the app and thus the fixed content will still see smaller screen dimensions. But if you're supplying a graphic for the native device resolution, it will look sharp when rendered on the screen.

In contrast to a fixed layout, where you always see the same content, an adaptive layout is one in which a view shows more stuff when more screen space is available. Such a layout is most easily achieved with a CSS grid where proportional rows and columns will automatically scale up and down; elements within grid cells will then find themselves resized accordingly. This is demonstrated in the Visual Studio/Blend project templates, especially the Grid App project. On a typical 1366x768 display you'll see a few items on a screen, as shown at the top of Figure 8-5. Switch over to a 27" 2560x1440 and you'll see a lot more, as you can see at the bottom of the figure.

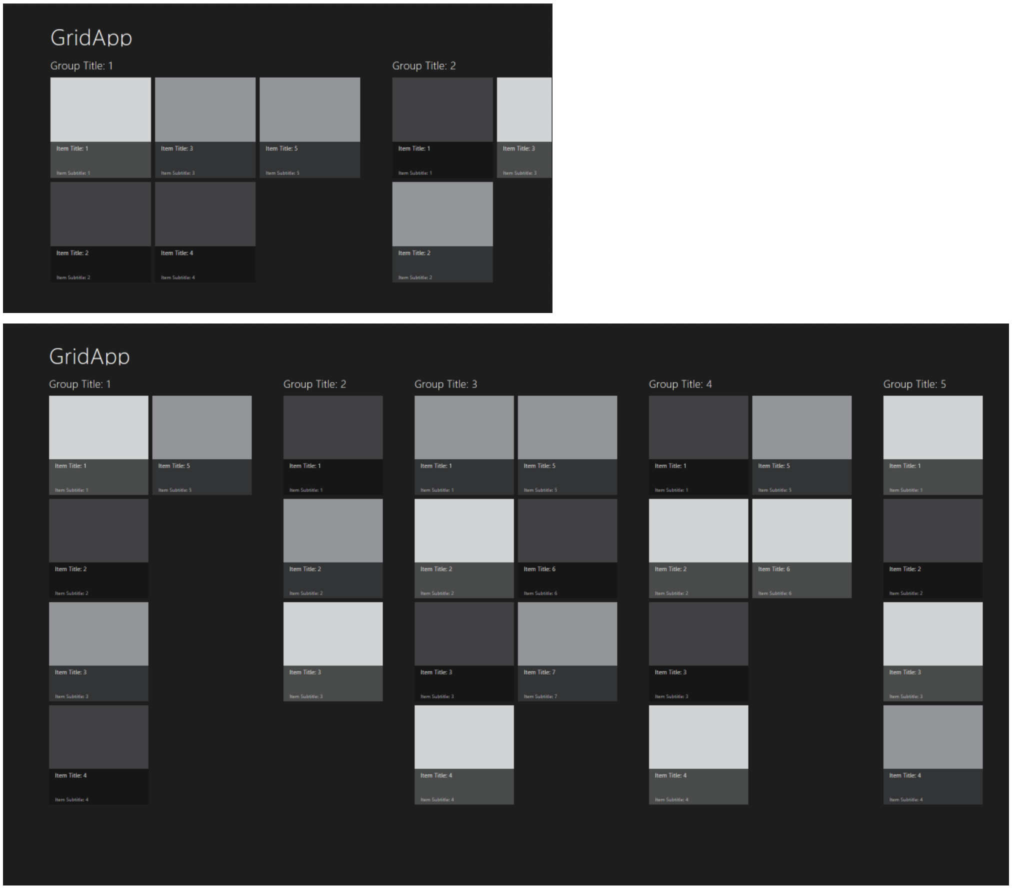


FIGURE 8-5 Adaptive layout in the Grid App project template shown for a 1366x768 display (top) and a 2560x1440 display (bottom).

To be honest, the Grid App project template doesn't do anything particularly special for view sizes. Because it uses CSS grids and proportional cells, the cell containing the ListView control automatically becomes bigger. The ListView control is listening for `window.onresize` on its own, and it reflows itself to show more items when it has more space; we don't need to instruct it directly. In any case, the Grid App template does demonstrate the basic strategy:

- Use a CSS grid (or flexbox) where possible to handle adaptive layout automatically.
- Listen for `window.onresize` as necessary to reposition and resize elements manually, such as an HTML `canvas` element.
- Have controls listen to `window.onresize` to adapt themselves directly.

As another reference point, refer to the Adaptive layout with CSS sample, which takes the same approach as the Grid App project template, relying on controls to resize themselves. In the sample, you will see that the app isn't doing any direct calculations based on view size.

> **Hint** If you have an adaptive layout and want a background image specified in CSS to scale to its container (rather than being repeated), style `background-size` to either `contain` or `100% 100%`.

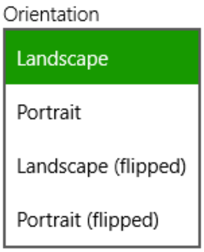
Finally, let me again recommend the Guidelines for scaling to screens topic in the documentation, which goes into many more design questions, such as:

- Which regions of a view are fixed and which are adaptive?
- How do adaptive regions make use of available space, including the directions in which that region adapts?
- How do adaptive and fixed regions relate in the wireframe?
- How does the view's layout overall make use of space—that is, how does whitespace itself expand so that content doesn't become too dense? This can mean collapsing some areas of the UI in narrower views to lighten the density when there's less room.
- How does the view make use of multicolumn text?

Answering these sorts of questions will help you understand how the layout should adapt.

Handling Orientations

If you've worked with any kind of accelerometer-equipped device, you know that one of the most natural things to do is rotate it 90 degrees in some direction to reorient an app. On other systems, the user can change orientation manually through PC Settings > PC & Devices > Display > Orientation:



Orientation

Landscape

Portrait

Landscape (flipped)

Portrait (flipped)

When this happens, an app receives a `window.onresize` event and is expected to update its view accordingly. A key point here is that landscape orientations support multiple side-by-side views, but portrait does not. In fact, Windows locks the orientation when you have multiple apps showing side by side. To switch to portrait, an app must be running full screen by itself.

If you structure your CSS appropriate for different view states and handle resize events as needed, you typically don't care about the actual physical orientation of the device—you care about only the size of the view in which you need to lay out your content. Still, you can determine the physical orientation a number of ways. One is through the `ApplicationView.orientation` property that was noted earlier in "Sidebar: View Properties in the ApplicationView Object." It has possible values of `portrait` and `landscape`.

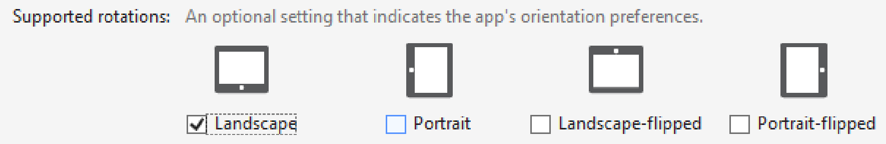
A more specific value comes from the `Windows.Graphics.Display.DisplayInformation` class (obtained via `DisplayInformation.getForCurrentView()`) and its `currentOrientation` property. This identifies one of the four rotation quadrants that are possible for a device relative to its native orientation: `landscape`, `portrait`, `landscapeFlipped`, and `portraitFlipped`. There's also an `orientationchanged` event if you need it (see scenario 3 of the Display orientation sample for usage of this event).

Thirdly, the APIs in `Windows.Devices.Sensors`—specifically, the `SimpleOrientationSensor` and `OrientationSensor` classes—can provide more information from the hardware itself, including exact rotation angles. These are covered in Chapter 12, "Input and Sensors."

> **Note** Internet Explorer 11 and the app host for Windows 8.1 also support the W3C orientation standards for orientation on the `screen` object through the `msOrientation` property, the `onmsorientationchange` event, and the `msLockOrientation` and `msUnlockOrientation` methods.

Some apps, of course, would prefer to *not* have their views resized in response to accelerometer-induced rotation. A full screen video player, for example, wants to remain in landscape irrespective of the device orientation, allowing you to watch videos while laying sideways on a couch with a tablet propped up on a chair![73]

Locking the orientation can be done in two places. First, you can specify the orientations the app supports in the app manifest on the Supported Rotations on the Application tab:



Supported rotations: An optional setting that indicates the app's orientation preferences.

☑ Landscape □ Portrait □ Landscape-flipped □ Portrait-flipped

By default, all these options are unchecked which means the app will be resized (and thus redrawn) when the device orientation changes (and checking all of them means the same thing). When you check a subset of the options, the following effects apply:

- If the device is not in a supported rotation when the app is launched, the app is launched in the nearest supported rotation. For example, if you check the Portrait and Landscape-flipped rotations (I have no idea why you'd choose that combination!) and the device is in Landscape mode, the app will launch into Portrait.

[73] Apps like movie players, aside from locking the orientation, typically need to keep the screen on even when the user hasn't interacted with the system for a long time. For this, see the `Windows.System.Display.DisplayRequest` API.

- When the app is in the foreground, rotating the device to a nonsupported orientation has no effect on the app.
- When the user switches away from the app, the device orientation is restored unless the new foreground app also has preferences. Of course, it's likely that the user will have rotated the device to match the app's preference, in which case that's the new device orientation.
- When the user switches back to the app, it will switch to the nearest supported rotation.
- In all cases, when the app's preference is in effect, system edge gestures work relative to that orientation—that is, the left and right edges are relative to the app's orientation.

You can achieve the same effects at run time—changing preferences dynamically and overriding the manifest—by setting the `autoRotationPreferences` property of the aforementioned `DisplayInformation` class. The preference values come from the `DisplayOrientations` enumeration and can be combined with the bitwise OR (`|`) operator. For example, here are the bits of code to set a Portrait preference and the combination of Portrait and Landscape-flipped:

```
var wgd = Windows.Graphics.Display;

wgd.DisplayInformation.autoRotationPreferences = wgd.DisplayOrientations.portrait;
wgd.DisplayInformation.autoRotationPreferences =
    wgd.DisplayOrientations.portrait | wgd.DisplayOrientations.landscapeFlipped;
```

> **Note** Orientation preferences set in the manifest and through the `autoRotationPreferences` property *do not work* with nonaccelerometer hardware, such as desktop monitors and also the Visual Studio simulator (they are ignored). The rotations that the simulator performs happen on the level of the display driver, which is different than the WinRT display orientation. To test these preferences, in other words, you'll need a real accelerometer-equipped device.

To play around with these settings, refer to the Device auto rotation preferences sample. Its different scenarios set one of the orientation preferences so that you can see the effect when the device is rotated; scenario 1 for its part clears all preferences and restores the usual behavior for apps. You can also change rotation settings in the manifest to see their effects, but those are again overridden as soon as you change the preferences through any of the scenarios.

For a live demonstration of this sample, see Video 8-1. This is definitely one feature that you just can't show in static images!

Screen Resolution, Pixel Density, and Scaling

I don't know about you, but when I first read that the minimum view widths were *always* 500 and 320 pixels—real pixels, not a percentage of the screen width—it set me wondering. Wouldn't that give a significantly different user experience on different monitors? The answer is actually no. Consider the narrow 320px width. 320 pixels is about 25% of the baseline 1366x768 target display, which means that the remaining 75% of the screen is a familiar 1024x768 . And on a 10-inch screen, it means that this narrow width is about the 2.5 physical inches wide. So far so good.

With a large monitor, on the other hand, let's say a 2560x1440 monster, those 320 pixels would only be 12.5% of the width, so the layout of the whole screen looks quite different. However, given that such monitors are in the 24-inch range, those 320 pixels still end up being about 2.5 physical inches wide, meaning that the narrow view gives essentially the same visual experience as before, just now with much more vertical space to play with and much more remaining screen space for other app views.

This now brings up the question of *pixel density*—what happens if your app ends up on a really small screen that also has a very high resolution, like a 7″ 1920x1200 (323dpi) screen? Obviously, 320 pixels on the latter display would be barely an inch wide. Anyone got a magnifying glass? And as new and even higher density displays are produced in the years ahead, it seems like the problem will just get harder to deal with.

Fortunately, this isn't anything a Store app has to worry about…almost. The main user benefit for such displays is greater sharpness, not greater density of information. Touch targets need to be the same size on any size display no matter how many pixels it occupies, because human fingers don't change with technology! To accommodate this, Windows automatically scales down the effective resolution that's reported to apps, which is to say that whatever coordinates you use within your app (in HTML, CSS, and JavaScript) are automatically scaled *up* to the necessary device resolution when the UI is actually rendered. This happens within the low-level HTML/CSS rendering engine in the app host so that everything is drawn directly against native device pixels for maximum sharpness.

As for the "almost" above, the one place where you do need to care about pixel density is with raster graphics, as we discussed in Chapter 3, "App Anatomy and Performance Fundamentals," for your splash screen and tiles. We'll return to this shortly in "Graphics that Scale Well" below.

Display sizes and pixel densities can both be tested again using the Visual Studio simulator or the Device tab in Blend. The latter, shown in Figure 8-6, indicates the applicable DPI and scaling factor. 100% scale means the device resolution is reported directly to an app. 140% and 180%, on the other hand, indicate that scaling is taking place. Doing a little math, you can see that the effective resolution that the app sees is generally near the standard 1366x768 or 1024x768 sizes:

Display Size	Physical Resolution	Scaling	Effective Resolution
7″	1920x1200	140% (1.4)	1371x857 (1920/1.4 by 1200/1.4)
7.5″	1440x1080	140% (1.4)	1028x771
10.6″	1920x1080	140% (1.4)	1371x771
10.6″	2560x1440	180% (1.8)	1422x800

In all these cases, layouts designed for 1024x768 and 1366x768 are quite sufficient. Of course, you'll need to adapt to the exact pixel dimensions and to the larger 100% resolutions, but you don't have to worry about resizing touch targets and such based on pixel density—all that happens transparently.

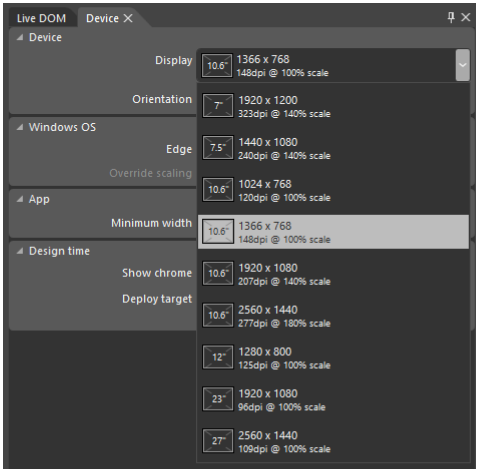
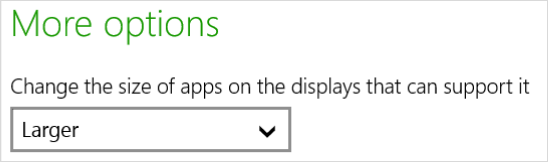


FIGURE 8-6 Options for display sizes and pixel densities in Blend's Device tab.

Scaling can also happen on 100% displays if the user goes to PC Settings > PC & Devices > Display and sets the Larger option in the control shown below. This sets the scaling to 140% and is enabled only for larger monitors where the effective resolution would not fall below 1024x768.



More options

Change the size of apps on the displays that can support it

Larger

As noted earlier, the effective/scaled resolution is what you'll see for a view in the `window.innerWidth` and `window.innerHeight` properties, the `document.body.clientWidth` and `document.body.clientHeight` properties, and the `clientWidth` and `clientHeight` properties of any element that occupies 100% of the body. Within `window.onresize`, you can also use these (or the `args.view.outerWidth` and `args.view.outerHeight` properties) to adjust the app's layout for changes in the overall display. Of course, if you're using something like the CSS grid with fractional rows and columns to do your layout, much of that will be handled automatically.

In all cases, these dimensions will already reflect automatic scaling for pixel densities, so they are the dimensions against which to do layout. If you want to know the physical *display* dimensions, on the other hand, you'll find these in the `window.screen.width` and `window.screen.height` properties. Other aspects of the display can be found in the `DisplayInformation` class (in `Windows.Graphics.Display`) as we used before for orientation. The properties of interest here are `logicalDPI` and the current `resolutionScale`. The latter is a value from the `ResolutionScale` enumeration, one of `scale100Percent`, `scale140Percent`, and `scale180Percent`. The actual values of these identifiers are 100, 140, and 180 so that you can use `resolutionScale` directly in calculations. Also, the `logicaldpichanged` event tells you when the scaling factor changes, as when the user changes the app's size option in PC Settings or switches to a different display.

Sidebar: A Good Opportunity for Remote Debugging

Working with different device capabilities provides a great opportunity to work with remote debugging as described on Running Windows Store apps on a remote machine. This will help you test your app on different displays without needing to set up Visual Studio on each one, and it also gives you the benefit of multimonitor debugging. You only need to install and run the remote debugging tools on the target machine and make sure it's connected to the same local network as your development machine. The Remote Debugging Monitor running on the remote machine will announce itself to Visual Studio running on your development machine. Note that the first time you run the app remotely, you'll be prompted to obtain a developer license for that machine, so it will need to be connected to the Internet during that time.

Graphics That Scale Well

In addition to layout, variable view sizes and pixel densities present a challenge to apps in making sure that graphical assets always look their best. You can certainly draw graphics directly with the HTML5 canvas, but oftentimes that's not possible and you have to use predrawn assets of some kind.

HTML5 scalable vector graphics (SVGs) are very handy here. You can include inline SVGs in your HTML (including page fragments), or you can keep them in separate files and refer to them in an img.src attribute. One of the easiest ways to use an SVG is to place an img element inside a proportionally sized cell of a CSS grid and set the element's width and height styles to 100%. The SVG will then automatically scale to fill the cell, and since the cell will resize with its container, everything is handled automatically.

One caveat with this approach is that the SVG will be scaled to the aspect ratio of the containing grid cell, which isn't always what you want. To control this behavior, make sure the SVG has viewBox and preserveAspectRatio attributes where the viewBox aspect ratio matches that defined by the SVG's width and height properties:

```
<svg
    xmlns:svg="http://www.w3.org/2000/svg"
    xmlns="http://www.w3.org/2000/svg"
    xmlns:xlink="http://www.w3.org/1999/xlink"
    version="1.0"
    width="300"
    height="150"
    viewBox="0 0 300 150"
    preserveAspectRatio="xMidYMid meet">
```

Of course, you don't always have nice vector graphics. Bitmaps that you include in your app package, pictures you load from files, and raster images you obtain from a service won't be so easily scalable. In these cases, you'll need to be aware of and apply the current scaling factor appropriately.

For assets in your app package, we already saw how to work with varying pixel densities in Chapter 3 through the .scale-100, .scale-140, and .scale-180 file name suffixes (the Here My Am! App has such variants). These work for any and all graphics in your app, just as they do for the splash screen, tile images, and other images referenced by the manifest. So, if you have a raster graphic named banner.png, you'll create three graphics in your app package called banner.scale-100.png, banner.scale-140.png, and banner.scale-180.png. You then just refer to the base name as in `<img src= "images/banner.png">` and `background-image: url('images/banner.png')`, and the Windows resource loader will magically load the appropriately scaled graphic automatically. (If files with .scale-* suffixes aren't found, it will look for banner.png directly.) We'll see even more such magic in Chapter 19, "Apps for Everyone, Part 1," when we also include variants for different languages and contrast settings that introduce additional suffixes of their own.

If your developer sensibilities object to this file-naming scheme, know that you can also use similarly named folders instead. That is, create scale-100, scale-140, and scale-180 folders in your images folder and place appropriate files with unadorned names (like banner.png) therein.

> **Tip** If you have an app with a fixed layout, you can address pixel density issues by simply using graphical assets that are scaled to 200% of your standard design. This is because a fixed layout can be scaled to arbitrary dimensions, so a 200% image scales well in all cases. Such an app does not need to provide 100%, 140%, and 180% variants of its images.

In CSS you can also use media queries with `max-resolution` and `min-resolution` settings to control which images get loaded. Remember, however, that CSS will see the logical DPI, not the physical DPI, so the *approximate* cutoffs for each scaling factor are more or less as follows:

```
@media all and (max-resolution: 133dpi) {
    /* 100% scaling */
}

@media all and (min-resolution: 134dpi) {
    /* 140% scaling */
}

@media all and (min-resolution: 172dpi) {
    /* 180% scaling */
}
```

As explained in the Guidelines for scaling to pixel density, such media queries are especially useful for images you obtain from a remote source, where you might need to amend the specific URI or the URI query string. See "Using Local and Web Images" in Chapter 16, "Alive with Activity," for how tile updates handle this for scale, contrast, and language.

The "more or less" part is that the cutoff numbers here are not that exact. What I show above comes from empirical tests, even though the documentation suggest 134, 135, and 174 dpi, respectively. Still, it's the best we can do in CSS at present. One ramification of this is that you should *never* refer to scale-specific graphics (using *.scale-nnn* in filenames). This is because the Windows Store will download only those scale graphics that are needed on the device (see Package Bloat? below), and if there's a mismatch between your media queries and what the system actual reports, you can end up referring to images that aren't present.

> **Package bloat?** When providing multiple variations of graphics for scales, languages, and contrasts, the natural concern is that they will cause your app package to get bigger and bigger, to the point where those assets dwarf the rest of the app code. It's actually not anything to worry about. Sure, when you create a package in Visual Studio and upload it to the Windows Store, it will contain all those assets together. However, if you structure resources with the appropriate folder names and/or file suffixes for the variations, the Store automatically breaks out those resources into separate packs so that your customers download only the resources that they actually need for their configuration.

Programmatically, you can again obtain `logicalDpi` and `resolutionScale` properties from the `DisplayInformation` object. Its `logicaldpichanged` event (a WinRT event) can also be used to check for changes in the `resolutionScale`, since the two are always coupled.

If your app manages a cache of graphical assets, by the way, especially those downloaded from a service, organize them according to the `resolutionScale` for which that graphic was obtained. This way you can obtain a better image if and when necessary, or you can scale down a higher resolution image that you already obtained. It's also something to be aware of with any app settings you might roam, because the pixel density and screen size may vary between a user's devices.

> **Note** `img` elements on a page that have scale variations in the app's resources are not automatically reloaded when the scale changes, as when moving a view between monitors with different DPI. The Scaling according to DPI sample recommends using the `WinJS.Resources.oncontextchanged` event instead, which fires for conditions that change what resources are loaded—contrast, language, and scale. For details, see js/scenario1.js in the sample, especially the comments in the `refresh` method.

Multiple Views

Many years ago, when I was very actively giving presentations at many developer conferences, I so much wished that Microsoft PowerPoint could present my slides on the main display for the audience while I got a view on my display that showed the next slide as well as my notes. You might be laughing because PowerPoint has had this feature for quite some time (I wasn't the only one who made the request!). The point, though, is that likely app scenarios include a single app managing multiple independent views, possibly and oftentimes on separate monitors, and the user rearranging and resizing those views, which includes dragging them between monitors. Any one view, though, is limited to a single display (that is, a view cannot span displays).

Projection scenarios like PowerPoint are clearly one of the primary uses for multiple views; a game might also use a second view for various controls and output windows so they don't interfere with the main display. A different kind of use is where an app has several independent (or loosely coupled) functions that can operate independently and can benefit from being placed side by side with views from other apps (multitasking at its best!). A customer-management system for a mobile sales rep might have one view for appointments and contacts, another for data entry forms, and a third for managing presentations and media. Although these functions *could* be part of the same layout within a single app view, they don't need to be. Having them as separate views—which means they operate on different threads and can share data only via the file system—allows the user to place them in relation to other apps such as a web browser (for customer research) and an email app.

The caveat with multiple views is that they run on separate threads and therefore each have their own script context. Views can only communicate with each other through `postMessage` calls (like we use to communicate between the local and web contexts), but they can exchange data also via appdata settings and/or files, which they share in common.

A view is created through the [`MSApp.createNewView`]() API, provided by the app host and not WinRT because it's specific to apps written in HTML and JavaScript. The one argument you give to this method is a string with the `ms-appx` URI of an in-package HTML page to use for the new view (only `ms-appx` URIs are allowed, meaning the view runs in the local context). That HTML file can, of course, reference whatever own stylesheets and script (including WinJS) that it needs. All of this will be loaded as part of the `createNewView` call, even though the view is not yet visible. The usual DOM events like `DOMContentLoaded` will be raised in the process, and you can use the `WinJS.Application` events as always (provided that you call `app.start()` to drain the queue).

The `MSAppView` object that `createNewView` returns will have a unique `viewId` property to identify it, along with `postMessage` and `close` methods. The latter is what the app clearly uses to close the view. The secondary view receives messages through the `window.onmessage` event, as usual. The new view can also call `window.close` to close itself, and the user can also close the view with touch/mouse gestures or Alt+F4. As for sending data back to the app, the view calls `MSApp.getViewOpener` to obtain the `MSAppView` object for the app's primary view—its `postMessage` method will raise a `window.onmessage` event in the primary view. These relationships are shown in Figure 8-7.

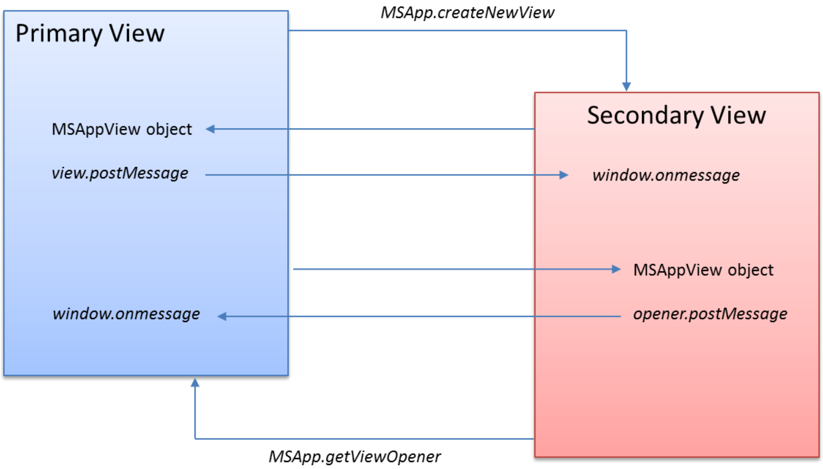


FIGURE 8-7 A primary view creates a secondary view through `MSApp.createNewView`. The secondary view retrieves a reference to the primary view through `MSApp.getViewOpener`. The result in both cases is an `MSAppView` object whose `postMessage` call sends a message to the other view.

Let me note up front that if a secondary view closes itself with window.close, the app won't receive any kind of event. Typically, then, the secondary view will use postMessage to inform the app that it's closing. This is also where the view should subscribe to the ApplicationView.onconsolidated event to detect when the user closes it directly, and post a message to the app. We'll see an example shortly.

Now creating a new view doesn't actually make it visible—the view's HTML, CSS, and JavaScript are all loaded and run, but nothing will have appeared on the screen. For that you need to use one of two APIs: the ProjectionManager or the ApplicationViewSwitcher, both of which are in Windows.UI.ViewManagement.

The ProjectionManager is meant for full-screen scenarios and works with a single secondary view. Its projectionDisplayAvailable property, first off, tells you whether projection is possible and, along with the projectiondisplayavailablechanged event, is what you use to enable a projection feature in an app. To make the view visible you pass its viewId to startProjectingAsync along with the ID of the primary view (the one from ApplicationView.getForCurrentView().id). Later on, call stopProjectingAsync to close the view and swapDisplaysForViewsAsync to do what it implies.

For a simple example, refer to the SimpleViewProjection example in this chapter's companion content. On startup it gets the primary view and the ProjectionManager (js/default.js):

```
var view = vm.ApplicationView.getForCurrentView();
var projMan = vm.ProjectionManager;
```

The Start button in the example is itself enabled or disabled according to the ProjectionManager.

```
btnStart.addEventListener("click", startProjection);

projMan.onprojectiondisplayavailablechanged = function () {
    checkEnableProjection();
}

function checkEnableProjection() {
    btnStart.disabled = !projMan.projectionDisplayAvailable;
}
```

When you click Start, the app creates the view and starts projecting, using the id's of both views:

```
viewProjection = MSApp.createNewView("ms-appx:///projection/projection.html");

projMan.startProjectingAsync(viewProjection.viewId, view.id).done(function () {
    // enable/disable buttons
});
```

Other buttons will call stopProjectingAsync and swapDisplaysForViewAsync, and other code handles selectively enabling the UI as needed. There's also a button to do a postMessage to the projection with the current time.

```
function sendMessage() {
    var date = new Date();
    var timeString = date.getHours() + ":" + date.getMinutes() + ":" + date.getSeconds()
        + ":" + date.getMilliseconds();
    var msgObj = { text: timeString };
    viewProjection.postMessage(JSON.stringify(msgObj),
        document.location.protocol + "//" + document.location.host);
}
```

The projection page is in projection/projection.html with associated .css and .js files. Within its activated handler it retrieves the opener view and listens for the `consolidated` event (js/projection.js):

```
var opener = MSApp.getViewOpener();

var thisView = Windows.UI.ViewManagement.ApplicationView.getForCurrentView();
thisView.onconsolidated = function () {
    sendCloseMessage();
}
```

where `sendCloseMessage` is a function that just does a `postMessage` with a "close" indicator so the main app can reset its UI. The same thing happens when you click the Close button in the projection, after which it calls `window.close`.

For another demonstration, refer to the Projection sample in the SDK. It basically does the same thing as the simple example above, adding a little more to set view properties like the title. I'm not showing any code from the sample, however, because it buries the basic API calls like `createNewView` and `getViewOpener` deep inside helper classes called *ProjectionView.ViewManager* (for the primary view) and *ProjectionView.ViewLifeTimeControl* (for the secondary view). These classes (in js/viewLifetimeControl.js) provide a mini-framework around the various events like `consolidated` and visibility changes, and creates a message protocol between the primary and secondary views. It also does reference counting on secondary views to control their lifetime, and makes sure that the primary view is notified when a secondary view is closed. It's the kind of stuff you'd write if you started using multiple views, but it's a little too complicated to use as an introduction to the API!

This same mini-framework is also used in the SDK's Multiple Views sample, which demonstrates the methods of the `ApplicationViewSwitcher` class. Those methods—with some verbose names!—are:

Method	Description
`switchAsync(id)` (3 overloads)	Switches to a given view in the same space as the calling one. If the target view is already visible elsewhere on the screen, however, this will simply change focus to that view. A value from `ApplicationViewSwitchingOptions` can be used to customize the transition: `default` (standard transition), `skipAnimation` (no transition), or `consolidateViews` (closes the calling view).
`tryShowAsStandaloneAsync` (3 overloads)	Shows the new view adjacent to the calling view, with options from `ViewSizePreference` to determine the desired state of the new and calling views: `default`, `useLess`, `useHalf`, `useMore`, `useMinimum`, `useNone`. You can experiment with these variations in scenario 1 of the sample.
`disableShowingMainViewOnActivation`	Prevents the primary view of the app from showing on subsequent activations, that is, when it's appropriate to activate the app directly into a secondary view. Showing the primary view is always the default unless this method is called. This is shown in scenario 2 of the sample, with the actual call in js/default.js.

prepareForCustomAnimatedSwitchAsync	Tells the views whether to run default animations on a switch; by specifying the skipAnimation option, you can attach a completed handler to this promise to perform custom animations. See scenario 3 of the sample.

Because the details and variations of these APIs get rather complex, I'll leave it to you to explore the sample directly. Note that in the main scenarios of the app you won't find any calls to switchAsync—these are made in the secondary view (js/secondaryView.js) to switch back to the primary view. In each case the view closes itself as part of the switch, which isn't a required behavior, of course.

To show the basic switching behavior, I've made a few small modifications to a copy of this sample in the companion content. First, the button in the secondary view to switch back to the main view does not close the secondary one. Then I've added a button to scenario 1 of the main app to call switchAsync on the selected secondary view. You'll see how it brings that view up in the same space as the original one, rather than alongside. If you switch back from the secondary view, the main view will appear in that space.

Things get interesting when you create an adjacent view first and then in the main app switch to a secondary view in the space. You'll see two secondary views at once. If you switch to the main app in either, you'll find that the Switch to Main View button in the other secondary view just changes focus to the already-visible app. Otherwise you'd be seeing the main app view twice, which would be very confusing!

Pannable Sections and Styles

In Chapter 7, "Collection Controls," we spent a little time looking at when a ListView control was the right choice and when it wasn't. One of the primary cases where developers have inappropriately attempted to use a ListView is to implement a home or hub page that contains a variety of distinct content groups arranged in columns, as shown in Figure 8-8 and explained on Navigation design for Windows Store apps. At first glance this might look like a ListView, but because the data it's representing isn't actually a collection, just a layout of fixed content, it makes sense to use other options for the job. One option is the WinJS.UI.Hub control, which works great for layouts like that in Figure 8-8, and we'll see that control fairly soon. The other option, which I want to discuss first, is to simply use tried-and-true HTML and CSS for the job!

I point this out because with all the great controls that WinJS provides, it's easy to forget that everything you know about HTML and CSS still applies in Store apps. After all, those controls are in themselves just blocks of HTML and CSS with some additional methods, properties, and events. So let's look at how we might create a hub page directly, as it gives us an opportunity to learn about a few features that controls like the Hub employ.

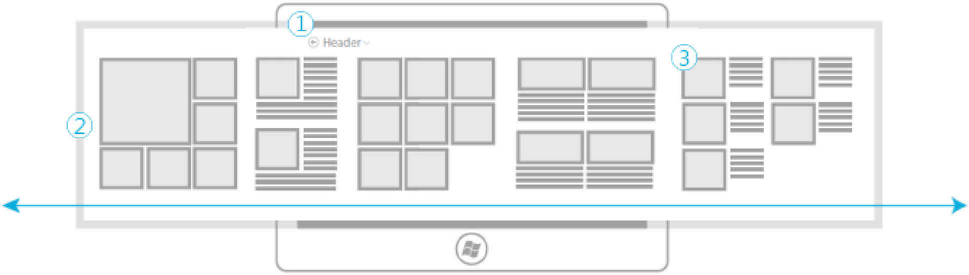


FIGURE 8-8 The layout of a typical home or hub page of a Store app with a fixed header (1), a horizontally pannable section (2), and content sections or categories (3).

Laying Out the Hub

Let's start by asking how we'd use straight HTML and CSS to implement the whole pannable area of the hub page in Figure 8-8. Referring first to Laying out an app page, we know that the padding between groups or sections should be four units of 20px each, or 80px. Let's say that each section should be square, except for the second one which is only half the width. On a baseline 1366x768 display, the height of each section would be 768px minus 128px (for the header) minus the minimum 50px on the bottom, which leaves 590px (if we add headings for each section, subtract another 40px).

Thus, a square section on the baseline display would be 590px wide (we'd set the actual height to 100% of its containing grid cell). The total width of the pannable area will be:

```
(590 * 4 full-size sections) + (295 * 1 half-width section) + (80 * 4 for the separator gaps)
```

This equals 2975px. To this we'll add border columns of 120px on the left (according to the silhouette) and 80px on the right, for a total of 3175px.

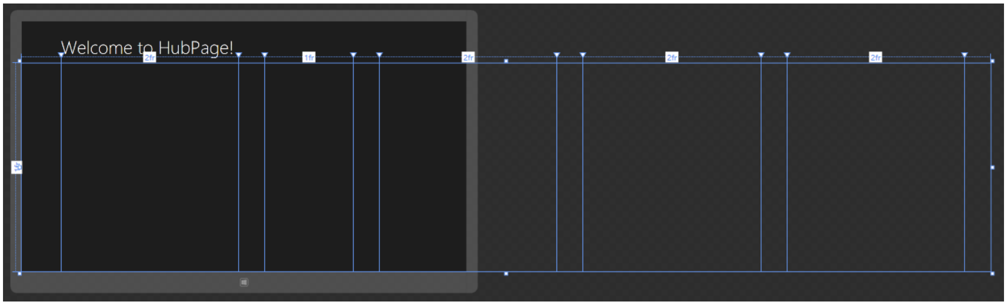
To create the whole region with exactly this layout, we can use a CSS grid within a block element. To demonstrate this, run Blend and create a new project with the Navigation App template (so we just get a basic page with the silhouette and not all the secondary pages). Within the `section` element of pages/home/home.html, create another `div` element and give is a class of *hubSections*:

```
<section aria-label="Main content" role="main">
    <div class="hubSections">
    </div>
</section>
```

In pages/home/home.css, add a few style rules. Give `overflow-x: auto` to the `section` element, and lay out the grid in the *hubSections* `div`, using added columns on the left and right for spacing (removing the `margin-left: 120px` from the `section` and adding it as the first column in the `div`):

```
.homepage section[role=main] {
    overflow-x: auto;
}
.homepage .hubSections {
    width: 2975px;
    height: 100%;
    display: -ms-grid;
    -ms-grid-rows: 1fr 50px;
    -ms-grid-columns: 120px 2fr 80px 1fr 80px 2fr 80px 2fr 80px 2fr 80px;
}
```

With just these styles we can see the layout taking shape in Blend by zooming out in the artboard:



Now let's create the individual sections, each one starting as a `div` in pages/home/home.html:

```
<section aria-label="Main content" role="main">
    <div class="hubSections">
        <div class="hubSection1"></div>
        <div class="hubSection2"></div>
        <div class="hubSection3"></div>
        <div class="hubSection4"></div>
        <div class="hubSection5"></div>
    </div>
</section>
```

and styled into their appropriate grid cells with 100% `width` and `height`. I'm showing *hubSection1* here; the others are the same with just a different column number (4, 6, 8, and 10, respectively):

```
.homepage .hubSection1 {
    -ms-grid-row: 1;
    -ms-grid-column: 2; /* 4 for hubSection2, 6 for hubSection3, etc. */
    width: 100%;
    height: 100%;
}
```

All of this is implemented in the HubPage example in this chapter's companion content.

Laying Out the Sections

Now we can look at the contents of each section. Depending on your content and how you want those sections to interact, you can again just use layout (CSS grids or perhaps flexbox) or controls like `Repeater` or `ListView`. *hubSection3* and *hubSection5* have gaps at the end, so they might be collection controls with variable items. Note that if we created lists with more than 9 or 6 items, respectively, we'd want to adjust the column size in the overall grid and make the `section` element width larger, but let's assume the design calls for a maximum of 9 and 6 items in those sections.

Let's also say that we want each section to be interactive, where tapping an item would navigate to a details page. (Not shown in this example are group headers to navigate to a group page.) We'll just use a ListView in each, where each ListView has a separate data source. For *hubSection1* we'll need to use cell spanning, but the rest of the groups can just use a simple GridLayout. The key consideration with all of these is to style the items so that they fit nicely into the basic dimensions we're using. And referring again back to the silhouette, the spacing between image items should be 10px and the spacing between columns of mixed content (*hubSection4* and *hubSection5*) should be 40px (which can be set with appropriate CSS margins).

> **Hint** If you need to make certain areas of your content unselectable, use the `-ms-user-select` attribute in CSS for a `div` element. Refer to the Unselectable content areas with -ms-user-select CSS attribute sample. How's that for a name?

Panning Styles and Railing

To make an element pannable, you use the standard CSS `overflow-x` (horizontal), `overflow-y` (vertical), or `overflow` (bi-directional) styles where the values can be `visible` (the default, where content visible overflows its element boundaries), `scroll` (content is clipped to the element and scrollbars are always visible), `hidden` (content is clipped and cannot be panned), and `auto` (content is clipped and panning is enabled when necessary). Most often, as in the HubPage example we just saw, you'll use `auto` so that an element pans with auto-hiding scrollbars.

If an element is pannable, you can control the appearance of the scrollbar with `-ms-overflow-style`. Its default setting, `auto`, provides the auto-hiding scrollbar (`-ms-autohiding-scrollbar` does the same). Other options include `scrollbar` (always visible) and `none` (never visible). Using `none` is what creates a pannable region that never shows any kind of scrollbar, which is certainly appropriate when you don't want anything interfering with the content. The one caveat is that panning with the mouse requires a mousewheel—if you want drag-to-pan behavior, you'll need to handle pointer events directly (see Chapter 12). And speaking of the mouse wheel, the `-ms-scroll-translation` style allows you to translate vertical mousewheel events to horizontal panning.

By default, when an element is styled panning in both vertical and horizontal directions, panning via touch (or the touchpad) follows the movement of the touchpoint in both directions together. This doesn't work well, however, for content that the user will typically want to pan in only one dimension. For this you can use *railing* to lock panning (again, only for touch and touchpad) along the axis that the user is panning most, as shown in Figure 8-9. And once panning is locked in that dimension, it stays in effect until the touch point is released, meaning that arbitrary movements will not change the lock. If the user does happen to pan evenly in both directions (a narrow 45-degree line in each quadrant), bi-directional panning can go into effect, but the user would have to deliberately try to make this happen.

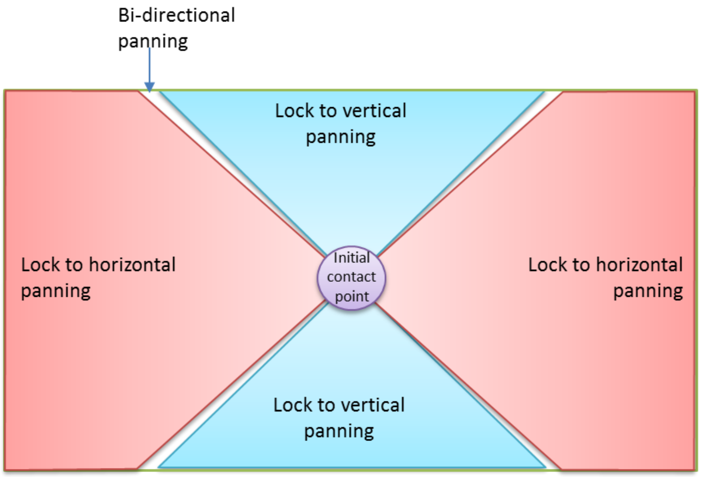


FIGURE 8-9 The concept of rails, where the white areas indicate regions where the touch point is moved equally along both axes and bi-directional panning can happen. Note that railing is determined along 45-degree lines from the touch point and has nothing to do with the shape of the region itself.

Setting up railing is quite simple, as demonstrated in scenario 1 of the HTML scrolling, panning, and zooming sample. Assuming that the panning area (a `div` containing an image in this case) allows bi-directional panning (`overflow: auto`), rails are added with `-ms-scroll-rails: railed` (js/panning.js):

```
.Railed {
    overflow: auto;
    -ms-scroll-rails: railed;
}
```

and they are removed by setting `-ms-scroll-rails: none`:

```
.Unrailed {
    overflow: auto;
    -ms-scroll-rails: none;
}
```

At present it isn't possible to adjust the rail thresholds nor to set vertical and horizontal rails independently. In any case, the effects of rails are shown in Video 8-2.

A related style for touch is `touch-action`, which sets allowable panning directions, allowable zooming gestures, and allowable cross-slide gestures, a few of which are shown in scenario 6 of the sample.

Parallax panning A parallax effect is when you see multiple layers of a page panning at different speeds, thereby giving a sense of visual depth. The Windows Start screen itself does this, where the tiles pan faster than the background. Such an effect is possible through HTML and CSS, and it's easy to find guidance on the subject. Be specifically mindful of the properties you're animating for the effect. If you use CSS transforms exclusively, those animations will run "independently" in the GPU. If you animate any other positional properties, the animations will run on the UI thread and likely perform poorly on lower-end hardware.

Panning Snap Points and Limits

If you run the HubPage example and pan around a bit using inertial touch gestures (that is, those that continue panning after you've released your finger, explained more in Chapter 12), you'll notice that panning can stop in any position along the way. You or your designers might like this, but it also makes sense in many scenarios to automatically stop on a section or group boundary. This can be accomplished for touch interactions using CSS styles for *snap points* as described in the following table. These are styles that you add to a pannable element alongside `overflow` styles, otherwise they have no effect. Documentation for these (and some others) can be found on the CSS reference for Touch: Zooming and Panning. Be clear that snap points are a touch-only feature (including touchpads); if you want to provide the same kind of behavior with mouse and/or keyboard input, you'll need to do such work manually along the lines of how the FlipView control handles transition between items.

Style	Description	Value Syntax
-ms-scroll-snap-points-x	Defines snap points along the x-axis	`snapInterval`(start<length>, step<length>) | `snapList`(list<lengths>)
-ms-scroll-snap-type	Defines what type of snap points should be used for the element: `none` turns off snap points, `mandatory` always adjusts panning to land on a snap-point (which includes ending inertial panning), and `proximity` changes the panning only if a panning motion naturally ends "close enough" to a snap point. Using `mandatory`, then, will enforce a one-section/item-at-a-time panning behavior, whereas `proximity` would pan past interim snap points if enough inertia is applied. Note also that dragging with a finger (not using an inertia gesture) will allow the user to pan directly past snap points.	`none` | `proximity` | `mandatory`
-ms-scroll-snap-x	Shorthand to combine `-ms-scroll-snap-type` and `-ms-scroll-snap-points-x`	`<-ms-scroll-snap-type> <-ms-scroll-snap-points-x>`
-ms-scroll-snap-y	Shorthand to combine `-ms-scroll-snap-type` and `-ms-scroll-snap-points-y`	`<-ms-scroll-snap-type> <-ms-scroll-snap-points-y>`

In the table, `<length>` is a floating-point number, followed by an absolute units designator (`cm`, `mm`, `in`, `pt`, or `pc`) or a relative units designator (`em`, `ex`, or `px`).

To add snap points for each of our hub sections in the Hub Page example, we only need to add two snap points styles after `overflow-x`:

```
.homepage section[role=main] {
    overflow-x: auto;
    -ms-scroll-snap-type: mandatory;
    -ms-scroll-snap-points-x: snapList(0px, 670px, 1045px, 1715px, 2385px, 3055px);
}
```

Note that the snap points indicated here include the 120px left border so that each one aligns the section underneath the header text. The 0px point thus snaps to the first section, with the box being 120px in so that it alights to the header. The 670px point for the second reflects that 120px plus the 590px width of the first section. 1045 is 670 plus a half-width section (295px) plus 80px, and so on. We also set snap points all the way to what would be the position *past* the last section—that is, the rightmost panning point. This makes sure that you can fully pan into the last section in narrow views.

With these changes you'll now find that panning around stops nicely (with animations) on the section boundaries—see Video 8-3. For a hub page like this, proximity snapping is usually more appropriate. Mandatory snap points are intended more for items that can't be interacted with or consumed without seeing their entirety, such as flipping between pictures, articles, and so on. (The FlipView control uses these.)

Additional demonstrations of snap points can be found in scenario 2 of the HTML scrolling, panning, and zooming sample. Snap points are not presently supported on the ListView control, as they are intended for use with your own layout.

Related to snap points are the styles that set minimum and maximum extents for panning: `-ms-scroll-limit-x-[min | max]`, `-ms-scroll-limit-y-[min | max]`, and the shorthand combined style `-ms-scroll-limit`. These specifically limit the values of the element's `scrollLeft` (x axis) and `scrollTop` (y axis).

The `-ms-scroll-chaining` style is also important when limits are in effect (whether set explicitly or by the size of the content). By default, if you pan content to its limit within an element and continue panning outside the boundaries of that element, the element's contents will compress a little and then bounce back to normal size when you release the pointer. By setting this style to `chained`, a pan that goes outside the boundary of one element picks up and pans the next nearest pannable element, and no bounce effect occurs. Scenarios 4 and 5 of the sample demonstrates some of this, and you can read more about chaining in Guidelines for panning.

Zooming Snap Points and Limits

While we're on the subject and looking at the HTML scrolling, panning, and zooming sample, I should mention the `-ms-content-zoom-*` styles that are also in the CSS reference for Touch: Zooming and Panning, a few of which are used in scenario 3 of the sample:

Style	Description	Value Syntax
`-ms-content-zooming`	Indicates whether zooming is enabled for an element.	`none` | `zoom`
`-ms-content-zoom-limit-min` `-ms-content-zoom-limit-max`	Specifies minimum and maximum zoom extents, typically in terms of percentages.	`<number>%`

-ms-content-zoom-limit	Shorthand to combine -ms-content-zoom-limit-min and -max.	<min>% <max>%
-ms-content-zoom-snap-points	Defines zooming snap points.	snapInterval(start<zoom %>, step<zoom %>) | snapList(list<zoom %>)
-ms-content-zoom-snap-type	Determines what type of snap points should be used for the element. See the description in the previous section for panning snap types.	none | proximity | mandatory
-ms-content-zoom-snap	Shorthand to combine snap points and type.	<-ms-zoom-snap-type> <-ms-zoom-snap-points>
-ms-content-zoom-chaining	Specifies whether zoom behavior carries over to the nearest zoomable parent once limits are reached for an element.	none | chained

The Hub Control and Hub App Template

Although it's certainly possible, as seen previously, to create any layout you want for an app with straight HTML and CSS (and to enable panning and zooming behaviors), WinJS helps out hub page design quite a bit through the WinJS.UI.Hub control (which isn't available on Windows Phone, use WinJS.UI.Pivot instead)). The control is organized much like the raw layout we saw earlier: the root div for the control plays hosts to any number of sections but has the added feature that it supports a large "hero" image on the left side, as we'll see, along with both horizontal and vertical layouts. The Hub also supports semantic zoom, meaning that you can host it within a semantic zoom control and use either another Hub or a ListView for the zoomed-out view. All in all, then, it's a good control to use unless you have layout needs that it can't accommodate.

Each section in the Hub is defined by a WinJS.UI.HubSection control. Each section can have an invocable header and can host whatever other content you want, including templates, repeaters, and ListView controls. That content can be expressed as however many child elements you like, as the HubSection will always create another div container for them alongside one that it creates for the header.

Apart from the usual suspects like addEventListener, the following methods and properties are found on the Hub control:

Member	Description
headerTemplate (property)	Gets or sets a template or rendering function for the header.
indexOfFirstVisible (property)	Gets or sets the index of the first visible section (leftmost for left-to-right languages, rightmost for right-to-left languages). The section need only be partially visible.
indexOfLastVisible (property)	Gets or sets the index of the last visible section (rightmost for left-to-right languages, leftmost for right-to-left languages). The section need only be partially visible.
loadingState (property) loadingstatechanged (event)	Property contains "loading" if the Hub is still rendering section, "complete"; the event is fired when the property changes (with the new state in eventArgs.detail.loadingState).
orientation	Gets or sets the orientation from the WinJS.UI.Orientation enumeration, either "horizontal" or "vertical".
scrollPosition (property)	Gets or sets the panning position of the entire Hub. This is useful when navigating to other pages and back again; by saving the position when navigating away you can reset that position when navigating back.
sectionOnScreen (property)	Gets or sets the index of the first fully visible section.

`sections` (property)	Gets or sets a `Binding.List` of the `HubSection` objects inside the hub.
`zoomableView` (property)	Returns an object with `IZoomableView` to support semantic zoom.
`contentanimating` (event)	Raised when the hub is about to play an animation effect. The `eventArgs.detail` object contains the `type` of animation, the `index` of the animated section, and that `section` control.
`headerinvoked` (event)	Fired when an interactive header is clicked or tapped, or when the header has the focus and the spacebar or Enter key is pressed. The `eventArgs.detail` object contains the `index` of the header and the `section` object to which it belongs.

You can find reference code for the Hub in two places: the HTML Hub control sample and the Hub App project template in Visual Studio. The project template is in many ways similar to the Grid App project—it uses the same data source (in js/data.js once you create a project with it) and navigates between hub, section, and item pages that correspond to the Grid App template's groupedItems, groupDetail, and itemDetail pages; the latter two in both templates are virtually the same.

Here, let's look at the control through the lens of the sample because it isolates the Hub's capabilities more clearly. Once we've done that, you should be able to look through the Hub App project template and understand what's going on.

When you run the sample, you'll see that each scenario has a Launch Full Screen Sample button because the Hub is definitely meant to occupy an entire page. When you look through the project structure, all the pages with the Hub control are found in the *pages* folder—the individual scenarios just navigate to those pages, which then provide a back button to return to the sample's home page. (Of course, if you use the Hub on your own home page, like the Hub App project template, you certainly wouldn't have a back button! You could also take it as an exercise to convert the navigation in the sample to using multiple views, as described earlier in this chapter.)

Scenario 1 navigates to pages/basichub.html whose full layout is shown in Figure 8-10. Four sections labeled Images, ListView, Video, and Form Controls describe exactly what kind of content that section contains. In this particular case, the headers are static and do not include a hero image—we'll see those shortly. The video, though, is playing inline so that you can see the media controls for it. Again note that a typical app would *not* have a back button on its home page!

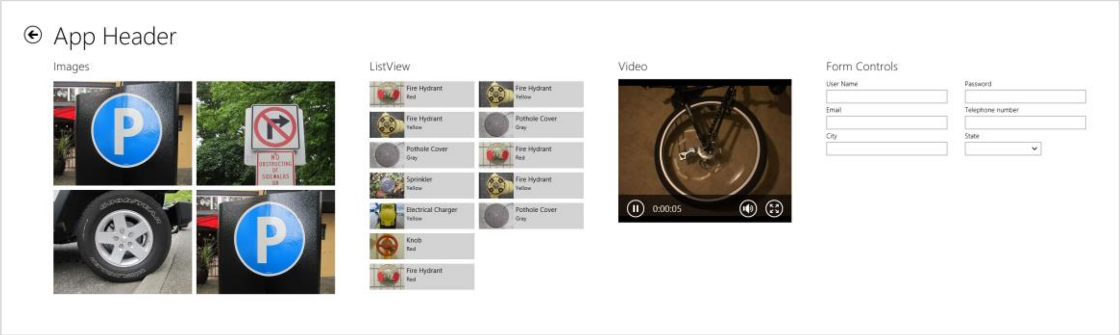


FIGURE 8-10 The full layout for scenario 1 of the HTML Hub Control sample.

In pages/basichub.html, the Hub control and its sections are hosted in div elements. Omitting the content of each section (especially the <select> element in the form with <option> elements for all fifty United States!), we can see the overall structure clearly:

```
<div data-win-control="WinJS.UI.Hub">
    <div class="section1" data-win-control="WinJS.UI.HubSection"
        data-win-options="{header: 'Images', isHeaderStatic: true}">
    </div>

    <div id="list" class="section2" data-win-control="WinJS.UI.HubSection"
        data-win-options="{header: 'ListView', isHeaderStatic: true}">
    </div>

    <div class="section3" data-win-control="WinJS.UI.HubSection"
        data-win-options="{header: 'Video', isHeaderStatic: true}">
    </div>

    <div class="section4" data-win-control="WinJS.UI.HubSection"
        data-win-options="{header: 'Form Controls', isHeaderStatic: true}">
    </div>
</div>
```

As you can see, each HubSection control has a header property (read-write) to provide its header text and an isHeaderStatic property (read-write) to indicate whether that header is inert (true) or can be clicked or tapped (false, the default). In the latter case, a chevron character > will be appended to the header text and clicking a header will raise a headerinvoked event on the overall Hub control, not the section.

The section control, in fact, has only one other property, contentElement (read-only), which will be the div that the HubSection creates to contain whatever child elements you declare for the control.

Speaking of child elements, here's that markup for the first three sections (slightly condensed; I'm omitting section 4 because its full markup takes two pages!):

```
<div data-win-control="WinJS.UI.Hub">
    <div class="section1" data-win-control="WinJS.UI.HubSection"
        data-win-options="{header: 'Images', isHeaderStatic: true}">
        <div class="imagesFlexBox">
            <img class="imageItem" src="/images/circle_image1.jpg" />
            <img class="imageItem" src="/images/circle_image3.jpg" />
            <!-- And so on for nine total images -->
        </div>
    </div>
    <div id="list" class="section2" data-win-control="WinJS.UI.HubSection"
        data-win-options="{header: 'ListView', isHeaderStatic: true}">
        <div id="listView" class="win-selectionstylefilled"
            data-win-control="WinJS.UI.ListView"
            data-win-options="{ itemTemplate: smallListIconTextTemplate,
                itemDataSource: select('.pagecontrol').winControl.myData.dataSource,
                selectionMode: 'none', tapBehavior: 'none', swipeBehavior: 'none' }">
        </div>
    </div>
    <div class="section3" data-win-control="WinJS.UI.HubSection"
        data-win-options="{header: 'Video', isHeaderStatic: true}">
        <video class="promoVideo" src="images/cycle.mp4" controls></video>
    </div>
    <!-- section4 omitted -->
</div>
```

Tip Do you see the `select('.pagecontrol').winControl.myData.dataSource` reference for the ListView's `itemDataSource` property in section 2? The *pageControl* class identifies the page control element defined in html/basichub.js. That element's `winControl` property gets you to the object defined with `WinJS.UI.Pages.define`, wherein you'll find the `myData` property that returns a `Binding.List`.

The item template for the ListView is, of course, defined earlier in html/basichub.html. Styles are pulled in from pages/commonstyles.css, which we'll look at soon after we finish up with the other Hub control features.

Scenario 2 of the sample now adds a hero image at the beginning of the Hub, the full result of which is shown in Figure 8-11 and the limited 1366x768 view in Figure 8-12. In reality, the hero image is not actually a concept of the Hub control at all—it's simply another `HubSection` that's been added to the top of the list (pages/heroimage.html):

```
<div data-win-control="WinJS.UI.Hub">
    <div class="hero" data-win-control="WinJS.UI.HubSection"
        data-win-options="{header: 'Hero'}"></div>
    <div class="section1" data-win-control="WinJS.UI.HubSection"
        data-win-options="{header: 'Images', isHeaderStatic: true}">
```

where the styling for the *hero* class in pages/heroimage.css sets the image, the width, and margins, and hides the section header (using the `win-hub-section-header` selector):

```
.hero {
    background-image: url(/images/circle_hero.jpg);
    background-size: cover;
}

.win-hub-horizontal .hero {
    width: 944px;
    margin-left: -80px;
    margin-right: 80px;
}

.win-hub-section.hero .win-hub-section-header {
    visibility: hidden;
}
```

No magic here—just straight CSS, although it's important to note that the `HubSection` control extends fully top to bottom within the `Hub`, so the background image bleeds to the edges.

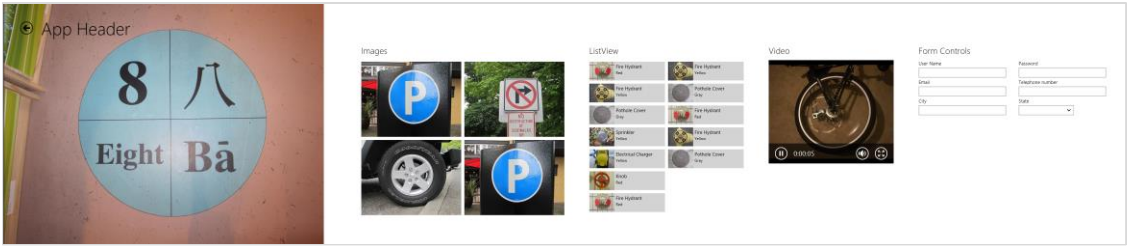


FIGURE 8-11 The full layout for scenario 2 of the HTML Hub Control sample. The hero image is just another `HubSection` added to the beginning of the Hub control, where the image, width, and margins are defined in CSS. The header for that section is also hidden via CSS.

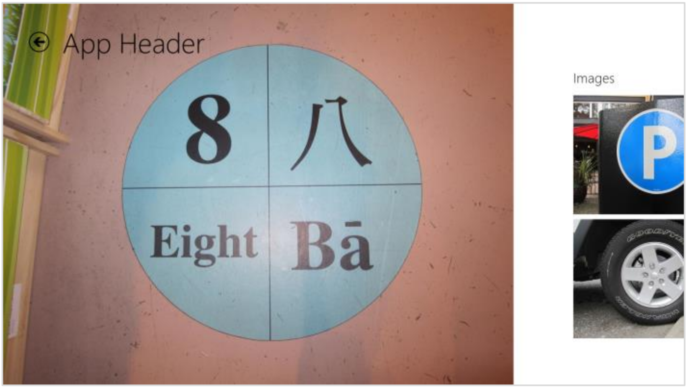
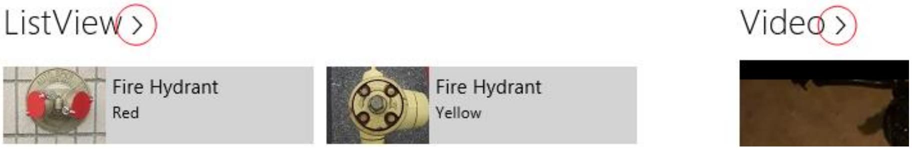


FIGURE 8-12 The appearance of the hero image of scenario 2 on a 1366x768 display.

Scenario 3 enables interactive headers for the ListView and Video sections simply by removing the `isHeaderStatic` options from the markup (defaulting to `false`; pages/interactiveheaders.html):

```
<div id="list" class="section2" data-win-control="WinJS.UI.HubSection"
    data-win-options="{header: 'ListView'}">
    <!-- ... -->
</div>

<div class="section3" data-win-control="WinJS.UI.HubSection"
    data-win-options="{header: 'Video'}">
    <video class="promoVideo" src="images/cycle.mp4" controls></video>
</div>
```

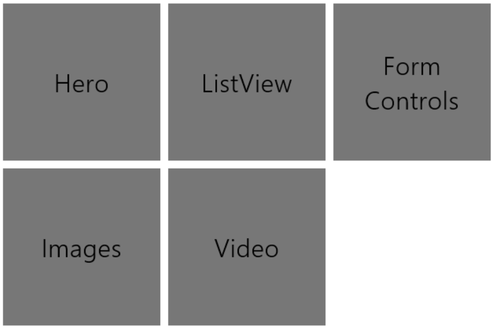
This makes it possible to navigate to the header with the Tab key, and it adds chevrons to the header text (circled in red):



Invoking a header (click, tap, spacebar, or Enter key) then raises the Hub's `headerinvoked` event. This scenario's code picks them up as follows to navigate to subsidiary pages. The section's index and element are included in the `eventArgs.detail` object (pages/interactiveheaders.js):[74]

```
// Inside the page control
ready: function (element, options) {
    this._hub = element.querySelector(".win-hub").winControl;
    this._onHeaderInvokedBound = this.onHeaderInvoked.bind(this);
    this._hub.addEventListener("headerinvoked", this._onHeaderInvokedBound);
},

//navigate to deeper levels by invoking interactive headers
onHeaderInvoked: function (ev) {
    var index = ev.detail.index;     //Section index
    var section = ev.detail.section; //Section element

    //check that the correct section is invoked
    if (index === 2) {
        WinJS.Navigation.navigate("/pages/listview.html");
    }

    if (index === 3) {
        WinJS.Navigation.navigate("/pages/video.html");
    }
},

unload: function () {
    this._hub.removeEventListener("headerinvoked", this._onHeaderInvokedBound);
},
```

Supporting semantic zoom, which is shown in scenario 4, is nothing new—the sample (pages/semanticzoom.html) just places the `Hub` control as the first child of a `WinJS.UI.SemanticZoom` control and declares a ListView for the zoomed-out view that just navigates to the sections:



[74] The sample actually uses `window.addEventListener`, which works only because the otherwise unhandled event bubbles up to the `window` level. Adding it on `this._hub` as shown in the code here is more direct.

Scenario 5 (pages/verticallayout.html) takes one more step to handle a narrow 320px view by switching the Hub control into a vertical `orientation` (with appropriate styles in pages/verticallayout.css). When vertical, the Hub simply lays out the sections vertically, and the sample also handles semantic zoom in this state are well. To illustrate all this, Figure 8-13 shows the 500px wide view of the app, where it still uses the horizontal orientation, then the 320px narrow view, and then the narrow zoomed-out view.

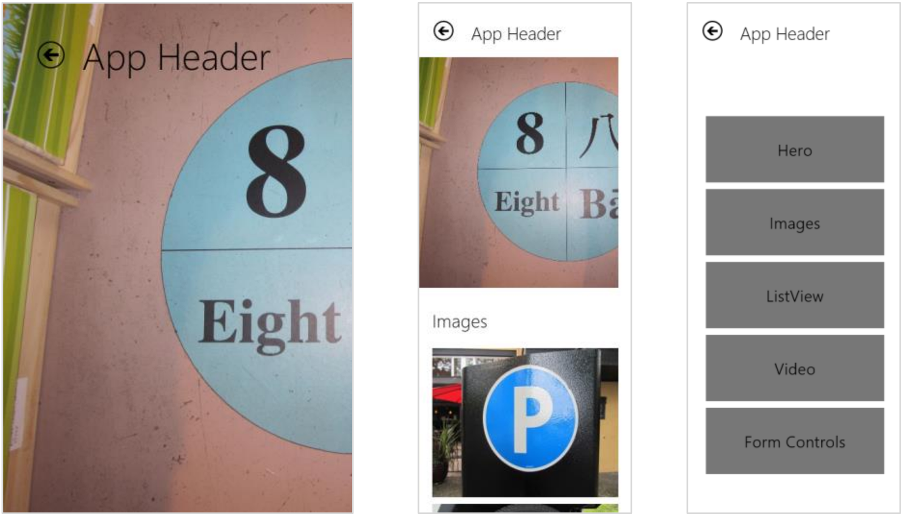


FIGURE 8-13 Scenario 5 at 500px wide (left), 320px wide (middle) with the vertical Hub layout, and 320px wide zoomed out (right) with a vertical ListView.

The only bit of code needed to handle this is the `updateHubLayout` function that's called on `window.onresize` (pages/verticallayout.js; the arguments to the function are the Hub and the zoomed-out ListView control, respectively):

```
function updateHubLayout(hub, listview) {
    if (document.body.clientWidth < 500) {
        if (hub.orientation !== WinJS.UI.Orientation.vertical) {
            hub.orientation = WinJS.UI.Orientation.vertical;
            listview.layout = new WinJS.UI.ListLayout();
        }
    }
    else {
        if (hub.orientation !== WinJS.UI.Orientation.horizontal) {
            hub.orientation = WinJS.UI.Orientation.horizontal;
            listview.layout = new WinJS.UI.GridLayout();
        }
    }
}
```

The code here changes orientation whenever the view width falls below 500px. In many cases you might want to make the switch when the aspect ratio goes to a portrait orientation, which can be done simply by changing the `if` statement to compare with `document.body.clientHeight` and changing the CSS media query to check for an orientation change rather than widths:

```
// pages/verticallayout.js
if (document.body.clientWidth < document.body.clientHeight)

/* pages/verticallayout.css */
/* Original was @media (min-width: 320px) and (max-width:499px) */
@media (orientation: portrait)
```

Hub Control Styling

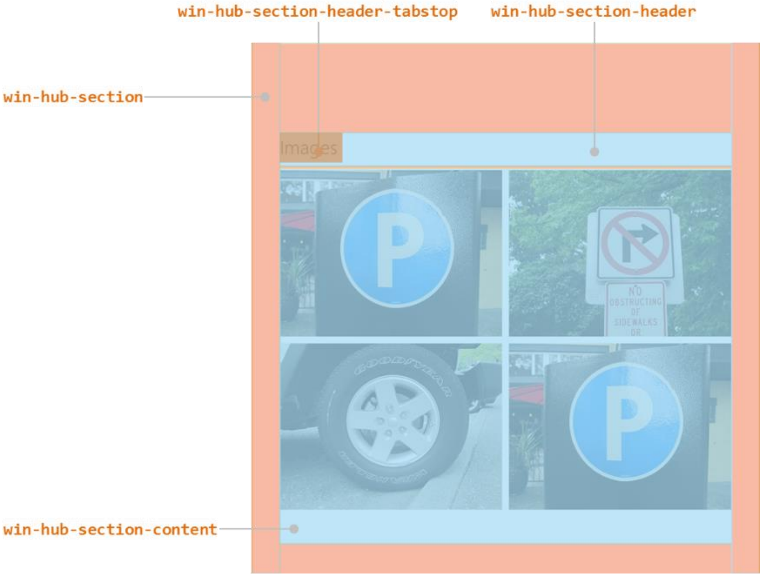
Styling the hub control is a matter of styling its various parts, like we've seen in earlier chapters with the ListView, the FlipView, and the ItemContainer controls. To lay the `div` structure out clearly, a bit of markup like this:

```
<div data-win-control="WinJS.UI.Hub">
    <div class="hero" data-win-control="WinJS.UI.HubSection"
        data-win-options="{header: 'Hero'}"></div>
    <div class="section1" data-win-control="WinJS.UI.HubSection"
        data-win-options="{header: 'Images', isHeaderStatic: true}"></div>
</div>
```

turns into a deeper `div` hierarchy with various `win-hub-*` classes assigned (and a few other elements):

```
<!-- The root element (win-vertical if vertically oriented) -->
<div class="win-hub win-horizontal">
    <!-- The panning region with snap points for the sections -->
    <div class="win-hub-viewport">
        <!-- The container for sections -->
        <div class="win-hub-surface">
            <!-- hero section -->
            <div class="hero win-hub-section">
                <!-- header -->
                <div class="win-hub-section-header">
                    <button class="win-hub-section-header-tabstop
                        win-hub-section-header-interactive"></button>
                    <h2 class="win-hub-section-header-content">Hero</h2>
                    <span class="win-hub-section-header-chevron"></span>
                </div>
                <!-- content -->
                <div class="win-hub-section-content">
                </div>
            </div>

            <!-- section1 -->
            <div class="section1 win-hub-section">
                <!-- same structure for header and content but without
                     win-hub-section-header-interactive for non-invocable header -->
            </div>
        </div>
    </div>
</div>
```

The most common classes to style differently than the defaults are win-hub-section, win-hub-section-header, and win-hub-section-content to change margins and/or padding. The win-hub-section-header-tabstop class identifies the button element in the header, regardless of whether it's interactive.



Because the win-hub-surface background is transparent by default, you can set a background color on win-hub-viewport to have it show through or you can use a background image that will stay fixed behind the sections. Typically you'd use a low-contrast image (like those on the Start screen) to not distract from the rest of the content.

Beyond that, the content within your sections is, of course, under your complete control, so you can style all that however you wish.

Using the CSS Grid

Starting back in Chapter 2, we've already been employing CSS grids for many purposes. Personally, I love the grid model because it so effortlessly allows for relative placement of elements and scaling easily to different screen sizes.

Because the focus of this book is on the specifics of the Windows platform, I'll leave it to the W3C specs on http://www.w3.org/TR/css3-grid-layout/ and http://dev.w3.org/csswg/css3-grid-align/ to explain all the details. These specs are essential references for understanding how rows and columns are sized, especially when some are declared with fixed sizes, some are sized to content, and others are declared such that they fill the remaining space. The nuances are many!

Because the specs themselves are still in the draft stages as of this writing, it's good to know exactly which parts of those specs are actually supported by the HTML/CSS engine used for Store apps.

For the element containing the grid, the supported styles are simple. First use the -ms-grid and -ms-inline-grid display models (the display: style). We'll come back to -ms-inline-grid later.

Second, use -ms-grid-columns and -ms-grid-rows on the grid element to define its arrangement. If left unspecified, the default is one column and one row. The repeat syntax such as -ms-grid-columns: (1fr)[3]; is supported, which is most useful when you have repeated series of rows or columns, which appear inside the parentheses. As examples, all the following are equivalent:

```
-ms-grid-rows:10px 10px 10px 20px 10px 20px 10px;
-ms-grid-rows:(10px)[3] (20px 10px)[2];
-ms-grid-rows:(10px)[3] (20px 10px) 20px 10px;
-ms-grid-rows:(10px)[2] (10px 20px)[2] 10px;
```

How you define your rows and columns is the interesting part, because you can make some fixed, some flexible, and some sized to the content using the following values. Again, see the specs for the nuances involving max-content, min-content, minmax, auto, and fit-content specifiers, along with values specified in units of px, em, %, and fr. Windows Store apps can also use vh (viewport height) and vw (viewport width) as units.

Within the grid now, child elements are placed in specific rows and columns, with specific alignment, spanning, and layering characteristics using the following styles:

- -ms-grid-column identifies the 1-based column of the child in the grid.
- -ms-grid-row identifies the 1-based row of the child in the grid.
- -ms-grid-column-align and -ms-grid-row-align specify where the child is placed in the grid cell. Allowed values are start, end, center, and stretch (default).
- -ms-grid-column-span and -ms-grid-row-span indicate that a child spans one or more rows/columns.
- -ms-grid-layer controls how grid items overlap. This is similar to the z-index style as used for positional element. Since grid children are not positioned directly with CSS and are instead positioned according to the grid, -ms-grid-layer allows for separate control.

Be very aware that row and column styles are 1-based, not 0-based. Re-program your JavaScript-oriented mind to remember this, as you'll need to do a little translation if you track child elements in a 0-based array.

Also, when referring to any of these -ms-grid* styles as properties in JavaScript, drop the hyphens and switch to camel case, as in msGrid, msGridColumns, msGridRowAlign, msGridLayer, and so on.

Overall, grids are fairly straightforward to work with, especially within Blend where you can immediately see how the grid is taking shape. Let's now take a look at a few tips and tricks that you might find useful.

Overflowing a Grid Cell

One of the great features of the grid, depending on your point of view, is that overflowing content in a grid cell doesn't break the layout at all—it just overflows. (This is very different from tables!) What this means is that you can, if necessary, offset a child element within a grid cell so that it overlaps an adjacent cell (or cells). Besides not breaking the layout, this makes it possible to animate elements moving between cells in the grid, if desired.

A quick example of content that extends outside its containing grid cell can be found in the GridOverflow example with this chapter's companion content. For the most part, it creates a 4x4 grid of rectangles, but this code at the end of the doLayout function (js/default.js), places the first rectangle well outside its cell:

```
children[0].style.width = "350px";
children[0].style.marginLeft = "150px";
children[0].style.background = "#fbb";
```

This makes the first element in the grid wider and moves it to the right, thereby making it appear inside the second element's cell (the background is changed to make this obvious). Yet the overall layout of the grid remains untouched.

I'll cast a little doubt on this being a great feature because you might not want this behavior at times, hoping instead that the grid would resize to the content. For that behavior, use an HTML table.

Centering Content Vertically

Somewhere in your own experience with CSS, you've probably made the bittersweet acquaintance with the vertical-align style in an attempt to place a piece of text in the middle of a div, or at the bottom. Unfortunately, it doesn't work: this particular style works only for table cells and for inline content (to determine how text and images, for instance, are aligned in that flow).

As a result, various methods have been developed to do this, such as those discussed in http://blog.themeforest.net/tutorials/vertical-centering-with-css/. Unfortunately, just about every technique depends on fixed heights—something that can work for a website but doesn't work well for the adaptive layout needs of a Windows Store app. And the one method that doesn't use fixed heights uses an embedded table. Urk.

Fortunately, both the CSS grid and the flexbox (see "Item Layout" later on) easily solve this problem. With the grid, you can just create a parent div with a 1x1 grid and use the -ms-grid-row-align: center style for a child div (which defaults to cell 1, 1):

```
<!-- In HTML -->
<div id="divMain">
    <div id="divChild">
        <p>Centered Text</p>
    </div>
</div>

/* In CSS */
#divMain {
    width: 100%;
    height: 100%;
    display: -ms-grid;
    -ms-grid-rows: 1fr;
    -ms-grid-columns: 1fr;
}

#divChild {
    -ms-grid-row-align: center;
    -ms-grid-column-align: center;

    /* Horizontal alignment of text also work with the following */
    /* text-align: center; */
}
```

The solution (below) is even simpler with the `flexbox` layout, where `flex-align: center` handles vertical centering, `flex-pack: center` handles the horizontal, and a child `div` isn't needed at all. You can use this styling for any centering needs, such as placing content in the middle of a fixed layout.

```
<!-- In HTML -->
<div id="divMain">
    <p>Centered Text</p>
</div>

/* In CSS */
#divMain {
    width: 100%;
    height: 100%;
    display: -ms-flexbox;
    -ms-flex-align: center;
    -ms-flex-direction: column;
    -ms-flex-pack: center;
}
```

Code for both these methods can be found in the CenteredText example for this chapter, with the Grid method commented out in css/default.css. (This example also demonstrates the use of ellipses later on, so there's more text in the markup.)

Scaling Font Size

One particularly troublesome area with HTML is figuring out how to scale a font size with an adaptive layout. I'm not suggesting you do this with the standard typography recommended by Windows app design as we saw earlier in this chapter. It's more a consideration when you need to use fonts in some other aspect of your app such as large letters on a tile in a game.

With an adaptive layout, you typically want certain font sizes to be proportional to the dimensions of its parent element. (It's not a concern if the parent element is a fixed size, because then you can fix the size of the font.) Unfortunately, percentage values used in the `font-size` style in CSS are based on the default font size (1em), not the size of the parent element as happens with `height` and `width`. What you'd love to be able to do is something like `font-size: calc(height * .4)`, but, well, the value of other CSS styles on the same element are just not available to `calc`.

One exception to this is the `vh` value (which can be used with `calc`). If you know, for instance, that the text you want to scale is contained within a grid cell that is always going to be 10% of the viewport height and if you want the font size to be half of that, you can just use `font-size: 5vh` (5% of viewport height).

Another method is to use an SVG for the text, wherein you can set a `viewBox` attribute and a `font-size` relative to that `viewBox`. Scaling the SVG to a grid cell will effectively scale the font:

```
<svg viewBox="0 0 600 400" preserveAspectRatio="xMaxYMax">
   <text x="0" y="150" font-size="200" font-family="Verdana">
      Big SVG Text
   </text>
</svg>
```

You can also use JavaScript to calculate the desired font size programmatically based on the `clientHeight` property of the parent element. If that element is in a grid cell, the font size (and line height) can be some percentage of that cell's height, thereby allowing the font to scale with the cell.

Item Layout

So far in this chapter we've explored page-level layout, which is to say, how top-level items are positioned on a page, typically with a CSS grid. Of course, it's all just HTML and CSS—you can use tables, line breaks, and anything else supported by the rendering engine so long as you adapt well to view sizes.

It's also important to work with item layout in the flexible areas of your page. That is, if you set up a top-level grid to have a number of fixed-size areas (for headings, title graphics, control bars, etc.), the remaining area can vary greatly in size as the window size changes. Let's look now at some of the tools we have within those specific regions: CSS transforms, flexbox, nested and inline grids, multicolumn text, CSS figures, and CSS connected frames.

I don't intend to teach you all the details of CSS, so let me give you a few other resources. A general reference for these and all other CSS styles that are supported for Windows Store apps (such as background, borders, and gradients) can be found on the [Cascading Style Sheets](#) topic (note the [`border-image`](#) style that's available in Windows 8.1 and Internet Explorer 11). Also check out *CSS for Windows 8 App Development*, by Jeremy Foster (APress, 2013). The CSS specifications themselves can be found on http://www.w3.org/; specifically start with the HTML & CSS standards reference. I also highly recommend the well-designed and curated resources from Smashing Magazine for learning the many nuances of CSS, which I must admit still seems mysterious to me at times!

CSS 2D and 3D Transforms

It's quite impossible to think about layout for elements without taking CSS transforms into consideration. Transforms are very powerful because they make it possible to change the display of an element without actually affecting the document flow or the overall layout. This is very useful for animations and transitions; transforms are used heavily in the WinJS animations library that provides the Windows look and feel for all the built-in controls. As we'll explore in Chapter 14, "Purposeful Animations," and as we've seen with the ExtendedSplashScreen example in the companion content, you can make direct use of this library as well.

CSS transforms can be used directly, of course, anytime you need to translate, scale, or rotate an element. Both 2D and 3D transforms (http://dev.w3.org/csswg/css3-2d-transforms/ and http:/www.w3.org/TR/css3-3d-transforms/) are supported for Windows Store apps, specifically these styles:[75]

CSS Style	**JavaScript Property (element.style.)**
`backface-visibility`	`backfaceVisibility`
`perspective`, `perspective-origin`	`perspective`, `perspectiveOrigin`
`transform`, `transform-origin`, and `transform-style`	`transform`, `transformOrigin`, and `transformStyle`

Full details can be found on the Transforms reference. Know also that because the app host uses the same underlying engines as Internet Explorer, transforms enjoy all the performance benefits of hardware acceleration. Be aware when doing animations and transitions, however, that hardware acceleration only happens for `transform` and `opacity` styles, not for `perspective`.

Flexbox

Just as the grid is magnificent for solving many long-standing problems with page layout, the CSS flexbox module, documented at http://www.w3.org/TR/css3-flexbox/, is excellent for handling variable-sized areas wherein the content wants to "flex" with the available space. To quote the W3C specification:

> *In [this] box model, the children of a box are laid out either horizontally or vertically, and unused space can be assigned to a particular child or distributed among the children by assignment of 'flex' to the children that should expand. Nesting of these boxes (horizontal inside vertical, or vertical inside horizontal) can be used to build layouts in two dimensions.*

The specific display styles for Store apps are `display: -ms-flexbox` (block level) and `display: -`

[75] At the time of writing, the `-ms-*` prefixes on these styles were no longer needed but are still supported.

ms-inline-flexbox (inline). For a complete reference of the other supported properties, see the Flexible Box ("Flexbox") Layout documentation:[76]

CSS Style	JavaScript Property (element.style.)	Values
-ms-flex-align	msFlexAlign	start | end | center | baseline | stretch
-ms-flex-direction	msFlexDirection	row | column | row-reverse | column-reverse | inherit
-ms-flex-flow	msFlexFlow	<direction> <pack> where <direction> is an -ms-flex-direction value and <pack> is an -ms-flex-pack value.
-ms-flex-order	msFlexOrder	<integer> (ordinal group)
-ms-flex-pack	msFlexPack	start | end | center | justify
-ms-flex-wrap	msFlexWrap	none | wrap | wrap-reverse

As with all styles, Blend is a great tool in which to experiment with different flexbox styles because you can see the effect immediately. It's also helpful to know that flexbox is used in a number of places around WinJS. The ListView control in particular takes advantage of it, allowing more items to appear when there's more space. The FlipView uses flexbox to center its items, and the Ratings, DatePicker, and TimePicker controls all arrange their inner elements using an inline flexbox. It's likely that your own custom controls will do the same.

Nested and Inline Grids

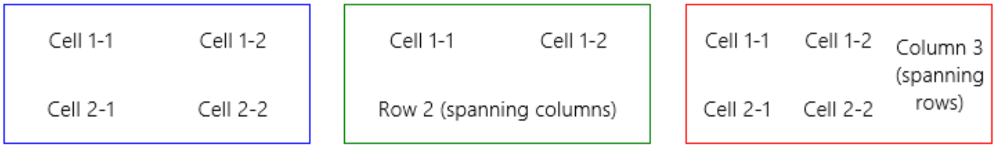
Just as the flexbox has both block level and inline models, there is also an inline grid: display: -ms-inline-grid. Unlike the block level grid, the inline variant allows you to place several grids on the same line. This is shown in the InlineGrid example for this chapter, where we have three div elements in the HTML that can be toggled between inline (the default) and block level models:

```
//Within the activated handler
document.getElementById("chkInline").addEventListener("click", function () {
    setGridStyle(document.getElementById("chkInline").checked);
});

setGridStyle(true);


//Elsewhere in default.js
function setGridStyle(inline) {
    var gridClass = inline ? "inline" : "block";

    document.getElementById("grid1").className = gridClass;
    document.getElementById("grid2").className = gridClass;
    document.getElementById("grid3").className = gridClass;
}
```

[76] If you're accustomed to the -ms-box* styles for flexbox, Microsoft aligned to the W3C specifications as they existed at that time for Windows 8 and Internet Explorer 10. As of Windows 8.1 and Internet Explorer 11 and this writing, the [-]ms prefixes are apparently no longer needed, but they are still what's recognized by Visual Studio's IntelliSense, so I'm showing them as such here.

```
/* default.css */
.inline {
    display: -ms-inline-grid;
}

.block {
    display: -ms-grid;
}
```

When using the inline grid, the elements appear as follows:

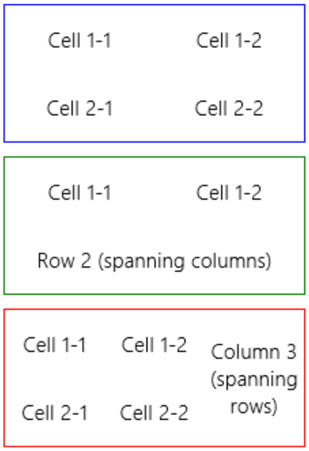


Cell 1-1 Cell 1-2
Cell 2-1 Cell 2-2

Cell 1-1 Cell 1-2
Row 2 (spanning columns)

Cell 1-1 Cell 1-2 Column 3 (spanning rows)
Cell 2-1 Cell 2-2

When using the block level grid, we see this instead:



Cell 1-1 Cell 1-2
Cell 2-1 Cell 2-2

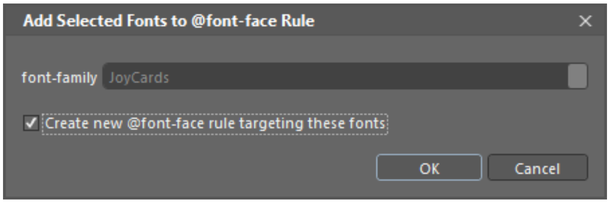
Cell 1-1 Cell 1-2
Row 2 (spanning columns)

Cell 1-1 Cell 1-2 Column 3 (spanning rows)
Cell 2-1 Cell 2-2

Fonts and Text Overflow

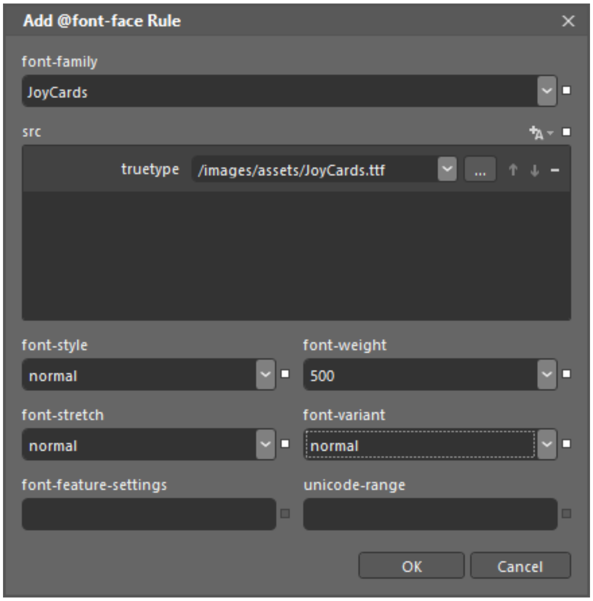
As discussed earlier, typography is an important design element for Store apps, and for the most part the standard font styles using Segoe UI are already defined in the default WinJS stylesheets. The very helpful CSS typography JS sample in the Windows SDK compares the HTML header elements and the `win-type-*` styles, demonstrating font fallbacks and how to use bidirectional fonts (left to right and right to left directions).

Speaking of fonts, custom font resources using the `@font-face` rule in CSS are allowed in Store apps. For local context pages, the `src` property for the rule must refer to an in-package font file (that is, a URI that begins with `/` or `ms-appx:///`). Pages running in the web context can load fonts from remote sources. Blend for Visual Studio 2013 supports this directly. First import a font into your project in Blend: right-click a project folder in the Project tab and select Add Existing Item, navigate to your font file, and press OK. As soon as you do so, Blend will bring up a series of two dialog boxes asking if you'd like to create an `@font-face` rule:


Add Selected Fonts to @font-face Rule

font-family: JoyCards

☑ Create new @font-face rule targeting these fonts


Add @font-face Rule

font-family: JoyCards

src: truetype /images/assets/JoyCards.ttf

font-style: normal

font-weight: 500

font-stretch: normal

font-variant: normal

font-feature-settings:

unicode-range:

Completing these dialogs (such as adding multiple font files in the second dialog) will add the appropriate CSS to your stylesheet. For a single font it looks like this:

```
@font-face {
    src: url('/images/assets/JoyCards.ttf') format('truetype');
    font-family: JoyCards;
    font-style: normal;
    font-weight: 500;
    font-stretch: normal;
    font-variant: normal;
}
```

which makes the font available in the CSS Properties pane alongside all other installed fonts.

Another piece of text and typography is dealing with text that overflows its assigned region. You can use the CSS `text-overflow: ellipsis;` style to crop the text with a ..., and the WinJS stylesheets contain the `win-type-ellipsis` class for this purpose. In addition to setting `text-overflow`, this class also adds `overflow: hidden` (to suppress scrollbars) and `white-space: nowrap`. It's basically a style you can add to any text element when you want the ellipsis behavior.

The W3C specification on text overflow, http://dev.w3.org/csswg/css3-ui/#text-overflow, is a helpful reference as to what can and cannot be done here. One of the limitations of the current spec is that multiline wrapping text doesn't work with ellipsis. That is, you can word-wrap with the `word-wrap: break-word` style, but it won't cooperate with `text-overflow: ellipsis` (`word-wrap` wins). I also investigated whether flowing text from a multiline CSS region (see next section) into a single-line region with ellipsis would work, but `text-overflow` doesn't apply to regions. If you want to do the work, you can use complex CSS approaches or you can just shorten the text and insert ellipses manually if it spans multiple lines. An MSDN forum post on multiline ellipses shows some options.

For a demonstration of ellipsis and word-wrapping, see the CenteredText example for this chapter. By default, the example shows ellipsis. To see word wrapping, remove the `win-type-ellipsis` class from the `divChild` element in default.html and add `word-wrap: break-word` to the `.textbox` class in css/default.css.

Multicolumn Elements and Regions

Translating the multicolumn flow of content that we're so accustomed to in print media has long been a difficult proposition for web developers. While it's been easy enough to create elements for each column, there was no inherent relationship between the content in those columns. As a result, developers have had to programmatically determine what content could be placed in each element, accounting for variations like font size or changing the number of columns based on the screen width or changes in device orientation.

CSS3 provides for doing multicolumn layout within an element (see http://www.w3.org/TR/css3-multicol). With this, you can instruct a single element to lay out its contents in multiple columns, with specific control over many aspects of that layout. The specific styles supported for Windows Store apps (with no pesky little vendor prefixes!) are as follows:

CSS Styles	**JavaScript Property (element.style.)**
`column-width` and `column-count` (`columns` is the shorthand)	`columnWidth`, `columnCount`, and `columns`
`column-gap`, `column-fill`, and `column-span`	`columnGap`, `columnFill`, and `columnSpan`
`column-rule-color`, `column-rule-style`, and `column-rule-width` (`column-rule` is the shorthand for separators between columns)	`columnRuleColor`, `columnRuleStyle`, and `columnRuleWidth` (`columnRule` is the shorthand)
`break-before`, `break-inside`, and `break-after`	`breakBefore`, `breakInside`, and `breakAfter`
`overflow: scroll` (to display scrollbars in the container)	`overflow`

The reference documentation for these can be found on Multi-column layout, and Blend, of course, provides a great environment to explore how these different styles work. If you're placing a multicolumn element within a variable-size grid cell, you can set `column-width` and let the layout engine add and remove columns as needed, or you can use media queries or JavaScript to set `column-count` directly.

CSS3 multicolumn again only applies to the contents of a single element. While highly useful, it does impose the limitation of a rectangular element and rectangular columns (spans aside). Certain apps like magazines need something more flexible, such as the ability to flow content across multiple elements with more arbitrary shapes, and columns that are offset from one another.

To support irregular columns, CSS Regions (see http://dev.w3.org/csswg/css3-regions/) are supported in Store apps (see Regions reference). Regions allow arbitrarily (that is, absolutely) positioned elements like images to interact with inline content.

The key style for a positioned element is the `float: -ms-positioned` style which should accompany `position: absolute`. Basically that's all you need to do: drop in the positioned element, and the layout engine does the rest. It should be noted that CSS Hyphenation, yet another module, relates closely to all this because doing dynamic layout on text immediately brings up such matters. Fortunately, Store apps support the `-ms-hyphens` and the `-ms-hyphenation-*` styles (and their equivalent JavaScript properties). The hyphenation spec is located at http://www.w3.org/TR/css3-text/; documentation for Store apps is found on the Text styles reference.

The second part of the story consists of named flows and region chains (which are also part of the Regions spec). These provide the ability for content to flow across multiple container elements, as shown in Figure 8-14. Region chains can also allow the content to take on the styling of a particular container, rather than being defined at the source. Each container, in other words, gets to set its own styling and the content adapts to it, but commonly all the containers share similar styling for consistency.

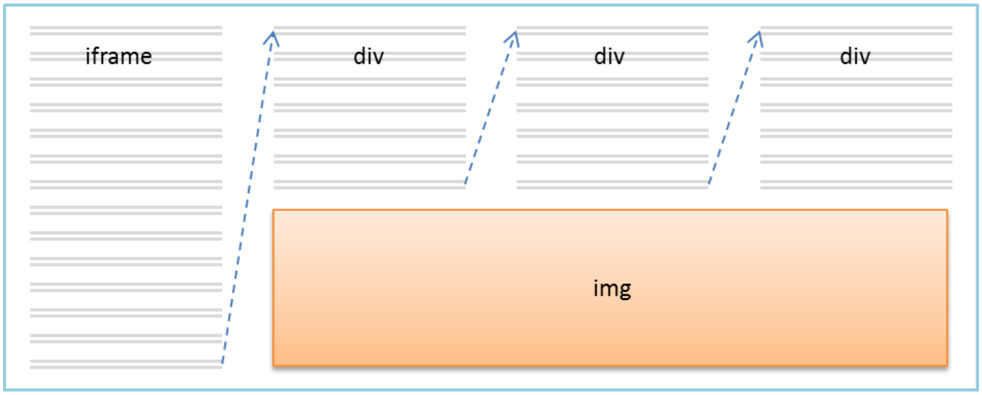


FIGURE 8-14 CSS region chains to flow content across multiple elements.

How this all works is that the source content is defined by an `iframe` that points to an HTML file (and the `iframe` can be in the web or local context, of course). It's then styled with `-ms-flow-into: <element>` (`msFlowInfo` in JavaScript) where `<element>` is the id of the first container:

```
<!-- HTML -->
<iframe id="s1-content-source" src="/html/content.html"></iframe>
<div class="s1-container"></div>
```

```
<div class="s1-container"></div>
<div class="s1-container"></div>

/* CSS */
#s1-content-source {
    -ms-flow-into: content;
}
```

Note that -ms-flow-into prevents the iframe content from displaying on its own.

Container elements can be any *nonreplaced* element—that is, any element whose appearance and dimensions are not defined by an external resource, such as img—and can contain content between its opening and closing tabs, like a div (the most common) or p. Each container is styled with -ms-flow-from: <element> (msFlowFrom in JavaScript) where the <element> is the first container in the flow. The layout then happens in the order elements appear in the HTML (as above):

```
.s1-container {
    -ms-flow-from: content;
    /* Other styles */
}
```

This simple example was taken from the Static CSS regions sample (Windows 8), which also provides a few other scenarios; the Dynamic CSS regions sample (Windows 8) is also helpful. (Note: these are Windows 8 samples and will need to be retargeted for Windows 8.1; they were not updated for 8.1, but they work just fine.) In all cases, though, be aware that styling for regions is limited to properties tha t affect the container and not the content—content styles are drawn from the iframe HTML source. This is why using text-overflow: ellipsis doesn't work, nor will font-color and so forth. But styles like height and width, along with borders, margin, padding, and other properties that don't affect the content can be applied.

What We've Just Learned

- Layout that is consistent with Windows design principles—specifically the silhouette and typography—helps users focus immediately on content rather than having to figure out each specific app.
- The principle of "content before chrome" allows content to use 75% or more of the display space rather than 25% as is common with chrome-heavy desktop or web applications.
- An app can create and manage multiple views that can be displayed on separate monitors or adjacent to one another on the same monitor (in landscape mode).
- The user is always in control of view size and placement. All views can be sized down to 500px wide by the height of the display, and an app can optionally support a 320px narrow view.
- Every page of an app (including an extended splash screen) can encounter all view sizes, so an app design must show how those views are handled. Media queries and the Media Query

Listener API, along with `window.onresize`, can be used to handle view sizing declaratively and programmatically.

- Apps can specify a preferred orientation in their manifest and also lock the orientation at run time.
- Handling varying screen sizes is accomplished either through a grid-based adaptive layout or a fixed layout utilizing CSS transforms to scale of its content.
- The chief concern with pixel density is providing graphics that scale well. This means either using vector graphics or providing scaled variants of each raster graphic.
- Pannable HTML sections can use snap points to automatically stop panning at particular intervals within the content and can use railing to limit panning to one dimension at a time. Snap points are also available for zooming.
- The `WinJS.UI.Hub` control supports creating a home or hub page of an app with varied and content that does not come from a single collection. Straight HTML/CSS layout can also be used to achieve this.
- Windows Store apps can take advantage of a wide range of CSS 3 options, including the grid, flexbox, transforms, multicolumn text, and regions. The CSS grid is a highly useful mechanism for adaptive page-level layout, and it can also be used inline. The CSS flexbox is most useful for inline content, though it has uses at the page level as well, as for centering content vertically and horizontally.

Chapter 9

Commanding UI

For consumers coming anew to Windows 8, Windows 8.1, and Windows Store apps, one of their first reactions might be "Where are the menus? Where is the ribbon? How do I tell this app to do something with the items I selected from a list?" This will be a natural response until users become more accustomed to where commands live, giving another meaning, albeit a mundane one, to the dictum "Blessed are those who have not seen, and yet believe!"

With the design principle of "content before chrome," UI elements that exist solely to invoke actions and don't otherwise contain meaningful content fall into the category of "chrome." As such, they are generally kept out of sight until needed, as are system-level commands like the Charms bar. The user indicates his or her desire for those commands through an appropriate gesture. A swipe on the top or bottom edge of the display, a right mouse button click, or the Win+Z key combination brings up app-specific commands at the top and bottom. A swipe on the left edge of the display, a mouse click on the upper left corner, or Win+Tab allows for switching between apps. And a swipe on the right edge of the display, a mouse click on the upper-right or lower-right corner, or Win+C reveals the Charms bar. (Win+Q, Win+H, and Win+i open the Search, Share, and Settings charms directly.) An app responds to the different charms through particular contracts, as we'll see in a number of the chapters that follow.

App-specific commands, for their part, are provided through an app bar control: `WinJS.UI.AppBar`. In many ways, the app bar is the equivalent of a menu and ribbon for Windows Store apps, because you can create all sorts of UI within it and even show menu elements. Menus, supplied by the `WinJS.UI.Menu` control, can also pop up from specific points on the app's main display, such as a menu attached to a header. There is also a special case app bar for navigation: `WinJS.UI.NavBar`.

The app/nav bar and menus are specific instances of the more generic `WinJS.UI.Flyout` control, which is used directly for messages or actions that the user can cancel or ignore; such flyouts are dismissed simply by clicking or tapping outside the flyout's window. (This is like pressing a Cancel button.) For important messages that require action—that is, where the user must choose between a set of options—apps employ `Windows.UI.Popups.MessageDialog`. Dialog boxes are a familiar concept from the world of desktop applications and have long been used for collecting all kinds of information and adjusting app settings. In Windows Store app design, however, dialog boxes are used only to ask a question and get a simple answer, or just to inform the user of some condition that did not arise directly from user interaction. Settings are specifically handled through the Settings charm, as we'll see in Chapter 10, "The Story of State, Part 1."

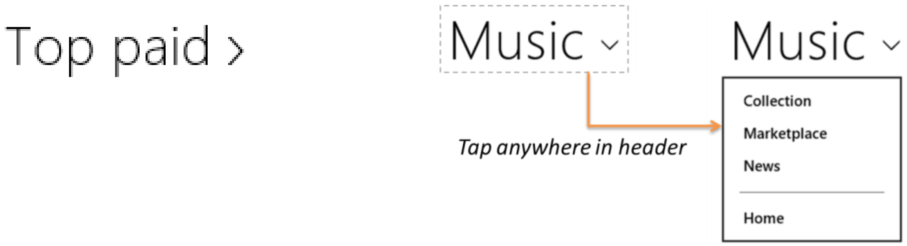
An important point with all of these command controls is that they don't participate in page layout: they instead "fly out" and remain on top of the current page. This means we thankfully don't need to worry about their impact on layout…with one small exception that I'll keep secret for now.

To begin with, though, let's take a step back to think about an app's commands as a whole and where those commands are ideally placed.

Where to Place Commands

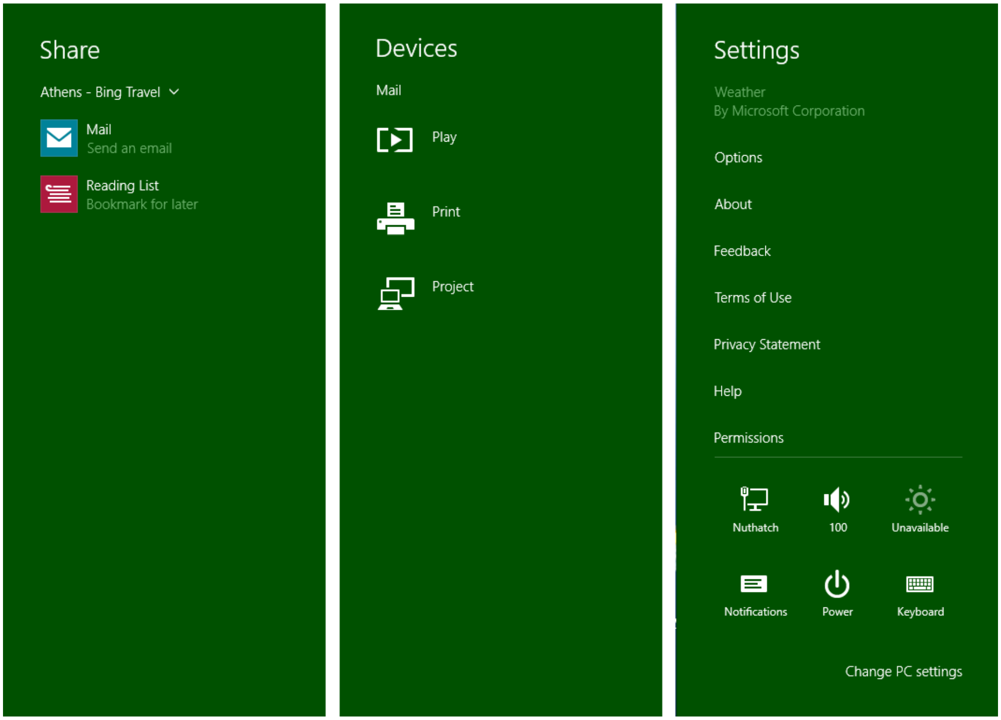
The placement of commands is really quite central to app design. Unlike the guidelines—or lack thereof!—for desktop application commands, which has resulted in quite a jumble, the Windows Developer Center offers two rather extensive topics on this subject: Commanding design and Laying out your UI. These are must-reads for any designer working on an app, because they describe the different kinds of commanding UI and how to gain the best smiling accolades from Windows design pundits. These are also good topics for developers because they can give you some idea of what you might expect from your designers. Let's review that guidance, then, as an introductory tour to the various options:

- The user should be able to complete their most important scenarios using just the *app canvas*, so commands that are essential to a workflow should appear directly on-screen. The overall purpose here is to minimize the distraction of unnecessary commands. Nonessential commands should be kept out of view, except for navigation options where a single navigation command uses a forward chevron (below left) and those with multiple options use a down chevron and a drop-down menu (below right):

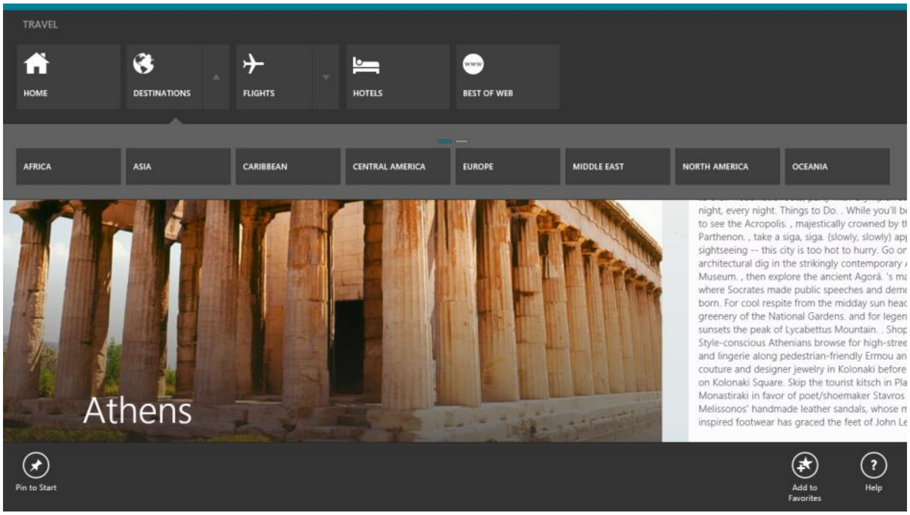


- Use *Charms* for common app commands where possible. That is, instead of supplying individual commands to share with specific targets such as email apps, your contacts, social network apps, and the like, use the Share charm. Instead of supplying your own Print commands, rely on the Device charm. And instead of creating pages within your navigation hierarchy for app settings, Help, About, permissions, license agreements, privacy statements, and login/account management, simplify your life and use the Settings charm! (Refer also to "Sidebar: Logins and License Agreements.") Examples of the charms are shown in the image below, which also illustrates that many app commands can leverage the Charms bar, which means less clutter in the rest of your commanding UI. Again, we'll cover how to respond to Charms events in later chapters.

  An exception to this guidance for charms is Search, which typically is placed on the app canvas with the `WinJS.UI.SearchBox` control, but an app can also use the Search contract to work with the Search Charm. We'll learn about search in Chapter 15, "Contracts."

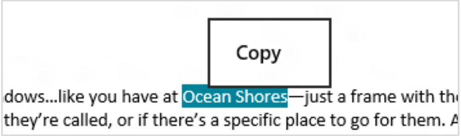


- Commands that can't be placed in Charms and don't need to be on the app canvas are then placed within the *nav bar* and *app bar*, as shown below in the Travel app. Both bars are invoked together by swiping in from either the top or bottom edges and are the closest analogies to traditional menus:
  - The top bar should contain navigation commands.
  - The bottom app bar contains all other commands that are sensitive to the context or selection, as well as global (nonselection) commands. Context and global commands are placed on different sides of the app bar.
  - App bar and nav bar commands can display menus to group related commands to reduce clutter; below, the Destinations button in the nav bar opens up the secondary list.



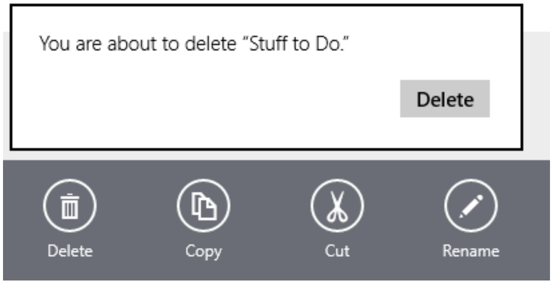
night, every night. Things to Do. . While you'll be
to see the Acropolis. , majestically crowned by th
Parthenon. , take a siga, siga. (slowly, slowly) app
sightseeing -- this city is too hot to hurry. Go on
architectural dig in the strikingly contemporary A
Museum. , then explore the ancient Agorá. 's ma
where Socrates made public speeches and demo
born. For cool respite from the midday sun head
greenery of the National Gardens. and for legen
sunsets the peak of Lycabettus Mountain. . Shop
Style-conscious Athenians browse for high-stree
and lingerie along pedestrian-friendly Ermou an
couture and designer jewelry in Kolonaki before
on Kolonaki Square. Skip the tourist kitsch in Pla
Monastiraki in favor of poet/shoemaker Stavros
Melissonos' handmade leather sandals, whose m
inspired footwear has graced the feet of John Le

- *Context menus* can provide specific commands for particular content or a selection. For example, selected text typically provides a context menu for clipboard commands, as shown here in the Mail app.



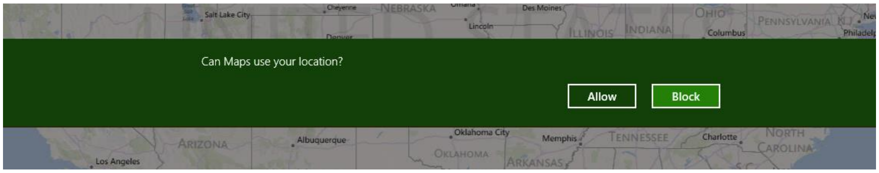
dows...like you have at Ocean Shores—just a frame with th
they're called, or if there's a specific place to go for them. A

- Confirmations and other questions (including collecting information) that you need to display *in response to a user action* should use a *flyout* control; see Guidelines and checklist for Flyouts. Tapping or clicking outside the control (or pressing ESC) is the same as canceling. Here's an example from the OneDrive app when using the Delete button:



You are about to delete "Stuff to Do."

- For blocking events that are not related to a user command but that affect the whole app, use a *message dialog*. A message dialog disables the rest of the app until you pay attention to it! A good example of this is a loss of network connectivity, where the user needs to be informed that some capabilities may not be available until connectivity is restored. User consent prompts for capabilities like geolocation, as shown below from the Maps app, is another place you see message dialogs. Note that a message dialog is used only when the app is in the foreground. Toast notifications, as we'll see in Chapter 16, "Alive with Activity," are how background apps and tasks get the user's attention.

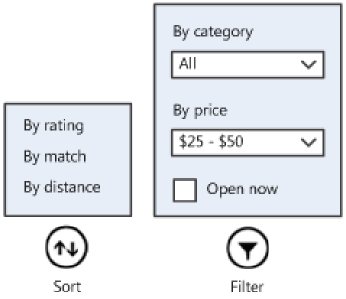

Can Maps use your location?

- Finally, other errors that don't require user action can be displayed either inline (on the app canvas) or through flyouts. See [Laying out your UI: errors]() for full details; we'll see some examples later on as well.

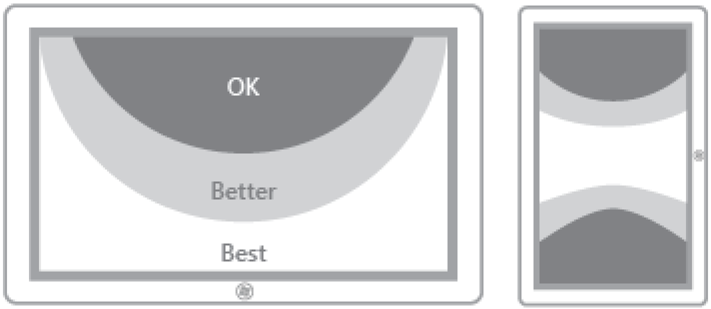
Where the app bar (on the bottom) is concerned, it's also important to organize your commands into sets, as this streamlines implementation as we'll see in the next section. For full guidance I recommend [Guidelines or app bars]() and [Commanding Design]() in the documentation, which provide many specifics on placement, spacing, and grouping. That guidance can be summarized as follows:

- First, make two groups of commands: one with those commands that appear throughout the entire app, regardless of context, and another with those that show only on certain pages. The app bar control is fairly simple to reconfigure at run time for different groups.

- Next, create command sets, such as those that are functionally related, those that toggle view types, and those that apply to selections. Remember that an app bar command can display a popup menu, as shown below, to provide a list of options and/or additional controls, including longer labels, drop-down lists, checkboxes, radiobuttons, and toggle switches. In this way you can combine closely related commands into a single one that gets more room to play than its little space on the app bar proper.



- For placement, put persistent commands on the right side of the app bar and the most common context-specific commands on the left. After that, populate additional commands toward the middle. This recommendation comes from the ergonomic realities of human hands: fingers and thumbs—even on the largest hands of basketball players!—grow only so long and can reach only so far on the screen without having to move one's hand. The most commonly used commands are best placed nearest to where a person's thumbs will be when holding a device, as indicated in the image below (from the Touch interaction design topic in the docs). Those spots are easier to reach (especially by those of us that can't grip a large ball with one hand!) and thus make the whole user experience more comfortable.



- The nav bar and app bar are always available in all views, regardless of size; in narrower views you can limit app bar commands to around 10 so that they fit into one or two rows, and also omit labels and tighten up the hit targets. The WinJS app bar does this automatically.
- Know too that the app bar is not limited to circular command buttons: you can create whatever custom layout you like, which is how the nav bar is implemented. With any custom layout, make sure that your elements are appropriately sized for touch interaction. More on this—including a small graphic of the aforementioned finger of a basketball player—can again be found on Guidelines for app bars as well as Touch interaction design under "Touch targets."

Sidebar: Logins and License Agreements

As noted above, Microsoft recommends that login/account management and license agreements/terms-of-use pages are accessed through the Settings charm, where an app adds relevant commands to the Settings pane that appears when the charm is invoked. These commands invoke subsidiary pages with the necessary controls for each functions. Of course, sometimes logins and license agreements need some special handling. For example, if your app *requires* a login or license agreement on startup, such controls can be shown on the app's first page, through the Web Authentication Broker (see Chapter 4, "Web Content and Services"), or in enterprise scenarios through the Credential Picker UI (see Appendix C, "Additional Networking Topics"). If the user provides a login and/or agrees to the terms of service, the app can continue to run. Otherwise, the app should show a page that indicates that a login or agreement is necessary to do something more interesting than stare at error messages.

If a login is *recommended* but not required, perhaps to enable additional features, you can place those controls directly on the canvas. When the user logs in, you can replace those controls with bits of profile information (user name and picture, for example, as on the Windows Start screen). If, on the other hand, a login is entirely optional, keep it within Settings.

In all cases, commands to view the license agreement, manage one's account or profile, and log in or out should still be available within Settings. Other app bar or on-canvas commands can invoke Settings programmatically, as we'll see in Chapter 10.

The App Bar and Nav Bar

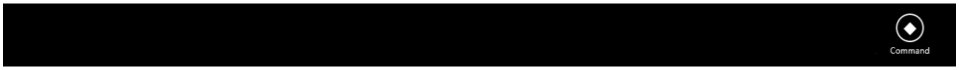
After placing essential commands on the app canvas, most of your app's commands will be placed in the app bar and navigation-specific commands in the top nav bar. Again, both bars are automatically brought up in response to various user gestures, such as a top or bottom edge swipe, Win+Z, or a right mouse button click. Whenever you perform one of these gestures, Windows looks for suitable controls on the current page and invokes them—you don't need to process any input events yourself. (Similarly, a click/tap outside the control or the ESC key dismisses them.)

> **Tip** To prevent the app/nav bar from appearing, you can do one of two things. First, to prevent an app bar or nav bar from appearing at all (for any gesture), set the control's `disabled` property to `true`. Second, if you want to prevent it for, say, a right-click on a particular element (such as a canvas), listen to the `contextmenu` (right click) event for that element and call `eventArgs.preventDefault()` within your handler.

For apps written in HTML and JavaScript, the app bar control is implemented as a WinJS control, `WinJS.UI.AppBar`, and the nav bar with `WinJS.UI.NavBar`. As with all other WinJS controls, you declare either or both controls in HTML and instantiate them with a call to `WinJS.UI.process` or `WinJS.UI.processAll`. For a first example, we don't need to look any farther than some of the Visual Studio/Blend project templates like the Grid App project, where a placeholder app bar is included in default.html (initially commented out):

```
<div id="appbar" data-win-control="WinJS.UI.AppBar">
   <button data-win-control="WinJS.UI.AppBarCommand"
         data-win-options="{id:'cmd', label:'Command', icon:'placeholder'}"></button>
</div>
```

The super-exciting result of this markup, using the WinJS ui-dark.css stylesheet, is as follows:



Because the app bar is declared in default.html, which is the container for all other page controls, *this same app bar will apply to all the pages in the app*. With this approach you can declare all your commands within a single app bar and assign different classes to the commands that allow you to easily show and hide command sets as appropriate for each page. This also centralizes those commands that appear on multiple pages, and you can wire up event handlers for them in your app's primary activation code (such as that in default.js).

Alternately, you can declare app/nav bars within the markup for individual page controls. Because the controls will be in the DOM, the Windows gestures will invoke it on each particular page. In the Grid App project, for example, you can move the markup above from default.html into groupedItems.html, groupDetail.html, and itemDetail.html with whatever modifications you like for each page. This might be especially useful if your app's pages don't share many commands in common.

In these cases, each page's `ready` method should take care of wiring up the commands on its particular app/nav bars. Note also that you can add handlers within a page's `ready` method even for central app/nav bars; it's just a matter of calling `addEventListener` on the appropriate child element within those controls. If it's in the DOM, you can add a listener!

Let's look now at how all this works through the HTML AppBar control sample and the HTML NavBar control sample. We'll look at app bars first, starting with the basics and the standard command-oriented configuration; then we'll look at how to display menus for some of those commands and see how to create custom layouts as is used for a top navigation bar. The WinJS NavBar control, in fact, is a derivative of a custom layout app bar, so we'll come to that at the end of this section.

> **Hint** Technically speaking, you can declare as many app/nav bars as you want in whatever pages you want, and they'll all be present in the DOM. However, the last one that gets processed in your markup will be the one that's topmost in the z-index by default and therefore the one to receive events. Windows does not make any attempt to combine app/nav bars, so because page controls are inserted into the middle of a host page like default.html, an app bar in default.html that's declared after the page control host element will appear on top. At the same time, if the page control's nav bar is larger than that in default.html, a portion of it might be visible. The bottom line: declare app/nav bars *either* in the host page or in a page control, but not both.

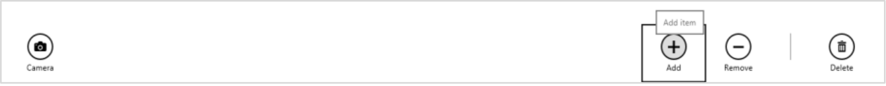
App Bar Basics and Standard Commands

As I just mentioned, an app bar can be declared once for an app in a container page like default.html or can be declared separately for each individual page control. The HTML AppBar control sample does the latter, because it provides very distinct app bars for its various scenarios.

Scenario 1 of the sample (html/create-appbar.html) declares an app bar with four commands and a separator:

```
<div id="createAppBar" data-win-control="WinJS.UI.AppBar" data-win-options="">
    <button data-win-control="WinJS.UI.AppBarCommand" data-win-options="{id:'cmdAdd',
        label:'Add', icon:'add', section:'global', tooltip:'Add item'}"></button>
    <button data-win-control="WinJS.UI.AppBarCommand" data-win-options="{id:'cmdRemove',
        label:'Remove', icon:'remove', section:'global', tooltip:'Remove item'}"></button>
    <hr data-win-control="WinJS.UI.AppBarCommand" data-win-options="{type:'separator',
        section:'global'}" />
    <button data-win-control="WinJS.UI.AppBarCommand" data-win-options="{id:'cmdDelete',
        label:'Delete', icon:'delete', section:'global', tooltip:'Delete item'}"></button>
    <button data-win-control="WinJS.UI.AppBarCommand" data-win-options="{id:'cmdCamera',
        label:'Camera', icon:'camera', section:'selection', tooltip:'Take a picture'}"></button>
</div>
```

This appears in the app as follows, using the WinJS ui-light.css stylesheet, in which we can also see a tooltip, a focus rectangle, and a hover effect on the Add command (I placed my mouse over the command to see all this):



In the markup, the app bar control is declared like any other WinJS control (this is becoming a habit!) using some containing element (a `div`) with `data-win-control="WinJS.UI.AppBar"`. Each page in this sample is loaded with `WinJS.UI.Pages.render` that conveniently calls `WinJS.UI.processAll` to instantiate the app bar. (It is also allowable, as with other controls, to create an app bar programmatically using the `new` operator.)

This example doesn't provide any specific options for the app bar in its `data-win-options`, but there are a number of possibilities:

- `disabled`, if set to `true`, creates an initially disabled app bar; the default is `false`.
- `layout` (a string) can be `commands` (the default) or `custom`, as we'll see in the "Custom App Bars" section later.
- `placement` (a string) can be either `top` or `bottom` (the default). The `top` option is the default for a nav bar.
- `sticky` changes the light-dismiss behavior of the app bar. With the default of `false`, the app bar will be dismissed when you click or tap outside of it. If this is set to `true`, the app bar will stay on the screen until either you change `sticky` to `false` and tap outside or you programmatically relieve the control from its duties with its `hide` method.

So, if you wanted a sticky nav bar with a custom layout to appear at the top of the screen, you'd use markup like this:

```
<div id="navBar" data-win-control="WinJS.UI.AppBar"
    data-win-options="{layout:'custom', placement:'top', sticky: true}">
```

Note that having two app bars in a page with different `placement` values will not interfere with each other. (Again, the `NavBar` derives from the `AppBar` and uses a custom layout and top placement by default, so you don't have use markup like the above.) Also, the `sticky` property for each placement operates independently. So if you want to implement an appwide top nav bar, you could declare that within default.html (or whatever your top-level page happens to be), and declare bottom app bars in each page control. Again, they're all just elements in the DOM!

As you can see, an app bar control can contain any number of child elements for its commands, each of which *must* be a `WinJS.UI.AppBarCommand` control within a `button` or `hr` element or else the app bar won't instantiate.

The properties and options of an app bar command are as follows:

- `id` The element identifier, which you can use with `document.getElementById` or the app bar's `getCommandById` method to wire up `click` handlers.

- `type` (a string) One of `button` (the default), `separator` (which creates a vertical bar), `flyout` (which triggers a popup specified with the `flyout` property; see "Command Menus" later), `toggle` (which creates a button with on/off states), and `content` (which allows for arbitrary controls as commands; see "Custom App Bars" later on). With `toggle`, the `selected` property of a command can also be used to set the initial value and to retrieve the state at run time.

- `label` The text shown below for the command button. You always want to use this instead of providing text for the `button` element itself, because such text won't be aligned properly in the control. (Try it and you'll see!) The app bar will also automatically hide labels in narrow views. Also, note that this property, along with `tooltip` below, is often localized using `data-win-res` attributes. We'll cover this in Chapter 19, "Apps for Everyone, Part 1," but for the time being you can look at the html/localize-appbar.html file in the sample (scenario 8) to see how it works.

- `tooltip` The (typically localized) tooltip text for the command, using the value of `label` as the default. Note that this is just text that gets passed to the element's `title` attribute, so using a full HTML-based `WinJS.UI.Tooltip` control here is not supported.

- `icon` Specifies the glyph that's shown in the command. Typically, this is one of the strings from the `WinJS.UI.AppBarIcon` enumeration, which contains 150 different options from the Segoe UI Symbol font. If you look in the ui.strings.js resource file of WinJS you can see how these are defined using codes like `\uE109`—the enumeration, in fact, simply provides friendly names for character codes `\uE100` through `\uE1E9`. But you're not limited by these. For one thing, you can use any other Unicode escape value `'\uXXXX'` from the Segoe UI Symbol font. (Note the single quotes.) You can also use a different font or use your own graphics as described in "Custom Icons" later.[77]

- `section` (a string) Controls the placement of the command. For left-to-right languages (such as English), the default value of `selection` places the command on the left side of the app bar and `global` places it on the right. For right-to-left languages (such as Hebrew and Arabic), the sides are swapped. These simple choices encourage consistent placement of these two categories of commands (and using any other random value here defaults to `selection`). This trains users' eyes to look for the most constant commands on one side and selection-specific commands on the other. Note that the commands in each section are laid out left-to-right (or right-to-left) in the order they appear in your markup.

[77] Three notes: First, within `data-win-options` the Unicode escape sequence can also be in the HTML form of *&#xNNNN;* I prefer the JSON form because it has less ceremony and is less prone to error. Second, you can use the Character Map desktop applet (charmap.exe) to examine all the symbols within any particular font. Third, if you need to localize an icon, you can specify the icon property in the `data-win-res` string because the `icon` property ultimately resolves to a string.

- firstElementFocus, lastElementFocus For commands of type content, gets or sets the element within that command's DOM tree that should receive the focus with the Home and End keys, respectively, and the arrow keys.
- onclick Can be used to declaratively specify a click handler; remember that any function named here in markup must be marked safe for processing. (See Chapter 5, "Controls and Control Styling," in the "Strict Processing and processAll Functions" section.) Click handlers can also be assigned programmatically with addEventListener, in which case the mark is not needed.
- disabled Sets the disabled state of a command if true; the default is false.
- extraClass Specifies one or more CSS classes that are attached to the command. These can be used to individually style command controls as well as to create sets that you can easily show and hide, as explained in the "Showing, Hiding, Enabling, and Updating Commands" section later.

If you want to generate commands at run time (not using the content type), you can do so by setting the app bar's commands property with an array of JSON AppBarCommand *descriptors* any time the app bar isn't visible (that is, when its hidden property is true). An array of such descriptors for the scenario 1 app bar in the sample would be as follows (this is provided in the modified sample included with this chapter; see js/create_appbar.js):

```
//Set the app bar commands property to populate it
var commands = [
    { id: 'cmdAdd', label: 'Add', icon: 'add', section: 'global', tooltip: 'Add item' },
    { id: 'cmdRemove', label: 'Remove', icon: 'remove', section: 'global',
       tooltip: 'Remove item' },
    { type: 'separator', section: 'global' },
    { id: 'cmdDelete', label: 'Delete', icon: 'delete', section: 'global',
       tooltip: 'Delete item' },
    { id: 'cmdCamera', label: 'Camera', icon: 'camera', section: 'selection',
       tooltip: 'Take a picture' }
];

appBar.commands = commands;
```

When the app bar is created, it will iterate through the commands array and create AppBarCommand controls for each item. If type isn't specified or if it's set to button, flyout, or toggle, then the command is a button element. A type of separator creates an hr element. The content type will create an empty div. Note that you should localize the label, tooltip, and possibly icon fields in each command declaration rather than using explicit text as shown here.

You can also use such an array directly within declarative markup, but this form cannot be localized and is thus discouraged (though I include comments that show how in the modified sample). At the same time, because the value of commands in markup is just a string, you can assign its value through data binding with an attribute like this in the app bar element:

```
data-win-bind="{ winControl.commands: Data.commands }"
```

where `Data.commands` can refer to a localized data source. In this case Data must be a global variable (like a namespace), and you must call `WinJS.Binding.processAll` on the app bar element (with no specific context) within a completed handler for `WinJS.UI.processAll` to make sure the app bar has been created first. Alternately, you can pass `Data` as the second argument to `Binding.processAll` and just use `commands` for the source in the `data-win-bind` string.

Note also that this approach does not work with the `data-win-res` attribute (as we'll see in Chapter 19 and which is also shown in scenario 8 of the sample) because the resource string won't be converted to JSON as part of the resource lookup. Attempting to play such a trick would be more trouble than it's worth anyway, so it's best to use either the HTML declarative form or a localized commands array at run time.

Also, be aware that `commands` is a rare example of a *write-only* property: you can set it, but you cannot retrieve the array from an app bar. The app bar uses this array only to configure itself and the array is discarded once all the elements are created in the DOM. At run time, however, you can use the app bar's `getCommandById` method to retrieve a particular command element.

Command Events

Speaking of the command elements, an app bar's `AppBarCommand` controls (other than separators) are all just `button` elements and thus respond to the usual events. Because each command element is assigned the `id` you specify, you can use `getElementById` as a prelude to `addEventListener`, but the more direct means is the app bar's `getCommandById` method. In scenario 1 of the HTML App Bar control sample, for instance, this code appears in the page's `ready` method (js/create-appbar.js):

```
var appBar = document.getElementById("createAppBar").winControl;
appBar.getCommandById("cmdAdd").addEventListener("click", doClickAdd, false);
appBar.getCommandById("cmdRemove").addEventListener("click", doClickRemove, false);
appBar.getCommandById("cmdDelete").addEventListener("click", doClickDelete, false);
appBar.getCommandById("cmdCamera").addEventListener("click", doClickCamera, false);
```

Of course, if you specify a handler for each command's `onclick` property in your markup (with each one having its `supportedForProcessing` property `true`), you can avoid all of this entirely!

It should also be obvious that you can wire up events from anywhere in your app, and you can certainly listen to any other events you want to, especially when doing custom app bar layouts with other UI. Also, know that the `click` event conveniently handles touch, mouse, and keyboard input alike, so you don't need to do any extra work there. In the case of the keyboard, by the way, the app bar lets you move between commands with the Tab key and the arrow keys; Enter or Spacebar will invoke the `click` handler.

App Bar Events and Methods

In addition to the app bar's `getCommandById` method we just saw, the app bar has several other methods and a handful of events. First, the methods:

- `show` displays an app bar if its `disabled` property is `false`; otherwise the call is ignored.

- hide dismisses the app bar.
- showCommands, hideCommands, and showOnlyCommands are used to manage command sets as described in the next section, "Showing, Hiding, Enabling, and Updating Commands."

As for events, there are a total of four that are common to the overlay-style UI controls in WinJS (that is, those that don't participate in layout):

- beforeshow occurs before a flyout becomes visible. For an app bar, this is when you could set the commands property depending on the state of the app at the moment or enable/disable specific commands.
- aftershow occurs immediately after a flyout becomes visible. For an app bar, if its sticky property is true, you can use this event to adjust the app's layout if you have a scrolling element that might be partially covered otherwise—see below.
- beforehide occurs before a flyout is hidden. For an app bar, you'd use this event to hide any supplemental UI created with the app bar and to readjust layout around a sticky app bar.
- afterhide occurs immediately after a flyout is hidden. For an app bar, this again could be a time to readjust the app's layout.

You can find an example of using the show method along with the aftershow and beforehide events in scenario 5 of the HTML AppBar control sample.

The matter with app layout identified above (and what I kept secret in the introduction to this chapter) arises because an app bar overlays and obscures the bottom portion of the page. If that page contains a scrolling element, an app bar with sticky set to true will, for mouse users, partly cover a vertical scrollbar and will make a horizontal scrollbar wholly inaccessible. If you're using a sticky app bar with such a page—and because Windows Store policy does not look kindly upon discrimination against mouse users!—you should use aftershow to reduce the scrolling element's height by the offsetHeight or clientHeight of the app bar control, thereby keeping the scrollbars accessible. When the app bar is hidden and afterhide fires, you can then readjust the layout. Always use a runtime value like clientHeight in these calculations as well, because it accommodates different view sizes and because the height of an app bar can vary with the number of commands and with view size.

To show this, scenario 7 of the sample has a horizontally panning ListView control that normally occupies most of the page; a scrollbar will appear along the very bottom when the mouse is used. If you select an item, the app bar is made sticky and then shown (see the doSelectItem function in js/appbar-listview.js):

```
appBar.sticky = true;
appBar.show();
```

The show method triggers both beforeshow and aftershow events. To adjust the layout, the appropriate event to use is aftershow, which makes sure the height of the app bar is valid. The sample handles this event in function called doAppBarShow (also in js/appbar-listview.js):

```
function doAppBarShow() {
    var listView = document.getElementById("scenarioListView");
    var appBarHeight = appBar.offsetHeight;
    // Move the scrollbar into view if appbar is sticky
    if (appBar. sticky) {
        var listViewTargetHeight = "calc(100% - " + appBarHeight + "px)";
        var transition = {
            property: 'height',
            duration: 367,
            timing: "cubic-bezier(0.1, 0.9, 0.2, 0.1)",
            to: listViewTargetHeight
        };
        WinJS.UI.executeTransition(listView, transition);
    }
}
```

> **Note** The SDK sample uses beforeshow instead of aftershow, with the result that sometimes the app bar still has a zero height and the layout is not adjusted properly. To guarantee that the app bar has its proper height for such calculations, use the aftershow event as demonstrated in the modified sample included with this chapter's companion content.

Here you can see that the appBar.offsetHeight value is simply subtracted from the ListView's height with an animated transition. (See Chapter 14, "Purposeful Animations.") The operation is reversed in doAppBarHide where the ListView height is simply reset to 100% with a similar animation. In this case, the event handler doesn't depend on the app bar's height at all, so it can use either beforehide or afterhide events. If, on the other hand, you need to know the size of the app bar for your own layout, use the beforehide event.

As an exercise, run scenario 8 of the SDK sample. Notice how the bottom part of the text region's vertical scrollbar is obscured by the sticky app bar. Try taking the code from scenario 7 to handle aftershow and beforehide to adjust the text area's height to accommodate the app bar and keep the scrollbar completely visible (they show and hide the app bar to see the scrollbar adjust). And no, I won't be grading you on this quiz: the solution is provided in the modified sample with this chapter.

Showing, Hiding, Enabling, and Updating Commands

In the previous section I mentioned using the beforeshow event to configure an app bar's commands property such that it contains those commands appropriate to the current page and the page state. This might include setting the disabled property for specific commands that are, for example, dependent on selection state. This can be done through the commands array, in markup, or again by using the app bar's getCommandById method:

```
appBar.getCommandById("cmdAdd").disabled = true;
```

Let me reiterate that the commands that appear on an app bar are specific to each page; it's not necessary to try to maintain a consistent app bar structure across pages. That is, if a command would always be disabled for a particular page, don't bother showing it at all. What's more important is that the app bar *for a page* is consistent, because it's a really bad idea to have commands appear and disappear depending on the state of the page. That would leave users guessing at how to get the page in the right state for certain commands to appear!

Speaking of changes, it is entirely allowable to modify or update a command at run time, which can eliminate the need to create multiple commands that your alternately show or hide. Because each command on the app bar is just a DOM element, you can really make any changes you want at any time. An example of this is shown in scenario 3 of the sample where the app bar is initially created with a Play button (html/custom-icons.html):

```
<button data-win-control="WinJS.UI.AppBarCommand"
       data-win-options="{id:'cmdPlay', label:'Play', icon:'play', tooltip:'Play this song'}">
</button>
```

This button's click handler uses the `doClickPlay` function in js/custom-icons.js to toggle between states:

```
var isPaused = true;

function doClickPlay() {
    var cmd = appBar.getCommandById('cmdPlay');

    if (!isPaused) {
        isPaused = true; // paused
        cmd.icon = 'play';
        cmd. label = 'Play';
        cmd. tooltip = 'Play this song';
    } else {
        isPaused = false; // playing
        cmd. icon = 'pause';
        cmd. label = 'Pause';
        cmd. tooltip = 'Pause this song';
    }
}
```

You can use something similar with a command to pin and unpin a secondary tile, as we'll see in Chapter 16. And again, the button is just an element in the DOM and updating any of its properties, including styles, will update the element on the screen once you yield control to the UI thread.

Now using `beforeshow` for the purpose of adjusting your commands is certainly effective, but you can accomplish the same goal in other ways. The strategy you use depends on the architecture of your app as well as personal preference. From the user's point of view, so long as the appropriate commands are available at the right time, it doesn't really matter how the app gets them there!

Thinking through your approach is especially important when dealing with narrow views, because the recommendation is that you limit your commands so that the app bar fits on one or two rows. This means that you will want to think through how to adjust the app bar for different view sizes, perhaps combining multiple commands into a popup menu on a single button.

One approach is to have each page in the app declare and handle its own app bar, which includes pages that create app bars on the fly within their `ready` methods. This makes the relationship between the page content and the app bar very clear and local to the page. The downside is that common commands—those that appear on more than one page—end up being declared multiple times, making them more difficult to maintain and certainly inviting small inconsistencies like ants to sugar. Nevertheless, if you have very distinct content in your various pages and few common commands, this approach might be the right choice. It is also necessary if your app uses multiple top-level pages rather than one page with page controls, as we discussed in Chapter 3, "App Anatomy and Performance Fundamentals," because each top-level HTML page has to declare its own app bar anyway.

For apps using page controls, another approach is to declare a single app bar in the top-level page and set its `commands` property within each page control's `ready` method. The drawback here is that because `commands` is a write-only property, you can't declare your common commands in HTML and append your page-specific commands later on, unless you go through the trouble of creating each individual `AppBarCommand` child element within each `ready` method. This kind of code is both tedious to write and to maintain.

Fortunately, there is a third approach that allows you to define a single app bar in your top-level page that contains *all* of your commands, for all of your pages, and then selectively show certain sets of those commands within each page's `ready` method. This is the purpose of the app bar's `showCommands`, `hideCommands`, and `showOnlyCommands` methods.

All three of these methods accept an array of commands, which can be either `AppBarCommand` objects or command id's. `showCommands` makes those commands visible and can be called multiple times with different sets for a cumulative result. On the opposite side, `hideCommands` hides the specified commands in the app bar, again with cumulative effects. The basic usage of these methods is demonstrated in scenario 5 of the sample.

`showOnlyCommands` then combines the two, making specific commands visible while hiding all others. If you declare an app bar with all your commands, you can use `showOnlyCommands` within each page's `ready` method to quickly and easily adjust what's visible. The trick is obtaining the appropriate array to pass to the method. You can, of course, hard-code commands into specific arrays, as scenario 5 of the sample does for `showCommands` and `hideCommands`. However, if you're thinking that this is A Classic Bad Idea, you're thinking like I'm thinking! Such arrays mean that any changes you make to app bar must happen in both HTML and JavaScript file, meaning that anyone having to maintain your code in the future will surely curse your name!

A better path to happiness and long life is thus to programmatically obtain the necessary arrays from the DOM, using each command's `extraClass` property to effectively define command sets. This enables you to call `querySelectorAll` to retrieve those commands that belong to a particular set.

Consider the following app bar definition, where for the sake of brevity I've omitted properties like `label`, `icon`, and `section`, as well as any other styling classes:

```
<div id="appbar" data-win-control="WinJS.UI.AppBar" data-win-options="{
   commands:[
       {id:'home', extraClass: 'menuView gameView scoreView'},
       {id:'play', extraClass: 'menuView gameView scoreView'},
        {id:'rules', extraClass: 'menuView gameView scoreView'},
        {id:'scores', extraClass: 'menuView gameView scoreView'},
        {id:'newgame', extraClass: 'gameView gameNarrowView'},
        {id:'resetgame', extraClass: 'gameView gameNarrowView'},
        {id:'loadgame', extraClass: 'gameView gameNarrowView'},
        {id:'savegame', extraClass: 'gameView gameNarrowView'},
        {id:'hint', extraClass: 'gameView gameNarrowView'},
        {id:'timer', extraClass: 'gameView gameNarrowView'},
        {id:'pause', extraClass: 'gameView gameNarrowView'},
        {id:'replaygame', extraClass: 'scoreView'},
        {id:'resetscores', extraClass: 'scoreView'}
    ]}">
</div>
```

In the extraClass properties we've defined four distinct sets: *menuView*, *gameView*, *gameNarrowView*, and *scoreView*. With these in place, a simple call to querySelectorAll provides exactly the array we need for showOnlyCommands. A generic function like the following can then be used from within each page's ready method (or elsewhere) to activate commands for a particular view:

```
function updateAppBar(view) {
    var appbar = document.getElementById("appbar").winControl;
    var commands = appbar.element.querySelectorAll("." + view); // The . is essential
    appbar.showOnlyCommands(commands);
}
```

With this approach, credit for which belongs to my colleague Jesse McGatha, the app bar is wholly defined in a single location, making it very easy to manage and maintain.

App Bar Styling

The extraClass property for commands can, of course, be used for styling purposes as well as managing command sets. It's very simple: whatever classes you specify in extraClass are added to the AppBarCommand controls created for the app bar.

There are also seven WinJS style classes utilized by the app bar, as described in the following table, where the first two apply to the app bar as a whole and the other five to the individual commands:

CSS class (app bar)	**Description**
win-appbar	Styles the app bar container; typically this style is used as a root for more specific selectors.
win-commandlayout	Styles the app bar commands layout; apps generally don't modify this style at all.
win-reduced	Automatically added when the app bar is too narrow to accommodate full size commands; this has the effect of removing labels and making the command buttons narrower.
CSS class (commands)	**Description**
win-command	Styles the entire AppBarCommand.
win-commandicon	Styles the icon box for the AppBarCommand.
win-commandimage	Styles the image for the AppBarCommand.
win-commandring	Styles the icon ring for the AppBarCommand.
win-label	Styles the label for the AppBarCommand.

Hint To get an app bar to show up in Blend for Visual Studio, right-click its element in the Live DOM and select the Activate AppBar menu, which will keep it up for as long as you need it. You can also invoke it within Interactive Mode and then switch back to Design Mode and it will remain. In Visual Studio's DOM Explorer and debugger, on the other hand, an app bar will be dismissed when you click/tap outside of it. To be able to access it, make it `sticky` or add a call to `show` in your page's `ready` method or your app's `activated` event.

Generally speaking, you don't need to override the `win-appbar` or `win-commandlayout` styles; instead, create selectors for a custom class related to these and then style the particular pieces you need. This can include pseudo-selectors like `button:hover`, `button:active`, and so forth.

Scenario 2 of the HTML Appbar Control sample shows many such selectors in action, in this case to set the background of the app bar and its commands to blue and the foreground color to green (a somewhat hideous combination, but demonstrative nonetheless).

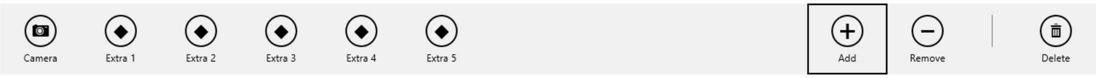
As a basis, scenario 2 (html/custom-color.html) adds a CSS class *customColor* to the app bar:

```
<div id="customColorAppBar" data-win-control="WinJS.UI.AppBar" class="customColor" ...>
```

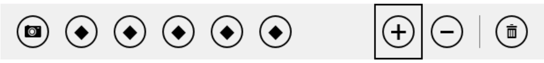
In css/custom-color.css it then styles selectors based on `.win-appbar.customColor`. The following rules, for instance, set the overall background color, the label text color, and the color of the circle around the commands for the `:hover` and `:active` states:

```
.win-appbar.customColor {
    background-color: rgb(20, 20, 90);
}
.win-appbar.customColor .win-label {
    color: rgb(90, 200, 90);
}
.win-appbar.customColor button:hover .win-commandring,
.win-appbar.customColor button:active .win-commandring {
    background-color: rgba(90, 200, 90, 0.13);
    border-color: rgb(90, 200, 90);
}
```

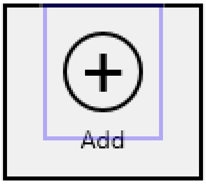
To help accommodate narrow layouts, the app bar automatically checks whether the total width of all the commands together is greater than the width of the app bar control in whatever view it's in. When this happens, it adds the `win-reduced` class to the app bar (and removes it when the app bar is again wider). In the WinJS stylesheets, `win-reduced` hides the labels, shrinks the margins, and otherwise tightens up the layout. For example, in the modified HTML AppBar control sample in the companion content, I've added a number of extra buttons to scenario 1 so that we can see the effect. With full size command buttons, the app bar appears like this:



And when `win-reduced` is in effect, they look like this (same scale):



To compare the two sizes, here's a reduced app bar command (with a color outline) overlaid on the full size command:



Note For the automatic size reduction to work properly, avoid setting the `margin`, `border`, or `padding` styles of `AppBarCommand` elements.

All of this styling, by the way, applies only to the standard command-oriented layout; that is, the various style classes are those that the `AppBar` control adds only when using the command layout. If you're using a custom layout, the app bar just contains whatever elements you want with whatever style classes you want, so you just handle styling as you would any other HTML.

Custom Icons

Earlier we saw that the `icon` property of an `AppBarCommand` typically comes from the Segoe UI Symbol font. Although this is suitable for most needs, you might want at times to use a character from a different font (some of us just can't get away from Wingdings!) or to provide custom graphics. The app bar supports both.

To use a different font for the whole app bar, simply add a class to the app bar and create a rule based on `win-appbar`:

```
win-appbar.customFont {
    font-family: "Wingdings";
}
```

To change the font of a specific command button, add a class to its `extraClass` property (such as *customButtonFont*) and create a rule with the following selector (as used in scenario 1 of the modified sample that will show if you add `extraClass: 'otherFont'` to a button):

```
button.otherFont .win-commandimage {
    font-family: "Wingdings";
}
```

To provide graphics of your own, do the following for a 100% resolution scale:

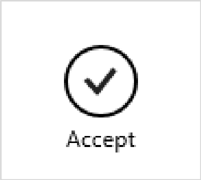
- Create a 160x80 pixel PNG sprite image with a transparent background, saving the file with the *.scale-100* suffix in the filename.

- This sprite is divided into two rows of four columns—that is, 40x40 pixel cells. The top row is for the light styling theme, and the bottom is for the dark theme.
- Each row has four icons for the following button states, in this order from left to right: default (rest), hover, pressed (active), and disabled.
- Each image is centered in its respective 40x40 cell, but remember that a ring will be drawn around the icon, so generally keep the image in the 20–30 pixel range vertically and horizontally. It can be wider or taller in the middle areas, of course, where the ring is widest.

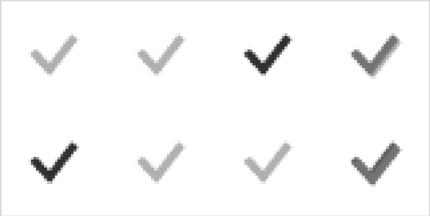
For other resolution scales, multiple the sizes by 1.4 (140%) and 1.8 (180%) and use the *.scale-140* and *.scale-180* suffixes in the image filename.

To use the custom icon, point the command's `icon` property to the base image URI (without the *.scale-1x0* suffixes)—for instance, `icon: 'url(images/icon.png)'`.

Scenario 3 of the HTML Appbar Control sample uses custom icon graphics for an Accept button:



The icon comes from a file called accept.png, which appears something like this—I've adjusted the brightness and contrast and added a border so that you can see each cell clearly:



The declaration for the app bar button then appears as follows (some properties omitted for brevity):

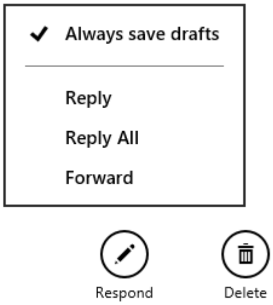
```
<button data-win-control="WinJS.UI.AppBarCommand"
       data-win-options="{id:'cmdAccept', label:'Accept', icon:'url(images/accept.png)' }">
```

Note that although the sample doesn't have variations of the icon for resolution scales, it does provide variants for high contrast themes, an important accessibility consideration that we'll come back to in Chapter 19. For this reason, the `button` element includes `style="-ms-high-contrast-adjust:none"` to override automatic adjustments for high contrast.

Command Menus

The next aspect of an app bar we need to explore in a little more depth are those commands whose `type` property is set to `flyout`. In this case the command's `flyout` property must identify a `WinJS.UI.Flyout` object or a `WinJS.UI.Menu` control (which is a flyout). As noted before, flyout or popup menus like this are used when there are too many related commands cluttering up the basic app bar, or when you need other types of controls that aren't quite appropriate on the app bar itself. It's said, though, that if you're tempted to use a button labeled "More", "Advanced", or "Other Stuff" because you can't figure out how to organize the commands otherwise, it's a good sign that the app itself is too complex! Seek ways to simplify the app's purpose so that the app bar doesn't just become a repository for randomness.

We'll be covering flyouts more fully a little later in this chapter, but let's see how to use one in an app bar, as demonstrated in scenario 6 of the HTML flyout control sample (not the app bar sample, mind you!):



In html/appbar-flyout.html of this sample we see the app bar button declared as follows:

```
<button data-win-control="WinJS.UI.AppBarCommand"
      data-win-options="{id:'respondButton', label:'Respond', icon:'edit', type:'flyout',
         flyout:select('#respondFlyout') }">
```

The *respondFlyout* element identified here is defined earlier in html/appbar-flyout.html; note that such an element must be declared prior to the app bar to make sure it's instantiated *before* the app bar is created:

```
<div id="respondFlyout" data-win-control="WinJS.UI.Menu">
   <button data-win-control="WinJS.UI.MenuCommand"
         data-win-options="{id:'alwaysSaveMenuItem',
            label:'Always save drafts', type:'toggle', selected:'true'}"></button>
   <hr data-win-control="WinJS.UI.MenuCommand"
      data-win-options="{id:'separator', type:'separator'}" />
   <button data-win-control="WinJS.UI.MenuCommand"
         data-win-options="{id:'replyMenuItem', label:'Reply'}"></button>
   <button data-win-control="WinJS.UI.MenuCommand"
         data-win-options="{id:'replyAllMenuItem', label:'Reply All'}"></button>
   <button data-win-control="WinJS.UI.MenuCommand"
```

```
          data-win-options="{id:'forwardMenuItem', label:'Forward'}"></button>
</div>
```

It should come as no surprise by now that the menu is just another WinJS control, `WinJS.UI.Menu`, where its child elements define the menu's contents. As all these elements are, once again, just elements in the DOM; their `click` events are wired up in js/appbar-flyout.js with the ever-present `addEventListener`. (By the way, the sample uses `document.getElementById` to obtain the elements in order to call `addEventListener`; you can use the app bar's `getCommandById` method instead.)

Each menu item, as you can see, is a `MenuCommand` object, and we'll come back to the details later—for the time being, you can see that those items have an `id`, a `label`, and a `type`, very similar to `AppBarCommand` objects.

That's pretty much all there is to it—the one added bit is that when a menu item is selected, you'll want to dismiss the menu and perhaps also the app bar (if it's not sticky). This is shown in the sample within js/appbar-flyout.js in a function called `hideFlyoutAndAppBar`:

```
function hideFlyoutAndAppBar() {
    document.getElementById("respondFlyout").winControl.hide();
    document.getElementById("appBar").winControl.hide();
}
```

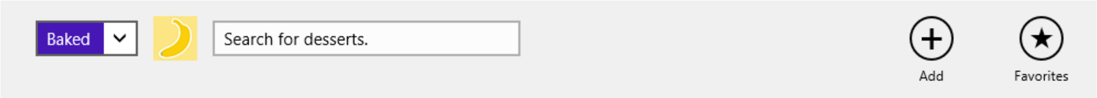
Custom App Bars

All this time we've been looking at the *standard commands layout* of the app bar, which is of course the simplest way to use the control. There will be times, however, when the standard commands layout isn't sufficient. Perhaps you want to place more interesting controls on the app bar, especially custom controls (like a color selector). For this you have two options.

The first is to place different controls alongside standard command buttons with the app bar's `layout` property set to `commands`. Here you must still have `AppBarCommand` controls for each child of the app bar, but those with `type: "content"` can contain any elements you'd like, though the root element must be a `div`. This way they act like standard commands and participate in keyboard navigation but aren't just circles and labels. Scenario 4 of the HTML AppBar control sample provides an example (html/custom-content.html):

```
<div id="customContentAppBar" data-win-control="WinJS.UI.AppBar">
    <div data-win-control="WinJS.UI.AppBarCommand" data-win-options="{ id: 'list',
        type: 'content', section: 'selection',
        firstElementFocus: select('.dessertType'), lastElementFocus:select('.dessertType') }">
        <select class="dessertType">
            <option>Baked</option>
            <option>Fried</option>
            <option>Frozen</option>
            <option>Chilled</option>
        </select>
    </div>
    <div tabindex="-1" data-win-control="WinJS.UI.AppBarCommand"
         data-win-options="{ id: 'banana', type: 'content', section: 'selection' }">
        <img src="../images/40Banana.png" />
    </div>
    <div data-win-control="WinJS.UI.AppBarCommand"
         data-win-options="{ id: 'search', type: 'content', section: 'selection' }">
        <input type="text" value="Search for desserts." />
    </div>
    <button data-win-control="WinJS.UI.AppBarCommand" data-win-options="{ id: 'cmdAdd',
        label: 'Add', icon: 'add', tooltip: 'Add a recipe' }"></button>
    <button data-win-control="WinJS.UI.AppBarCommand" data-win-options="{ id: 'cmdFavorites',
        label: 'Favorites', icon: 'favorite', tooltip:'Favorites' }"></button>
</div>
```

The result (combined with a slight bit in css/custom-content.css) is as follows:

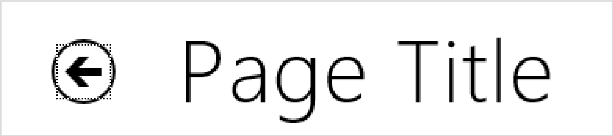


The other option is to set the app bar's `layout` to `custom` and place whatever HTML you want within the app bar control, styling it with CSS, and wiring up whatever events you need in JavaScript. As mentioned before, the WinJS `NavBar` control is just an app bar with a custom layout and top placement; if you implement a navigation bar of your own, you'd do the same.

Scenario 6 of the [HTML AppBar control sample]() provides an example, though in this case the result is not something I'd recommend for your own app design: it creates a page header and a backbutton (html/custom-layout.html):

```
<div id="customLayoutAppBar" data-win-control="WinJS.UI.AppBar" aria-label="Navigation Bar"
     data-win-options="{layout:'custom', placement:'top'}">
    <header aria-label="Navigation bar" role="banner">
        <button id="cmdBack" class="win-backbutton" aria-label="Back"></button>
        <div class="titleArea">
            <h1 class="win-type-xx-large" tabindex="0">
                Page Title
            </h1>
        </div>
    </header>
</div>
```

The result of this example is as follows (focus rectangles included!):



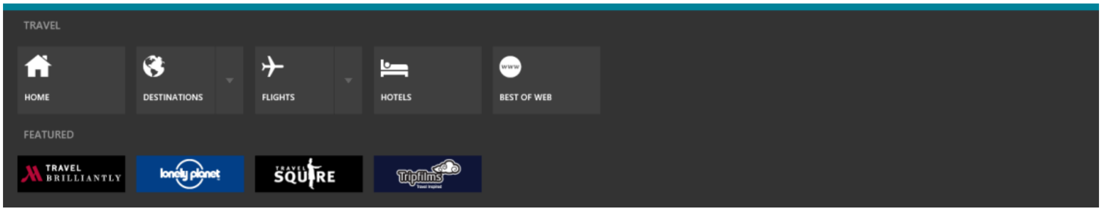
The point, though, is that a custom layout app bar is essentially nothing more than a container for whatever arbitrary content you want to place there, and that content is entirely under your control. The app bar in these cases is just taking care of showing and hiding that content at appropriate times and firing relevant events.

A custom layout app bar is typically what you'd use to implement a completely custom nav bar, using a `placement` of `top`. Before going down that road, however, let's check out the WinJS NavBar control that provides a lot of functionality in this department already.

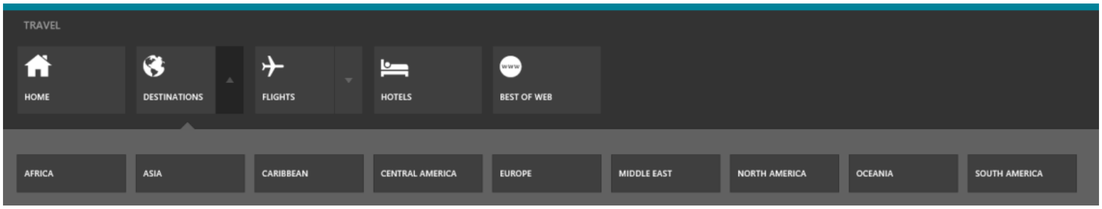
Nav Bar Features

Navigation with page controls, as we know from Chapter 3, is just a matter of calling `WinJS.Navigation.navigate` at the appropriate times with the appropriate target page. Assuming there's some piece of code like the `PageControlNavigator` to pick up the navigation events and take care of the page loading, an app can wire whatever controls it sees fit to `navigate` calls. This includes a top placement app bar that can have whatever design you would like to use, typically with a custom layout. With the [flat navigation pattern](), this app bar can contain just a horizontally-panning ListView (using `ListLayout`) with the relevant pages. With a [hierarchical system](), on the other hand, the implementation gets trickier as you want to have one list that opens up a secondary list and potentially has even a third level of options.

Fortunately, the WinJS `NavBar` control steps in to support this pattern, which you can see demonstrated in the inbox Travel app. Let's see some of the variations to understand what the control provides for us. For starters, here's the nav bar at full width:

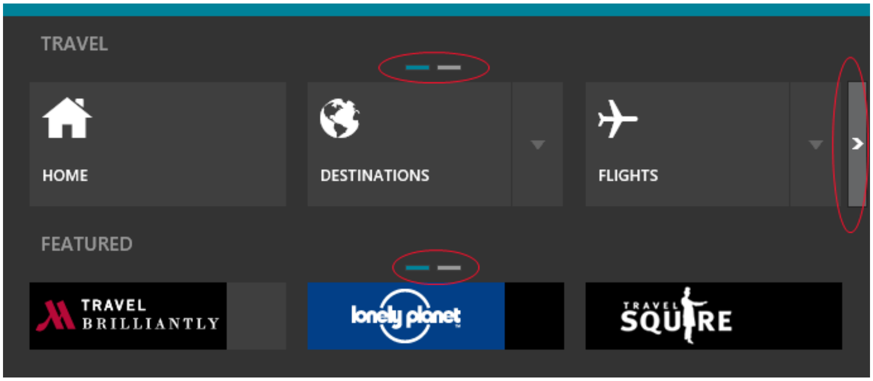


Tapping the down arrow next to Destinations or Flights opens up the second level in the navigation hierarchy, where again we see all the options when the screen is wide enough:

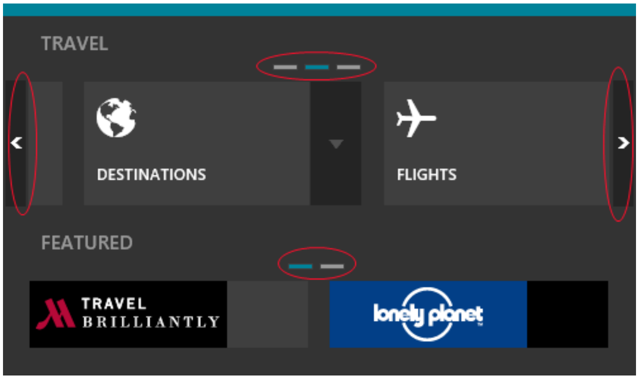


Notice how the down arrow by Destinations changed to an up arrow to indicate that this second level can be collapsed back to its previous state.

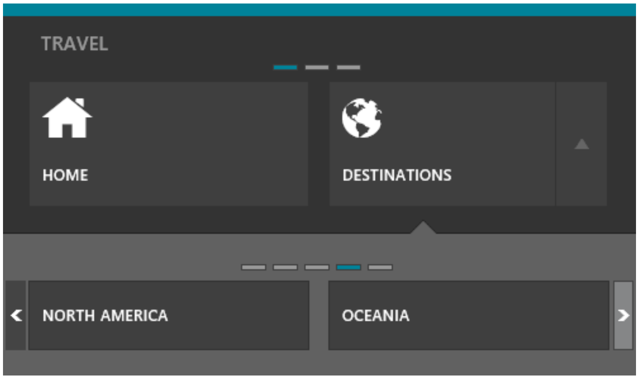
So far so good. Now if we resize the app to a narrower view, we'll clearly need a way to pan the commands left and right. In this case the NavBar provides panning arrows a'la FlipView (circled on the right side), as well as page indicators to show where you are in the list (circled in the middle):



Narrowing the view still further (down to 500px), we see that more indicators appear and that the flipping arrows appear on both sides of the list:



And we get the same effect on the second level navigation commands when we tap the down arrow by Destinations:



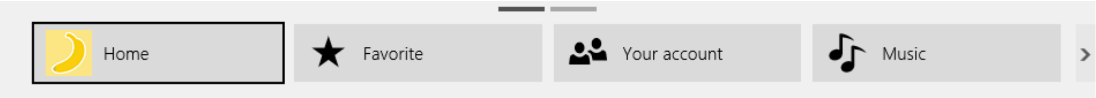
The `WinJS.UI.NavBar` control makes it straightforward to implement these kinds of patterns, as well as vertical layout, by leveraging much of what we already know of the app bar. The NavBar, in fact, derives directly from the AppBar and thus shares many of the same properties, methods, and events, such as sticky, show/hide, showOnlyCommands, aftershow, etc. As noted before, the NavBar's default placement is top, but it also supports bottom (check out Internet Explorer for a bottom navigation bar design). The layout property, however, is always custom (thus, the commands property is ignored), and the NavBar adds one event, childrenprocessed, to inform you when the NavBar has constructed itself fully. This event exists because processing NavBar children is done at "idle" priority, as explained in Chapter 3: the NavBar isn't typically visible when the page containing it first appears, so it's appropriate to do that processing only after other UI work is complete. Of course, if you invoke the NavBar, it will reprioritize this processing so that it completes more quickly.

Because the NavBar is a custom layout AppBar, you can place any controls on it that you like. More often, however, you'll want collection-like behavior for multiple navigation targets, perhaps with multiple levels. For this there's a special control, the WinJS.UI.NavBarContainer, whose children are instances of the NavBarCommand class, the latter of which has properties like label and icon to control its display just like the AppBarCommand.

Let's see two brief examples from the HTML NavBar control sample. First, here's the simple markup from scenario 1 (html/1-CreateNavBar.html):

```
<div id="createNavBar" data-win-control="WinJS.UI.NavBar">
    <div data-win-control="WinJS.UI.NavBarContainer">
        <div data-win-control="WinJS.UI.NavBarCommand"
            data-win-options="{ label: 'Home', icon: 'url(../images/homeIcon.png)' }"></div>
        <div data-win-control="WinJS.UI.NavBarCommand"
            data-win-options="{ label: 'Favorite', icon: 'favorite' }"></div>
        <div data-win-control="WinJS.UI.NavBarCommand"
            data-win-options="{ label: 'Your account', icon: 'people' }"></div>
        <div data-win-control="WinJS.UI.NavBarCommand"
            data-win-options="{ label: 'Music', icon: 'audio' }"></div>
        <div data-win-control="WinJS.UI.NavBarCommand"
            data-win-options="{ label: 'Video', icon: 'video' }"></div>
        <div data-win-control="WinJS.UI.NavBarCommand"
            data-win-options="{ label: 'Photos', icon: 'camera' }"></div>
        <div data-win-control="WinJS.UI.NavBarCommand"
            data-win-options="{ label: 'Settings', icon: 'settings' }"></div>
    </div>
</div>
```

The result of this, in a view that's narrow enough to see the page indicators, is as follows:



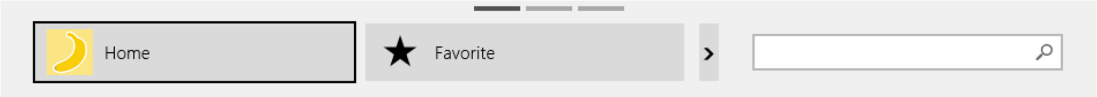
You can clearly see that the default shape of a NavBarCommand is a rectangle with the icon on the left and a label on the right (reversed for right-to-left languages, as is the paging direction). You can choose icons from the AppBarIcon enumeration or provide one of your own, and the labels act just like those on the AppBar where localization is concerned.

The second example is found in scenario 5, which uses the same markup as above plus a

WinJS.UI.SearchBox outside the NavBarContainer (see html/5-UseSearchControl.html):

```
<div id="useSearch" data-win-control="WinJS.UI.NavBar">
    <div class="globalNav" data-win-control="WinJS.UI.NavBarContainer">
        <!-- ... -->
    </div>
    <div class="SearchBox" data-win-control="WinJS.UI.SearchBox"></div>
</div>
```

With this the SearchBox appears to the right of the command container, and the container has adjusted itself to a smaller width by making more pages of commands:



In most of the sample's scenarios, note that the buttons don't actually navigate because none of the commands have a location property—this is the URI string that you want the command to pass to WinJS.Navigation.navigate, such as:

```
<div data-win-control="WinJS.UI.NavBarCommand"
     data-win-options="{ label: 'Home', icon: 'url(../images/homeIcon.png)',
                         location: '/pages/home/home.html' }">
</div>
```

Scenario 4 is the only part of the sample that actually navigates, and it does so between the different scenarios simply through the location property.

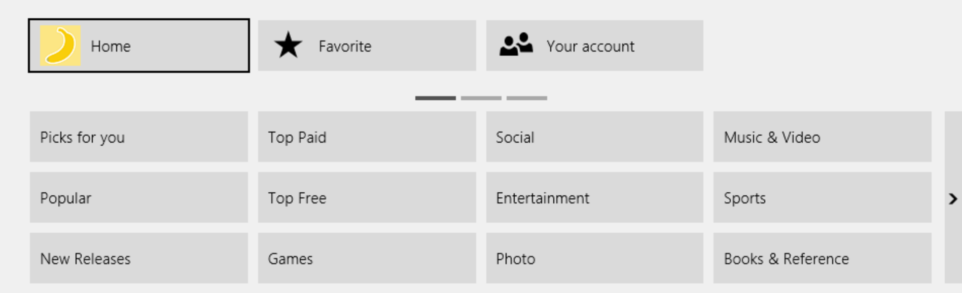
Besides label, icon, and location, the NavBarCommand has these additional properties:

- tooltip The typically localized tooltip text for the command, using the value of label as the default. As with the app bar, only text is supported here (not HTML or a Tooltip control) because it just passes on to the element's title attribute.
- state An app-provided object that is passed with the location to Navigation.navigate (in the *initialState* argument). If you want to dynamically customize this state—such as storing selection information from the current page to pass to the target page—update the object within a handler for the either the WinJS.Navigation.onbeforenavigate event or the NavBarContainer.oninvoked event (see below).
- splitButton If true, adds a down arrow or "split button" to the command.
- splitOpened Indicates (true or false) whether the split button is open on this command. By setting this flag you'll trigger the container's splitToggle event.

We'll come back to the split button operation in a bit, because it's necessary to learn a little more about the NavBarContainer beforehand. This control is what we use to organize commands—NavBarCommand controls, that is—into meaningful groups, and it provides for page indicators, paging arrows, and opening another NavBarContainer for to a split button.

Tip There's nothing in the NavBarContainer that says it has to exist inside a NavBar; you can use the control by itself wherever you like. Just be aware that it will still call WinJS.Navigation.navigate!

A NavBar can contain multiple NavBarContainer controls as its immediate children, in which case they are stacked vertically. In the earlier examples from the Travel app, the Travel group of commands is in one container and the Featured group is in another. This is why they have separate page indicators and page navigation arrows. Scenario 2 of the sample shows a similar result:



In this case, as with scenarios 1 and 5, the upper NavBarContainer is declared without any options and all its commands are declared inline. Of course, because the NavBarContainer is itself a kind of collection control, it would make perfect sense to hand it a WinJS.Binding.List to describe its contents, which would be especially useful if you have a dynamic navigation hierarchy or just a large list of commands, as in the lower container above. This is the purpose of the data property.

To use the data property, build a List of options objects for the NavBarCommand controls you want: each object in the List is just passed to the NavBarCommand constructor as the options argument. Then you can just declare the NavBar like so (html/2-UseData.html):

```
<div class="categoryNav" data-win-control="WinJS.UI.NavBarContainer"
    data-win-options="{ data: Data.categoryList, maxRows: 3 }"></div>
```

where you also see the maxRows property that limits the vertical height of the container. Data.categoryList is defined in the page's init method (see js/2-UseData.js):

```
WinJS.Namespace.define("Data");

var categoryNames = ["Picks for you", "Popular", "New Releases", "Top Paid", "Top Free",
    "Games", "Social", "Entertainment", "Photo", "Music & Video",
    "Sports", "Books & Reference", "News & Weather", "Health & Fitness", "Food & Dining",
    "Lifestyle", "Shopping", "Travel", "Finance", "Productivity",
    "Tools", "Security", "Business", "Education", "Government"];

var categoryItems = [];
for (var i = 0; i < categoryNames.length; i++) {
    categoryItems[i] = {
        label: categoryNames[i]
    };
}
```

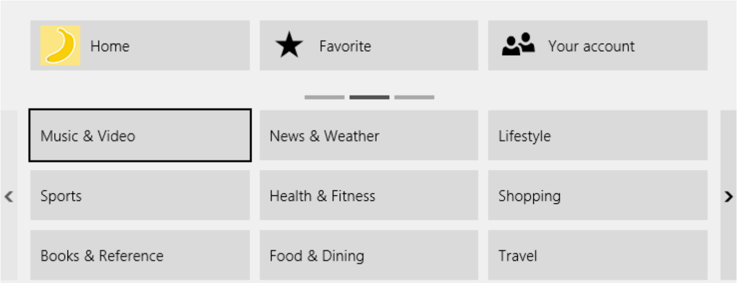
```
Data.categoryList = new WinJS.Binding.List(categoryItems);
```

Tip When declaratively referring to a Binding.List that you generate within a page control, be sure to create that list within the page's init method rather than ready. This is because init is called prior to WinJS.UI.processAll, which instantiates the NavBarContainer, whereas ready is called after. By creating the List within init, you make sure it exists before the container's data-win-options is processed.

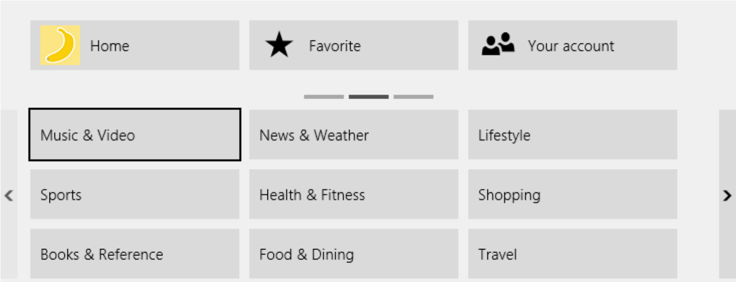
Take note that the NavBarContainer works directly with a Binding.List (not its dataSource) and is thus limited to in-memory collections.

The NavBarContainer also supports these additional members:

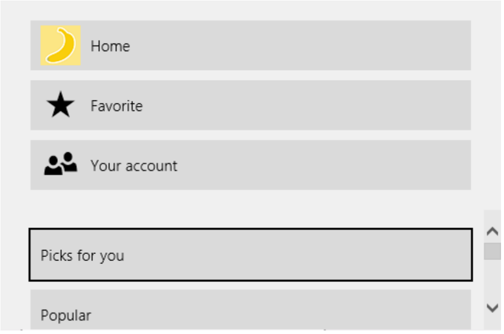
- template (Property) Gets or sets a WinJS.BindingTemplate or rendering function that creates the DOM for each item in the data collection. The template must render a single root element but can otherwise contain whatever controls you'd like.
- layout (Property) A value from the [WinJS.UI.Orientation]() enumeration, either horizontal (the default) or vertical. The horizontal layout works with paging, as we've seen, with the maxRows property (default is 1) controlling the layout in each page. The vertical layout pans continuously without page indicators or arrows and ignores maxRows.
- currentIndex (Property) Gets or sets the index of the item with the keyboard focus.
- fixedSize (Property) When true, the width of each command in the container is determined by its styling, which means there could be gaps on the sides. When false (the default), the container will size the commands so that they fill the horizontal width of the container.
- forceLayout (Method) As with other collection controls, call this when making the control visible again after changing its display style to something other than none.
- invoked (Event) Fired when a command in the container has been invoked in response to a click, tap, or the Space or Enter keys being pressed. The eventArgs.detail object contains the index and navbarcommand object of the command that was invoked, along with the data item that was used to create the command. Note that navigation to the command's location will have already started when this event is fired.
- splitToggle (Event) Fired when a command with splitButton: true changes its splitOpen state. The eventArgs.detail object contains the same properties as invoked above plus the new state in the opened property. You use this to show or hide a secondary NavBarContainer.
- The layout and fixedSize properties are demonstrated in scenario 3 of the [HTML NavBar control sample](). By default it uses dynamic width, so both containers fill the width:



Click the Switch To Fixed Width button, and you'll see that at certain view widths you get a gap on the right side (or left in right-to-left languages):



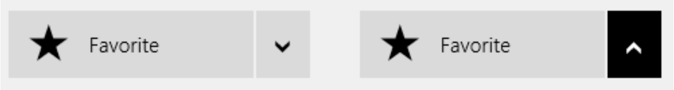
If you size the view all the way down to 500px (and it must be 500px, because the manifest isn't set for anything smaller!), the sample changes the `layout` to `vertical` with the following result:



As for implementing additional levels of navigation hierarchy, this is where we make use of the `splitButton` option for a command along with the `splittoggle` event on the container. As shown in scenario 6, a command with `splitButton: true` (html/6-UseSplitButton.html):

```
<div data-win-control="WinJS.UI.NavBarCommand" data-win-options="{ label: 'Favorite',
   icon: 'favorite', splitButton: 'true' }"></div>
```

appears with a down arrow (if in the closed state, left) or an up arrow (if in the open state, right):



By itself, a split button just makes this one visual change when you invoke it, updates its splitOpen property, and—as a result—causes its container to fire a splitToggle event. This latter event is then where you then show or hide whatever additional controls you want to attach to it. Note that I didn't specifically say a NavBarContainer—though you'll probably want to use a NavBarContainer to implement the expected UI pattern, this is not required by any means.

The key here is to make those other controls appear as an overlay within the NavBar, which is exactly what the generic WinJS.UI.Flyout control is meant for (and which we'll learn about right after we talk NavBar styling). Simply said, a flyout is a separate piece of transient UI that won't appear until you call its show method. In scenario 6, the flyout for the Favorite button above is, in fact, declared separately from the NavBar itself (html/6-UseSplitButton.html):

```
<div id="useSplit" data-win-control="WinJS.UI.NavBar">
    <div class="globalNav" data-win-control="WinJS.UI.NavBarContainer">
        <div data-win-control="WinJS.UI.NavBarCommand" data-win-options="{ label: 'Home',
            icon: 'url(../images/homeIcon.png)' }"></div>
        <div data-win-control="WinJS.UI.NavBarCommand" data-win-options="{ label: 'Favorite',
            icon: 'favorite', splitButton: 'true' }"></div>
        <div data-win-control="WinJS.UI.NavBarCommand"
             data-win-options="{ label: 'Your account', icon: 'people' }"></div>
    </div>
</div>

<div id="contactFlyout" data-win-control="WinJS.UI.Flyout"
     data-win-options="{ placement: 'bottom' }">
    <div id="contactNavBarContainer" data-win-control="WinJS.UI.NavBarContainer" }">
        <div data-win-control="WinJS.UI.NavBarCommand"
             data-win-options="{ label: 'Family' }"></div>
        <div data-win-control="WinJS.UI.NavBarCommand"
             data-win-options="{ label: 'Work' }"></div>
        <div data-win-control="WinJS.UI.NavBarCommand"
             data-win-options="{ label: 'Friends' }"></div>
        <div data-win-control="WinJS.UI.NavBarCommand"
             data-win-options="{ label: 'Blocked' }"></div>
    </div>
</div>
```

The sample then implements a handler for the container's splitToggle event to show or hide this flyout as needed (js/6-UseSplitButton.js):

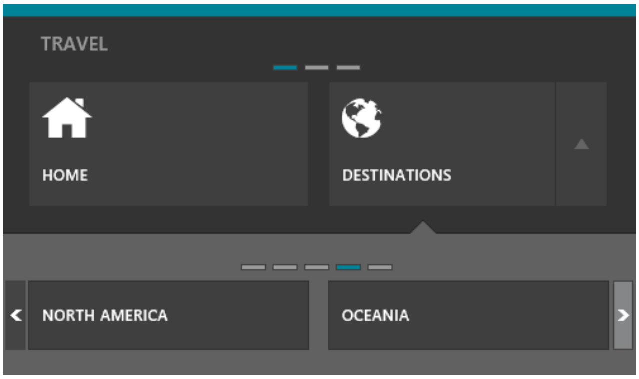
```
setupNavBarContainer: function () {
    var navBarContainerEl = document.body.querySelector('#useSplit .globalNav');

    navBarContainerEl.addEventListener("splittoggle", function (e) {
        var flyout = document.getElementById("contactFlyout").winControl;
        var navbarCommand = e.detail.navbarCommand;

        if (e.detail.opened) {
            flyout.show(navbarCommand.element);
            var subNavBarContainer = flyout.element.querySelector('.win-navbarcontainer');
            if (subNavBarContainer) {
                // Switching the navbarcontainer from display none to display block requires
                // forceLayout in case there was a pending measure.
                subNavBarContainer.winControl.forceLayout();
                // Reset back to the first item:
                subNavBarContainer.currentIndex = 0;
            }
            flyout.addEventListener('beforehide', go);
        } else {
            flyout.removeEventListener('beforehide', go);
            flyout.hide();
        }

        function go() {
            flyout.removeEventListener('beforehide', go);
            navbarCommand.splitOpened = false;
        }
    });
```

Notice the use of the flyouts beforehide event here to make sure the command's splitOpened flag is set to false when the flyout is dismissed. This is necessary because the flyout can be dismissed independently of the split button.

Because showing and hiding the flyout is under your complete control, the splitToggle event is also where you can perform animations for your subsidiary navigation controls.

Nav Bar Styling

In the previous section, as we looked at NavBar examples from both the Travel app and the HTML NavBar control sample, you'll have easily noticed some styling differences other than the basic light or dark theme (which you can switch in the sample, by the way). The Travel app, for example, definitely uses its own template for the top-level commands and adds some color theming along the top and in the page indicators:



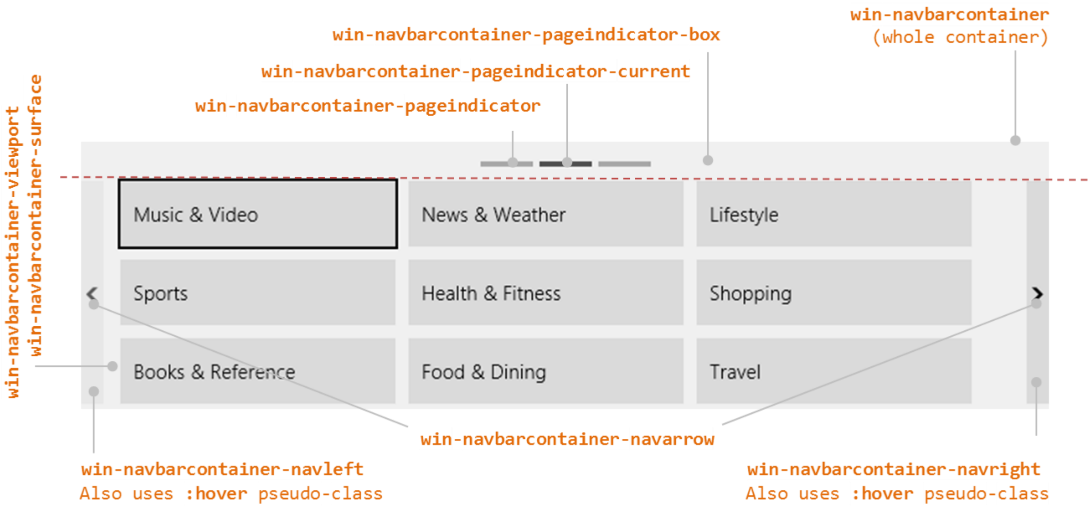
These appearances are easy to control, as the NavBar, NavBarContainer, and NavBarCommand classes all provide the usual win-* style hooks for their different parts.

> **Hint** As with the app bar, to keep it visible in Blend for Visual Studio, right-click its element and select Activate NavBar, or activate it in Interactive Mode and switch back to Design Mode. You can also make the nav bar sticky or add a call to show in an appropriate place, if you need to keep it visible within Visual Studio's debugger and DOM Explorer.

The NavBar as a whole has just one relevant class, win-navbar, that's added to the control's root element. This is where you can add something like the Travel app's top color border—try this in css/scenario1.cs:

```
#createNavBar.win-navbar {
    border-top: 10px solid #008299;
}
```

Within the NavBar, any of your own child elements (like the TRAVEL and FEATURED labels in the Travel app) have whatever classes you assign to them. If you have NavBarContainer elements, those are composed of a considerable hierarchy of parts:


win-navbarcontainer
(whole container)
win-navbarcontainer-pageindicator-box
win-navbarcontainer-pageindicator-current
win-navbarcontainer-pageindicator
win-navbarcontainer-viewport
win-navbarcontainer-surface
Music & Video
News & Weather
Lifestyle
Sports
Health & Fitness
Shopping
Books & Reference
Food & Dining
Travel
win-navbarcontainer-navarrow
win-navbarcontainer-navleft
Also uses :hover pseudo-class
win-navbarcontainer-navright
Also uses :hover pseudo-class

To color the page indicators like the Travel app, for example, uses these rules:

```
#createNavBar .win-navbarcontainer-pageindicator {
    background-color: #636363;
}

#createNavBar .win-navbarcontainer-pageindicator-current {
    background-color: #008299;
}
```

The viewport and pageindicator-box (if I may drop the win-navbarcontainer- prefix) are sibling elements, each of which has its own portion within the container where you can style background colors, margins/padding, etc.

Within the viewport we then have the pannable surface area alongside the left and right navigation arrows. The navarrow style specifically styles the arrow inside the buttons; the navleft and navright selectors (including pseudo-styles) affect the surrounding controls.

Within the win-navbarcontainer-surface element is where you'll find the NavBarCommand elements, each of which has this hierarchy of elements and classes, through which you can specifically address whichever part you want:

```
<div class="win-navbarcommand">
    <div class="win-navbarcommand-button">
        <div class="win-navbarcommand-button-content">
            <div class="win-navbarcommand-icon"></div>
            <div class="win-navbarcommand-label"></div>
        </div>
        <div class="win-navbarcommand-splitbutton"></div>
    </div>
</div>
```

Alternately, you might find it easier to provide a template for the command items so that you can control the elements that are built up for each one.

As for the little triangle that you see on the secondary flyout in the Travel app (to point up to the Destinations button), that's just a little element in the Flyout that uses a background-image; it's not part of the NavBarContainer with the list of items.

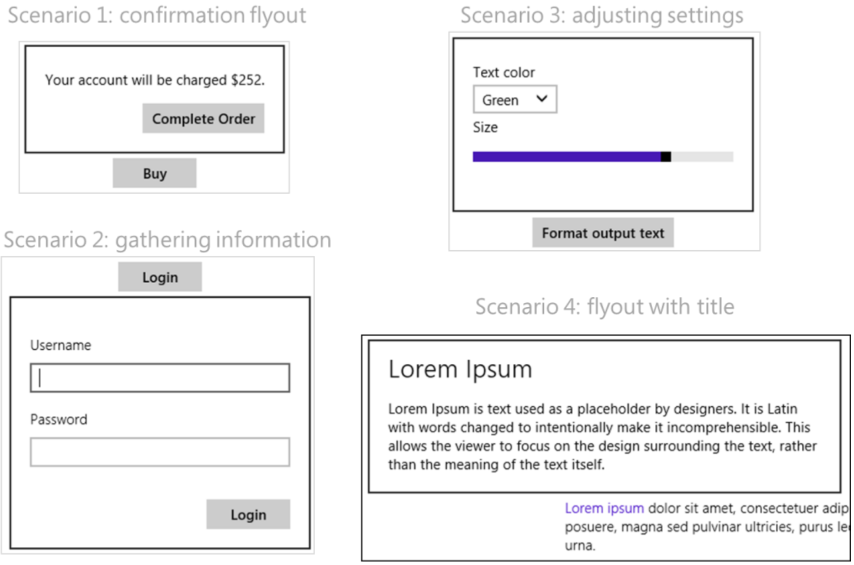
Flyouts and Menus

Going back to our earlier discussion about where to place commands, a flyout control—WinJS.UI.Flyout—is used for confirmations, collecting information, and otherwise answering questions in response to a user action. The menu control—WinJS.UI.Menu—is then a particular kind of flyout that contains WinJS.UI.MenuCommand controls rather than arbitrary HTML. In fact, Menu is directly derived from Flyout using WinJS.Class.define, so they share much in common. As flyouts, they also share some feature in common with the app bar and nav bar, as all of them, in fact, derive from a common internal base class (WinJS.UI._Overlay).

Tip In addition to the Flyout control that you'll employ in an app, there is also a system flyout that appears in response to some API calls, such as creating or removing a secondary tile (see Chapter 16—specifically Figure 16-3 and the "Secondary Tiles" section). Although visually the same, the system flyout will trigger a blur event to the app whereas the WinJS flyout, being part of the app, does not. As a result, a system flyout will cause a non-sticky app bar to be dismissed. To prevent this, it's necessary to set the appbar's sticky property to true before calling APIs with system flyouts. This is demonstrated in scenario 7 of the Secondary tiles sample.

Styling in Blend As with app bars and nav bars, you can right-click a flyout element in Blend's Live DOM and select Activate Flyout to make it visible for styling, or you can activate it in Interactive Mode and switch back to Design Mode. To work with it in Visual Studio's debugger or the DOM Explorer, you'll need to make it sticky, otherwise switching to Visual Studio will dismiss it.

Before we look at the details, let's see a number of visual examples from the [HTML flyout control sample]() in which we already saw a popup menu on an app bar command. The `Flyout` controls used in scenarios 1–4 are shown in Figure 9-1. Notice the variance of content in the flyout itself and how the flyout is always positioned near the control that invoked it, such as the Buy, Login, and Format output text buttons and the *Lorem ipsum* hyperlink text. These examples illustrate that a flyout can contain a simple message with a button (scenario 1, for warnings and confirmations), can contain fields for entering information or changing settings (scenarios 2 and 3), and can have a title (scenario 4). Scenario 5, for its part, contains the example of a popup header menu with `Menu` that we'll see a little later.



Scenario 1: confirmation flyout

Your account will be charged $252.

Scenario 2: gathering information

Scenario 3: adjusting settings

Text color: Green

Scenario 4: flyout with title

Lorem Ipsum

Lorem Ipsum is text used as a placeholder by designers. It is Latin with words changed to intentionally make it incomprehensible. This allows the viewer to focus on the design surrounding the text, rather than the meaning of the text itself.

Lorem ipsum dolor sit amet, consectetuer adip
posuere, magna sed pulvinar ultricies, purus le
urna.

Figure 9-1 Examples of flyout controls from the HTML flyout control sample.

There are two key characteristics of flyout controls, including menus. One is that flyouts can be dismissed programmatically, like an app bar, when an appropriate control within the flyout is invoked. This is the case with the Complete Order button of scenario 1 and the Login button of scenario 2.

The second characteristic, also shared with the app bar, is the light dismiss behavior: clicking or tapping outside the control dismisses it, as does the ESC key, which means light dismiss is the equivalent of pressing a Cancel or Close button in a traditional dialog box. The benefit here is that we don't need a visible button for this purpose, which helps simplify the UI. At the same time, notice in scenario 3 of Figure 9-1 that there is no OK button or other control to confirm changes you might make in the flyout. With this particular design, changes are immediately applied such that dismissing the flyout does not reverse or cancel them. If you don't want that kind of behavior, you can place something like an Apply button on the flyout and not make changes until that button is pressed. In this case, dismissing the flyout would cancel the changes.

I'll again encourage you to read the Guidelines for Flyouts topic that goes into detail about how and when to use the different designs that are possible with this control. It also outlines when *not* to use the control: for example, don't use flyouts to surface errors not related to user action (use a message dialog instead), don't use them for primary commands (use the app bar), don't use them for text selection context menus, and avoid them for UI that is part of a workflow and should be directly on the app canvas. These guidelines also suggest keeping a flyout small and focused (omitting unnecessary controls) and making sure a flyout is positioned close to the object that invoked it. Let's now see how that works in the code.

WinJS.UI.Flyout Properties, Methods, and Events

Most of the properties, methods, and events of the `WinJS.UI.Flyout` control are exactly the same as we've already seen for the app bar. The `show` and `hide` methods control its visibility, a `hidden` property indicates its visible state, and same the `beforeshow`, `aftershow`, `beforehide`, and `afterhide` events fire as appropriate. The `afterhide` event is typically used to detect dismissal of the flyout.

Like the app bar, the flyout also has a `placement` property, but it has different values that are only meaningful in the context of the flyout's own `alignment` and `anchor` properties. In fact, all three properties are optional parameters to the `show` method because they determine where, exactly, the flyout appears on the screen; the default `placement` and `alignment` can also be set on the control itself because these are optional with `show`. (Note also that if you don't specify an anchor in the `show` method; the `anchor` property must already be set on the control or `show` will throw an exception.)

The `anchor` property identifies the control that invokes the flyout or whatever other operation might bring up a flyout (as for confirmation). The `placement` property (a string) then indicates how the flyout should appear in relation to the `anchor`: `top`, `bottom`, `left`, `right`, or `auto` (the default). Typically, you use a specific `placement` only if you don't want the flyout to possibly obscure important content. Otherwise, you run the risk of the flyout element being shrunk down to fit the available space. The flyout's *content* will remain the same size, mind you, so it means that—ick!—you'll get scrollbars! So, unless you have a really good reason and a note from your doctor, stick with `auto` placement so that the control will be placed where it can be shown full size. Along these same lines, if you're supporting the 320px minimum width, limit your flyouts also to that size.

The `alignment` property, for its part (also a string), when used with a `placement` of `top` or `bottom`, determines how the flyout aligns to the edge of the `anchor`: `left`, `right`, or `center` (the default). The content of the flyout itself is aligned through CSS as with any other HTML.

If you need to style the flyout control itself, you can set styles in the `win-flyout` class, like fonts, default alignments, margins, and so on. As with other WinJS style classes like this, use `win-flyout` as a basis for more specific selectors unless you really want to style every flyout in the app. Typically you also exclude win-menu from the rule so that menu flyouts aren't affected by such styling. For example, most of the scenarios in the HTML flyout control sample have rules like this:

```
.win-flyout:not(.win-menu) button,
.win-flyout:not(.win-menu) input[type="button"] {
    margin-top: 16px;
    margin-left: 20px;
    float: right;
}
```

Finally, if for some reason you need to know when a flyout is loaded, listen to the DOMNodeInserted method on document.body:

```
document.body.addEventListener("DOMNodeInserted", insertionHandler, false);
```

Flyout Examples

A flyout control is created like any other WinJS control with data-win-control and data-win-options attributes and processed by WinJS.UI.process/processAll. Flyouts with relatively fixed content will typically be declared in markup where you can use data binding on specific properties of the elements within the flyout. Flyouts that are very dynamic, on the other hand, can be created directly from code by using new WinJS.UI.Flyout(<element>, <options>), and you can certainly change its child elements at any time. It's all just part of the DOM! (Am I repeating myself?)

Like I said before (apparently I am repeating myself), a Flyout control can contain arbitrary HTML, styled as always with CSS. The flyout for scenario 1 in the sample appears as follows in html/confirm-action.html (condensed slightly):

```
<div id="confirmFlyout" data-win-control="WinJS.UI.Flyout"
    aria-label="{Confirm purchase flyout}">
    <div>Your account will be charged $252.</div>
    <button id="confirmButton">Complete Order</button>
</div>
```

The login flyout in scenario 2 is similar, and it even employs an HTML form to attach the Login button to the Enter key (html/collect-information.html):

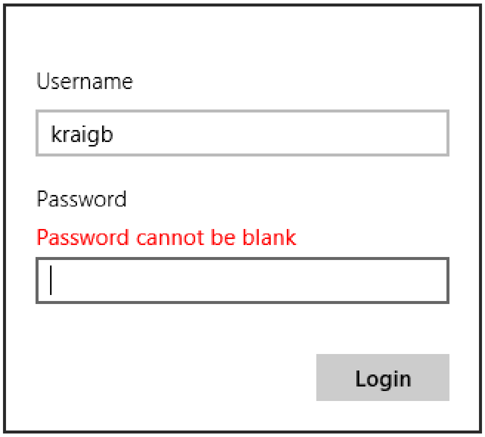
```
<div id="loginFlyout" data-win-control="WinJS.UI.Flyout" aria-label="{Login flyout}">
    <form onsubmit="return false;">
        <p>
            <label for="username">Username <br /></label>
            <span id="usernameError" class="error"></span>
            <input type="text" id="username" />
        </p>
        <p>
            <label for="password">Password<br /></label>
            <span id="passwordError" class="error"></span>
            <input type="password" id="password" />
        </p>
        <button id="submitLoginButton">Login</button>
    </form>
</div>
```

The flyout is displayed by calling its show method. In scenario 1, for instance, the button's click event is wired to the showConfirmFlyout function (js/confirm-action.js), where the Buy button is given as the anchor element. Handling the Complete Order button just happens through a click handler attached to that element, and here we want to make sure to call hide to programmatically dismiss the flyout. Finally, the afterhide event is used to detect dismissal:

```
var bought;

var page = WinJS.UI.Pages.define("/html/confirm-action.html", {
    ready: function (element, options) {
        document.getElementById("buyButton").addEventListener("click",
            showConfirmFlyout, false);
        document.getElementById("confirmButton").addEventListener("click",
            confirmOrder, false);
        document.getElementById("confirmFlyout").addEventListener("afterhide",
            onDismiss, false);
    }

function showConfirmFlyout() {
    bought = false;
    var buyButton = document.getElementById("buyButton");
    document.getElementById("confirmFlyout").winControl.show(buyButton);
}

// When the Buy button is pressed, hide the flyout since the user is done with it.
function confirmOrder() {
    bought = true;
    document.getElementById("confirmFlyout").winControl.hide();
}

// On dismiss of the flyout, determine if it closed because the user pressed the buy button.
// If not, then the flyout was light dismissed.
function onDismiss() {
    if (!bought) {
        // (Sample displays a dismissal message on the canvas)
    }
}
```

Tip To create a default button in a flyout, use an <input type="submit"> element. Just be sure that it doesn't steal Enter key behavior from other buttons when the flyout isn't showing.

Handling the login controls in scenario 2 is pretty much the same, with some added code to make sure that both a username and password have been given. If not, the Login button handler displays an inline error and sets the focus to the appropriate input field:


Username

kraigb

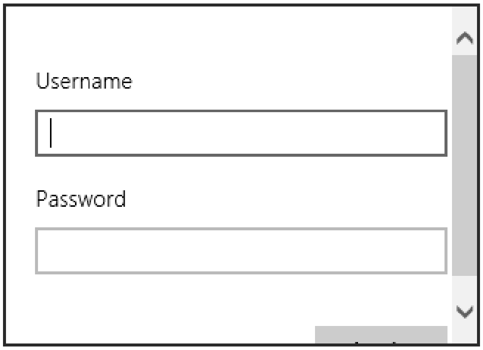
Password

Password cannot be blank

As the flyout of scenario 2 is a little larger, the default `placement` of `auto` on a 1366x768 display (as in the simulator) makes it appear below the button that invokes it. There isn't quite enough room above that button. So try setting `placement` to `top` in the call to `show`:

```
function showLoginFlyout() {
    // ...
    document.getElementById("loginFlyout").winControl.show(loginButton, "top");
}
```

Then you can see how the flyout gets scrollbars because the overall control element is too short:


Username

Password

What was that word I used before? "Ick"?

To move on, scenario 3 again declares a flyout in markup, where it contains some `label`, `select`, and `input` controls. In JavaScript, though, it listens for change events on the latter and applies those new values to the output element on the app canvas:

```
var page = WinJS.UI.Pages.define("/html/change-settings.html", {
    ready: function (element, options) {
        // ...
        document.getElementById("textColor").addEventListener("change", changeColor, false);
        document.getElementById("textSize").addEventListener("change", changeSize, false);
        }
    });

// Change the text color
function changeColor() {
    document.getElementById("outputText").style.color =
        document.getElementById("textColor").value;
}

// Change the text size
function changeSize() {
    document.getElementById("outputText").style.fontSize =
        document.getElementById("textSize").value + "pt";
}
```

If this flyout had an Apply button rather than applying the changes immediately, its `click` handler would obtain the current selection and slider values and use them like `changeColor` and `changeSize`.

Finally, in scenario 4 we see a flyout with a title, which is just a piece of larger text in the markup; the flyout control itself doesn't have a separate notion of a header:

```
<div id="moreInfoFlyout" data-win-control="WinJS.UI.Flyout" aria-label="{More info flyout}">
    <div class="win-type-x-large">Lorem Ipsum</div>
    <div>
        Lorem Ipsum is text used as a placeholder by designers...
    </div>
</div>
```

The point of this last example is to show that unlike traditional desktop dialog boxes, flyouts don't often need a title because they already have context within the app itself. Dialog boxes in desktop applications need titles because that's what appears in task-switching UI alongside other apps.

> **Hint** If you find that `beforeshow`, `aftershow`, `beforehide`, or `afterhide` events triggered from a flyout are getting propagated to a containing app bar or nav bar, which shares the same event names, include a call to `eventArgs.stopPropagation()` inside your flyout's handler.

Menus and Menu Commands

What distinguishes a `WinJS.UI.Menu` control from a more generic `Flyout` is that a menu expects that all its child elements are `WinJS.UI.MenuCommand` objects, similar to how the standard command layout of the app bar expects `AppBarCommand` objects (and won't instantiate if you declare something else). Other common characteristics between the menu control and other flyouts include:

- `show` and `hide` methods.
- `getCommandById`, `showCommands`, `hideCommands`, and `showOnlyCommands`, along with the `commands` property, meaning that you can use the same strategies to manage commands as discussed in "Showing, Hiding, Enabling, and Updating Commands" in the app bar section, including specifying commands using a JSON array rather than discrete elements.
- `beforeshow`, `aftershow`, `beforehide`, and `afterhide` events.

- anchor, alignment, and placement properties.

The menu also has two styles for its appearance—win-menu and win-command—that you use to create more specific selectors, as we've seen, for the entire menu and for the individual text commands.

MenuCommand objects are also very similar to AppBarCommand objects. Both share many of the same properties: id, label, type (a string: button, toggle, flyout, or separator), disabled, extraClass, flyout, hidden, onclick, and selected. Menu commands do not have icons, sections, and tooltips but you can see from type that menu items can be buttons (including just text items), checkable items, separators, and also another flyout. In the latter case, the secondary menu will replace the first rather than show up alongside, and to be honest, I've yet to see secondary menus used in a real app. Still, it's supported in the control!

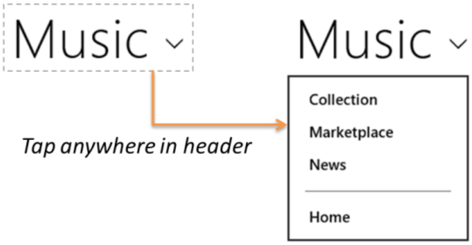
We've already seen how to use a flyout menu from an app bar command, which is covered in scenario 6 of the HTML flyout control sample (see the earlier "Command Menus" section). Another primary use case is to provide what looks like drop-down menu from a header element, covered in scenario 5. Here (see html/header-menu.html), the standard design is to place a down chevron symbol (&#xe099) at the end of the header:

```
<header aria-label="Header content" role="banner">
    <button class="win-backbutton" aria-label="Back"></button>
    <div class="titlearea win-type-ellipsis">
        <button class="titlecontainer">
            <h1>
                <span class="pagetitle">Music</span>
                <span class="chevron win-type-x-large">&#xe099</span>
            </h1>
        </button>
    </div>
</header>
```

Notice that the whole header is wrapped in a button, so its click handler can display the menu with show:

```
document.querySelector(".titlearea").addEventListener("click", showHeaderMenu, false);

function showHeaderMenu() {
    var title = document.querySelector("header .titlearea");
    var menu = document.getElementById("headerMenu").winControl;
    menu.anchor = title;
    menu.placement = "bottom";
    menu.alignment = "left";
    menu.show();
}
```

The flyout (defined as *headerMenu* in html/header-menu.html) appears when you click anywhere on the header (not just the chevron, as that's just a character in the header text):

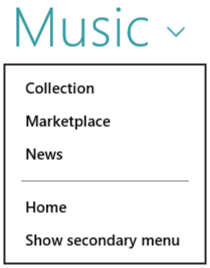


The individual menu commands are just `button` elements themselves, so you can attach `click` handlers to them as you need. As with the app bar, it's best to use the menu control's `getCommandById` to locate these elements because it's more direct than `document.getElementById` (as the SDK sample uses…sigh).

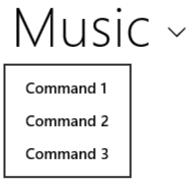
To see a secondary menu in action, try adding the following *secondaryMenu* element in html/header-menu.html before the *headerMenu* element and adding a `button` within *headerMenu* whose `flyout` property refers to *secondaryMenu*:

```
<div id="secondaryMenu" data-win-control="WinJS.UI.Menu">
    <button data-win-control="WinJS.UI.MenuCommand"
          data-win-options="{id:'command1', label:'Command 1'}"></button>
    <button data-win-control="WinJS.UI.MenuCommand"
          data-win-options="{id:'command2', label:'Command 2'}"></button>
    <button data-win-control="WinJS.UI.MenuCommand"
          data-win-options="{id:'command3', label:'Command 3'}"></button>
</div>

<div id="headerMenu" data-win-control="WinJS.UI.Menu">
    <!-- ... -->
    <button data-win-control="WinJS.UI.MenuCommand"
          data-win-options="{id:'showFlyout', label:'Show secondary menu',
          type:'flyout', flyout:'secondaryMenu'}"></button>
</div>
```

Also, go into css/header-menu.css and adjust the `width` style of *#headerMenu* to 200px. With these changes, the first menu will appear as follows where the color change in the header is the hover effect:



When you select *Show secondary menu*, the first menu will be dismissed and the secondary one will appear in its place:



Another example of a header flyout menu can be found in the Adaptive layout with CSS sample we saw in Chapter 8, "Layout and Views." It's implemented the same way we see above, with the added detail that it actually changes the page contents in response to a selection.

Context Menus

Besides the flyout menu that we've seen so far, there are also context menus as described in Guidelines for context menus. These are specifically used for commands that are directly relevant to a selection of some kind, like clipboard commands for text, and are invoked with a right mouse click on that selection, a tap-and-hold gesture, or the context menu key on the keyboard. Text and hyperlink controls already provide such menus by default. Context menus are also good for providing commands on objects that cannot be selected (like parts of an instant messaging conversation), because app bar commands can't be contextually sensitive to such items. They're also recommended for actions that cannot be accomplished with a direct interaction of some kind. However, don't use them on page backgrounds—that's what the app bar is for because the app bar will automatically appear with a right-click gesture.

> **Hint** If you process the right mouse button click event for an element, be aware that the default behavior to show the app/nav bar will be suppressed over that element. Therefore, use the right-click event judiciously, because users will become accustomed to right-clicking around the app to bring up the app/nav bar. Note also that you can programmatically invoke the app bar using its `show` method.

The Context menu sample gives us some context here—I know, it's a bad pun! In all cases, you need only listen to the HTML `contextmenu` event on the appropriate element; you don't need to worry about mouse, touch, and keyboard separately. Scenario 1 of the sample, for instance, has a nonselectable *attachment* element on which it listens for the event (html/scenario1.html):

```
document.getElementById("attachment").addEventListener("contextmenu",
    attachmentHandler, false);
```

In the event handler, you then create a `Windows.UI.Popups.PopupMenu` object (which comes from WinRT, not WinJS!), populate it with `Windows.UI.Popups.UICommand` objects (that contain an item label and click handler) or `UICommandSeparator` objects, and then call the menu's `showAsync` method (js/scenario1.js):

```
function attachmentHandler(e) {
    var menu = new Windows.UI.Popups.PopupMenu();
    menu.commands.append(new Windows.UI.Popups.UICommand("Open with", onOpenWith));
    menu.commands.append(new Windows.UI.Popups.UICommand("Save attachment",
        onSaveAttachment));
    menu.showAsync({ x: e.clientX, y: e.clientY }).done(function (invokedCommand) {
        if (invokedCommand === null) {
            // The command is null if no command was invoked.
        }
    });
}
```

Notice that the results of the showAsync method[78] is the UICommand object that was invoked; you can examine its id property to take further action. Also, the parameter you give to showAsync is a Windows.Foundation.Point object that indicates where the menu should appear relative to the mouse pointer or the touch point. The menu is placed above and centered on this point.

The PopupMenu object also supports a method called showForSelectionAsync, whose first argument is a Windows.Foundation.Rect that describes the applicable selection. Again, the menu is placed above and centered on this rectangle. This is demonstrated in scenario 2 of the sample in js/scenario2.js:

```
//In the contextmenu handler
menu.showForSelectionAsync(
    clientToWinRTRect(window.getSelection().getRangeAt(0).getBoundingClientRect()))
    .done(function (invokedCommand) {
//...

// Converts from client to WinRT coordinates, which take scale factor into consideration.
function clientToWinRTRect(rect) {
    var zoomFactor = document.documentElement.msContentZoomFactor;
    return {
        x: (rect.left + document.documentElement.scrollLeft - window.pageXOffset) * zoomFactor,
        y: (rect.top + document.documentElement.scrollTop - window.pageYOffset) * zoomFactor,
        width: rect.width * zoomFactor,
        height: rect.height * zoomFactor
    };
}
```

This scenario also demonstrates that you can use a contextmenu event handler on text to override the default commands that such controls otherwise provide.

Two final notes for context menus. First, even though the menus are created with WinRT APIs, they do not cause a blur event for the app as a whole, unlike system flyouts like the message dialog. Second, because context menus originate in WinRT, they don't exist in the DOM and are not DOM-aware, which explains the use of other WinRT constructs like Point and Rect rather than plain JavaScript objects. Message dialogs, our final subject for this chapter, share this characteristic.

[78] The sample actually calls then and not done here. If you're wondering why such consistencies exist, it's because the done method was introduced mid-way during the production of Windows 8 when it became clear that we needed a better mechanism for surfacing exceptions within chained promises. As a result, a few SDK samples and code in the documentation still use then instead of done when handling the last promise in a chain. It still works; it's just that exceptions in the chain will be swallowed, thus hiding possible errors.

Message Dialogs

Our last piece of commanding UI for this chapter is the message dialog. Like the context menu, this flyout element comes not from WinJS but from WinRT via the Windows.UI.Popups.MessageDialog API. Again, this means that the message dialog simply appears on top of the current page and doesn't participate in the DOM. Message dialogs automatically dim the app's current page and block input events from the app until the user responds to the dialog. They will also cause a window.onblur event in the app.

The Guidelines for message dialogs topic explains the use cases for this UI:

- To display urgent information that the user must acknowledge to continue, especially conditions that are not related to a user command of some kind.
- Errors that apply to the overall app, as opposed to a workflow where the error is better surfaced inline on the app canvas. Loss of network connectivity is a good example of this.
- Questions that *require* user input and cannot be light dismissed like a flyout. That is, use a message dialog to block progress when user input is essential to continue.

The interface for message dialogs is very straightforward. You create the dialog object with a new Windows.UI.Popups.MessageDialog. The constructor accepts a required string with the message content and an optional second string containing a title. The dialog also has content and title properties that you can use independently. In all cases the strings support only plain text (not HTML).

You then configure the dialog through its commands, options, defaultCommandIndex (the command tied to the Enter key), and cancelCommandIndex (the command tied to the ESC key).

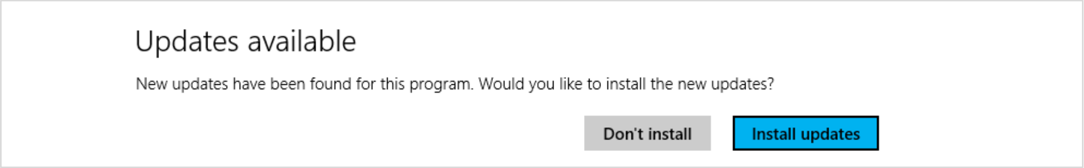
The options come from the MessageDialogOptions enumeration where there are only two members: none (the default, for no special behavior) and acceptUserInputAfterDelay (which causes the message dialog to ignore user input for a short time to prevent possible clickjacking; this exists primarily for Internet browsers loading arbitrary web content and isn't typically needed for most apps).

The commands property then contains up to three UICommand objects, the same ones used in context menus. Each command again contains an id, a label, and an invoked property to which you assign the handler for the command. Note that the defaultCommandIndex and cancelCommandIndex properties work on the indices of the commands array, not the id properties of those commands. Also, if you don't add any commands of your own, the message dialog will default to a single Close command.

Finally, once the dialog is configured, you display it with a call to its showAsync method. Like the context menu, the result is the selected UICommand object that's given to the completed handler you provide to the promise's then/done method. Typically, you don't need to obtain that result because the selected command will have triggered the invoked handler you associated with it.

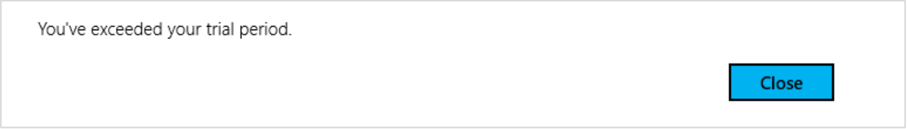
Note If the Search, Share, Devices, or Settings charm is invoked while a message dialog is active, or if an app is activated to service a contract, a message dialog will be dismissed without any command being selected. The completed handler for `showAsync` will be called, however, with the result set to the default command. Be aware of his if you're using the completed handler to process such commands.

The Message dialog sample—one of the simplest samples in the whole Windows SDK!—demonstrates various uses of this API. Scenario 1 displays a message dialog with a title and two command buttons, setting the second command (index 1) as the default. This appears as follows:


Updates available

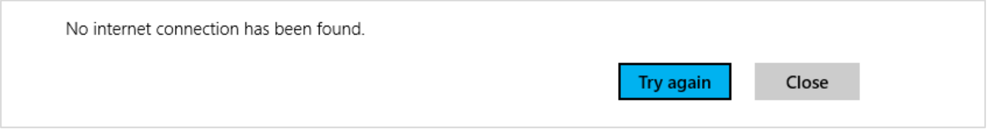
New updates have been found for this program. Would you like to install the new updates?

Scenario 2 shows the default Close command with a message and no title:


You've exceeded your trial period.

Scenario 3 is identical to scenario 1 but uses the completed handler of the `showAsync().done` method to process the selected command. You can use this to see the effect of invoking a charm while the dialog is shown.

Finally, scenario 4 assigns the first command to be the default and marks the second as the cancel command, so the message is dismissed with that command or the ESC key:


No internet connection has been found.

And that's really all there is to it!

Improving Error Handling in Here My Am!

To complete this chapter and bring together much of what we've discussed, let's make some changes to Here My Am!—last seen in Chapter 4—to improve its handling of various error conditions. As it stands right now, Here My Am! doesn't behave very well in a few areas:

- If the Bing Maps control script fails to load from a remote source, the code in html/map.html just throws an exception and the app terminates.
- If we're using the app on a mobile device and have changed our location, there isn't a way to

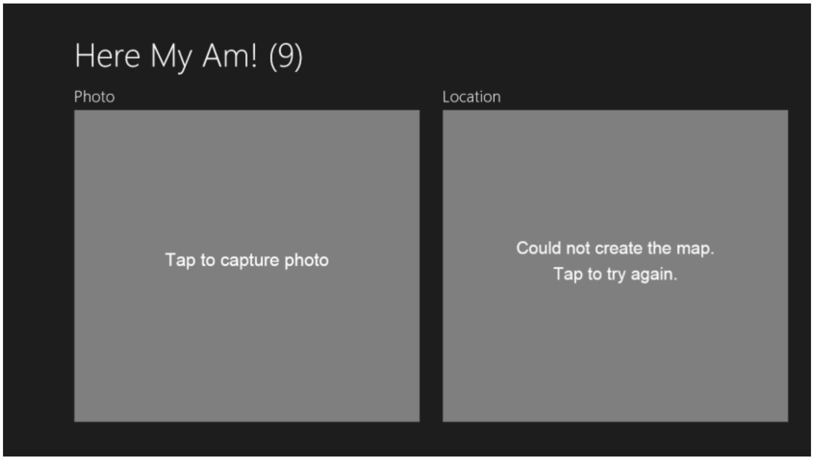
refresh the location on the map other than dragging the pin; that is, the geolocation API is used only on startup.

- When WinRT's geolocation API is trying to obtain a location without a network connection, a several-second timeout period occurs, during which the user doesn't have any idea what's happening.
- If our attempt to use WinRT's geolocation API fails, typically due to timeout or network connectivity problems, but also possibly due to a denial of user consent, there isn't any way to try again.
- If the app's view is smaller than 500px, the camera capture UI will not appear.

The Here My Am! (9) app for this chapter addresses these concerns. First, I've added code to html/map.html to generate a placeholder image for the Location area just like we have in pages/home/home.js for the Photo area. This way a failure to load the Bing maps script will display that message in place of the map (the display style of `none` is removed in that case):

```
<img id="errorImage" style="display: none; width: 100%; height: 100%;" src="#" />
```

I've also added a `click` handler to the image that reloads the webview contents with `document.location.reload(true)`. With this in place, we prevent the exceptions that were previously raised when the map couldn't be created, which caused the app to be terminated. Here's how it looks now if the map can't be created:



To test this, you need to disconnect from the Internet, uninstall the app (to clear any cached map script; otherwise, it will continue to load!), and run the app again. It should hit the error case at the beginning of the `init` method in html/map.html, which shows the (dynamically-generated) error image by removing the default `display: none` style and wiring up the `click` handler. Then reconnect the Internet and click the image, and the map should reload, but if there are continued issues the error message will again appear.

The second problem—adding the ability to refresh our location—is easily done with an app bar. I've added such a control to default.html with one command:

```
<div id="appbar" data-win-control="WinJS.UI.AppBar" data-win-options="">
    <button data-win-control="WinJS.UI.AppBarCommand"
            data-win-options="{id:'cmdRefreshLocation', label:'Refresh location',
            icon:'globe', section:'global', tooltip:'Refresh your location'}"></button>
</div>
```

This command is wired up within pages/home/home.js in the page control's `ready` method:

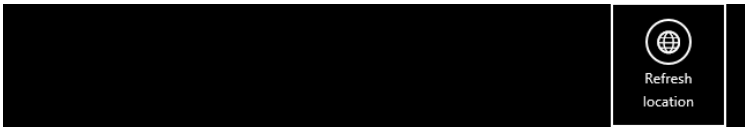
```
var appbar = document.getElementById("appbar").winControl;
appbar.getCommandById("cmdRefreshLocation").addEventListener("click",
this.tryRefresh.bind(this));
```

where the `tryRefresh` handler, also in the page control, hides the app bar and calls another new method, `refreshPosition`, where I moved the code that obtains the geolocation and updates the map:

```
tryRefresh: function () {
    //Hide the app bar and retry
    var appbar = document.getElementById("appbar").winControl.hide();
    this.refreshPosition();
},
```

I also needed to tweak the `pinLocation` function within html/map.html. Without a location refresh command, this function was only ever called once on app startup. Because it can now be called multiple times, we need to remove any existing pin on the map before adding one for the new location. This is done with a call to `map.entities.pop` prior to the existing call to `map.entities.push` that pins the new location.

The app bar now appears as follows, and we can refresh the location as needed. (If you aren't on a mobile device in your car, try dragging the first pin to another location and then refreshing to see the pin return to your current location.)



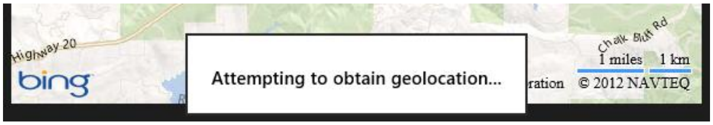
For the third problem—letting the user know that geolocation is trying to happen—we can show a small flyout message just before attempting to call the WinRT geolocator's `getGeopositionAsync` call. The flyout is defined in pages/home/home.html (our page control) to be centered along the bottom of the map area itself:

```
<div id="retryFlyout" data-win-control="WinJS.UI.Flyout" aria-label="{Trying geolocation}"
     data-win-options="{anchor: 'map', placement: 'bottom', alignment: 'center'}">
    <div class="win-type-large">Attempting to obtain geolocation...</div>
</div>
```

The refreshPosition function that we just added to pages/home/home.js makes a great place to display the flyout just before calling getGeopositionAsync:

```
refreshPosition: function () {
    document.getElementById("retryFlyout").winControl.show();

    locator.getGeopositionAsync().done(function (position) {
        //...

        //Always hide the flyout
        document.getElementById("retryFlyout").winControl.hide();

        //...
    }, function (error) {
        //...

        //Always hide the flyout
        document.getElementById("retryFlyout").winControl.hide();
    });
},
```

Note that we want to hide the flyout inside the completed and error handlers so that the message stays visible while the async operation is happening. If we placed a single call to hide outside these handlers, the message would flash only very briefly before being dismissed, which isn't what we want. As we've written it, the user will have enough time to see the notice along the bottom of the map (subject to light dismiss):


Attempting to obtain geolocation...

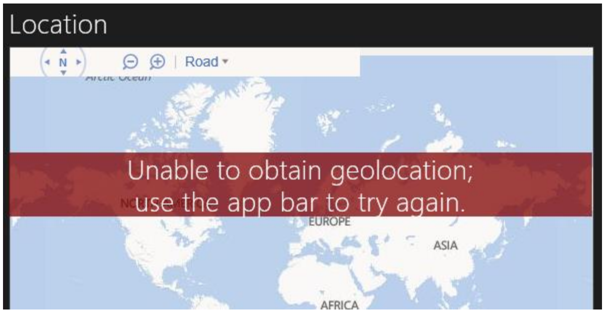
The next piece is to notify the user when obtaining geolocation fails. We could do this with another flyout with a Retry button, or with an inline message as below. We would *not* use a message dialog in this case, however, because the message could appear in response to a user-initiated refresh action. A message dialog might be allowable on startup, but with an inline message combined with the flyout we already added we have all the bases covered.

For an inline message, I've added a floating div that's positioned about a third of the way down on top of the map. It's defined in pages/home/home.html as follows, as a sibling of the map webview:

```
<div id="locationSection" class="subsection" aria-label="Location section">
    <h2 class="group-title" role="heading">Location</h2>
    <iframe id="map" class="graphic" src="ms-appx-web:///html/map.html"
           aria-label="Map"></iframe>
    <div id="noLocation" class="errorOverlay win-type-x-large">
        Unable to obtain geolocation;<br />use the app bar to try again.
    </div>
</div>
```

The styles for the `.errorOverlay` and `#noLocation` rules in pages/home/home.css provide for its placement in the same grid cell as the map, with a semitransparent background; most of the styling is placed in the `.errorOverlay` rule, so we can use it for a camera capture message too:

```
.errorOverlay {
    display: block;
    -ms-grid-column: 1;
    -ms-grid-row: 2;
    width: 100%;
    text-align: center;
    background-color: rgba(128, 0, 0, 0.75);
    opacity: 0.9999;
}
#noLocation {
    -ms-grid-row-align: start;
    margin-top: 20%;
}
```



Location

Unable to obtain geolocation; use the app bar to try again.

Important When overlaying any element on top of a webview, you must follow the rules for independent composition that normally apply to animation, otherwise the webview will render as *black*. As demonstrated here in the `.errorOverlay` rule, the key piece is that you set the `opacity` style to something other than 1.0 or 0.0; those values will fail "layer candidacy" and cause the webview to go black. In this case I'm using an RGBA `background-color` for the semitransparent red overlay and `opacity: 0.9999` (effectively solid, but different from 1.0). Alternately, I could use `background-color: rgb(128, 0, 0)` and set `opacity: 0.75`, though in that case the white text *also* becomes partly transparent with the rest of the element, reducing contrast.

This message will appear if the user denies geolocation consent at startup or allows it but later uses the Settings charm to deny the capability. You can use these variations to test the appearance of the message. It's also possible, if you run the app the first time without network connectivity, for this message to appear on top of the map error image; this is why I've positioned the geolocation error toward the top so that it doesn't obscure the message in the image. But if you've successfully run the app once and then lose connectivity, the map should still get created because the Bing maps script will have been cached.

With display:block in the CSS, the error message is initially visible, so we make sure to hide it on startup, setting display: none. If we get to the error handler for getGeolocationAsync, we set style.display to block again, which reveals the element:

```
document.getElementById("noLocation").style.display = "block";
```

We again hide the message within the tryRefresh function, which is again invoked from the app bar command, so that the message stays hidden unless the error persists:

```
tryRefresh: function () {
    document.getElementById("noLocation").style.display = "none";
    //...
},
```

We can reuse the .errorOverlay class to add a similar message on top of the Photo area when the view width falls below 500px. In this case we need another element in the *photoSection* div:

```
<div id="cannotCapture" class="errorOverlay win-type-x-large">
    Widen the view to capture a photo.
</div>
```

In home.css we can make this initially hidden and then show it automatically through the media query for max-width: 499px:

```
#cannotCapture {
    -ms-grid-row-align: end;
    margin-bottom: 15%;
    display: none;
}

@media screen and (orientation: portrait) and (max-width: 499px) {
    /* Other styles omitted */
    #cannotCapture {
        display: block;
    }
}
```

This shows a small message on top of the Photo in narrow views:

Widen the view to capture a photo.

A little code added to the top of the capturePhoto function also prevents calls to the camera capture UI for narrow widths (this prevents excess debug output too):

```
if (window.innerWidth < 500) {
    return;
}
```

One more piece that could be added, if desired, is a message dialog if connectivity is lost and we can't update our position. I've not done this in the app because it's simply relying on the geolocation

APIs directly, which on some devices might not depend on network connectivity at all. In any case, if this is an appropriate scenario for your app, use the NetworkInformation.onnetworkstatuschanged event we met in Chapter 4. This is a case where a message dialog is appropriate because such a condition does not arise from direct user action.

Also, it's worth noting that if we used the Bing Maps SDK control in this app, the script we're normally loading from a remote source would instead exist in our app package, thereby eliminating the first error case altogether. We'll make this change in the next revision of the app.

What We've Just Learned

- In Windows Store app design, commands that are essential to a workflow should appear on the app canvas or on a popup menu from an element like a header. Those that can be placed on the Setting charm should also go there; doing so greatly simplifies the overall app implementation. Those commands that remain typically appear on an app bar or nav bar, which can contain flyout menus for some commands. Context menus (Windows.UI.Popups.-PopupMenu) can also be used for specific commands on content.
- The app bar is a WinJS control (WinJS.UI.AppBar) on which you can place standard commands or other command controls, using the commands layout, or any HTML of your choice, using the custom layout. Custom icons are also possible, using different fonts or custom graphics. An app can have both a top and a bottom app bar, where the top is typically used for navigation and can employ the WinJS.UI.NavBar control or a custom layout. App bars and nav bars can be sticky to keep them visible instead of being light-dismissed.
- The app/nav bar's showCommands, hideCommands, and showOnlyCommands methods, along with the extraClass property of commands, make it easy to define an app bar in a single location in the app and to selectively show specific command sets by using querySelectorAll with a class that represents that set.
- The NavBar, being a custom layout AppBar, can host arbitrary HTML but is designed to host NavBarContainer controls that provide a paging UI for collections of NavBarCommand objects. A container can use a WinJS.Binding.List as a data source for commands and also supports vertical layout for narrow views.
- The WinJS.UI.Flyout control is used for confirmations and other questions in response to user action; they can also just display a message, collect additional information, or provide controls to change settings for some part of the page. Flyouts are light-dismissed, meaning that clicking outside the control or pressing ESC will dismiss it, which is the equivalent of canceling the question.
- Message dialogs (Windows.UI.Popups.MessageDialog) are used to ask questions that the user must answer or acknowledge before the app can proceed; a message dialog disables the rest of

the app. Message dialogs are best used for errors or conditions that affect the whole app; error messages that are specific to page content should appear inline.

- Command menus, as appear from an app bar command or an on-canvas control of some kind, are implemented with the `WinJS.UI.Menu` control.
- As an example of using many of these features, the Here My Am! app is updated in this chapter to greatly improve its handling of various error conditions.

Chapter 10

The Story of State, Part 1: App Data and Settings

Imagine when you travel if every hotel room you stayed in was automatically configured exactly how you like it—the right pillows and sheets, the right chairs, and the right food in the minibar rather than atrociously expensive and obscenely small snack tins. If you're sufficiently wealthy, of course, you can send people ahead of you to arrange these things, but such luxury remains naught but a dream for most of us.

Software isn't so bound by these limitations, fortunately. Sending agents on ahead doesn't involve booking airfare for them, providing for their income and healthcare, and contributing to their retirement plans. All it takes is a little connectivity, some cloud services, and—voila!—all of your settings can automatically travel with you—that is, between the different devices you're using.

This experience of *statefulness*, as it's called, is built right into Windows. You automatically expect that systemwide settings persist from session to session, so you don't have to reconfigure your profile picture, start screen preferences, Internet favorites, desktop theme, saved credentials, wireless network connections, printers, and so forth. But statefulness is not limited to one device. When you use a Microsoft account to log into Windows on a trusted PC, these settings are securely stored in the cloud and automatically transferred to other trusted devices where you use the same account (you can control them through PC Settings > OneDrive > Sync Settings). I was pleasantly surprised during the development of Windows 8 that I no longer needed to manually transfer all this data when I updated my machine from one release preview to another! Indeed, I was very pleased when I got a new laptop, turned it on for the first time at my in-law's house, and found that it had already connected to their WiFi access point using automatically roamed information.

With such an experience in place for system settings, users expect similar stateful behavior from apps. To continue the analogy, when we travel to new places and stay in hotels, most of us accept that we'll spend a little time upon arrival unpacking our things and setting up the room to our tastes. On the other hand, we expect the complete opposite from our homes: we expect continuity, which is to say, statefulness. Having moved twice in one year myself while writing the first edition of this book (once to a temporary home while our permanent home was being completed), I deeply appreciate the virtues of statefulness. Imagine that everything in your home got repacked into boxes every time you left, such that you had to spend hours, days, or weeks unpacking it all again! No, home is the place where we expect things to stay put, even if we do leave for a time. (I think this is exactly why many people enjoy traveling in a motor home.)

Windows Store apps should maintain a sense of continuity across sessions, across devices, and across process lifecycle events such as when an app is suspended, terminated by the system, and later restarted. In this way, apps feel more like a home than a temporary resting place; they become a place where users come to relax with the content they care about. And the less work users need to do to enjoy that experience, the better.

For stateful behavior across devices, a consistent experience means that app-specific settings on one device will appropriately roam to the same app installed on other devices. I say "appropriately" because some settings don't make sense to roam, especially those that are particular to the hardware in the device. On the other hand, if I configure email accounts in an app on one machine, I would certainly hope those show up on others! (I can't tell you how many times I've had to repeatedly set up my four active email accounts in Outlook on the desktop—ack!) In short, as a user I want my transition between devices—on the system level and with apps—to be both transparent and seamless, such that even newly installed apps that I've used on another device start up in an already-initialized state.

Managing statefulness in an app means a number of things. It means deciding what information is local to a device and what roams between devices. It means understanding when state is restored and when an app starts afresh. It means understanding the difference between app data—settings and configurations that are tied to the existence of an app—and user data—which lives independently. It also includes knowing how best to save certain kinds of state (such as credentials and file access permissions) and how to use state to provide a good offline experience and to improve performance through caching. We'll explore all of these aspects in this chapter, and the effort you invest in these can make a real difference in how users perceive your app and the ratings and reviews they'll give it in the Windows Store.

Many such settings will be completely internal to an app's code, but others can and should be directly configurable by the user. In the past, user configuration has given rise to an oft-bewildering array of nested dialog boxes with multiple tabs, each of which is adorned with buttons, popup menus, and long hierarchies of check boxes and radio buttons. As a result, there's been little consistency in configuration UI. Even the simple matter of where such options are located on menus has varied between Tools/Options, Edit/Preferences, and File/Info commands, among others!

Fortunately, the designers of Windows 8 recognized that most apps have settings of some kind—in fact, Windows guarantees this for all apps you acquire from the Windows Store. Thus they included Settings on the Charms bar alongside the other near-ubiquitous Search, Share, and Devices charms. For one thing, the Settings charm eliminates the need for users to remember where a particular app's settings are located, and app designers don't need to wonder how, exactly, to integrate settings into their overall content flow and navigation hierarchy. By being placed in the Settings charm, settings are effectively removed from an app's content structure, thereby simplifying the app's overall design. The app needs only to provide distinct pages or panes that are displayed when the user invokes the charm.

Clearly, an app's state and its Settings UI are intimately connected, as we'll see in this chapter. Along the way, we'll also have the opportunity to look a bit at the storage and file APIs in WinRT, along with some of the WinJS file I/O helpers and other storage options. Working with files, though, is much more relevant to user data, so we'll wait to complete the subject in Chapter 11, "The Story of State, Part 2: User Data, Files, and OneDrive."

The Story of State

An app's *state*—by which I mean persistent local and roaming state together—is clearly the kingpin of a stateful experience. State has a much longer lifetime than the app itself: it remains persistent, as it should, when an app isn't running and persists across different versions of the app. The state version is, in fact, managed separately from the app version, and roaming state will also persist in the cloud for some time even if the user doesn't have the app installed on any of their devices.

For all these reasons, it's helpful when telling the story of stateful apps to take the perspective of the state itself and ask questions like these:

- What kinds of state do we need to concern ourselves with?
- Where does state live?
- What affects and modifies that state?

To clearly understand the first question, let's first briefly revisit user data again. User data like documents, pictures, drawings, designs, music, videos, playlists, and so forth are things that a user creates and consume *with* an app but are not dependent on the app itself. User data implies that any number of apps might be able to load and manipulate it, and such data always remains on a system irrespective of the existence of apps. For this reason, user data *is not* part of an app's state. For example, while the *paths* or URIs of documents and other files might be remembered in a list of favorites or recently opened documents, the actual *contents* of those files are separate from that state.

User data doesn't have a strong relationship to app lifecycle events either: it's typically saved explicitly through a user-invoked command or implicitly on events like `visibilitychange`, rather than within a `suspending` handler. Again, the app might remember which file is currently loaded as part of its session state during `suspending`, but the file contents itself should be saved outside of this event, especially considering that you have only five seconds to complete whatever work is necessary!

Excluding whatever falls into the category of user data, whatever is left that's needed for an app to run and maintain its statefulness is what we refer to as *app state*. Such state is maintained on a per-user basis, is tied to the existence of a specific app, and is accessible by that app exclusively. As we've seen earlier in this book, app state is typically stored in user-specific folders that are wholly removed from the file system when an app is uninstalled (though of course roaming state still persists in the cloud and is downloaded again if the app is reinstalled). For this reason, never store anything in app state that the user might want *outside* your app. Similarly, avoid using document and media libraries to store state that wouldn't be meaningful to the user if the app is uninstalled.

App state falls into three basic categories:

- **Transient session state** State that normally resides in memory but is saved when the app is suspended in order to restore it after a possible termination. This includes unsaved form data, page navigation history, selection state, and so forth (but not window size, as that's always refreshed when an app is reactivated). As we saw in Chapter 3, "App Anatomy and Performance Fundamentals," being restarted after suspend/terminate is the *only* case in which an app restores transient session state. Session state is typically saved incrementally (as the state changes) or within the `suspending` or `checkpoint` events.

- **Local state** State that is typically loaded when an app is launched and that is specific to a device (and therefore not roamed). Local state includes lists of recently viewed items, temporary files and caches, and various behavioral settings that appear in the Settings panel like display units, preferred video formats, device-specific configurations, and so on. Local state is typically saved when it's changed because it's not directly tied to lifecycle events.

- **Roaming state** State that is shared between the same app running on multiple Windows devices where the same user is logged in, such as favorites, viewing position within videos, account configurations, game scores and progress, URIs for important files on cloud storage locations, perhaps some saved searches or queries, etc. Like local state, these might be manipulated through the Settings panel, but roaming state is subject to an overall quota (and, when exceeded, behaves like local state). Roaming state is also best saved when values are changed; we'll see more details on how this works later.

All this state is stored in the app data folders created when your app package is installed (roaming state is synced to the cloud from there). In these folders you can use settings containers for key-value properties (through the `Windows.Storage.ApplicationData` APIs) or create files with whatever structure you want (using the WinJS or `Windows.Storage` APIs).

At the same time, some types of state live elsewhere in the system, specifically those managed by other APIs. System-managed HTML5 features like IndexedDB and AppCache don't necessarily use your app data folders but are automatically cleaned up when the app is uninstalled. Permissions to programmatically access files and folders is another case. By default, an app can access only those files and folders in its app data or in those libraries for which it has declared a capability in its manifest. For all other arbitrary locations, permission is obtained only through user interaction with the file picker, because that implies user consent. To preserve programmatic access across app sessions, however, you need to save those permissions along with the file path, which is the purpose of the `Windows.Storage.AccessCache` APIs. Your app's section of the access cache, in other words, is considered part of your overall local state. (File pickers and the access cache are covered in Chapter 11.)

The other concern are credentials that you've collected from a user and would like to retrieve in the future. Never directly save credentials in your app data. Instead, use the credential locker API in `Windows.Security.Credentials.PasswordVault`, which we've already seen in Chapter 4, "Web Content and Services." The contents of the locker are isolated between apps and are roamed between a user's trusted PCs, so this constitutes part of roaming state. (Users can elect to not roam credentials by turning off PC Settings > OneDrive > Sync Settings > Other Settings > Passwords, in which case the credential locker still maintains them locally.)

Taken altogether, the different APIs I've just mentioned are those that you (or a third-party library) use to save, modify, and manage state, both from the running app and from background tasks.

Beyond this, two other events not under an app's control (that is, outside of the running app or a background task) can affect app state:

- **Disk Cleanup** If the user runs this tool and elects to clean up Temporary Files, older files in the app's TemporaryFolder might be deleted if disk space is low. The exact policy for when files get deleted is not documented, but the idea is that an app should always be ready to regenerate temp files if they've disappeared. Windows does this kind of lazy cleanup to avoid just blowing away newer and smaller app caches when it's not really necessary to reclaim the space.
- **Roaming from the cloud** If newer roaming state has been uploaded to the cloud from another device and then synced with the local device (in the app data RoamingState folder), a running app will receive a `Windows.Storage.ApplicationData.ondatachanged` event.

To bring all of this together, Figure 10-1 illustrates the different kinds of state, where they are located, what affects them, and the app lifecycle boundaries across which they persist. The APIs that we use to work with these forms of state is what makes up the bulk of this chapter. A good portion of this chapter, starting with "Settings Pane and UI," is concerned with how to surface those parts of your state that are user-configurable.

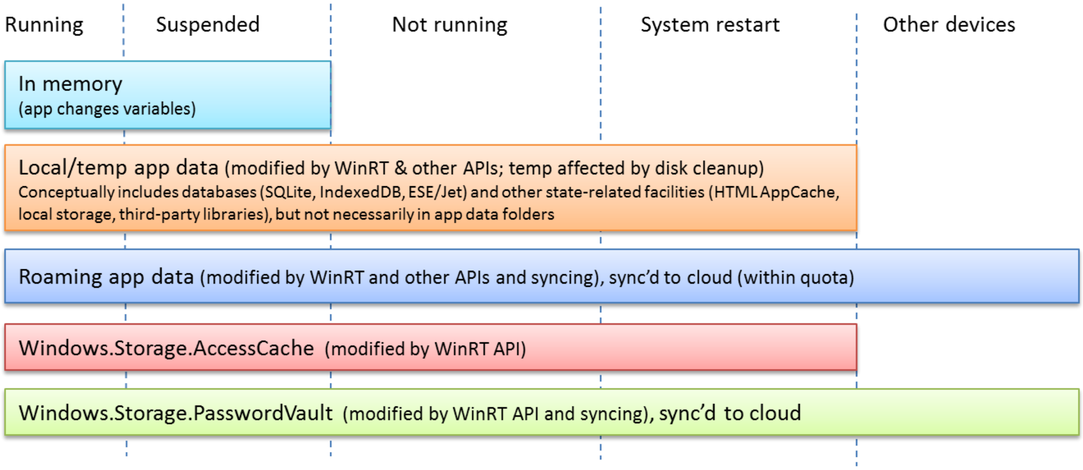
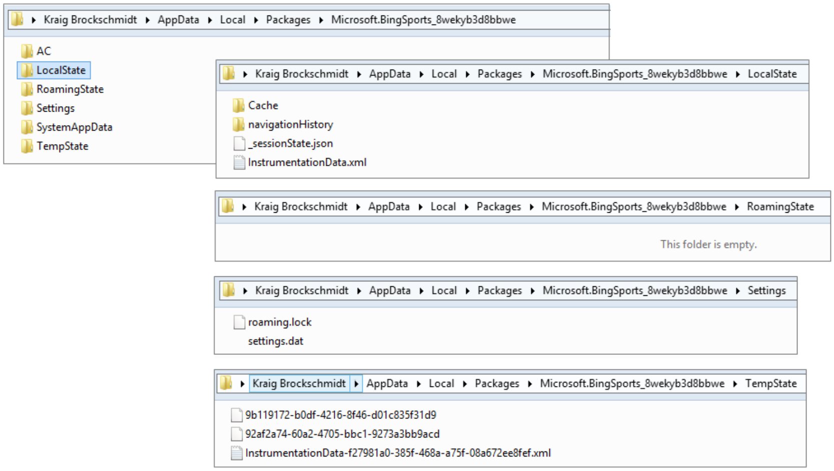

Running
Suspended
Not running
System restart
Other devices
In memory
(app changes variables)
Local/temp app data (modified by WinRT & other APIs; temp affected by disk cleanup)
Conceptually includes databases (SQLite, IndexedDB, ESE/Jet) and other state-related facilities (HTML AppCache, local storage, third-party libraries), but not necessarily in app data folders
Roaming app data (modified by WinRT and other APIs and syncing), sync'd to cloud (within quota)
Windows.Storage.AccessCache (modified by WinRT API)
Windows.Storage.PasswordVault (modified by WinRT API and syncing), sync'd to cloud

FIGURE 10-1 Different forms and locations of app data, how they persist across app lifecycle events, and the APIs that modify them.

App Data Locations

Now that we understand what kinds of information make up app state, let's delve deeper into the question, "Where does state live?" To review, when Windows installs an app for a user (and all Windows Store apps are accessible to only the user who installed them), it automatically creates a folder for the app in the current user's AppData folder (the one that gets deleted when you uninstall an app). It then creates LocalState, TempState, and RoamingState folders within that app folder. On the file system, if you point Windows Explorer to *%localappdata%\packages*, you'll see a bunch of folders for the different apps on your system. If you navigate into any of these, you'll see these folders along with one called "Settings," as shown in Figure 10-2 for the built-in Sports app. The figure also shows the varied contents of these folders.

In the LocalState folder of Figure 10-2 you can see a file named _sessionState.json. This is the file where WinJS saves and loads the contents of the `WinJS.Application.sessionState` object, as we saw in Chapter 3. Because it's just a text file in JSON format, you can easily open it in Notepad or some other JSON viewer to examine its contents. In fact, if you look at this file for the Sports app (that is, at the JSON file shown in the figure), you'll see a value like *{"lastSuspendTime":1340057531501}*. The Sports, News, Weather, and other apps show time-sensitive content, so they save when they were suspended and check elapsed time when they're resumed. If that time exceeds their refresh intervals, they can go get new data from their associated service. In the case of the Sports app, one of its Settings specifically lets the user set the refresh period.



Kraig Brockschmidt ▸ AppData ▸ Local ▸ Packages ▸ Microsoft.BingSports_8wekyb3d8bbwe

AC
LocalState
RoamingState
Settings
SystemAppData
TempState

Kraig Brockschmidt ▸ AppData ▸ Local ▸ Packages ▸ Microsoft.BingSports_8wekyb3d8bbwe ▸ LocalState

Cache
navigationHistory
_sessionState.json
InstrumentationData.xml

Kraig Brockschmidt ▸ AppData ▸ Local ▸ Packages ▸ Microsoft.BingSports_8wekyb3d8bbwe ▸ RoamingState

This folder is empty.

Kraig Brockschmidt ▸ AppData ▸ Local ▸ Packages ▸ Microsoft.BingSports_8wekyb3d8bbwe ▸ Settings

roaming.lock
settings.dat

Kraig Brockschmidt ▸ AppData ▸ Local ▸ Packages ▸ Microsoft.BingSports_8wekyb3d8bbwe ▸ TempState

9b119172-b0df-4216-8f46-d01c835f31d9
92af2a74-60a2-4705-bbc1-9273a3bb9acd
InstrumentationData-f27981a0-385f-468a-a75f-08a672ee8fef.xml

FIGURE 10-2 The Sports app's AppData folders and their contents.

Note If you look carefully at Figure 10-2, you'll see that all the app data–related folders, including RoamingState, are in the user's overall AppData/Local folder. There is also a sibling AppData/Roaming folder, but this applies only to roaming *user account* settings on intranets, such as when a domain-joined user logs in to another machine on a corporate network. This AppData/Roaming folder has no relationship to the AppData/Local.../RoamingState folder for Windows Store apps.

Programmatically, you refer to these locations in several ways. First, you can use the ms-appdata:/// URI scheme as we saw in Chapter 3, where ms-appdata:///local, ms-appdata:///roaming, and ms-appdata:///temp refer to the individual folders and their contents. (Note the triple slashes: it's a shorthand allowing you to omit the package name.) You can also use the object returned from the Windows.Storage.ApplicationData.current method, which contains all the APIs you need to work with state, as we'll see.

By the way, you might have some read-only state directly in file in your app package. You can reference these with URIs that just start with / (meaning ms-appx:///). You can also get to them through the StorageFolder object from the Windows.ApplicationModel.Package.current.installedLocation property. We'll come back to the StorageFolder class a little later.

App Data APIs (WinRT and WinJS)

We've answered the questions of "What kinds of state do we need to concern ourselves with?" and "Where does state live?" Now we can answer the third question, "What affects and modifies that state?"—a subject that will occupy the next 25 pages!

Much of the answer begins with the ApplicationData object that you get from the Windows.Storage.ApplicationData.current property, which is completely configured for your particular app. This object contains the following, where other object types are also found in the Windows.Storage namespace:

- localFolder, temporaryFolder, and roamingFolder Each of these properties is a StorageFolder object that allows you to create whatever files and additional folder structures you want in these locations (but note the roamingStorageQuota below).
- localSettings and roamingSettings These properties are ApplicationDataContainer objects that provide for managing a hierarchy of key-value settings pairs or composite groups of such pairs. All these are stored in the AppData/Settings folder in the settings.dat file.
- roamingStorageQuota This property contains the number of *kilobytes* that Windows will automatically roam for the app (typically 100). If the total data stored in roamingFolder and roamingSettings exceeds this amount, roaming will be suspended until the amount falls below the quota. You have to track how much data you store yourself if you think you might risk exceeding the quota.
- dataChanged An event indicating the contents of the roamingFolder or roamingSettings have been synchronized from the cloud, in which case an app should re-read its roaming state. It also indicates that some other part of the app (running code or a background task) has called the signalDataChanged method. (Note: dataChanged is a WinRT event; use removeEventListener as described in Chapter 3 in "WinRT Events and removeEventListener.")

- signalDataChanged A method that triggers a dataChanged event. This allows you to consolidate local and roaming updates in a single handler for the dataChanged event, including changes that occur in background tasks (that is, calling this method from a background task will trigger the event in the app if the app is not currently suspended.)
- version property and setVersionAsync method These provide for managing the version stamp on your app state. This version applies to the whole of your app state (local, temp, and roaming together); there are not separate versions for each. Note again that state version is a separate matter from app version, because multiple versions of an app can all use the same version of its state (which is to say, the same state structure).
- clearAsync A method that clears out the contents of all AppData folders and settings containers. Use this when you want to reinitialize your default state, which can be especially helpful if you've restarted the app because of corrupt state.
- clearAsync(<locality>) A variant of clearAsync that is limited to one locality (local, temp, or roaming), specified by a value from the ApplicationDataLocality enumeration, such as ApplicationDataLocality.local. In the case of local and roaming, the contents of both the folders and settings containers are cleared; temp affects only the TempState folder.

Let's now see how to use the APIs here to manage the different kinds of app data, which includes a number of WinJS helpers for the same purpose.

Hint App state APIs generate events in the Event Viewer if you've enabled the channel as described in Chapter 3 in "Debug Output, Error Reports, and the Event Viewer." To summarize, make sure that View > Show Analytics and Debug Logs menu item is checked. Then navigate to Application and Services Log, expand *Microsoft/Windows/AppModel-State*, and you'll find Debug and Diagnostic groups. Right-click either or both of these and select Enable Log to record those events.

Settings Containers

For starters, let's look at the localSettings and roamingSettings properties, which are typically referred to as *settings containers*. You work with these through the ApplicationDataContainer API, which is relatively simple. Each container has four read-only properties: a name (a string), a locality (again from ApplicationDataLocality, with local and roaming being the only values here), and collections called values and containers.

The top-level containers have empty names; the property will be set for child containers that you create with the createContainer method (and remove with deleteContainer). Those child containers can have other containers as well, allowing you to create a whole settings hierarchy. That said, these settings containers are intended to be used for small amounts of data, like user configurations; any individual setting is limited to 8K and any composite setting (see below) to 64K. With these limits, going beyond about a megabyte of total settings implies a somewhat complex hierarchy, which will be difficult to manage and will certainly slow to access. So don't be tempted to think of app data settings as a kind of database; other mechanisms like IndexedDB and SQLite are much better suited for that purpose, and you can write however much data you like as files in the various AppData folders (remembering the roaming quota when you write to `roamingFolder`).

For whatever container you have in hand, its `containers` collection is an `MapView` object through which you can enumerate its contents. The `values` collection, on the other hand, is a WinRT `PropertySet` object that you can more or less treat as an array. (For details on both of these types, refer to Chapter 6, "Data Binding, Templates, and Collections," in "Maps and Property Sets.") The `values` property in any container is itself read-only, meaning that you can't assign some other arbitrary array or property set to it but you can still manipulate its *contents* however you like.

We can see this in the Application data sample, which is a good reference for many of the core app data operations. Scenario 2, for example (js/settings.js), shows the simple use of the `localSettings.values` array:

```
var localSettings = Windows.Storage.ApplicationData.current.roamingSettings;
var settingName = "exampleSetting";
var settingValue = "Hello World";

function settingsWriteSetting() {
    roamingSettings.values[settingName] = settingValue;
}

function settingsDeleteSetting() {
    roamingSettings.values.remove(settingName);
}
```

Many settings, like that shown above, are just simple key-value pairs, but other settings will be objects with multiple properties. This presents a particular challenge: although you can certainly write and read the individual properties of that object within the `values` array, what happens if a failure occurs with one of them? That would cause your state to become corrupt.

To guard against this, the app data APIs provide for *composite settings*, which are groups of individual properties (again limited to 64K) that are guaranteed to be managed as a single unit. It's like the perfect group consciousness: either we all succeed or we all fail, with nothing in between! That is, if there's an error reading or writing any part of the composite, the whole composite fails; with roaming, either the whole composite roams or none of it roams.

A composite object is created using `Windows.Storage.ApplicationDataCompositeValue`, as shown in scenario 4 of the Application data sample (js/compositeSettings.js):

```
var roamingSettings = Windows.Storage.ApplicationData.current.roamingSettings;
var settingName = "exampleCompositeSetting";
var settingName1 = "one";
var settingName2 = "hello";

function compositeSettingsWriteCompositeSetting() {
    var composite = new Windows.Storage.ApplicationDataCompositeValue();
    composite[settingName1] = 1; // example value
    composite[settingName2] = "world"; // example value
    roamingSettings.values[settingName] = composite;
    }

function compositeSettingsDeleteCompositeSetting() {
    roamingSettings.values.remove(settingName);
}

function compositeSettingsDisplayOutput() {
    var composite = roamingSettings.values[settingName];
    // ...
}
```

The ApplicationDataCompositeValue object has, as you can see in the documentation, some additional methods and events to help you manage it, such as clear, insert, and mapchanged.

Composites are, in many ways, like their own kind of settings container, but they just cannot contain additional containers. It's important to not confuse the two. Child containers within settings are used only to create a hierarchy (refer to scenario 3 in the sample, js/settingsContainer.js). Composites, on the other hand, specifically exist to create more complex groups of settings *that act like a single unit*, a behavior that is not guaranteed for settings containers themselves.

As noted earlier, these settings are all written to the settings.dat file in your app data Settings folder. It's also good to know that changes you make to settings containers are automatically saved, though some built-in batching occurs to prevent excessive disk activity when you quickly change a number of values. In any case, you don't need to worry about the details—the system will make sure they're always saved before an app is suspended, before the system is shut down, and before roaming settings get synced to the cloud.

State Versioning

As a whole, everything you create in your local, roaming, and temp folders, as well as local and roaming settings, all constitute a version of your app's state structure. If you change that structure, you've created a new version of it.

The version of your state structure is set with ApplicationData.setVersionAsync, the value of which you can retrieve through ApplicationData.version (a read-only property). Windows primarily uses this version especially to manage copies of your app's roaming state in the cloud—it specifically maintains copies of each version separately, because the user could have different versions of the app that each use a distinct version of the state.[79]

As I mentioned before, though, remember that state version (controlled through setVersionAsync) is entirely separate from *app* version (as set in the manifest). You can have versions 1.0.0.0 through 4.3.9.3 of the app use version 1.0.0.0 of app data, or maybe version 3.2.1.9 of the app shifts to version

[79] You can maintain your own versioning system within particular files or settings, but I recommend against doing this with roaming data because it's hard to predict how Windows will manage synchronizing slightly different structures. Even with local state, trying to play complex versioning games is, well, rather complex and probably best avoided altogether.

1.0.1.0 of the app data, and version 4.1.1.3 moves to 1.2.0.0 of the app data. It doesn't really matter, so long as you keep it all straight and can migrate old versions of the app data to new versions!

Migration happens as part of the setVersionAsync call, whose second parameter is a function to handle the conversion. That is, when an updated app is first activated, it must check the version of its state because the contents of the app data folders and settings containers will have been carried forward from the previous app version. If the app finds an older version of state than it expects, it should call setVersionAsync with its conversion function. That function receives a SetVersionRequest object that contains currentVersion and desiredVersion properties, thereby instructing your function as to what kind of conversion is actually needed. Your code then goes through all your app state and migrates the individual settings and files accordingly. Once you return from the conversion handler, Windows will assume the migration is complete, meaning that it can resync roaming settings and files with the cloud. Of course, because the migration process will often involve asynchronous file I/O operations, you have a deferral mechanism like that we've seen with activation. Call the SetVersionRequest.getDeferral method to obtain the deferral object (a SetVersionDeferral), and call its complete method when all your async operations are done. Examples of this can be found in scenario 9 of the Application data sample.

It is also possible to migrate app data as soon as an app update has been installed. For this you use a background task for the servicingComplete trigger. See Chapter 16, "Alive with Activity," specifically "Background Tasks and Lock Screen Apps."

Folders, Files, and Streams

The local, roaming, and temp folders of your app data are where you can create whatever "unstructured" state you want, which means creating files and folders with whatever information you want to maintain. It's high time, then, that we start looking more closely at the file I/O APIs for Windows Store apps, bits and pieces of which we've already seen in earlier chapters. Here we'll round out the basics, and then Chapter 11 will provide the rest of the intricate details.

First, know that some APIs like URL.createObjectURL—which work with what are known as *blobs*—make it possible to do many things in an app without having to descend to the level of file I/O at all! We've already seen how to use this to set the src of an img element, and the same works for other elements like audio and video. The file I/O operations involved with such elements are encapsulated within createObjectURL. You can use blobs in other ways as well. You can convert a canvas element with canvas.msToBlob into something you can assign to an img element; similarly, you can obtain a binary blob from an HTTP request, save it to a file, and then source an img from that. We'll see some more of this in Chapter 13, "Media," and you can refer to the Using a blob to save and load content sample for more. Also, see "Q&A on Files, Buffers, Streams, and Blobs" later in this chapter.

For working directly with files, let's get a bearing on what we have at our disposal. The core WinRT APIs for files live within the `Windows.Storage` namespace. The key players are the [`StorageFolder`] and [`StorageFile`] classes, which clearly represent folder and file entities, respectively. These have a number of very important aspects:

- Both classes are best understood as *rich abstractions for pathnames*. Wherever you'd normally think to use a pathname to refer to some file system entity, you'll typically use one of these objects instead.
- As abstractions for pathnames, neither class maintains any kind of open file handles or such. Reading from or writing to a file requires a *stream* object instead, and it is the existence of a stream, not a `StorageFile`, that holds a file open and enforces access and sharing permissions.
- `StorageFolder` and `StorageFile` both derive from [`IStorageItem`], a generic interface that defines common members like `name`, `path`, `dateCreated`, and `attributes` properties and `deleteAsync` and `renameAsync` methods. For this reason, both classes together are generically referred to as "storage items."
- Both classes include static methods alongside their instance-specific methods, specifically those that return `StorageFolder` or `StorageFile` instances for specific pathnames or URIs.

The key reason why we have these abstractions is that the "file system" in Windows includes anything that can *appear* as part of the file system. This includes cloud storage locations, removable storage, and even other apps that present their contents in a file system–like manner, especially through the file picker. Pathnames by themselves simply cannot refer to entities that don't exist on a physical file system device, so we need objects like `StorageFolder` and `StorageFile` to represent them. Such abstractions allow the file and folder pickers to reach outside the file system and also make it possible to share such references between apps, as through the Share contract. We'll see more of this in later chapters.

The basic operations of `StorageFolder` and `StorageFile` objects are shown in Figure 10-3. As you'd expect, a `StorageFolder` has methods to create, retrieve, and enumerate folders; methods to create, retrieve, and enumerate files; methods to enumerate files and folders together; and methods to delete or rename itself. A `StorageFile`, similarly, has methods to move, copy, delete, and or rename itself. Most important, though, are those methods that *open* a file—resulting in a stream—through which you can then read or write data.

Tip Some file extensions are reserved by the system and won't be enumerated, such as .lnk, .url, and others; a complete list is found on the [How to handle file activation] topic. Also note that the ability to access UNC pathnames requires the *Private Networks (Client & Server)* and *Enterprise Authentication* capabilities in the manifest along with declarations of the file types you want to access.

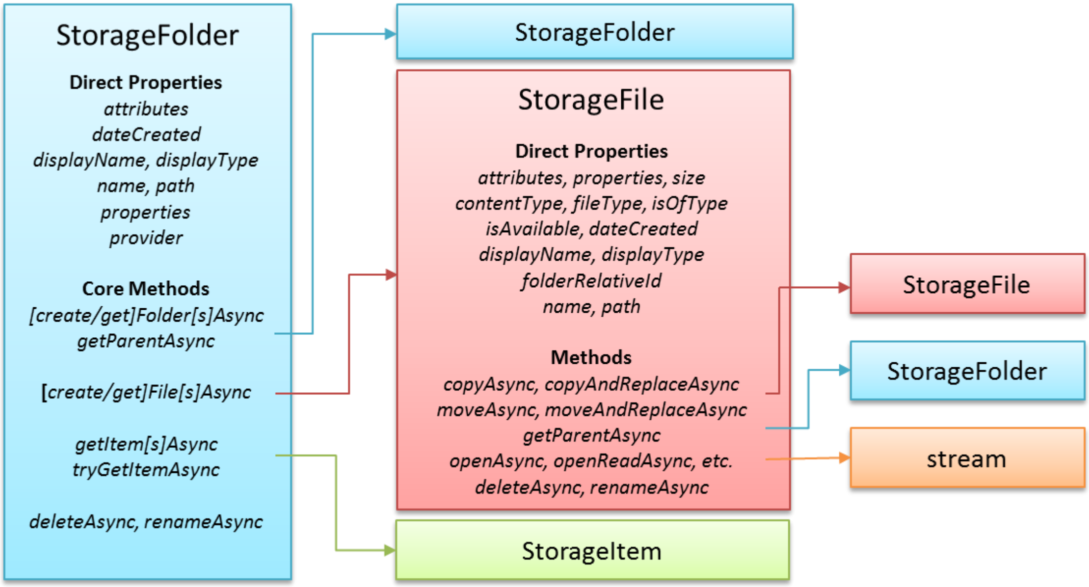


FIGURE 10-3 Core properties and methods of the `StorageFolder` and `StorageFile` classes and the objects they produce.

Given the relationship between the `StorageFolder` and `StorageFile` classes, nearly all file I/O in a Windows Store app starts by obtaining a `StorageFolder` object and then acquiring a `StorageFile` from it. Obtaining that first `StorageFolder` happens through one of the methods below (in a few cases you can also get to a `StorageFile` directly):

- **In-package contents** `Windows.ApplicationModel.Package.current.installedLocation` gets a `StorageFolder` through which you can load data from files in your package (all files therein are read-only).
- **App data folders** `Windows.Storage.ApplicationData.current.localFolder`, `roamingFolder`, or `temporaryFolder` provides `StorageFolder` objects for your app data locations (read-write).
- **Arbitrary file system locations** An app can allow the user to select a folder or file directly using the file pickers invoked through `Windows.Storage.Pickers.FolderPicker` plus `FileOpenPicker` and `FileSavePicker`. This is the preferred way for apps that don't need to enumerate contents of a library (see next bullet). This is also the only means through which an app can access safe (nonsystem) areas of the file system without additional declarations in the manifest. Alternately, a user can launch a file directly from Windows Explorer from any location, and the default app associated with that file type receives the `StorageFile` upon activation.
- **Libraries** `Windows.Storage.KnownFolders` provides `StorageFolder` objects for the Pictures, Music, and Videos libraries, as well as Removable Storage. Given the appropriate capabilities in your manifest, you can work with the contents of these folders. (Attempting to obtain a folder

without the correct capability will throw an Access Denied exception.)

- **Downloads** The `Windows.Storage.DownloadsFolder` object provides a `createFolderAsync` method through which you can obtain a `StorageFolder` in that location. It also provides a `createFileAsync` method to create a `StorageFile` directly. You would use this API if your app manages downloaded files directly. Note that `DownloadsFolder` itself provides only these two methods; it is not a `StorageFolder` in its own right, to prevent apps from interfering with one another's downloads.
- **Arbitrary paths** The static method `StorageFolder.getFolderFromPathAsync` returns a `StorageFolder` for a given pathname *if and only if* your app already has permissions to access it; otherwise, you'll get an Access Denied exception. A similar static method exists for files called `StorageFile.getFileFromPathAsync`, with the same restrictions; the static method `StorageFile.getFileFromApplicationUriAsync` opens files with `ms-appx://` (package) and `ms-appdata:///` URIs. Other schema are not supported.
- **Access cache** Once a folder or file object is obtained, it can be stored in the `AccessCache` that allows an app to retrieve it sometime in the future with the same programmatic permissions. This is primarily needed for folders or files selected through the pickers because permission to access the storage item is granted only for the lifetime of that in-memory object—we'll see more in Chapter 11. The short of it is that you should always use this API, as demonstrated in scenario 7 of the File access sample, where you'd normally think to save a file path as a string. Again, `StorageFolder.getFolderFromPathAsync` and `StorageFile.getFileFromPathAsync` will throw Access Denied exceptions if they refer to any locations where you don't already have permissions. Also, pathnames will typically not work for files provided by another app through the file picker, because the `StorageFile` object might not, in fact, refer to any kind of file system entity.

Beyond just enumerating a folder's contents, you often want only a partial list filtered by certain criteria (like file type), along with thumbnails and other indexed file metadata (like music album and track info, picture titles and tags, etc.) that you can use to group and organize the files. This is the purpose of file, folder, and item *queries,* as well as extended properties and thumbnails. We already saw a little with the FlipView app we built using the Pictures Library in Chapter 7, "Collection Controls," and we'll return to the subject in Chapter 11.

In the end, of course, we usually want to get to the *contents* of a particular file. This is the purpose of the `StorageFile.open*` methods, each variant providing a different kind of access. The result in each case is some kind of *stream*, an object that's backed by a series of bytes and transparently accommodates the flow of those bytes between storage systems with different latencies, such as memory and disk, disk and network, memory and network, and so on. Each stream type has certain characteristics based its means of access:

- `openAsync` and `openReadAsync` provide random-access byte streams for read/write and read-only, respectively. The streams are objects with the `IRandomAccessStream` and `IRandomAccessStreamWithContentType` interfaces, respectively, both in the

Windows.Storage.Streams namespace. The first of these works with a pure binary stream; the second works with data+type information, as would be needed with an http response that prepends a content type to a data stream.

- openSequentialReadAsync provides a read-only Windows.Storage.Streams.IInputStream object through which you can read file contents in blocks of bytes but cannot skip back to previous locations. You should always use this method when you need only to consume (read) the stream as it has better performance than a random access stream (the source can optimize for sequential reads).
- openTransactedWriteAsync provides a Windows.Storage.StorageStreamTransaction that's basically a helper object around an IRandomAccessStream with commitAsync and close methods to handle transactions. This is necessary when saving complex data to make sure that the whole write operation happens atomically and won't result in corrupted files if interrupted. Scenario 5 of the File access sample shows this.

The StorageFile class also provides these static methods: createStreamedFileAsync, createStreamedFileFromUriAsync, replaceWithStreamedFileAsync, and replaceWithStreamedFileFromUriAsync. These provide a StorageFile that you typically pass to other apps through contracts, as we'll see in Chapter 15, "Contracts." The utility of these methods is that the underlying file isn't accessed at all until data is first requested from it, *if* such a request ever happens.

Pulling all this together, here's a bit of code using the raw API we've seen thus far to create and open a "data.tmp" file in our temporary AppData folder and to write a given string to it. This bit of code is in the RawFileWrite example for this chapter. Let me be clear that what's shown here utilizes *low-level* APIs in WinRT and is **not** what you typically use, as we'll see in the next section. It's instructive nonetheless, as you might occasionally need to exercise precise control over the process:

```
var fileContents = "Congratulations, you're written data to a temp file!";
writeTempFileRaw("data.tmp", fileContents);

function writeTempFileRaw(filename, contents) {
    var ws = Windows.Storage;
    var tempFolder = ws.ApplicationData.current.temporaryFolder;
    var outputStream;

    //Promise chains, anyone?
    tempFolder.createFileAsync(filename, ws.CreationCollisionOption.replaceExisting)
    .then(function (file) {
        return file.openAsync(ws.FileAccessMode.readWrite);
    }).then(function (stream) {
        outputStream = stream.getOutputStreamAt(0);
        var writer = new ws.Streams.DataWriter(outputStream);
        writer.writeString(contents);
        return writer.storeAsync();
    }).done(/* Completed handler if necessary */);
}
```

Good thing we learned about chained async operations a while back! Starting with a StorageFolder from the ApplicationData object, we call its createFileAsync to get a StorageFile. Then we open that file to obtain a random access stream. At this point the file is open and locked, so no other apps can access it.

Now a random access stream itself doesn't have any methods to read or write data. For that we use the [DataReader](#) and [DataWriter](#) classes in Windows.Storage.Streams. The DataReader takes an IInputStream object, which you obtain from IRandomAccessStream.getInputStreamAt(<offset>). The DataWriter, as shown here, takes an IOutputStream similarly obtained from IRandomAccessStream.getOutputStreamAt(<offset>).

The DataReader and DataWriter classes then offer a number of methods to read or write data, either as a string (as shown above with writeString), as binary (for instance, DataReader.loadAsync), or as specific data types. With the DataWriter, we have to end the process with a call to its storeAsync, which commits the data to the backing store. With both DataReader and DataWriter, discarding the objects will close the files and invalidate the streams.

Two tips First, if you use a transacted output stream, you also need to call its flushAsync method in the chain after DataWriter.storeAsync. Second, the DataReader and DataWriter constructors do not validate their backing streams. That happens only during loadAsync, storeAsync, and other read/write operations, which will throw confusing exceptions if the backing stream is invalid. When in doubt, double-check the stream passed to the constructor.

Now, you might be saying, "You've got to be kidding me! Four chained async operations just to write a simple string to a file! Who designed this API?" Indeed, when we started building the first Store apps within Microsoft, this is all we had and we asked these questions ourselves! After all, doing some basic file I/O is typically the first thing you add to a Hello World app, and this was anything but simple. To make matters worse, at that time we didn't yet have promises for async operations in JavaScript, so we had to write the whole thing with raw nested async operations. Such were the days.

Fortunately, simpler APIs were already available and more came along shortly thereafter. These are the APIs you'll typically use when working with files, as we'll see in the next section. It is nevertheless important to understand the structure of the low-level code above because the DataReader and DataWriter classes are very important mechanisms for working with a variety of different I/O streams and are essential for data encoding processes. Having control over the fine details also supports scenarios such as having different components in your app that are all contributing to the file structure. So it's good to take a look at the DataReader/DataWriter reference documentation along with the [Reading and writing data sample](#) to familiarize yourself with the capabilities.

Tip You don't see any reference to a *close* method on the file or stream in the RawFileWrite example because that's automatically taken care of in the DataWriter (the DataReader does it as well). A stream does, in fact, have a close method that will close its backing file, which is what the DataReader and DataWriter objects call when they're disposed. If, however, you separate a stream from these objects through their detachStream methods, you must call the stream's close yourself.

When you're developing apps that write to files and you see errors indicating that the file is still open, check whether you've properly closed the streams involved. For more on this, see "Q&A on Files, Streams, Buffers, and Blobs" a little later in this chapter.

FileIO, PathIO, and WinJS Helpers (plus FileReader)

Simplicity is a good thing where file I/O is concerned, and the designers of WinRT made sure that the most common scenarios didn't require a long chain of async operations like we saw in the previous section. The `Windows.Storage.FileIO` and `PathIO` classes provide such a streamlined interface—and without having to muck with streams! The only difference between the two is that the `FileIO` methods take a `StorageFile` parameter whereas the `PathIO` methods take a pathname string, the latter assuming that you already have programmatic access to that path. Beyond that, both classes offer the same methods called `[read | write]BufferAsync` (these work with byte arrays), `[append | read | write]LinesAsync` (these work with arrays of strings), and `[append | read | write]TextAsync` (these work with singular strings). With strings, `WinJS.Application` simplifies matters even further for your appdata folders: its `local`, `roaming`, and `temp` properties, which implement an interface called `IOHelper`, provide `readText` and `writeText` methods that relieve you from even having to touch a `StorageFile` object.

Using the `FileIO` class, the code in the previous section can be reduced to the following, which can also be found in the RawFileWrite example (js/default.js):

```
function writeTempFileSimple(filename, contents) {
    var ws = Windows.Storage;
    var tempFolder = ws.ApplicationData.current.temporaryFolder;

    tempFolder.createFileAsync(filename, ws.CreationCollisionOption.replaceExisting)
    .then(function (file) {
        ws.FileIO.writeTextAsync(file, contents);
    });
}
```

And here's the same thing written with `WinJS.Application.temp.writeText`, which is an async call and returns a promise if you need it:

```
WinJS.Application.temp.writeText(filename, fileContents);
```

Additional examples can be found in Scenario 3 of the File access sample. One other option you have—*provided the file already exists*—is using the `PathIO.writeTextAsync` method with an `ms-appdata` URI like so:

```
Windows.Storage.PathIO.writeTextAsync("ms-appdata:///temp/" + filename, contents);
```

Reading text from a file through the async `readText` method is equally simple, and WinJS provides two other methods through `IOHelper`: `exists` and `remove`.[80] That said, these WinJS helpers are

[80] If you're curious why async methods like `readText` and `writeText` don't have *Async* in their names, this was a conscious

available for *only* your AppData folders and not for the file system more broadly. For areas outside app data, you must use the FileIO and PathIO classes.

You also have the HTML5 FileReader class available for use in Windows Store apps, which is part of the W3C File API specification. As its name implies, it's suited only for reading files and cannot write them, but one of its strengths is that it can work both with files and blobs. Some examples of this are found in the Using a blob to save and load content sample.

Encryption and Compression

WinRT provides two capabilities that might be helpful to your state management: encryption and compression. Encryption is provided through the Cryptography and Cryptography.Core API, both part of the Windows.Security namespace. Cryptography contains methods for basic *encoding* and decoding (base64, hex, and text formats); Cryptography.Core handles *encryption* according to various algorithms. As demonstrated in the Secret saver encryption sample, you typically encode data in some manner with the Cryptography.CryptographicBuffer.convertStringToBinary method and then create or obtain an algorithm and pass that with the data buffer to Cryptography.Core.CryptographicEngine.encrypt. Methods like decrypt and convertBinaryToString perform the reverse.

Compression is a little simpler in that it's just a built-in API through which you can make your data smaller (say, to decrease the size of your roaming data). The Windows.Storage.Compression API for this is composed of Compressor and Decompressor classes, both of which are demonstrated in the Compression sample. Although this API can employ different compression algorithms, including one called MSZIP, it does *not* provide a means to manage .ZIP *files* and the contents therein. For this purpose either employ a third-party JavaScript library or write a WinRT component in C++ that utilizes a higher-performance library (see Chapter 18, "WinRT Components").

Both the encryption and compression APIs utilize a WinRT structure called a *buffer*, which is another curious beast like the random access stream in that it doesn't have its own methods to manipulate it. Here again you use the DataReader and DataWriter classes, as described in the next section.

Q&A on Files, Streams, Buffers, and Blobs

As we've started to see in this chapter, the APIs for working with files and state begin to involve a plethora of object types and interfaces, all of which deal with managing and manipulating piles of data in some manner. Here's the complete roster I'm referring to:

- **File system entities**, represented by the StorageFolder and StorageFile classes and the IStorageItem interface, all in the Windows.Storage namespace of WinRT.
- **Streams**, represented by classes in Windows.Storage.Streams. Here we find a veritable

choice on the part of the WinJS designers to follow existing JavaScript conventions where such a suffix isn't typically used. The WinRT API, on the other hand, is language-independent and thus has its own convention with the *Async* suffix.

pantheon of types: FileInputStream, FileOutputStream, FileRandomAccessStream, IInputStream, IOutputStream, IRandomAccessStream, InMemoryRandomAccessStream, InputStreamOverStream, OutputStreamOverStream, RandomAccessStream, and RandomAccessStreamOverStream.

- **Buffers**, also in Windows.Storage.Streams, limited here to the Buffer class and the IBuffer interface.
- **Blob and MSStream objects**, which come from the app host via the HTML API and serve to bridge gaps between the HTML5 world and WinRT.

In this section I want to bring all of these together, because you often encounter one or more of these types at an inconvenient point in your app development, like where you're just trying to complete a seemingly simple task but end up getting lost in a jungle, so to speak. And I haven't found a topic in the documentation that makes sense of them all. But instead of boring you with every last detail, I've reduced matters down to a few key questions, the answers to which have, for me, made much better sense of the landscape. So here goes.

Question #1: Why on earth doesn't the StorageFile class have a *close* method?

Answer: As noted earlier, StorageFolder and StorageFile are abstractions for pathnames and thus only represent entities on the extended/virtual file system but not the *contents* of those entities (which is what streams are for). If you're even asking this question, it means you need to update your mental model. After all, one of the first things we tinker with when learning to code is file I/O. We learn to open a file, read its contents, and close the file, thus establishing a basic mental model at a young age for anything with the name of "file." Indeed, if you learned this through the C runtime library (OK, so I'm dating myself), you used a function like fopen with a pathname to open a file and get a handle. Then you called fread with that handle to read data, followed by fclose. All those methods nicely site next to each other in the API reference.

Coming to WinRT, you see StorageFile.openAsync quickly enough but then can't find the *read* and *close* equivalents anywhere nearby, which breaks the old mental model. What gives? The main difference is that whereas fopen and its friends are synchronous, WinRT is an asynchronous API. In that asynchronous world, the request to open a StorageFile produces some result later on. So, technically, the file isn't actually "open" until that result is delivered.

That result, to foreshadow the next question, is some kind of stream, which is the analog to a file handle. That stream is what manages access to the file's contents, so when you want to read from "the file" ***à la*** fread you actually read from a stream that's connected to the file contents. When you want to "close the file" in the way that you think of with fclose, you close and dispose of the *stream* through its own close method. This is why the StorageFile object does not have a *close*: it's the stream that holds the backing entity open, so you must close the stream.

Again, all of this is important because many file-like entities actually have no pathname at all thanks to the unification of local-, cloud-, and app-based entities. This means that whatever thingy a `StorageFile` represents might not be local to the device or even exist as a real file anywhere in the known universe—backing data for a `StorageFile` can, in fact, be generated on the fly and fed into the stream. Lots of work might be involved, then, in getting a stream through which you can get to that entity's contents, and that's the complexity that the WinRT API is handling on your behalf. Be grateful!

And to repeat another point from earlier, the APIs in `Windows.Storage.FileIO` and `PathIO` classes shield you from streams altogether (see Question #4).

Question #2: My God, what are all those different stream types about?

Answer: Trust me, I feel your pain! Let's sort them out.

A stream is just an abstraction for a bit bucket. Streams don't make any distinction about the data in those buckets, only about how you can get to those bits. Streams are used to access file contents, pass information over sockets, talk to devices, and so forth.

Streams come in two basic flavors: original and jalapeño. Oops! I'm writing this while cooking dinner...sorry about that. They come in two sorts: sequential and random access. This differentiation allows for certain optimizations to be made in the implementation of the stream:

- A sequential stream can assume that data is accessed (read or written) once, after which it no longer needs to be cached in memory. Sequential streams do not support seeking or positioning. When possible, it's always more memory-efficient to use a sequential stream.
- A random access stream needs to keep that data around in case the consuming code wants to rewind or fast-forward (seek or position).

As mentioned with Question #1, all streams have a `close` method that does exactly what you think. If the stream is backed by a file, it means closing the file. If the stream is backed by memory, it means freeing that memory. If it's backed by a socket, it means closing the socket. You get the idea.

In the sequential group there is a further distinction between "input" streams, which support reading (but not writing), and "output" streams, which support writing (but not reading). These reflect the reality that communication with many kinds of backing stores is inherently unidirectional—for example, downloading or uploading data through HTTP requests.

The primary classes in this group are `FileInputStream` and `FileOutputStream`. Also, the `IInputStream` and `IOutputStream` interfaces serve as the basic abstractions. (Don't concern yourself with `InputStreamOverStream` and `OutputStreamOverStream`, which are wrappers for lower-level COM `IStream` objects.)

An input stream has a method `readAsync` that copies bytes from the source into a buffer (an abstraction for a byte array, see Question #3). An output stream has two methods, `writeAsync`, which copies bytes from a buffer to the stream, and `flushAsync`, which makes sure the data is written to the backing entity before it deems the flushing operation is complete. When working with such streams, you always want to call flushAsync, using its completed handler for any subsequent operations (like a copy) that are dependent on that completion.

Now for the random access group. Within Windows.Storage.Streams we find the basic types: FileRandomAccessStream, InMemoryRandomAccessStream, and RandomAccessStream, along with the abstract interfaces IRandomAccessStream. (RandomAccessStreamOverStream again builds on the low-level COM IStream and isn't something you use directly.)

The methods of IRandomAccessStream are common among classes in this group. It provides:

- Properties of canRead, canWrite, position, and size.
- Methods of seek, cloneStream (the clone has an independent position and lifetime), getInputStreamAt (returns an IInputStream), and getOutputStreamAt (returns an IOutputStream).

The getInputStreamAt and getOutputStreamAt methods are how you obtain a sequential stream for some *section* of the random access stream, allowing more efficient read/write operations. You often use these methods to obtain a sequential stream for some other API that requires them.

If we now look at FileRandomAccessStream and InMemoryRandomAccessStream (whose backing data sources I trust are obvious), they have everything we've seen already (properties like position and canRead, and methods like close and seek) along with two more methods, readAsync and writeAsync, which behave just like their counterparts in sequential input and output streams (using those buffers again).

As for the RandomAccessStream class, it's a curious beast that contains *only* static members—you never have an instance of this one. It exists to provide the generic helper methods copyAsync (with two variants) and copyAndCloseAsync that transfer data between input and output streams. This way other classes like FileRandomAccessStream don't need their own copy methods. To copy content from one file into another, you call FileRandomAccessStream.getInputStreamAt on the source (for reading) and FileRandomAccessStream.getOutputStreamAt on the destination (for writing), and then you pass those to RandomAccessStream.copyAsync (to leave those streams open) or copyAndCloseAsync (to automatically do a flushAsync on the destination and close on both).

The other class to talk about here, RandomAccessStreamReference, also supplies other static members. When you read "reference," avoid thinking about reference types or pointers or anything like that—it's more about having read-only access to resources that might not be writable, like something at the other end of a URI. Its three static methods are createFromFile (which takes a StorageFile), createFromStream (which takes an IRandomAccessStream), and createFromUri (which takes a Windows.Foundation.Uri that you construct with a string). What you then get back from each of these static methods is an *instance* of RandomAccessStreamReference (how's that for confusing?). That instance has just one method, openReadAsync, whose result is an IRandomAccessStream-WithContentType (the same thing as an IRandomAccessStream with an extra string property contentType to identify its data format).

To sum up, remember the difference between sequential streams (input or output) and random access streams, and remember that streams are just ways to talk to the bits (or bytes) that exist in some backing entity like a file, memory, or a socket. When a stream exists, its backing entity is "open" and typically locked (depending on access options) until the stream is closed or disposed.

Question #3: So now what do I do with these buffer objects, when neither the Buffer class nor the IBuffer interface have any methods?

Answer: I totally agree that this one stumped me for a while, which is one reason I've included this Q&A in this book!

As in question #2, straightforward stream object methods like readAsync and writeAsync result in this odd duck called a Buffer instead of just giving us a byte array. The problem is, when you look at the reference docs for Buffer and IBuffer, all you see are two properties: length and capacity. "That's all well and good," you say (and I have said too!), "but how the heck do you get at the data itself?" After all, if you just opened a file and read its contents from an input stream into a buffer, that data exists *somewhere*, but this buffer thing is just a black box as far as you can see!

Indeed, looking at the Buffer itself, it's hard to understand how one even gets created in the first place, because the object itself also lack methods for storing data (the constructor takes only a capacity argument). This gets confusing when you need to provide a buffer to some API, like ProximityDevice.PublishBinaryMessage (for near-field communications): you need a buffer to call the function, but how do you create one?

To make things clear, first understand that a buffer is just an abstraction for an untyped byte array, and it exists because marshaling byte arrays between Windows and language-specific environments like JavaScript and C# is a tricky business. Having an abstract class makes such marshaling easier to work with in the API.

Next, when you are about to call FileRandomAccessStream.readAsync, you do, in fact, create an empty buffer with new Buffer(), which readAsync will populate. On the other hand, if you need to create a new buffer with real data, there are two ways to go about it:

- One way is our friend Windows.Storage.Streams.DataWriter. Create an instance with new DataWriter(), and use its write* methods to populate it with whatever you want (including writeBytes, which takes an array). Once you've written those contents, call its detachBuffer and you have your populated Buffer object.
- The other way is super-secret or maybe just unintentional, but nonetheless it's useful here. You have to look way down in Windows.Security.Cryptography.—wait for it!—CryptographicBuffer.createFromByteArray. This API has nothing to do with cryptography *per se* and simply creates a new buffer with a byte array you provide. This is simpler than using

DataWriter if you have a byte array; DataWriter is better, though, if you have data in any other form, such as a string.

How, then, do you get data *out* of a buffer? With the Windows.Storage.Streams.DataReader class. Create an instance of DataReader with the *static* method DataReader.fromBuffer, after which you can call methods like readBytes, readString, and so forth.[81]

In short, the methods that you would normally expect to find on a class like Buffer are found instead within DataReader and DataWriter, because these reader/writer classes also work with streams. That is, instead of having completely separate abstractions for streams and byte arrays with their own read/write methods for different data types, those methods are centralized within the DataReader and DataWriter objects that are themselves initialized with either a stream or a buffer. DataReader and DataWriter also take care of the details of closing streams for you when appropriate.

In the end, this reduces the overall API surface area once you understand how they relate.

Question #4: Now doesn't all this business with streams and buffers make simple file I/O rather complicated?

Answer: Yes and no, and for the most part we've already answered this question earlier in "FileIO, PathIO, and WinJS Helpers" with the RawFileWrite code examples. To reiterate, the low-level API gives you full control over the details; the higher-level APIs to simplify common scenarios.

Indeed, we can go even one step lower than our *writeTempFileRaw* function. In that code we used DataWriter.storeAsync, which is itself just a helper for the truly raw process that involves buffers. Here is the *lowest-level* implementation for writing a file (see js/default.js in the example):

```
var fileContents = "Congratulations, you're written data to a temp file!";
writeTempFileReallyRaw("data-raw.tmp", fileContents);

function writeTempFileReallyRaw(filename, contents) {
    var ws = Windows.Storage;
    var tempFolder = ws.ApplicationData.current.temporaryFolder;
    var writer;
    var outputStream;

    //All the control you want!
    tempFolder.createFileAsync(filename, ws.CreationCollisionOption.replaceExisting)
    .then(function (file) {
        //file is a StorageFile
        return file.openAsync(ws.FileAccessMode.readWrite);
    }).then(function (stream) {
```

[81] If you use new with DataReader, you provide an IInputStream argument instead—with buffers you have to use the static method. The reason for this is that JavaScript cannot differentiate overloaded methods by *arity* only (number of arguments), thus the designers of WinRT have had to make a few oddball choices like this where the more common usage employs the constructor and a static method is used for the less common option.

```
        //Stream is an RandomAccessStream. To write to it, we need an IOuputStream
        outputStream = stream.getOutputStreamAt(0);
        //Create a buffer with contents
        writer = new ws.Streams.DataWriter(outputStream);
        writer.writeString(contents);
        var buffer = writer.detachBuffer();
        return outputStream.writeAsync(buffer);
    }).then(function (bytesWritten) {
        console.log("Wrote " + bytesWritten + " bytes.");
        return outputStream.flushAsync();
    }).done(function () {
        writer.close(); //Closes the stream too
    });
}
```

Again, within this structure, you have the ability to inject any other actions you might need to take at any level, such as encrypting the contents of the buffer or cloning a stream. But if you don't need to see the buffers you can cut that part out by using DataWriter.storeAsync. If you don't need to play with the streams directly, you can use the FileIO class to hide those details. And if you have programmatic access to the desired location on the file system, you can even forego using StorageFile by instead using the PathIO class or the WinJS.Application.local, roaming, and temp helpers.

So, yes, file I/O can be complicated but only if you *need* it to be. The APIs are designed to give you the control you need when you need it but not burden you when you don't.

Question #5: What are MSStream and Blob objects in HTML5, and how do they relate to the WinRT classes?

Answer: To throw another stream into the river, so to speak, when working with the HTML5 APIs, specifically those in the File API section, we encounter MSStream and Blob types. (See the W3C File API and the Blob section therein for the standards.) As an overview, the HTML5 File APIs—as provided by the app host to Store apps written in JavaScript—contain a number of key objects that are built to interoperate with WinRT APIs:

- Blob A piece of immutable binary data that allows access to ranges of bytes as separate blobs. It has size, type, and msRandomAccessStream properties (the latter being a WinRT IRandomAccessStream). It has two methods: slice (returning a new blob from a subsection) and msClose (releases a file lock).
- MSStream Technically an extension of this HTML5 File API that provides interop with WinRT. It lives outside of WinRT, of course, and is thus available to both the local and web contexts in an app. The difference between a blob and a stream is that a stream doesn't have a known size and can thus represent partial data received from an in-process HTTP request. You can create an MSStreamReader object and pass an MSStream to its readAsBlob method to get the blob once the rest is downloaded.
- URL Creates and revokes URLs for blobs, MSStream, IRandomAccessStreamWithContentType,

IStorageItem, and MediaCapture. It has only two methods: createObjectURL and revokeObjectURL. (Make note that even if you have a oneTimeOnly URL for an image, it's cached, so it can be reused.)

- File Derived from Blob, has name and lastModifiedDate properties. A File in HTML5 is just a representation of a Blob that is known to be a file. Access to contents is through the HTML5 FileReader or FileReaderSync objects, with readAs* methods: readAsArrayBuffer, readAsDataURL, readAsText.
- MSBlobBuilder Used only if you need to append blobs together, with its append method, and then the getBlob method to obtain the result.

The short of it is that when you get MSStream or Blob objects from some HTML5 API (like an XmlHttpRequest with responseType of "ms-stream," as when downloading a file or video, or from the canvas' msToBlob method), you can pass those results to various WinRT APIs that accept IInputStream or IRandomAccessStream as input. To use the canvas example, if you call canvas.msToBlob, the msRandomAccessStream property of that blob can be fed directly into the Windows.Graphics.-Imaging APIs for transform or transcoding. A video stream can be similarly manipulated using the APIs in Windows.Media.Transcoding. You might also just want to write the contents of a stream to a StorageFile (especially when the backing store isn't local) or copy them to a buffer for encryption.

The aforementioned MSStreamReader object, by the way, is where you find methods to read data from an MSStream or blob. Do be aware that these methods are synchronous and will block the UI thread if you're working with large data sets. But MSStreamReader will work just fine in a web worker.

On the flip side of the WinRT/HTML5 relationship, the MSApp object in JavaScript provides methods to convert from WinRT types to HTML5 types. One such method, createStreamFromInputStream, creates an MSStream from an IInputStream, allowing to take data from a WinRT source and call URL.createObjectURL, assigning the result to something like an img.src property. Similarly, MSApp.createBlobFromRandomAccessStream creates an MSStream from an IRandomAccessStream, and MSApp.createFileFromStorageFile converts a WinRT StorageFile object into an HTML5 File object.

Let me reiterate that URL.createObjectURL, which is essential to create a URI you can assign to properties of various HTML elements, can work with both HTML objects (blobs and MSStream) and WinRT objects (IRandomAccessStreamWithContentType, IStorageItem, and MediaCapture). In some cases you'll find that you don't need to convert a WinRT stream into an MSStream or Blob explicitly—URL.createObjectURL does that automatically. This is what makes it simple to take a video preview stream and display it in a <video> element, as we'll see in Chapter 13. You just set up the MediaCapture object in WinRT, assign the result from URL.createObjectURL on that object to video.src, and call video.play() to stream the video preview directly into the element.

Using App Data APIs for State Management

Now that we've seen the nature of state-related APIs, let's see how they're used to manage for different kinds of state and any special considerations that come into play.

Transient Session State

As described before, transient session state is whatever an app saves when being suspended so that it can restore itself to that state if it's terminated by the system and later restarted. Being terminated by the system is again the only time this happens, so what you include in session state should always be scoped to giving the user the illusion that the app was running the whole time. In some cases, as described in Chapter 3, you might not in fact restore this state, especially if it's been a long time since the app was suspended and it's unlikely the user would remember where they left off. That's a decision you need to make for your own app regarding the experience you want to provide for your customers.

Session state should be saved within the AppData `localFolder` or the `localSettings` object. It should *not* be saved in a temp area because the user could run the Disk Cleanup tool while your app is suspended or terminated in which case session state might be deleted (see next section). And because this type of state is specific to a device, you would not use roaming areas for it.

Regardless of what information you save as session state, be sure that it is completely saved by the time you return from any `suspending` event handler, using deferrals of course if you need to do async work (also described in Chapter 3). Remember that you have only a few seconds to complete this task, so apps often save session state incrementally while the app is running rather than wait for the `suspending` event specifically.

As we saw in Chapter 3, it's a good idea to just maintain your session variables directly in the WinJS `sessionState` object so that it's always current with your state. WinJS then automatically saves the contents of `sessionState` in its own handler for `WinJS.Application.oncheckpoint` (a wrapper for `suspending`). It does this by calling `JSON.stringify(sessionState)` and writing the resulting text to a file called _sessionState.json within the `localFolder`.

If you need to store additional values within `sessionState` just before it's written, do that in your own `checkpoint` handler. A good example of such data is the navigation stack for page controls, which is available in `WinJS.Navigation.history`; you could also copy this data to `sessionState` within the `PageControlNavigator.navigated` method (in navigator.js as provided by the project templates). In any case, WinJS will always call its own `checkpoint` handler after yours is done to make sure any changes to `sessionState` are saved.

When an app is restarted after being terminated, WinJS also automatically loads _sessionState.json into the `sessionState` object, so everything will be there when your own activation code is run.

If you don't use the WinJS `sessionState` object and just manage your state directly (in settings containers and other files), you can save your session variables whenever you like (including within checkpoint). You then restore them directly within your activated event for previousExecutionState == terminated.

It's also a good practice to build some resilience into your handling of session state: if what gets loaded doesn't seem consistent or has some other problem, revert to default session values. Remember too that you can use the localSettings container with composite settings to guarantee that groups of values will be stored and retrieved as a unit. You might also find it helpful during development to give yourself a simple command to clear your app state in case things get fouled up, but just uninstalling your app will clear out all that as well. At the same time, it's not necessary to provide your users with a command to clear session state: if your app fails to launch after being terminated, the previousExecutionState flag will be notRunning the next time the user tries, in which case you won't attempt to restore the state.

Similarly, if the user installs an update after an app has been suspended and terminated and the app data version changes, the previousExecutionState value will be reset. If you don't change the state version for an app update, your session state can carry forward. (This is a behavioral change between Windows 8 and Windows 8.1—the former resets previousExecutionState when an app update is installed, but the latter does not.)

Sidebar: Using HTML5 sessionStorage and localStorage

If you prefer, you can use the HTML5 localStorage object for both session and other local state; its contents get persisted to the AppData localFolder. The contents of localStorage are not loaded until first accessed and are limited to 10MB per app; the WinRT and WinJS APIs, on the other hand, are limited only by the capacity of the file system.

As for the HTML5 sessionStorage object, it's not really needed when you're using page controls and maintaining your script context between app pages—your in-memory variables already do the job. However, if you're actually changing page contexts by using <a> links or setting document.location, then sessionStorage can still be useful. You can also encode information into URIs, as is commonly done with web apps.

Both sessionStorage and localStorage are also useful within iframe or webview elements running in the web context, as the WinRT APIs are not available. At the same time, you can load WinJS into a web context page (this is supported) and the WinJS.Application.local, roaming, and temp objects still work using in-memory buffers instead of the file system.

Local and Temporary State

Unlike session state that is restored only in particular circumstances, local app state is composed of those settings and other values that are *always applied* when an app is launched. Anything that the user can set directly falls into this category, unless it's also part of the roaming experience, in which case it is still loaded on app startup. Any other cached data, saved searches, display units, preferred media formats, and device-specific configurations also fall into this bucket. In short, if it's not pure session state and not part of your app's roaming experience, it's local or temporary state. (Remember that credentials should be stored in the Credential Locker instead of in your app data and that recent file lists and frequently used file lists should use `Windows.Storage.AccessCache`.)

All the same APIs we've seen also work for this form of state, including all the WinRT APIs, the `WinJS.Application.local` and `temp` objects, and HTML `localStorage`. You can also use the HTML5 IndexedDB APIs, SQLite, and the HTML AppCache—these are just other forms of local state even though they aren't necessarily stored in your AppData folders.

It's very important to version-stamp your local and temp state because it will always be preserved across an app update (unless temp state has been cleaned up in the meantime). With any app update, be prepared to load old versions of your state or migrate it with `setVersionAsync`, or simply decide that a version is too old and purge it (`Windows.Storage.ApplicationData.current.clearAsync`) before setting up new defaults. As mentioned before, it's also possible to migrate state from a background task. (See Chapter 16.)

Generally speaking, local and temp app data are the same—they have mostly the same APIs and are stored in parallel folders. Temp, however, doesn't support settings and settings containers. The other difference, again, is that the contents of your temp folder (along with the HTML5 app cache) are subject to the Windows Disk Cleanup tool when Temporary Files is selected. This means that your temp data could disappear at any time when the user wants to free up some disk space. You could also employ a background task with a maintenance trigger for doing cleanup on your own (again see Chapter 16, in "Tasks for Maintenance Triggers").

For these reasons, temp should be used for storage that optimizes your app's performance but *not* for anything that's critical to its operation. For example, if you have a JSON file in your package that you parse or decompress on first startup such that the app runs more quickly afterwards, and if you don't make any changes to that data from the app, you might elect to store that in temp. The same is true for graphical resources that you might have fine-tuned for the specific device you're running on; you can always repeat that process from the original resources, so it's another good candidate for temp data. Similarly, if you've acquired data from an online service as an optimization (that is, so that you can just update the local copy incrementally), you can always reacquire it. This is especially helpful for providing an offline experience for your app, though in some cases you might want to let the user choose to save it in local instead of temp (an option that would appear in Settings along with the ability to clear the cache).

Sidebar: HTML5 App Cache

Store apps can employ the HTML 5 app cache as part of an offline/caching strategy. It is most useful in `iframe` and webview elements (running in the web context) where it can be used for any kind of content. For example, an app that reads online books can show such content in a webview, and if those pages include app cache tags, they'll be saved and available offline. In the local context, the app cache works for nonexecutable resources like images, audio, and video but not for HTML or JavaScript.

IndexedDB, SQLite, and Other Database Options

Many forms of local state are well suited to being managed in a database. In Windows Store apps, the IndexedDB API is available through the `window.indexedDB` and `worker.indexedDB` objects. For complete details on using this feature, I'll refer you to the W3C specifications, the Indexed Database API reference for Store apps, and the IndexedDB sample. (Aaron Powell's db.js wrapper for IndexedDB might also be of interest.)

It's very important to understand that some app and systemwide limits are imposed on IndexedDB because there isn't a means through which the app or the system can shrink a database file and reclaim unused space:

- IndexedDB has a 250MB limit per app and an overall system limit of 375MB on drives smaller than 32GB, or 4% (to a maximum 20GB) for drives over 32GB. So it could be true that your app might not have much room to work with anyway, in which case you need to make sure you have a fallback mechanism. (When the limit is exceeded the APIs will throw a Quota Exceeded exception. And if you want to use a third-party tool to explore the database contents, you can find the .edb files in *%localappdata%/Microsoft/Internet Explorere/Indexed DB*.)
- IndexedDB as provided by the app host does not support complex key paths—that is, it does not presently support multiple values for a key or index (multientry).
- By default, access to IndexedDB is granted only to HTML pages that are part of the app package and those declared as content URIs. (See "App Content URIs" in Chapter 4.) Random web pages you might host in a webview will not be given access, primarily to preserve space within the 250MB limit for those pages you really care about in your app. However, you can grant access to arbitrary pages by including the following tag in your home page and not setting the webview's contents until the `DOMContentLoaded` or `activated` event has fired:

```
<meta name="ms-enable-external-database-usage" content="true" />
```

Besides IndexedDB a few other database options for Store apps exist. For a local relational database, the most popular option is SQLite, which you can install as a Visual Studio extension from the SQLite for Windows Runtime page.[82] Full documentation and other downloads can be found at http://sqlite.org, and Tim Heuer's blog on SQLite provides many details for Windows Store apps. The key thing for apps written in JavaScript is that you can't talk directly to a compiled SQLite DLL, so a little more work is necessary.

One solution that's emerged in the community is a WinRT wrapper component for the DLL called SQLite3-WinRT, available on GitHub, which provides a familiar promise-oriented async API for your

[82] It's a very robust solution—it's apparently used to operate commercial airliners like the massive Airbus A380.

database work. There is also a version called SQL.js, which is SQLite compiled to JavaScript via Emscripten. This gives you more of the straight SQLite API, but be mindful that as JavaScript it's always going to be running on the UI thread.

Another local, nonrelational database option are the Win32 "Jet" or Extensible Storage Engine (ESE) APIs (on which the IndexedDB implementation is built). For this you'll need to write a WinRT Component wrapper in C++ (the general process for which is in Chapter 18), because JavaScript cannot get to those APIs directly.

There's also LINQ for JavaScript, a project on CodePlex that allows you to use SQL-like queries against JavaScript objects. (LINQ stands for Language INtegerated Queries, a concept introduced with .NET languages that has proven very popular and convenient.) If your data is small enough to be loaded into memory from serialized JSON, this could make a suitable database for your app.

An alternate possibility for searchable file-backed data is to use the *system index* by creating a folder named "Indexed" in your local and/or roaming AppData folder. The contents of the files in this folder, including metadata (properties), will be indexed by the system indexer and can be queried using Advanced Query Syntax (AQS). (The general query APIs are explained in "Custom Queries" in Chapter 11; content indexing is covered in Chapter 15 in "Indexing and Searching Content.") You can also do property-based searches for Windows properties, making this approach a simple alternative to database solutions.

You can probably find other third-party libraries that would fulfill your needs for local data storage, perhaps just using the file system with JSON or XML. I haven't investigated any specific ones, however, so I can't make recommendations here. (If a library uses Win32 APIs under the covers, make sure that they use only those listed on Win32 and COM for Windows Store apps.)

If you're willing to work with online databases, you have a couple of additional technologies to choose from. First, Windows Azure Mobile Services makes it very easy to create and manage online tables (stored in an online SQL Server database), which you can query through simple client-side libraries. For more, see Get started with data in Mobile Services. Note that if you're planning to use push notifications in your app, a subject that we'll return to in Chapter 16, you'll likely want to use tables in Mobile Services for that purpose, so you might as well get started now.

Speaking of online SQL Server databases, you can work with them directly through the OData protocol, provided that you have configured the necessary data services on the server side (a REST interface). When that's in place, you can use the client-side OData Library for JavaScript or the datajs library to do all your work, because they handle the details of making the necessary HTTP requests.

Finally, OneDrive is an option for working with cloud-based files. We'll talk more about OneDrive in Chapter 11, but you can refer to the Live Connect Developer Center in the meantime.

Roaming State

The automatic roaming of app state between a user's devices (up to 81 of them!) is one of the most interesting and enabling features introduced for Windows Store apps. Seldom does such a small piece of technology like this so greatly reduce the burden on app developers!

It works very simply. First, your AppData `roamingFolder` and your `roamingSettings` container behave exactly like their local counterparts. As long as their combined size is less than the `roamingStorageQuota` (in `Windows.Storage.ApplicationData.current`), Windows will copy that data to the cloud (where it maintains a copy for each discrete version of your state) and then from the cloud to other devices where the same user is logged in and has the same app installed. In fact, Windows attempts to copy roaming data for an app during its installation process so that it's there when the app is first launched on that device.

If the app is running simultaneously on multiple devices, the last writer of any particular file or setting always wins. When roaming state gets synced from the latest copy on the cloud, apps will receive the `Windows.Storage.ApplicationData.ondatachanged` event. So an app should always read the appropriate roaming state on startup and refresh that state as needed within `datachanged`. You should always employ this strategy too in case Windows cannot bring down roaming state for a newly installed app right away (such as when the user installed the app and lost connectivity). As soon as the roaming state appears, you'll receive the `datachanged` event. Scenario 5 of the Application data sample provides a basic demonstration of this.

> **Tip** Roaming state is meant to keep multiple apps synchronized to state in the cloud. It is not intended as a message-passing system—that is, having one app write to a file and having the app on another device clearing out that file once it "receives" the data. If you need to pass messages, use another service of your own.

Deciding what your roaming experience looks like is really a design question more than a development question. It's a matter of taking all app settings that are not specific to the device hardware (that is, those not related to screen size, video capabilities, the presence of particular peripherals or sensors, etc.), and thinking through whether it makes sense for each setting to be roamed. A user's favorites, for example, are appropriate to roam *if* they refer to data that isn't local to the device. That is, favorite URIs or locations on a cloud storage service like OneDrive or Flickr are appropriate to roam; favorites and recently used files in a user's local libraries are not. The viewing position within a cloud-based video, like a streaming movie, would be appropriate to roam, as would be the last reading position in a magazine or book. But again, if that content is local, then maybe not. Account configurations like email settings are often good candidates so that the user doesn't have to configure the app again on other devices.

At the same time, you might not be able to predict whether the user will want to roam certain settings. In this case, the right choice is to give the user a choice! That is, include options in your Settings UI to allow the user to customize the roaming experience to their liking, especially as a user might have devices for both home and work where they want the same app to behave differently. For instance, with an RSS Reader the user might not want notifications on their work machine whenever new articles arrive but would want real-time updates at home. The set of feeds itself, on the other hand, would probably always be roamed, but then again the user might want to keep separate lists.

To minimize the size of your roaming state and stay below the quota, you might employ the `Windows.Storage.Compression` API for file-based data, as described earlier. For this same reason, never use roaming state for *user data*. Instead, use an online service like OneDrive to store user data in the cloud, and then roam URIs to those files as part of the roaming experience. Put another way, think in terms of roaming *references* to content, not content itself. Also consider putting caps on otherwise open-ended data sets (like favorites) to avoid exceeding the quota.

By now you probably have a number of other questions forming in your mind about how roaming works: "How often is data synchronized?" "How do I manage different versions?" "What else should I know?" These are good questions, and fortunately there are good answers!

- Assuming there's network connectivity, an app's roaming state is roamed within 30 minutes on an active machine. It's also roamed immediately when the user logs on or locks the machine. Locking the machine is always the best way to force a sync to the cloud. Note that if the cloud service is aware of only a single device for a user (that is, for any given a Microsoft account), synchronization with the cloud service happens only about once per day. When the service is aware that the user has multiple machines, it begins synchronizing within the 30-minute period. If the app is uninstalled on all but one machine, synchronization reverts to the longer period.

- When saving roaming state, you can write values whenever you like, such as when those settings are changed. Don't worry about grouping your changes: Windows has a built-in debounce period to combine changes together and reduce overall network traffic.

- If you have a group of settings that must be roamed together, manage these as a composite setting in your `roamingSettings` container.

- Files you create within the `roamingFolder` are not be roamed so long as you have the file open for writing (that is, as long as you have an open stream). For this reason it's a good idea to make sure that all streams are closed when the app is suspended.

- Windows allows each app to have a "high priority" setting that is roamed within one minute, thereby allowing apps on multiple devices to stay much more closely in sync. This one setting—which can be a composite setting—must exist in the root of your `roamingSettings` with the name *HighPriority*—that is, `roamingSettings.values["HighPriority"]`. That setting must also be 8K or smaller to maintain the priority. If you exceed 8K, it roams with normal priority. (And note that the setting must be a single or composite setting; a settings *container* with the same name roams with normal priority.) See scenario 6 of the Application data sample for a demonstration.

- On a trusted PC, systemwide user settings like the Start page configuration are automatically roamed independent of apps. This includes encrypted credentials saved by apps in the credential locker (if enabled in PC Settings); apps should never attempt to roam passwords. Apps that create secondary tiles (as we'll see in Chapter 16) can indicate whether such tiles should be copied to a new device when the app is installed.

- When multiple *state* versions are in use by different versions of an app, Windows manages each version of the state separately, meaning that newer state won't be roamed to devices with apps that use older state versions. In light of this, it's a good idea to not be too aggressive in versioning your state because it breaks the roaming connection between apps.
- The cloud service retains multiple versions of roaming state so long as multiple versions are in use by the same Microsoft account. Only when all instances of the app have been updated or uninstalled will older versions of the roaming state be eligible for deletion.
- When an updated app encounters an older version of roaming state, it should load it according to the old version but call `setVersionAsync` to migrate to the new version.
- Avoid using secondary versioning schemes within roaming state such that you introduce structural differences without changing the state version through `setVersionAsync`. Because the cloud service is managing the roaming state by this version number, and because the last writer always wins, a version of an app that expects to see some extra bit of data, and in fact saved it there, might find that it's been removed because a slightly older version of the app didn't write it.
- Even if all apps are uninstalled from a user's devices, the cloud service retains roaming state for "a reasonable time" (maybe 30 days) so that if a user reinstalls the app within that time period they'll find that their settings are still intact. To avoid this retention and explicitly clear roaming state from the cloud, use the `clearAsync` method.
- To debug roaming state, check out the Roaming Monitor Tool available in the Visual Studio Gallery. It provides status information on the current sync state, a Sync Now button to help with testing, and a browser for roaming state and a file editor. (At the time of writing, however, this tool is only available for Visual Studio 2012 for Windows 8 and has not been updated for Windows 8.1; it might appear directly in Visual Studio and not as an extension.)

For additional discussion, refer to Guidelines for roaming app data.

Settings Pane and UI

We've now seen all the different APIs that an app can use to manage its state where storage is concerned. The question that remains is how to present settings that a user can configure directly. That is, within the whole of an app's state, there will be some subset that you allow a user to control directly, as opposed to indirectly through other actions. Many bits of state are tracked transparently or, like a navigation history, might reflect user activity but aren't otherwise explicitly shown to or configurable by the user. Other pieces of state—like preferences, accounts, profile pictures, and so forth—can and should be exposed directly to the user. This is the purpose of the Settings charm.

When the user invokes the Settings charm (which can also be done directly with the Win+i key), Windows displays the Settings pane, a piece of UI that is populated with various settings commands as well as system functions along the bottom. Some examples are shown in Figure 10-4.

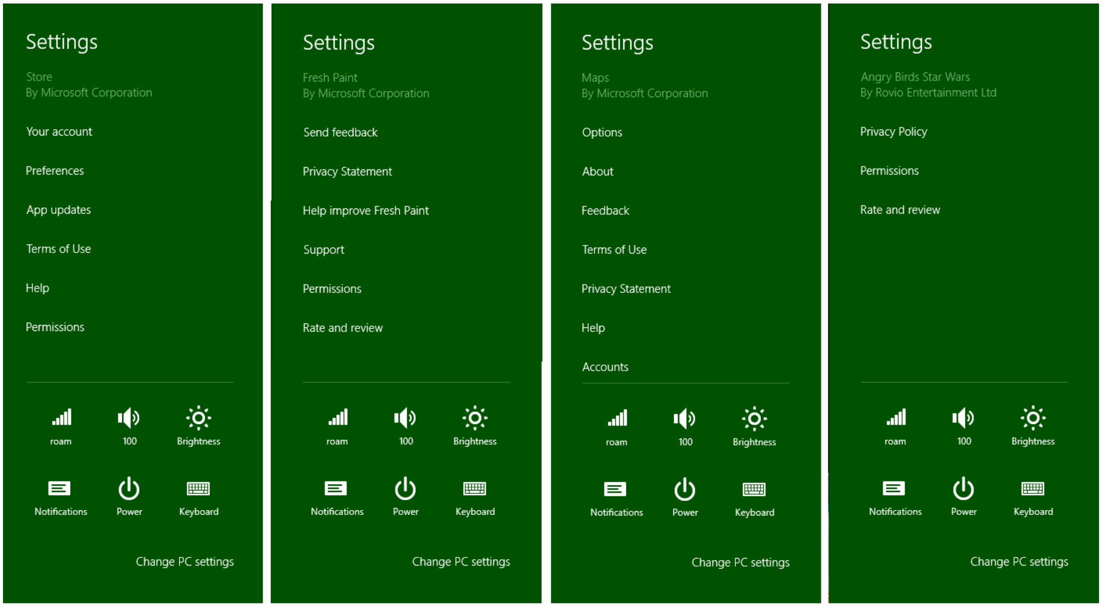


FIGURE 10-4 Examples of commands on the top-level settings pane. Notice that the lower section of the pane always has system settings and the app name and publisher always appear at the top. Permissions and Rate And Review are added automatically for apps acquired from the Store. Rate And Review is not included for side-loaded apps (nor the Store app itself).

What appears in the Settings charm for an app should be those settings that affect behavior of the app as a whole and are adjusted only occasionally, or commands like Feedback and Support that simply navigate to a website. Options that apply only to particular pages or workflows should not appear in Settings: place them directly on the page (the app canvas) or in the app bar. Of course, such options are still part of your app state—just not part of the Settings charm UI!

Details and options that typically appear in the Settings charm include the following:

- Display preferences like units, color themes, alignment grids, and defaults.
- Roaming preferences that allow the user to control exactly what settings get roamed, such as to keep configurations for personal and work machines separate.
- Account and profile configurations, along with commands to log in, log out, and otherwise manage those accounts and profiles. Passwords should never be stored directly or roamed, however; use the Credential Locker instead.
- Behavioral settings like online/offline mode, auto-refresh, refresh intervals, preferred video/audio streaming quality, whether to transfer data over metered network connections, the

location from which the app should draw data, and so forth.

- A feedback link where you can gather specific information from the user, or you can use a feedback panel that records data to telemetry you collect and upload separately.
- Additional information about the app, such as Help, About, a copyright page, a privacy statement, license agreements, and terms of use. Oftentimes these commands will take the user to a separate website, which is perfectly fine.

I highly recommend that you run the apps that are built into Windows and explore their use of the Settings charm. You're welcome to explore how Settings are used by other apps in the Store as well, but those might not always follows the design guidelines as consistently—and consistency is essential to settings!

Speaking of which, apps can add their own commands to the Setting pane but they are not obligated to do so. Windows guarantees that something always shows up for the app in this pane: it automatically displays the app name and publisher, a Rate And Review command that takes you to the Windows Store page for the app, an Update command if an update is available from the Store (and auto-update is turned off), and a Permissions command if the app has declared any capabilities in its manifest that are subject to user consent (such as Location, Camera, Microphone, etc.). Note that Rate And Review and Update won't appear for apps you run from Visual Studio or for side-loaded apps, because they weren't acquired from the Store.

One of the beauties of the Settings charm is that it appears as a flyout on top of the app, meaning you never need to incorporate settings pages into your app's navigation hierarchy. Furthermore, the Settings charm is always available no matter where you are in the app, so you don't need to think about having such a command on your app bar, nor do you ever need a general settings command on your app canvas. That said, you can invoke the Settings charm programmatically, such as when you detect that a certain capability is turned off and you prompt the user about that condition. You might ask something like "Do you want to turn on geolocation for this app?" and if the user says Yes, you can invoke the Settings charm. This is done through the settings pane object returned from `Windows.UI.ApplicationSettings.SettingPane.getForCurrentView`, whose `show` method displays the UI (or throws a kindly exception if the app doesn't have the focus). The `edge` property of the settings pane object also tells you if it's on the left or right side of the screen, depending on the left-to-right or right-to-left orientation of the system as a whole (a regional variance).

And with that we've covered all the methods and properties of this object! Yet the most interesting part is how we add our own commands to the settings pane. But let's first look at a few design considerations as described on Guidelines for app settings.

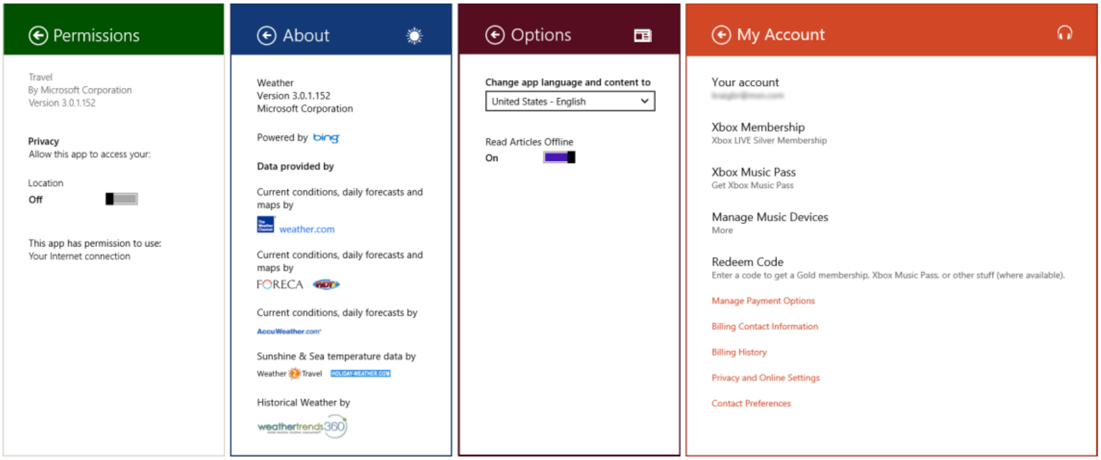
Design Guidelines for Settings

Beyond the commands that Windows automatically adds to the settings pane, an app can provide up to eight others, typically around four; anything more than eight will throw an exception. Because settings are global to an app, the commands you add are always the same: they are not sensitive to context. To say it another way, the *only* commands that should appear on the settings pane are those that are global to the app (refer back to Figure 10-4 for examples); commands that apply only to certain pages or contexts within a page should appear on the app bar or app canvas.

Each app-supplied command can do one of two things. First, a command can simply be a hyperlink to a web page. Some apps use links for their Help, Privacy Statement, Terms of Use, License Agreements, and so on, which will open the linked pages in a browser. The other option is to have the command invoke a secondary flyout panel with more specific settings controls or simply a webview to display web-based content. You can provide Help, Terms of Use, and other textual content in both these ways rather than switch to the browser.

> **Note** As stated in the App certification requirements, section 4.1.1, apps that collect personal information in any way, or even just use a network, must have a privacy policy or statement. This must be included on the app's product description page in the Store and must also be accessible through a command in your Settings pane.

Secondary flyouts are created with the `WinJS.UI.SettingsFlyout` control; some examples are shown in Figure 10-5. Notice that the secondary settings panes can be sized however needed, but they should fall between 346px and 646px. You should style the flyout header (using the `win-header` class) to use your app's primary color for the background and style the entire flyout with a border color that's 20% darker. Also note that the Permissions flyout, shown on the left of Figure 10-5, is provided by Windows automatically, is configured according to capabilities declared in your manifest, and uses the system colors to specifically differentiate system settings. Some capabilities like geolocation are controlled in this pane; other capabilities like Internet and library access are simply listed because the user is not allowed to turn them on or off.



Permissions

Travel
By Microsoft Corporation
Version 3.0.1.152

Privacy
Allow this app to access your:

Location
Off

This app has permission to use:
Your Internet connection

About

Weather
Version 3.0.1.152
Microsoft Corporation

Powered by bing

Data provided by

Current conditions, daily forecasts and maps by
weather.com

Current conditions, daily forecasts and maps by
FORECA

Current conditions, daily forecasts by
AccuWeather.com

Sunshine & Sea temperature data by
Weather 2 Travel HOLIDAY-WEATHER.COM

Historical Weather by
weathertrends360

Options

Change app language and content to
United States - English

Read Articles Offline
On

My Account

Your account
[redacted]

Xbox Membership
Xbox LIVE Silver Membership

Xbox Music Pass
Get Xbox Music Pass

Manage Music Devices
More

Redeem Code
Enter a code to get a Gold membership, Xbox Music Pass, or other stuff (where available).

Manage Payment Options

Billing Contact Information

Billing History

Privacy and Online Settings

Contact Preferences

FIGURE 10-5 Examples of secondary settings panes in the Travel, Weather, News, and Music apps. The first three are 346px wide; the fourth is 646px. Notice that each app-provided pane is appropriately branded and provides a back button to return to the main Settings pane. The Permissions pane is provided by the system and thus reflects the system theme (that is, it cannot be customized).

A common group of settings are those that allow the user to configure their roaming experience—a group of settings that determine what state is roamed (you see this on PC Settings > OneDrive > Sync Settings). It is also recommended that you include account/profile management commands within Settings, as well as login/logout functionality. As noted in Chapter 9, "Commanding UI," logins and license agreements that are necessary to run the app at all should be shown upon launch. For ongoing login-related functions, and to review license agreements and such, create the necessary commands and panes within Settings. Refer to Guidelines for login for more information on this subject. Guidelines for a Help command can also be found on Quickstart: add app help.

Behaviorally, settings panes are light-dismiss (returning to the app) and have a back button to return to the primary settings pane with all the commands. Because of the light-dismiss behavior, changing a setting on a pane applies the setting immediately: there is **no** OK or Apply button or other such UI. If the user wants to revert a change, she should just restore the original setting.

For this reason it's a good idea to use simple controls that are easy to switch back, rather than complex sets of controls that would be difficult to undo. The recommendation is to use toggle switches for on/off values (rather than check boxes), a button to apply an action (but without closing the settings UI), hyperlinks (to open the browser), text input boxes (which should be set to the appropriate type such as email address, password, etc.), radio buttons for groups of up to five mutually exclusive items, and a listbox (`select`) control for four to six text-only items.

In all your settings, think in terms of "less is more." Avoid having all kinds of different settings, because if the user is never going to find them, you probably don't need to surface them in the first place! Also, while a settings pane can scroll vertically, try to limit the overall size such that the user has to pan down only once or twice, if at all (that is, three pages on a 768px vertical display).

Some other things to avoid with Settings:

- Don't use Settings for workflow-related commands. Those belong on the app bar or on the app canvas, as discussed in Chapter 9.
- Don't use a top-level command in the Settings pane to perform an action other than linking to another app (like the browser). Top-level commands should never execute an action *within* the app.
- Don't use settings commands to navigate within the app.
- Don't use `WinJS.UI.SettingsFlyout` as a general-purpose control.

And on that note, let's now look at the steps to use Settings and the `SettingsFlyout` properly!

Populating Commands

The first part of working with Settings is to provide your specific commands when the Settings charm is invoked. Unlike app bar commands, these should always be the same no matter the state of the app; if you have context-sensitive settings, place commands for those in the app bar.

The two ways to implement this process in an app written in HTML and JavaScript are using WinRT directly or using the helpers in WinJS. Let's look at these for a simple Help command.

To know when the charm is invoked through WinRT, obtain the settings pane object through Windows.UI.ApplicationSettings.SettingsPane.getForCurrentView and add a listener for its commandsrequested event (this is a WinRT event, so be sure to remove the listener if necessary):

```
// The n variable here is a convenient shorthand
var n = Windows.UI.ApplicationSettings;
var settingsPane = n.SettingsPane.getForCurrentView();
settingsPane.addEventListener("commandsrequested", onCommandsRequested);
```

Within your event handler, create SettingsCommand objects for each command, where each command has an id, a label, and an invoked function that's called when the command is tapped or clicked. These can all be specified in the constructor:

```
function onCommandsRequested(e) {
    // n is still the shortcut variable to Windows.UI.ApplicationSettings
    var commandHelp = new n.SettingsCommand("help", "Help", helpCommandInvoked);
    e.request.applicationCommands.append(commandHelp);
}
```

A command is added to the Settings pane by adding it to the e.request.applicationCommands vector; above we use the vector's append method, but you could also use insertAt. You'd make such a call for each command, or you can pass an array of such commands to the vector's replaceAll method. What then happens within the invoked handler for each command is the interesting part, and we'll come back to that soon.

You can also prepopulate the applicationCommands vector outside of the commandsrequested event; this is perfectly fine because your settings commands should be constant for the app anyway. Here's an example, which also shows the vector's replaceAll method:

```
var n = Windows.UI.ApplicationSettings;
var settingsPane = n.SettingsPane.getForCurrentView();
var vector = settingsPane.applicationCommands;

//Ensure no settings commands are currently specified in the settings charm
vector.clear();

var commands = [ new settingsSample.SettingsCommand("Custom.Help", "Help", OnHelp),
              new n.SettingsCommand("Custom.Parameters", "Parameters", OnParameters)];
vector.replaceAll(commands);
```

This way, you don't actually need to register for or handle commandsrequested directly.

Now because most apps will likely use settings in some capacity and will typically employ flyouts for each command, WinJS provides some shortcuts to this whole process. First, instead of listening for the WinRT event, simply assign a handler to WinJS.Application.onsettings (which is a wrapper for commandsrequested):

```
WinJS.Application.onsettings = function (e) {
    // ...
};
```

In your handler, create a JSON object describing your commands and store that object in `e.detail.applicationcommands`. Mind you, this is *different* from the WinRT object—just setting this property accomplishes nothing. What comes next is passing the now-modified event object to the static `WinJS.UI.SettingsFlyout.populateSettings` method as follows (taken from scenario 2 of the App settings sample, js/2-AddFlyoutToCharm.js):

```
WinJS.Application.onsettings = function (e) {
    e.detail.applicationcommands =
        { "help": { title: "Help", href: "/html/2-SettingsFlyout-Help.html" } };
    WinJS.UI.SettingsFlyout.populateSettings(e);
};
```

The `populateSettings` method traverses the `e.details.applicationcommands` object and calls the WinRT `applicationCommands.append` method for each item. This gives you a more compact means to accomplish what you'd do with WinRT, and it also simplifies the implementation of setti ngs commands, as we'll see in a moment.

Tip The `populateSettings` method is just a helper function and isn't anything you're required to use. You can easily see its implementation in the WinJS ui.js file and even make a copy of the code to customize it however you like.

As you can see above, the JSON in `e.detail.applicationcommands` has this format:

{ <command id>: { title: <command label>, href: <path to in-package flyout markup> }}

The `href` property here must refer to an **in-package** HTML file that describes the content of the `SettingsFlyout` that WinJS will invoke for that command. That is, you cannot use `href` to specify an arbitrary URI to launch a browser, as commonly employed for Terms of Service, Privacy Statement, and other such commands.

To intermix external URI commands with flyouts, you need to use a combination of both the WinJS and WinRT APIs (or you can just bring the external content into a flyout with a webview). Fortunately, within `WinJS.Application.onsettings`, the event args for the original WinRT `commandsrequested` event is available in the `e.detail.e` property. That is, within the WinJS event, `e.detail.e.request.applicationCommands` is the WinRT vector. Thus, you can call `WinJS.UI.SettingsFlyout.populateSettings` for those commands that use flyouts and then create and add other commands through `e.detail.e.request.applicationCommands.append` or `insertAt`. You'd use `insertAt`, clearly, if you want to place a command at a specific point in the list rather than add it to the end.

Caveat You can call `populateSettings` only once, because WinJS internally stores the list of commands in another internal object. Any subsequent call with a different list of commands will cause any previous commands to be visible but unresponsive. In short, don't do it.

Implementing Commands: Links and Settings Flyouts

Technically speaking, you can do anything you want within the invoked function for a settings command. Truly! Of course, as described in the design guidelines earlier, recommendations exist for how to use settings and how not to use them. For example, settings commands shouldn't act like app bar commands that affect content, nor should they navigate within the app itself. Ideally, a settings command does one of two things: launch a hyperlink (to open a browser) or display a secondary settings pane.

In the first case, launching a hyperlink uses the Windows.System.Launcher.launchUriAsync API as follows:

```
function onCommandsRequested(e) {
    // n is still the shortcut variable to Windows.UI.ApplicationSettings
    var commandHelp = new n.SettingsCommand("help", "Help", helpCommandInvoked);
    e.request.applicationCommands.append(commandHelp);
}

function helpCommandInvoked(e) {
    var uri = new Windows.Foundation.Uri("http://example.domain.com/help.html");
    Windows.System.Launcher.launchUriAsync(uri).done();
}
```

In the second case, settings panes are implemented with the WinJS.UI.SettingsFlyout control. Again, technically speaking, you're not required to use this control: you can display any UI you want within the invoked handler. The SettingsFlyout control, however, supplies enter and exit animations and fires events like [before | after][show | hide][83]. And because the flyout automatically handles vertical scrolling with any HTML you place within it, including other controls (almost), there's no reason *not* to use it.

Note The one limitation is that nesting flyouts is not presently supported in WinJS, so you cannot have a secondary flyout appear in the context of a settings flyout.

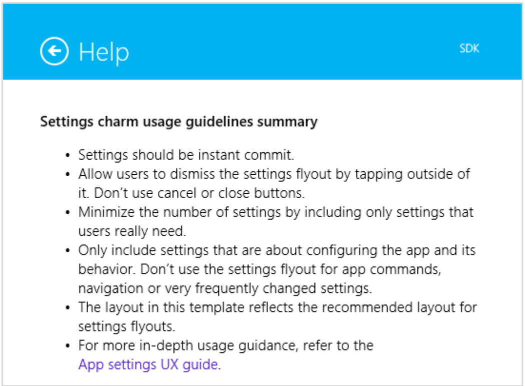
Because it's a WinJS control, you can declare a SettingsFlyout for each one of your commands in markup (making sure WinJS.UI.process/processAll is called, which handles any other controls in the flyout). For example, scenario 2 of the App settings sample defines the following flyout for its Help command (html/2-SettingsFlyout-Help.html, omitting the text content and reformatting a bit); the result of this is shown in Figure 10-6:

```
<div data-win-control="WinJS.UI.SettingsFlyout" id="helpSettingsFlyout"
    aria-label="Help settings flyout" data-win-options="{settingsCommandId:'help'}">
    <!-- Use either 'win-ui-light' or 'win-ui-dark' depending on the contrast between the
        header title and background color; background color reflects app's personality -->
    <div class="win-ui-dark win-header" style="background-color:#00b2f0">
```

[83] How's that for a terse combination of four event names? It's also worth noting that the document.body.DOMNodeInserted event will also fire when a flyout appears.

```
        <button type="button" id="backButton" class="win-backbutton"></button>
        <div class="win-label">Help</div>
        <img src="../images/smallTile-sdk.png" style="position: absolute; right: 40px;"/>
    </div>
    <div class="win-content ">
        <div class="win-settings-section">
            <h3>Settings charm usage guidelines summary</h3>
            <!-- Other content omitted -->
            <li>For more in-depth usage guidance, refer to the
                <a href="http://msdn.microsoft.com/en-us/library/windows/apps/hh770544">
                App settings UX guide</a>.</li>
        </div>
    </div>
</div>
```

As always, the `SettingsFlyout` control has options (just one, `settingsCommandId` for obvious purpose) as well as a few applicable `win-*` style classes: `win-settingsflyout`, which styles the whole control, most especially for width and your border color, and `win-ui-light` and `win-ui-dark`, which apply a light or dark theme to the contents of the flyout. In this example, we use the dark theme for the header while the rest of the flyout uses the default light theme, which is inherited from the app's global stylesheet (that is, default.html pulls in ui-light.css).



Settings charm usage guidelines summary

- Settings should be instant commit.
- Allow users to dismiss the settings flyout by tapping outside of it. Don't use cancel or close buttons.
- Minimize the number of settings by including only settings that users really need.
- Only include settings that are about configuring the app and its behavior. Don't use the settings flyout for app commands, navigation or very frequently changed settings.
- The layout in this template reflects the recommended layout for settings flyouts.
- For more in-depth usage guidance, refer to the App settings UX guide.

FIGURE 10-6 The Help settings flyout (truncated vertically) from scenario 2 of the App settings sample. Notice the hyperlink on the bottom.

In any case, you can see that everything within the control is just markup for the flyout contents, nothing more, and you can wire up events to controls in the markup or in code. You're free to use hyperlinks here, such as to launch the browser to open a fuller Help page. You can also use a webview to host web content within a settings flyout; for an example, see the Here My Am! example in this chapter's companion content, specifically html/privacy.html.

> **Tip** If you load any stylesheets in settings flyouts, they will be added into the global stylesheet and can possibly affect the rest of the app. Make sure, then, to avoid conflicts by scoping your selectors specifically to elements inside the flyout.

So, how do we get a flyout to show when a command is invoked on the top-level settings pane? The easy way is to let WinJS take care of the details using the information you provide to WinJS.UI.SettingsFlyout.populateSettings. When you specify a reference to the flyout's markup, like we saw earlier (from js/2-AddFlyoutToCharm.js):

```
WinJS.Application.onsettings = function (e) {
    e.detail.applicationcommands =
        { "help": { title: "Help", href: "/html/2-SettingsFlyout-Help.html" } };
    WinJS.UI.SettingsFlyout.populateSettings(e);
};
```

then WinJS will automatically invoke a flyout with that markup when the command is invoked, calling WinJS.UI.processAll along the way. This is why in most of the scenarios of the sample you don't see any explicit calls to showSettings, just a call to populateSettings. But you can use showSettings to programmatically invoke a flyout, as we'll now see.

Programmatically Invoking Settings Flyouts

In addition to being a control that you use to define a specific flyout, WinJS.UI.SettingsFlyout has a couple of other static methods beyond populateSettings: show and showSettings. The show method specifically brings out the top-level Windows settings pane—that is, Windows.UI.ApplicationSettings.SettingsPane. A call to show is what you typically wire up to a flyout's back button so that you return to the main Settings pane.

> **Note** Although it's possible to programmatically invoke your own settings panes, you cannot do so with the system-provided Permissions command. If you have a condition for which you need the user to change a permission, such as enabling geolocation, the recommendation is to display an error message that instructs the user to use the Permissions command (as Here My Am! in this chapter does) and that perhaps opens the main Setting pane. You cannot invoke the Rate And Review settings command either, but you can launch the ms-windows-store:REVIEW?PFN=<package_family_name> URI to achieve that end. See "Connecting Your Website and Web-Mapped Search Results" in Chapter 20, "Apps for Everyone, Part 2."

The showSettings method, for its part, shows a *specific* settings flyout that you define in your app. The signature of the method is showSettings(<id> [, <page>]) where <id> identifies the flyout you're looking for and the optional <page> parameter identifies an HTML document to look in if a flyout with <id> isn't found in the current document. That is, showSettings always starts by looking in the current document for a SettingsFlyout element that has a matching settingsCommandId property or a matching HTML id attribute. If such a flyout is found, that UI is shown.

If the markup in the previous section (with Figure 10-7) was contained in the same HTML page that's currently loaded in the app, the following line of code will show that flyout:

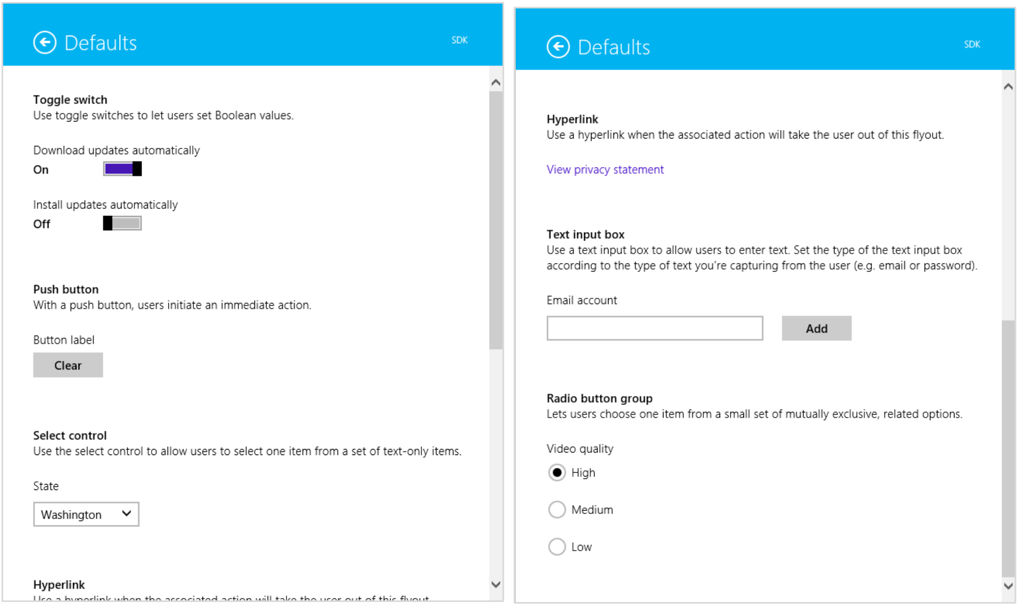
```
WinJS.UI.SettingsFlyout.showSettings("help");
```

In this case you could also omit the href part of the JSON object passed to populateCommands, but only again if the flyout is contained within the current HTML document already.

It usually makes more sense to separate your settings flyouts from the rest of your markup and then use the page parameter to showSettings, passing a URI for a page in your app package. The App settings sample uses this to place the flyout for each scenario into a separate HTML file. You can also place all your flyouts in one HTML file, so long as they have unique ids. Either way, showSettings loads the flyout's HTML into the current page using WinJS.UI.Pages.load (which calls WinJS.UI.processAll), scans that DOM tree for a matching flyout with the given <id>, and shows it. Failure to locate the flyout will throw an exception.

Scenario 4 of the [App settings sample](#) shows this form of programmatic invocation. This is also a good example (see Figure 10-7) of a vertically scrolling flyout (js/4-ProgrammaticInvocation.js):

```
WinJS.UI.SettingsFlyout.showSettings("defaults", "/html/4 -SettingsFlyout-Settings.html");
```



Defaults

Toggle switch
Use toggle switches to let users set Boolean values.

Download updates automatically
On

Install updates automatically
Off

Push button
With a push button, users initiate an immediate action.

Button label
Clear

Select control
Use the select control to allow users to select one item from a set of text-only items.

State
Washington

Hyperlink
Use a hyperlink when the associated action will take the user out of this flyout.

Defaults

Hyperlink
Use a hyperlink when the associated action will take the user out of this flyout.

View privacy statement

Text input box
Use a text input box to allow users to enter text. Set the type of the text input box according to the type of text you're capturing from the user (e.g. email or password).

Email account
Add

Radio button group
Lets users choose one item from a small set of mutually exclusive, related options.

Video quality
● High
○ Medium
○ Low

FIGURE 10-7 The settings flyout from scenario 4 of the App settings sample, showing how a flyout supports vertical scrolling; note the scrollbar positions for the top portion (left) and the bottom portion (right).

A call to showSettings is thus exactly what you use within any particular command's invoked handler; it's what WinJS sets up within populateCommands. But it also means you can call showSettings from anywhere else in your code when you want to display a particular settings pane. For example, if you encounter an error condition in the app that could be rectified by changing an app setting, you can provide a button in a message dialog or notification flyout that calls showSettings to open that particular pane. And for what it's worth, the hide method of that flyout will dismiss it; it doesn't affect the top-level settings pane for which you must use Windows.UI.Application-

Settings.SettingsPane.getForCurrentView().hide.

You might use showSettings and hide together, in fact, if you need to navigate to a third-level settings pane. One of your own settings flyouts could contain a command that calls hide on the current flyout and then calls showSettings to invoke another. The back button of that subsidiary flyout (and it should always have a back button) would similarly call hide on the current flyout and showSettings to make its second-level parent reappear. That said, we don't recommend making your settings so complex that third-level flyouts are necessary, but the capability is there if you have a particular scenario that demands it.

Knowing how showSettings tries to find a flyout is also helpful if you want to create a SettingsFlyout programmatically. So long as such a control is in the DOM when you call showSettings with its id, WinJS will be able to find it and display it like any other. It would also work, though I haven't tried this and it's not in the sample, to use a kind of hybrid approach. Because showSettings loads the HTML page you specify as a page control with WinJS.UI.Pages.load, that page can also include its own script wherein you define a page control object with methods like processed and ready. Within those methods you could then make specific customizations to the settings flyout defined in the markup.

Sidebar: Changes to Permissions

A common question is whether an app can receive events when the user changes settings within the Permissions pane. The answer is no, which means that you discover whether access is disallowed only by handling Access Denied exceptions when you try to use the capability. To be fair, though, you always have to handle denial of a capability gracefully because the user can always deny access the first time you use the API. When that happens, you again display a message about the disabled permission (as shown with the Here My Am! app) and provide some UI to reattempt the operation. But the user still needs to invoke the Permissions settings manually. Refer to the Guidelines for devices that access personal data for more details, specifically in the "What if access to a device is turned off?" section.

Here My Am! Update

To bring together some of the topics we've covered in this chapter, the companion content includes another revision of the Here My Am! app with the following changes and additions (mostly to pages/home/home.js unless noted):

- It now incorporates the Bing Maps SDK so that the control is part of the package rather than loaded from a remote source. This eliminates the webview we've been using to host the map, so all the core code from html/map.html can move into js/default.js and we can eliminate the code needed to communicate between the app and the webview. Note that to run this sample in Visual Studio you need to download and install the SDK yourself (be sure to choose the version for Windows 8.1).

- Instead of copying pictures taken with the camera to app data, those are now copied to a HereMyAm folder in the Pictures library. The *Pictures Library* capability has been declared.
- Instead of saving a pathname to the last captured image file, which is used when the app is terminated and restarted, the `StorageFile` is saved in `Windows.Storage.AccessCache` to guarantee future programmatic access.
- An added appbar command allows you to use the File Picker to select an image to load instead of relying solely on the camera. This also allows you to use a camera app, if desired. Note that we use a particular `settingsIdentifier` with the picker in this case to distinguish from the picker for recent images. We'll again learn about the file pickers in Chapter 11.
- Another appbar command allows you to choose from recent pictures from the camera. This defaults initially to the Pictures library but uses a different `settingsIdentifier` so that subsequent invocations will default to the last viewed location.
- Additional commands for About, Help, and a Privacy Statement are included on the Settings pane using the `WinJS.Application.onsettings` event (see js/default.js). The first two display content from within the app whereas the third pulls down web content in a webview; all the settings pages are found in the html folder of the project, with styles in css/default.css.

What We've Just Learned

- Statefulness is important to Windows Store apps, to maintain a sense of continuity between sessions even if the app is suspended and terminated.
- App data is session, local, temporary, and roaming state that is tied to the existence of an app; it is accessible only by that app.
- User data is stored in locations other than app data (such as the user's music, pictures, and videos libraries, along with removable storage) and persists independent of any given app. Multiple apps might be able to open and manipulate user files.
- The `StorageFolder` and `StorageFile` classes in WinRT are the core objects for working with folders and files. All programmatic access to the file system begins, in fact, with a `StorageFolder`. The `Windows.Storage.FileIO` and `PathIO` classes simplify file access, as do helpers in `WinJS.Application`.
- WinRT offers encryption services through `Windows.Security.Cryptography`, as well as a built-in compression mechanism in `Windows.Storage.Compression`.
- Streams are the objects through which you general access file contents. Blobs and buffers interact with streams to handle different interchange needs between WinRT and the app host.

- App data is accessed through the `Windows.Storage.ApplicationData` API and accommodates both structured settings containers as well as file-based data. Additional APIs like IndexedDB and HTML5 `localStorage` are also available. Third-party libraries, such as SQLite and the OData Library for JavaScript, provide other options.
- It is important to version app state, especially where roaming is concerned, because versioning is how the roaming service manages what app state gets roamed to which devices based on what version apps are looking for.
- The size of roaming state is limited to a quota (provided by an API), otherwise Windows will not roam the data. Services like OneDrive can be used to roam larger files, including user data.
- The typical roaming period is 30 minutes or less. A single setting or composite named "HighPriority," so long as it's under 8K, will be roamed within a minute.
- To use the Settings pane, an app populates the top-level pane provided by Windows with specific commands. Those commands map to handlers that either open a hyperlink (in a browser) or display a settings flyout using the `WinJS.UI.SettingsFlyout` control. Those flyouts can contain any HTML desired, including webview elements that load remote content.
- Settings panes can be invoked programmatically when needed.

Chapter 11

The Story of State, Part 2: User Data, Files, and OneDrive

Every week I receive an email advertisement from a well-known electronics retailer (I did opt in) that typically highlights items across a variety of categories, like PCs, tablets, TVs, audio equipment, external hard drives, software, home security, and so forth. I've found it interesting over the couple of years I've been receiving these emails to observe how large a hard drive (or now SSD) you can get for around US$80. This is easy as the retailer seems to highlight items around that price point. I've watched how the same US$80 that bought about 320 gigabytes of storage two years ago will now acquire on the order of 2 terabytes or more (and may increase yet further by the time you read this).

What we call *user data,* a term we defined in Chapter 10, "The Story of State, Part 1," is the driving force behind the ever-growing need for storage. (Just hand a video camera to a six-year-old and—presto!—you have another gig of video files that you just can't bring yourself to delete.) The vast majority of what populates storage devices nowadays is all the stuff that we'll likely haul around with us across app changes, operating system changes, and device changes—or simply keep it away from all such concerns in the cloud. By its nature, user data is generally independent from the apps that create it. Any number of apps can manipulate that data (and associate themselves with the file types in question), and those apps can come and go while the data remains.

At the same time, the more user data expands the more we need great apps to efficiently manage it and present it in meaningful ways. There are many creative ways to present and interact with a user's pictures, videos, and music, regardless of where those files are stored. The same is true for all other types of data (like designs, drawings, and other documents), because you can easily query a folder to retrieve those files that match all sorts of different criteria, thereby helping the user sort through all their data more easily.

One of the key characteristics of the Windows platform is that the "file system" as it's presented to the user isn't merely a local phenomenon: it seamlessly integrates local, removable, network-based, and cloud-based locations, and it can even represent file-like or folder-like entities that other apps generate dynamically. The same is true for apps: the WinRT `StorageFolder` and `StorageFile` classes, which we met in Chapter 10 and will explore fully here, insulate you from the details of where such entities are physically located, how they are referenced, and how they are accessed. Two of the most important properties they provide are an *availability* flag and *thumbnail* representations, which are the fundamental building blocks of most types of browsing UI.

Speaking of cloud-based storage, we'll get to know OneDrive (formerly SkyDrive) much more in this chapter. Users with a Microsoft account get cloud storage on OneDrive for free, and the service is deeply integrated into the fiber of Windows as a whole. OneDrive is what hosts all the user's roaming data (from both apps and the system). Users also automatically see a OneDrive folder on their local system that is transparently synchronized with their cloud storage, and they can elect to maintain offline copies of whatever files and folders they choose (hence the matter of availability).

And speaking of properties, we'll see how the `StorageFile` class makes all sorts of additional properties and metadata available for whatever files you're working with and how the `StorageFolder` class makes it possible to query against that metadata.

But let's not get ahead of ourselves by rushing into details any more than you need to rush out with your US$80 to buy more storage. Let's instead take a step back and see how the different aspects of files, folders, and user data relate.

The Big Picture of User Data

To look at the broad scope of user data, we can ask a few questions of it similar to those we asked of app data in Chapter 10:

- Where does user data live?
- How does an app get to user data?
- What affects and modifies user data?
- How do apps associate themselves with specific user data formats?

Let's be clear again that user data, by definition, has no dependency on the existence of particular apps and is never part of any app's state. *Lists* of files and folders can certainly part of such state, such as recently used files, favorite folder locations, and so forth, but the data in those files is not app state. (Apps use the `Windows.Storage.AccessCache` to maintain such lists, because it preserves programmatic access permissions for `StorageFolder` and `StorageFile` objects across app sessions. We'll see the details later in this chapter.)

This makes it easy to answer the first question: from an app's point of view—which is what we care about in a book about building apps!—user data lives *anywhere and everywhere* outside your app data folders and your package folder. That outside realm again stretches from other parts of the local file system all the way out to the cloud, as illustrated in Figure 11-1.

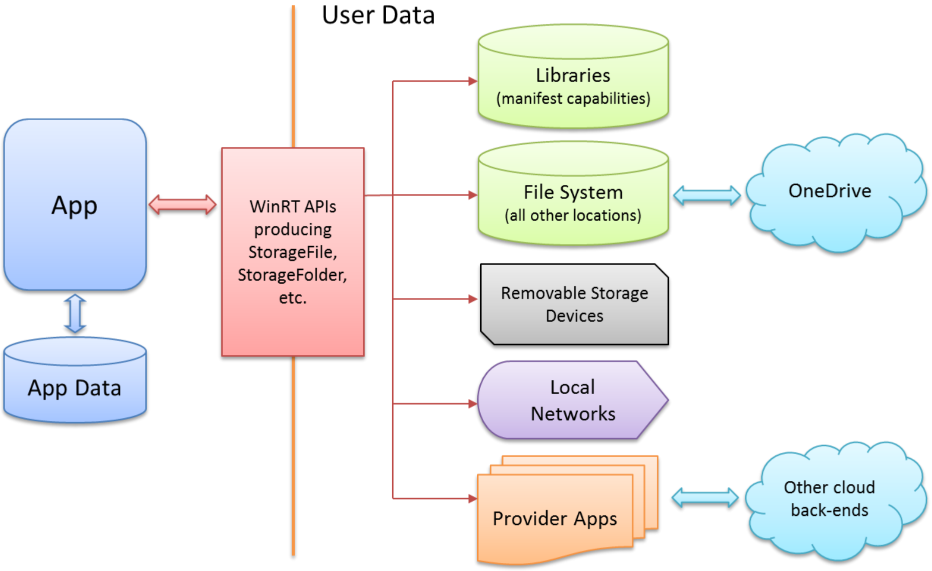

User Data
App
App Data
WinRT APIs producing StorageFile, StorageFolder, etc.
Libraries (manifest capabilities)
File System (all other locations)
OneDrive
Removable Storage Devices
Local Networks
Provider Apps
Other cloud back-ends

FIGURE 11-1 User data lives anywhere outside the app's package and app data folders. Access to user data locations happens through APIs that produce `StorageFile`, `StorageFolder`, and related objects, which represent files and folders regardless of location. Access to some locations like libraries, local networks, and removable storage are determined through manifest capabilities; the rest are typically accessed through the File Picker API. OneDrive is integrated as part of the local file system, and other apps can provide access to other cloud back-ends.

Windows makes it easy—seamless, really—for users to navigate across all these locations to select the files and folders they care about. This includes items served up by other provider apps (the Sound Recorder is an example), as well as built-in integration with OneDrive.

The right way to think about OneDrive integration is that it's simply a folder on the local file system with automatic synchronization with the cloud that's also aware of considerations like connection cost on metered networks. You use it like a local folder, meaning that you can programmatically enumerate the contents of OneDrive folders and get file metadata like thumbnails through the `StorageFile` object. All this works because Windows automatically maintains local placeholder or "smart" files that contain the metadata but not the file contents. The `StorageFile.isAvailable` property tells you whether a file's contents exist locally, which is helpful for visually distinguishing which files in a gallery view are available offline (a local copy exists) and which are online-only.

Regardless of availability, you can still attempt to open a file and read its contents. Windows will automatically download a local copy of the file as part of the process (if it has connectivity, of course, and if current cost policy on a metered network allows it). This doesn't complicate the programming model, mind you, because you always have to handle errors even for local files. Anyway, the bottom line is that it's super-easy to work with OneDrive in an app—other than using availability to style items in your UI, you work with the `StorageFolder` and `StorageFile` APIs as you would with any other location. Again, those objects insulate you from having to worry about the underlying details of whatever is providing the item in question—truly convenient!

Using OneDrive directly Although OneDrive is directly integrated with Windows, you can always use the service directly through its REST API (as you would do with other cloud storage providers). Details can be found on the OneDrive reference, on the OneDrive core concepts (which includes a list of supported file types), and in the PhotoSky sample. A backgrounder on this and other Windows Live services can also be found on the Building Windows 8 blog post entitled Extending "Windows 8" apps to the cloud with SkyDrive.

This brings us to our second question: how does an app get to an arbitrary user data location in the first place? That is, how does it acquire a `StorageFolder` or `StorageFile` object for stuff outside of its app data locations? Where user data is concerned, this happens in only three ways:

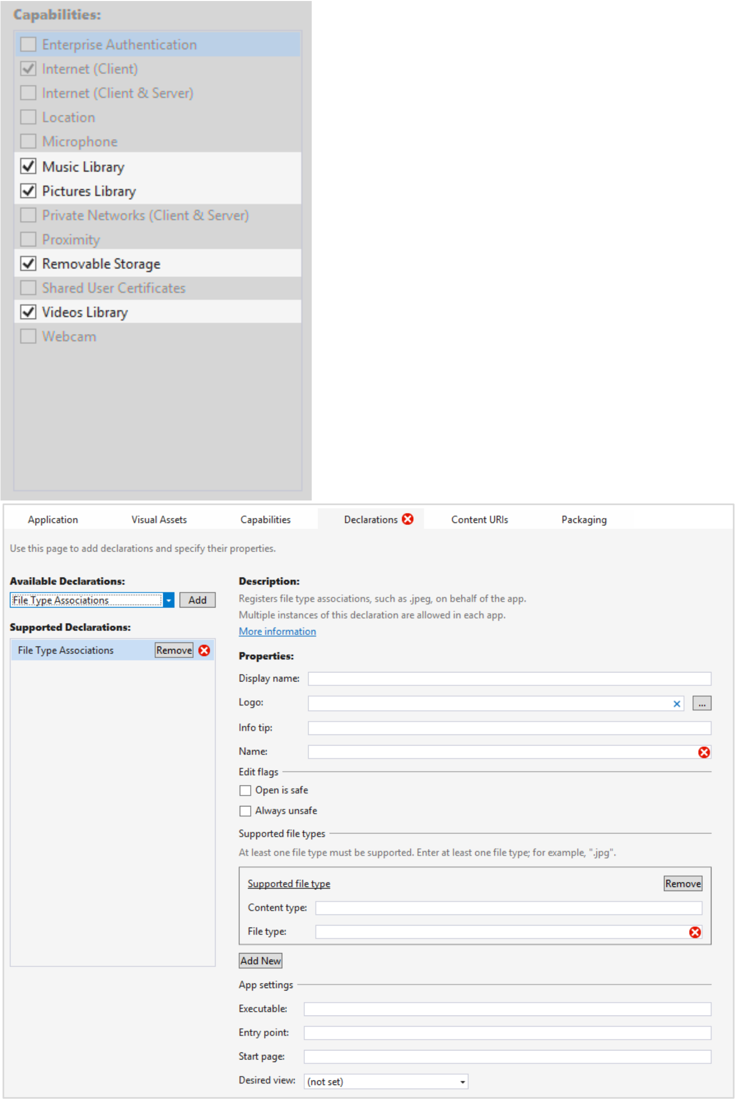
- Let the user choose a file or folder through the File Picker UI, invoked through one of three classes in `Windows.Storage.Pickers`: `FolderPicker`, `FileOpenPicker`, and `FileSavePicker`, each of which is tailored for its particular purpose.
- Acquire a `StorageFolder` for a known library, folder, or removable storage, or acquire a `StorageFile` from one of these.
- The user launches a file for which the app has declared an association in its manifest.

Let me be very clear up front that the first option—using the File Picker UI—*should always be your first choice if you need to access only a single file at a time*. Accessing libraries (and thus declaring the necessary capabilities) is necessary only if you need to enumerate the contents of a library to create a gallery/browsing experience directly in the app. Otherwise the File Pickers do a fabulous job of browsing content across all available locations and don't require you to declare any capabilities. You can also instruct the pickers to use a particular library as its default location, making the whole process even more seamless for your users.

The reason for channeling access through the pickers is that accessing arbitrary locations requires user consent. That consent is implicit in having the user specifically navigate to a file or folder through a picker UI, and doing it that way is much more natural for most users than showing a long pathname in a message dialog! Furthermore, some files and folders—especially those that aren't on the file system and those that are generated dynamically—might not even have user-readable names. The pickers, which we'll see visually in "The File Picker UI" later, provide a friendly, graphical means to this end that apps can also extend through *picker providers*.

Again, the `Windows.Storage.AccessCache` API is how you save a `StorageFolder` or `StorageFile` object along with the user consent implied by a picker. This is essential to remember. I've seen many developers slip into thinking of files and folders in terms of pathnames and just save those strings in their app state. This never preserves access, however, so always think in terms of the `StorageFolder` and `StorageFile` abstractions and APIs like the `AccessCache` that work with them.

When it is appropriate to work with a library directly, the options are quite specific. First you have the Windows.Storage.KnownFolders object, which contains StorageFolder objects for the Pictures, Music, and Videos libraries, as well as Removable Storage. Access to each of these requires the declaration of the appropriate capability in your manifest, as shown in Figure 11-2, without which the attempt to retrieve a folder will throw an Access Denied exception. With Removable Storage you must also declare a file type association, also shown in Figure 11-2. (The static methods StorageFolder.getFolderFromPathAsync and StorageFile.getFileFromPathAsync can access locations with string pathnames if the app has programmatic access through manifest capabilities.)



Capabilities:

- □ Enterprise Authentication
- ☑ Internet (Client)
- □ Internet (Client & Server)
- □ Location
- □ Microphone
- ☑ Music Library
- ☑ Pictures Library
- □ Private Networks (Client & Server)
- □ Proximity
- ☑ Removable Storage
- □ Shared User Certificates
- ☑ Videos Library
- □ Webcam

Use this page to add declarations and specify their properties.

Available Declarations: File Type Associations

Supported Declarations: File Type Associations

Description:

Registers file type associations, such as .jpeg, on behalf of the app.

Multiple instances of this declaration are allowed in each app.

More information

Properties:

Display name:

Logo:

Info tip:

Name:

Edit flags

□ Open is safe

□ Always unsafe

Supported file types

At least one file type must be supported. Enter at least one file type; for example, ".jpg".

Supported file type

Content type:

File type:

App settings

Executable:

Entry point:

Start page:

Desired view: (not set)

FIGURE 11-2 Capabilities related to user data in the manifest editor (left) and the file type association editor (right). The red X on the Declarations tab indicates the minimal required fields for an association.

Where is the Documents library? If you look at the KnownFolders object, you'll also see that there's a documentsLibrary property, but there is no Documents capability in the manifest editor. I call this out because the capability was visible in Windows 8 (that is, Visual Studio 2012), and declaring that capability required one or more file type associations as with Removable Storage. The capability still exists in Windows 8.1 but must be added manually by editing the manifest XML. Declaring it will trigger a more extensive (and time-consuming) process when you submit the app to the Windows Store, which includes verifying that you have a company account (not an individual account), verifying an Extended Validation Certificate, and reviewing your written justifications for using the library. In the end, few apps used the capability in Windows 8, and most of those that did were better off using the file pickers in the first place, hence the stringent requirements.

Note also that you call tell the File Picker API to use the Documents library as the default location, in which case it maps to the user's OneDrive root. A user can also still navigate to their local Documents folder through the pickers if they want, which is the recommended approach for most apps.

Another way to access libraries is through the StorageLibrary objects obtained through the static method Windows.Storage.StorageLibrary.getLibraryAsync.[84] The StorageLibrary object is meant for apps that provide a UI through which a user can manage one of their media libraries. It contains a folders property (a vector of StorageFolder objects) and two methods: requestAddFolderAsync and requestRemoveFolderAsync. Note that the StorageLibrary object does not give you a StorageFolder for the library's root, because you can obtain that through KnownFolders already.

The other known location is represented by the Windows.Storage.DownloadsFolder object, whose only methods, createFolderAsync and createFileAsync, allow you to create folders and files, respectively (but not open or enumerate existing content). The DownloadsFolder object—having only these two methods—is useful only for apps that download user data files and wouldn't otherwise prompt the user for a target location. This is all we'll say of the object in this chapter.

Now that we know how to get to files and folders, we can answer the third question we posed at the beginning of this section: "What affects and modifies user data?" This is something of an open question because user data isn't tied to any particular app and can thus always be modified independently of those apps. The user can rename, copy, move, and delete files and folders, of course, and can use any number of tools to modify file contents (including old-time methods like *copy con* from the command prompt!). We're primarily interested in the answer to this question from an app's point of view, and here the answer is simple: access to and modification of user data starts with the StorageFolder and StorageFile objects. Through these you can modify metadata, for one, as well as open files to get to their data streams. For the most part, we've already seen in Chapter 10 how to get to the data through methods like StorageFile.openAsync and the kinds of things we can do with the resulting streams, buffers, and blobs.

[84] There is also a documents option for this API that has the same requirements as KnownFolder.documentsLibrary.

In this chapter, we'll review a few of those basics and we'll complete the story with all the other features of these objects. This includes extended properties for media files, enumerating and filtering folder contents with file queries (using the Windows.Storage.Search API), and the additional capabilities offered by the StorageLibrary object, as noted earlier. Of special interest is how to use file metadata like thumbnails to create gallery experiences, which avoids the expensive overhead (in time and memory) of opening files and reading their contents for that purpose.

This brings us to the last question—"How do apps associate themselves with specific user data formats?"—and also the third way an app gets a StorageFile object: through a file association. In this case the answer is again simple: apps declare such associations in their manifests. By doing so, those apps appear in the Windows app selector UI when an otherwise unassigned file is launched, and an app will always be there as an option if the user wants to change the default association through PC Settings > Search and Apps > Defaults > Choose Default Apps by File Type.

Such launching happens through the Windows Explorer on the desktop or programmatically through WinRT APIs in Windows.System.Launcher. In either case, the associated app gets activated with an activation kind of file, and the activation event args will contain the appropriate StorageFile objects. We'll see the details toward the end of this chapter.

Sidebar: Enterprise File Protection

The Windows.Security.EnterpriseData.FileRevocationManager APIs are an additional set of capabilities with the file system, but they are not covered in this book. These help you manage copy protection for any StorageItem with what is called *selective wipe* (described in the Security documentation). This accommodates enterprise users who bring their own mobile devices to work and use them to access corporate data. IT departments, of course, want to make sure that such data doesn't get leaked outside the enterprise environment. Access to files and folders can be granted in a protected manner through FileRevocation-Manager.protectAsync such that they can be revoked remotely through a server-issued command. Revoked files are completely inaccessible even though they still technically exist on the file system.

For use of the API, refer to the File Revocation Manager sample.

Using the File Picker and Access Cache

Although the File Picker doesn't sound all that glamorous, it's actually, to my mind, one of the coolest features in Windows. "Wait a minute!" you say, "How can a UI to pick a file or folder be, well, *cool*!" The reason is that this is *the* place where the users can browse and select from their entire world of data. That world—as I've said several times already—includes locations well beyond what we normally think of as the local file system (local drives, removable drives, and the local network). Those added locations are made available by what are called *file picker providers:* apps that specifically take a library of data that's otherwise buried behind a web service, within an app's own database, or even generated on the fly and make it appear as if it's part of the local file system.

Think about this for a moment (as I invited you to do way back in Chapter 1, "The Life Story of a Windows Store App"). When you want to work with an image from a photo service like Flickr or Picasa, for example, what do you typically have to do? The first step is to download that file to the local file system within some app that gives you an interface to that service (which might be a web app). Then you can make whatever edits and modifications you want, after which you typically need to upload the file back to the service. Well, that's not so bad, except that it's time consuming, it forces you to switch between multiple apps, and eventually it litters your system with a bunch of temporary files, the relationship of which to your online service is quickly forgotten.

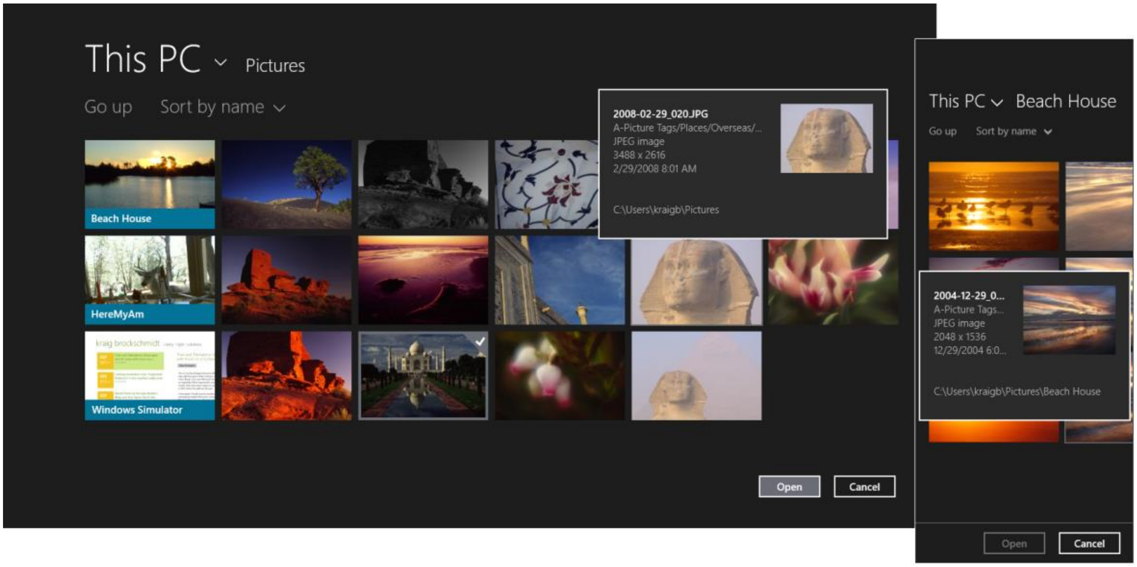
Having a file picker provider that can surface such data directly, both for reading and writing, eliminates all those intermediate steps and eliminates the need to switch apps. This means that a provider for a photo service makes it possible for other apps to load, edit, and save online content as if it all existed on the local file system. Consuming apps don't need to know anything about those other services, and they automatically have access to more services as more provider apps are installed. What's more, providers can also make data that isn't normally stored as files appear as though they are. For example, the Sound Recorder app that's built into Windows is a file picker provider that lets you record a new audio file and return it just as if it had already been present on the file system. All of this gives users a very natural means to flow in and out of data no matter where it's stored. Like I said, I think this is a very cool feature!

In this section, we'll first look at the File Picker UI so that we know what's going to appear when we use the File Picker API in `Windows.Storage.Pickers`. Then we'll see the `Windows.Storage.AccessCache` API, because it's in the context of the file picker that you'll typically be saving file permissions for later sessions.

We'll look more at the question of providers in Appendix D, "Contract Providers." Our more immediate concern is how to use these file pickers to obtain a `StorageFile` or `StorageFolder` object.

The File Picker UI

When a file picker is invoked, you'll see a full-screen view like that in Figure 11-3, depending on whether you want single or multiple selection, whether you're picking files or folders (or a save location), and whether you want only specific file types. In the case of Figure 11-3, the picker is invoked to choose a single image with a thumbnail view (which provides a rich tooltip control when you hover over an item). In a way, the file picker itself is like an app that's invoked for this purpose, and it's designed (with a dark gray background) to give full attention to the contents of the files. The pickers also provide semantic zoom capabilities, as you'd expect, and can be invoked in any sized view even down to the 320px minimum, as shown on the right of the figure.



2008-02-29_020.JPG
A-Picture Tags/Places/Overseas/...
JPEG image
3488 x 2616
2/29/2008 8:01 AM
C:\Users\kraigb\Pictures

2004-12-29_0...
A-Picture Tags...
JPEG image
2048 x 1536
12/29/2004 6:0...
C:\Users\kraigb\Pictures\Beach House

FIGURE 11-3 A single-selection file picker on the Pictures library in thumbnail view mode, with a hover tooltip showing for one of the items (the head of the Sphinx) and the selection frame showing on another (the Taj Mahal). The overlay on the right shows the file picker in a narrow 320px view. Bonus points if you can identify the location of the other ruins in the main view on the left! (And if you're wondering, these are all my own photos.)

In Figure 11-3, the Pictures heading shows the current location of the picker. The Sort By Name drop-down lets you choose other sorting criteria, and the This PC header is also a drop-down list that lets you navigate to different locations, as shown in Figure 11-4, including other areas of the file system (though never protected areas like the Windows folder or Program Files), network locations, *and* other provider apps. When choosing a picture (left side), notice how the list of apps is filtered to show just those that can provide pictures. When the picker is invoked to choose general files or other types like music (right side), additional apps like the Sound Recorder can appear.

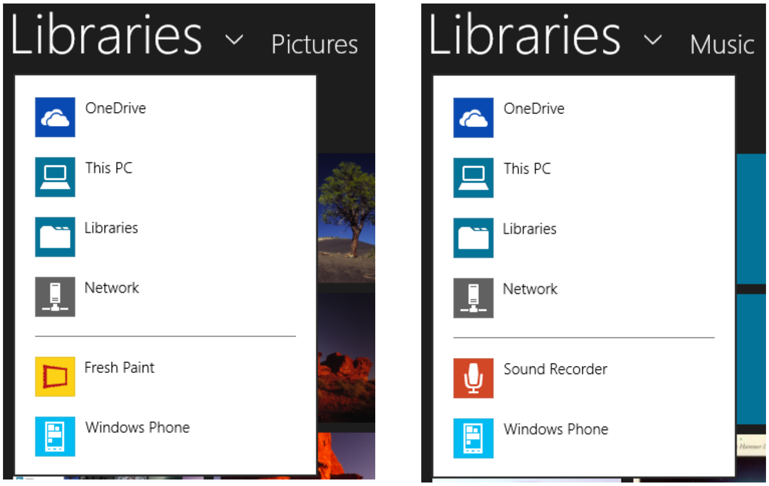


FIGURE 11-4 Selecting other picker locations; notice that OneDrive and apps are listed along with file system locations. The picker on the left is invoked to select pictures, so only picture-providing apps appear; the picker on the right is invoked to select any type of file, so additional providers appear.

Choosing another file system or network location navigates there, of course, from which you can browse into other folders. As OneDrive is built into Windows, it's treated like another network location, as shown in Figure 11-5, and it's also the default location for the file save picker (controlled through PC Settings > OneDrive > File Storage > Save Documents to OneDrive by Default).

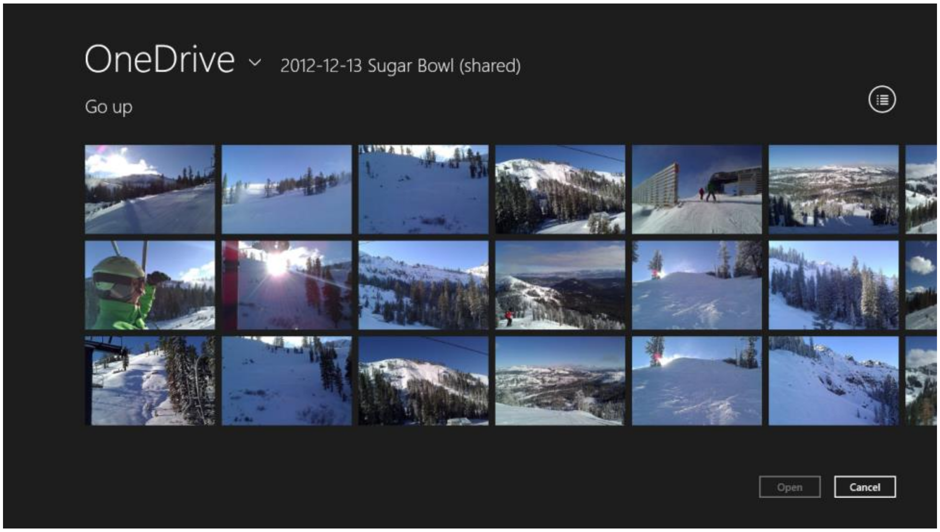


FIGURE 11-5 When picking files from OneDrive, cloud storage appears like any other local or network location.

Selecting an app, on the other hand, launches that app through the file picker provider contract. In this case it appears within a system-provided—but app-branded—UI like that shown in Figure 11-6 and Figure 11-7. In these cases the heading reflects the name of the app but also provides the drop-down list that lets you navigate to other picker locations (which is important for multiple selections); the Open and Cancel buttons act as they do for other picker selections. In short, a provider app is just an extension to the File Picker UI, but it's a very powerful one. And ultimately such an app just returns an appropriate `StorageFile` object that makes its way back to the original app. There's quite a lot happening with just a single call to the File Picker API!

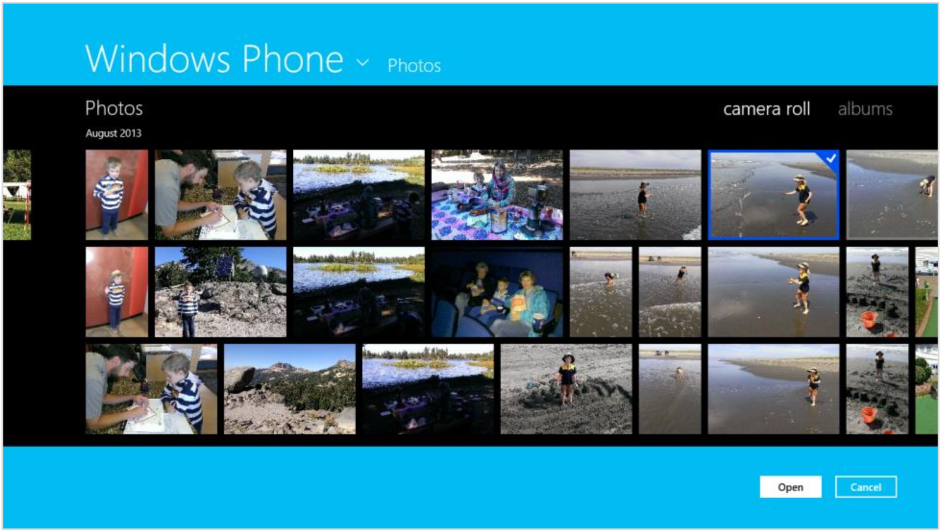


FIGURE 11-6 The Windows Phone app invoked through the file picker provider contract to select a single image.

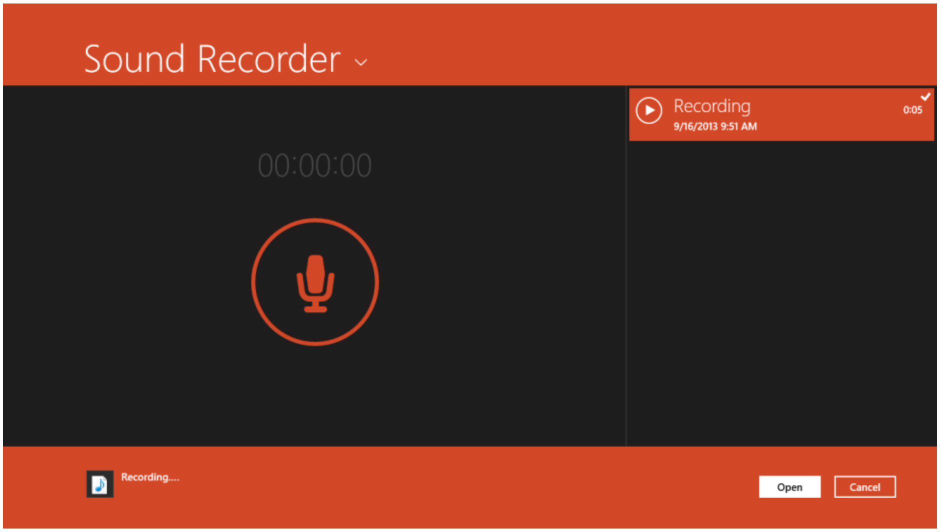


FIGURE 11-7 The Sound Recorder app invoked through the file picker provider contract. This is what appears after a sound has been recorded, and because the picker is invoked to select multiple files, the user can create and return multiple recordings at one time.

In Figure 11-3, Figure 11-5, and Figure 11-6, the picker is invoked to select a single file. In Figure 11-7, on the other hand, it is invoked to select multiple files, which can again come from the file system, network or cloud locations, or other apps—it doesn't matter! With multiple selection, selected items are placed into what's called the *basket* on the bottom of the screen. You can see this in Figure 11-7 and also in Figure 11-8 (where the picker is using list view mode rather than thumbnails). The purpose of the basket is to let you select items from one location, navigate to a new location through the header drop-down, select a few more, and then navigate to still other locations. In short, the basket holds whatever items you select from *whatever locations*. The basket in Figure 11-8 contains the sound recording from Figure 11-7, a picture from my phone, a picture from OneDrive, an MP3 file from my Music folder, and a couple videos from the current folder.

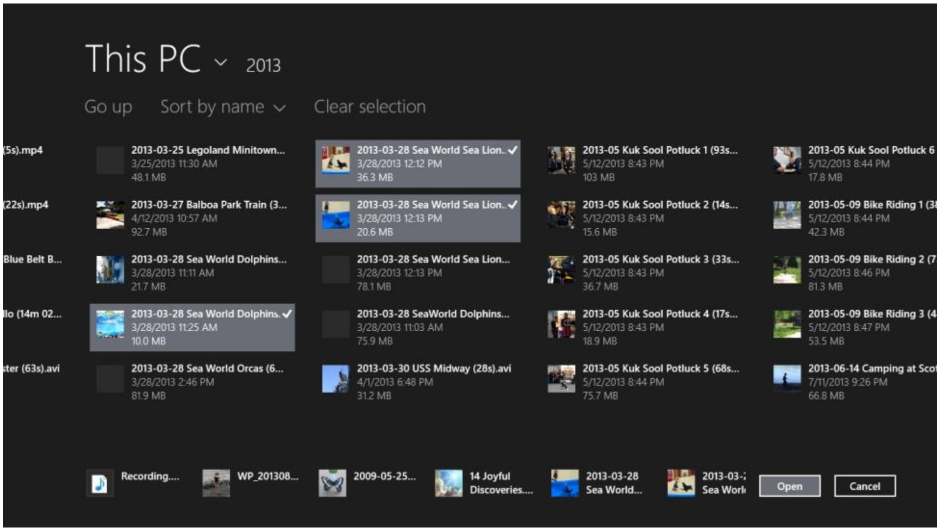
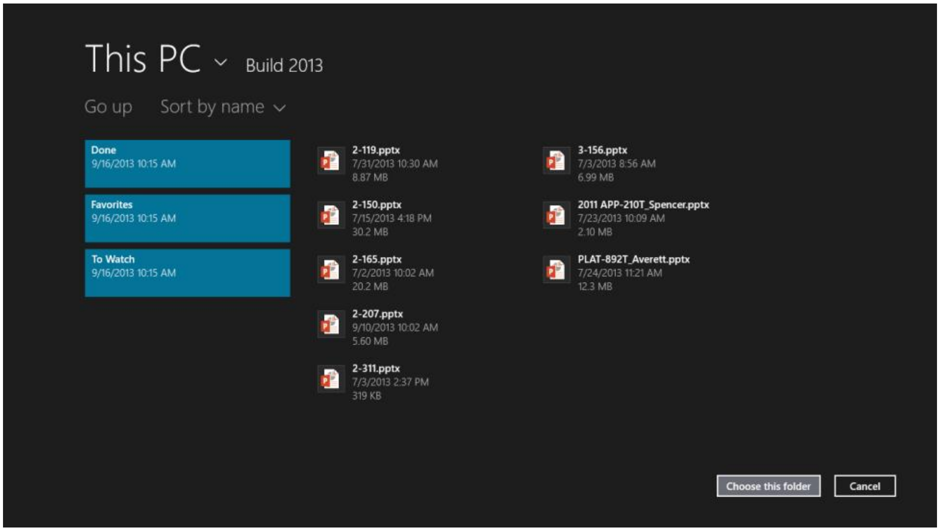

This PC ⌄ 2013
Go up
Sort by name ⌄
Clear selection
(5s).mp4
(22s).mp4
Blue Belt B...
llo (14m 02...
ster (63s).avi
2013-03-25 Legoland Minitown... 3/25/2013 11:30 AM 48.1 MB
2013-03-27 Balboa Park Train (3... 4/12/2013 10:57 AM 92.7 MB
2013-03-28 Sea World Dolphins... 3/28/2013 11:11 AM 21.7 MB
2013-03-28 Sea World Dolphins.. ✓ 3/28/2013 11:25 AM 10.0 MB
2013-03-28 Sea World Orcas (6... 3/28/2013 2:46 PM 81.9 MB
2013-03-28 Sea World Sea Lion.. ✓ 3/28/2013 12:12 PM 36.3 MB
2013-03-28 Sea World Sea Lion.. ✓ 3/28/2013 12:13 PM 20.6 MB
2013-03-28 Sea World Sea Lion... 3/28/2013 12:13 PM 78.1 MB
2013-03-28 SeaWorld Dolphins... 3/28/2013 11:03 AM 75.9 MB
2013-03-30 USS Midway (28s).avi 4/1/2013 6:48 PM 31.2 MB
2013-05 Kuk Sool Potluck 1 (93s... 5/12/2013 8:43 PM 103 MB
2013-05 Kuk Sool Potluck 2 (14s... 5/12/2013 8:43 PM 15.6 MB
2013-05 Kuk Sool Potluck 3 (33s... 5/12/2013 8:43 PM 36.7 MB
2013-05 Kuk Sool Potluck 4 (17s... 5/12/2013 8:43 PM 18.9 MB
2013-05 Kuk Sool Potluck 5 (68s... 5/12/2013 8:44 PM 75.7 MB
2013-05 Kuk Sool Potluck 6 5/12/2013 8:44 PM 17.8 MB
2013-05-09 Bike Riding 1 (3 5/12/2013 8:44 PM 42.3 MB
2013-05-09 Bike Riding 2 (7 5/12/2013 8:46 PM 81.3 MB
2013-05-09 Bike Riding 3 (4 5/12/2013 8:47 PM 53.5 MB
2013-06-14 Camping at Scot 7/11/2013 9:26 PM 66.8 MB
Recording....
WP_201308...
2009-05-25...
14 Joyful Discoveries....
2013-03-28 Sea World...
2013-03-: Sea Worl
Open
Cancel

FIGURE 11-8 The file picker in multiselect mode with the selection basket at the bottom. The picker's layout here is a "list" view mode (not thumbnails) that's set independently from the selection mode.

The picker can also be used to select a folder, as shown in Figure 11-9 (provider apps aren't shown from the heading drop-down in this case), or a save location and filename, as shown in Figure 11-10.



Done
9/16/2013 10:15 AM

Favorites
9/16/2013 10:15 AM

To Watch
9/16/2013 10:15 AM

2-119.pptx
7/31/2013 10:30 AM
8.87 MB

2-150.pptx
7/15/2013 4:18 PM
30.2 MB

2-165.pptx
7/2/2013 10:02 AM
20.2 MB

2-207.pptx
9/10/2013 10:02 AM
5.60 MB

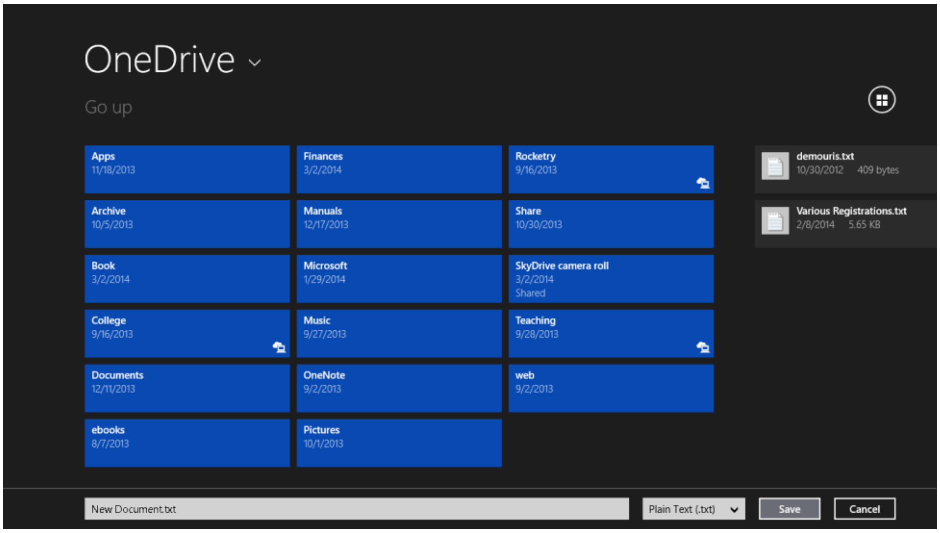
2-311.pptx
7/3/2013 2:37 PM
319 KB

3-156.pptx
7/3/2013 8:56 AM
6.99 MB

2011 APP-210T_Spencer.pptx
7/23/2013 10:09 AM
2.10 MB

PLAT-892T_Averett.pptx
7/24/2013 11:21 AM
12.3 MB

FIGURE 11-9 The file picker used to select a folder—notice that the button text changed and the picker shows the contents of the folder.



Apps
11/18/2013

Archive
10/5/2013

Book
3/2/2014

College
9/16/2013

Documents
12/11/2013

ebooks
8/7/2013

Finances
3/2/2014

Manuals
12/17/2013

Microsoft
1/29/2014

Music
9/27/2013

OneNote
9/2/2013

Pictures
10/1/2013

Rocketry
9/16/2013

Share
10/30/2013

SkyDrive camera roll
3/2/2014
Shared

Teaching
9/28/2013

web
9/2/2013

demouris.txt
10/30/2012 409 bytes

Various Registrations.txt
2/8/2014 5.65 KB

New Document.txt

Plain Text (.txt)

FIGURE 11-10 The file picker used to select a save location (defaulting to OneDrive) and filename (at the bottom). Files that match the specified save type are also shown alongside folders.

The File Picker API

Now that we've seen the visual results of the file picker, let's see how we invoke it from our app code through the API in `Windows.Storage.Pickers` (assume this namespace unless indicated). All the images we just saw came from the File picker sample, so we'll use that as the source of our code.

For starters, scenario 1 of the sample, in its pickSinglePhoto function (js/scenario1.js), uses the picker to obtain a single StorageFile for opening (reading and writing):

```
function pickSinglePhoto() {
    // Create the picker object and set options
    var openPicker = new Windows.Storage.Pickers.FileOpenPicker();
    openPicker.viewMode = Windows.Storage.Pickers.PickerViewMode.thumbnail;
    openPicker.suggestedStartLocation =
        Windows.Storage.Pickers.PickerLocationId.picturesLibrary;

    // Users expect to have a filtered view of their folders depending on the scenario.
    openPicker.fileTypeFilter.replaceAll([".png", ".jpg", ".jpeg"]);

    // Open the picker for the user to pick a file
    openPicker.pickSingleFileAsync().done(function (file) {
        if (file) {
            // Application now has read/write access to the picked file
        } else {
            // The picker was dismissed with no selected file
        }
    });
}
```

To invoke the picker, we create an instance of the FileOpenPicker class, configure it, and call its pickSingleFileAsync method. The result of pickSingleFileAsync as delivered to the completed handler is a StorageFile object, which will be null if the user canceled the picker. *Always* check that the picker's result is not null before taking further action on the file!

With the configuration, here we're setting the picker's viewMode to thumbnail (from the PickerViewMode enumeration), resulting in the view in Figure 11-3. The only other possibility here is list, the view shown in Figure 11-8.

We also set the suggestedStartLocation to the picturesLibrary, which is a value from the PickerLocationId enumeration; other possibilities are documentsLibrary (OneDrive or This PC > Documents), computerFolder (meaning This PC on Windows 8.1), desktop, downloads, homeGroup, musicLibrary, and videosLibrary.

No capabilities needed! It's very important to note that picker locations do *not* require you to declare any capabilities in your manifest. By using the picker, the user is giving consent for you to access whatever location he or she chooses. If you check the manifest in the File pickers sample, in fact, you'll see that no capabilities are declared whatsoever and yet you can still navigate anywhere other than protected system folders, including network locations.

OneDrive or Documents folder? If the user has PC Settings > OneDrive > File Storage > Save to OneDrive by Default turned on, the file picker will show OneDrive when you specify the documentsLibrary location (as in Figure 11-10). If the user turns this off, that location will bring up the user's local Documents folder instead.

The one other property we set is the fileTypeFilter (a vector/array of strings) to indicate the type of files we're interested in (PNG and JPEG). Beyond that, the FileOpenPicker also has a commitButtonText property, which sets the label of the primary button in the UI (the one that's not Cancel), and settingsIdentifier, a means to essentially remember different contexts of the file picker. For example, an app might use one identifier for selecting pictures, where the starting location is set to the pictures library and the view mode to thumbnails, and another id for selecting documents with a different location and perhaps a list view mode.

This sample, as you can also see, doesn't actually do anything with the file once it's obtained, but it's quite easy to imagine what we might do. We can, for instance, simply pass the StorageFile to URL.createObjectURL and assign the result to an img.src property for display. The same thing could be done with audio and video, possibilities that are all demonstrated in scenario 1 of the Using a blob to save and load content sample I mentioned in Chapter 10. That sample also shows reading the file contents through the HTML FileReader API alongside the other WinRT and WinJS APIs we've seen. You could also transcode an image (or other media) in the StorageFile to another format (as we'll see in Chapter 13, "Media"), retrieve thumbnails as shown in the File and folder thumbnail sample, or use the StorageFile methods to make a copy in another location, rename the file, and so forth. But from the file picker's point of view, its particular job was well done!

Returning now to the File pickers sample, picking multiple files is pretty much the same story as shown in the pickMultipleFiles function of scenario 2 (js/scenario2.js). Here we're using the list view mode and starting off in the documentsLibrary (which goes to either OneDrive or the local Documents folder depending on the user's choice in PC Settings). Again, these start locations *do not* require capability declarations in the manifest, which is fortunate here because the Documents library capability has many restrictions on its use!

```
function pickMultipleFiles() {
    // Create the picker object and set options
    var openPicker = new Windows.Storage.Pickers.FileOpenPicker();
    openPicker.viewMode = Windows.Storage.Pickers.PickerViewMode.list;
    openPicker.suggestedStartLocation =
        Windows.Storage.Pickers.PickerLocationId.documentsLibrary;
    openPicker.fileTypeFilter.replaceAll(["*"]);

    // Open the picker for the user to pick a file
    openPicker.pickMultipleFilesAsync().done(function (files) {
        if (files.size > 0) {
            // Application now has read/write access to the picked file(s)
        } else {
            // The picker was dismissed with no selected file
        }
    });
}
```

When picking multiple files, the result of pickMultipleFilesAsync is an array (technically a vector view) of StorageFile objects.

Scenario 3 of the sample shows a call to pickSingleFolderAsync defaulting to the desktop, where the result of the operation is a `StorageFolder`. Here you must indicate a `fileTypeFilter` that helps users pick an appropriate location where some files of that type exist or create a new location (js/scenario3.js):

```
function pickFolder() {
    // Create the picker object and set options
    var folderPicker = new Windows.Storage.Pickers.FolderPicker;
    folderPicker.suggestedStartLocation = Windows.Storage.Pickers.PickerLocationId.desktop;
    folderPicker.fileTypeFilter.replaceAll([".docx", ".xlsx", ".pptx"]);

    folderPicker.pickSingleFolderAsync().then(function (folder) {
        if (folder) {
            // Cache folder so the contents can be accessed at a later time
            Windows.Storage.AccessCache.StorageApplicationPermissions.futureAccessList
                .addOrReplace("PickedFolderToken", folder);
        } else {
            // The picker was dismissed with no selected file
        }
    });
}
```

You can see here that we save the folder in the access cache, which we'll come back to shortly. First let's look at the final file picker use case in scenario 4, where we use a [`FileSavePicker`] object and its `pickSaveFileAsync` method (js/scenario4.js), resulting in the UI of Figure 11-10 (assuming OneDrive is the default save location; I've also added the *myData* variable to illustrate what's being saved, even though it isn't in the sample):

```
function saveFile(myData) {
    // Create the picker object and set options
    var savePicker = new Windows.Storage.Pickers.FileSavePicker();
    savePicker.suggestedStartLocation =
        Windows.Storage.Pickers.PickerLocationId.documentsLibrary;
    // Drop-down of file types the user can save the file as
    savePicker.fileTypeChoices.insert("Plain Text", [".txt"]);
    // Default file name if the user does not type one in or select a file to replace
    savePicker.suggestedFileName = "New Document";

    savePicker.pickSaveFileAsync().done(function (file) {
        if (file) {
            // Prevent updates to the remote version of the file until we finish making
            // changes and call CompleteUpdatesAsync.
            Windows.Storage.CachedFileManager.deferUpdates(file);

            // write to file
            Windows.Storage.FileIO.writeTextAsync(file, myData).done(function () {
                // Let Windows know that we're finished changing the file so the other app
                // can update the remote version of the file (see Appendix D).
                // Completing updates might require Windows to ask for user input.
                Windows.Storage.CachedFileManager.completeUpdatesAsync(file)
                    .done(function (updateStatus) {
                        if (updateStatus ===
                            Windows.Storage.Provider.FileUpdateStatus.complete) {
                        } else {
                            // ...
                        }
                    }
                });
            });
        } else {
            // The picker was dismissed
        }
    });
}
```

The `FileSavePicker` has many of the same properties as the `FileOpenPicker`, but it replaces `fileTypeFilter` with `fileTypeChoices` (to populate the drop-down list) and includes `suggestedFileName` (a string), `suggestedSaveFile` (a `StorageFile`), and `defaultFileExtension` (a string).

What's interesting (and important!) in the code above are the interactions with the `Windows.Storage.CachedFileManager` API. This object helps file picker providers know when they should synchronize local and remote files, which is often necessary when a file consumer saves new content. Technically speaking, use of the `CachedFileManager` API isn't required—you can just write to the file and be done with it, especially for files you know are local and also for a single write as shown here. However, if you're doing multiple writes, placing your I/O within calls to `deferUpdates` and `completeUpdatesAsync` methods will make the process more efficient. For more details on the caching mechanism, refer to Appendix D.

Access Cache

As we've now seen, the File Picker UI allows users to navigate to and select files and folders in many different locations, and through the File Picker APIs an app gets back the appropriate `StorageFile` and `StorageFolder` objects for those selections. By virtue of the user having selected those files and folders, an app has full programmatic access through the `StorageFile` and `StorageFolder` APIs as it does for its appdata folders and those libraries declared in its manifest.

This is all well and good within any given app session. But how does an app preserve that same level of access across sessions (that is, when the app is closed and restarted later on, or across reboots)? Furthermore, how does an app save references to such files and folders in its state? Remember that the `StorageFile` and `StorageFolder` objects are essentially rich abstractions for pathnames, and they hide the fact that some entities cannot even be represented by a path to begin with because they use URIs with custom schema or some other provider-specific naming convention altogether.

These needs are met by the `Windows.Storage.AccessCache` API, which saves `StorageFile` and `StorageFolder` objects along with their permissions such that you can retrieve those same objects and permissions in subsequent sessions. Simply said, unless you know for certain that your app already has programmatic access to a given item and can definitely be represented by a pathname (which basically means appdata locations), *always* use the `AccessCache` API to save file/folder references instead of saving path strings. Otherwise you'll see Access Denied errors when you try to open the item again.

When you add a storage item to the cache—and I'll refer now to files and folders as *items* for convenience unless the distinction is important—what you get back is a string token. You save that token in your app state if you want to get back to a specific item later on, but you can also enumerate the contents of the cache at any time. What's also very powerful is that for the local file system, at least, the token will continue to provide access to its associated item even if that item is independently moved or renamed. That is, the access cache does its best to keep the tokens connected to their underlying files, but it's not an exact science. For this reason, if you find an invalid token in the cache, you'll want to remove it.

The access cache maintains two per-app lists for storage items: a future access list and a recently used list. You get to these through the `AccessCache.StorageApplicationPermissions` class and its `futureAccessList` and `mostRecentlyUsedList` properties (it has no others):

```
var futureList = Window.Storage.AccessCache.StorageApplicationPermissions.futureAccessList;
var mruList = Window.Storage.AccessCache.StorageApplicationPermissions.mostRecentlyUsedList;
```

Technically speaking, the two are almost identical; their methods and properties come from the same interface (see table below). You could, in fact, maintain your own recently used list by adding items to the `futureAccessList` and saving the returned tokens in some collection of your own. But because a recently used list is a common app scenario, the API's designers decided to save you the trouble. The `mostRecentlyUsedList` is thus limited to 25 items and automatically removes the oldest items when a new one is added that would exceed that limit. The `mostRecentlyUsedList` fires its `itemremoved` event in this case, which you can use to enumerate the current contents of the list and update your UI as necessary.

The `futureAccessList`, on the other hand, is there for any and all other items for which you want to preserve access. It has an upper limit of 1000 items and will throw an exception when it's full, so you have to remove items yourself.

The methods and properties of both lists (which come from the `IStorageItemAccessList` interface), as are follows:

Property	**Description**
`entries`	An `AccessListEntryView` that provides access to the collection of items.
`maximumItemsAllowed`	The maximum number of entries in this list; 1000 for the `futureAccessList`, 25 for the `mostRecentlyUsedList`.
Method	**Description**
`add`	Adds an item in the list, returning a token. A variant method allows you to attach an extra string of metadata to the entry.
`addOrReplace`	Replaces an existing item in the list, given its token; a variant method allows you to attach a metadata string. Note that this method has no return value, as the same token is reassigned to the new item.
`remove`	Removes an entry from the list given its token.
`clear`	Removes all entries from the list.
`checkAccess`	Given a `StorageItem`, returns `true` if it exists in the list and the app can access it, `false` otherwise.
`containsItem`	Given a token, returns true if the item exists in the list.

`getFileAsync`
`getFolderAsync`
`getItemAsync`
	Retrieve a `StorageFile`, `StorageFolder`, or `StorageItem` object from the list given its token. Variants of each method also take a combination of `AccessCacheOptions` values (combined with `|`) to limit which items are returned:
	• `none` The default, assuming no other flags.

	• disallowUserInput Returns an item only if the user need not provide any additional information to access it, such as credentials. • fastLocationsOnly Returns an item only if exists in a fast location, like the local file system. An item that would need to be downloaded first will not be returned. • useReadOnlyCachedCopy Returns a cached, read-only item that might not be the most recent (e.g., it's out of sync with its cloud backend). • suppressAccessTimeUpdate Preserves the items position in the mostRecentlyUseList and its access timestamp; does not affect the futureAccessList.

The update to the Here My Am! app we made in Chapter 10 shows some basic use of the access cache. First, here's how we save the current image's StorageFile in the futureAccessList and save its token in the app's sessionState (pages/home/home.js):

```
var list = Windows.Storage.AccessCache.StorageApplicationPermissions.futureAccessList;

if (app.sessionState.fileToken) {
    list.addOrReplace(app.sessionState.fileToken, newFile);
} else {
    app.sessionState.fileToken = list.add(newFile);
}
```

Notice how we use the list's addOrReplace method if we already have a token from a previous session; otherwise we add the item anew and save that token. I will say that when I first wrote this code, I didn't realize that addOrReplace does *not* return the same token you pass in, and I was assigning undefined to my fileToken variable. Such an assignment is unnecessary because I already have that token in hand.

Anyway, if the app is suspended and then terminated, we check for the token during activation and attempt to retrieve its StorageFile, which we use to rehydrate the image element (also in pages/home/home.js):

```
if (app.sessionState.fileToken) {
    var list = Windows.Storage.AccessCache.StorageApplicationPermissions.futureAccessList;

    list.getFileAsync(app.sessionState.fileToken).done(function (file) {
        if (file != null) {
            lastCapture = file;
            var uri = URL.createObjectURL(file);

            var img = document.getElementById("photoImg");
            img.src = uri;
            scaleImageToFit(img, document.getElementById("photo"), file);
        }
    });
}
```

Scenario 7 of the File access sample has additional demonstrations, letting you choose which list to work with. It shows that you can enumerate the entries collection of either list. As above, entries is an AccessListEntryView with a size property and first, getAt, getMany, and indexOf methods (basically a derivative of a vector view, which we saw in Chapter 6, "Data Binding, Templates, and Collections"). This type is projected into JavaScript as an array, so you can also use the [ ] operator and methods like forEach as scenario 7 of the sample shows (js/scenario7.hs):

```
var mruEntries =
    Windows.Storage.AccessCache.StorageApplicationPermissions.mostRecentlyUsedList.entries;
if (mruEntries.size > 0) {
    var mruOutputText = "The MRU list contains the following item(s):<br /><br />";
    mruEntries.forEach(function (entry) {
        mruOutputText += entry.metadata + "<br />";
    });
    outputDiv.innerHTML = mruOutputText;
}
```

Each entry in the AccessListEntryView is a simple AccessListEntry object that contains the item's token and a metadata property with the string you can include when adding an item to the list.

StorageFile Properties and Metadata

In Chapter 10 (in "Folders, Files, and Streams") we began looking at the many methods and properties of the StorageFile object, limiting ourselves to the basics because many of its features apply primarily to user data files rather than those you'll create for your app state. Now we're ready to delve into all its details, a topic that will take us quite deep down a few rabbit holes!

Note Many of the properties and methods described here for StorageFile also apply to StorageFolder object, but for convenience we'll focus on StorageFile.

Let's just assume that you've obtained a StorageFile object of interest through some means, be it a file picker, a media library, a file activation, one of the static StorageFile methods like getFileFromPathAsync or replaceWithStreamedFileAsync, and so on. As we've seen in Chapter 10, you can open the file (obtaining a stream) through openAsync, openReadAsync, openSequentialReadAsync, and openTransactedWriteAsync. You can also manage the file on the file system (as you would through Windows Explorer) with copyAsync, copyAndReplaceAsync, deleteAsync, moveAsync, moveAndReplaceAsync, and renameAsync. And the purpose of many other properties and methods, listed below, should be quite apparent from their descriptions, so we won't cover them here. Refer to the StorageFile reference and the File access sample for all that.

Property	**Description**
name	The simple filename string of the file, including the extension if appropriate.
Path	The full pathname string of the file.
displayName	The file's name string as appropriate for UI, typically without the extension.
fileType	The file's extension string. displayName + fileType is the same as name at least on the local file system.
contentType	The string MIME type of the file contents (e.g., "audio/wma").
displayType	The content type string as appropriate for UI (e.g., "Windows Media Audio file").
dateCreated	A Date object with the creation timestamp of the file.
Attributes	A value containing one of more FileAttributes values such as readOnly.

folderRelativeId	A unique identifier for the file within its parent folder, which distinguished files that nave the same name.
provider	A StorageProvider object describing the provider that's handling this file. The object simply contains id and displayName properties, such as "computer" and "This PC" or "OneDrive" for both.
Method	**Description**
isOfType	Tests whether the item is a file or folder (or just a generic StorageItem). See the StorageItemTypes enumeration.
getParentAsync	Retrieves the StorageFolder that contains this file.
isEqual	Tests whether the file and another StorageItem are the same entity, returning true or false.

What's left are just two properties and three methods that are among the most important to understand:

- The isAvailable property (described next in "Availability")
- The getThumbnailAsync and getScaledImageAsThumbnailAsync methods (described in "Thumbnails")
- The StorageFile.properties property and the getBasicPropertiesAsync method (described in "File Properties")

Availability

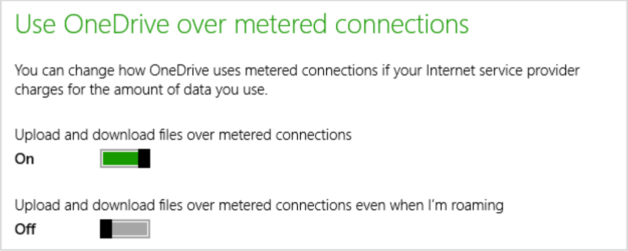
First is the isAvailable property that I mentioned earlier. When working with files that might originate in the cloud, it's generally unimportant for an app to know where it came from, how it might be downloaded, and so forth, because the provider that's behind the StorageFile can take care of all that transparently like Windows does for OneDrive. What you primarily need to know in an app is whether you can expect to open that file and get to its contents. The simple isAvailable flag (a Boolean) tells you that and relieves you from the burden of having to check network connectivity yourself. This is a big reason why placeholder files are often referred to as "smart" files!

The following table (thanks to Marc Wautier) indicates the different conditions that will set this flag to true or false:

Location	Online	Metered Network	Offline
Local file	true	true	true
OneDrive file marked "available offline"	true	true	true
OneDrive placeholder file (marked "online only")*	true	Based on user setting	false
Other network-based file	true	Based on user setting	false

* Opening a placeholder file will download it and mark it "available offline."

For a metered network, the user settings are found in PC Settings > OneDrive > Metered Connections, as shown below. These are simple on/off settings regardless of file size.



Use OneDrive over metered connections

You can change how OneDrive uses metered connections if your Internet service provider charges for the amount of data you use.

Upload and download files over metered connections

On

Upload and download files over metered connections even when I'm roaming

Off

Thumbnails

Next we have the `getThumbnailAsync` and `getScaledImageAsThumbnailAsync` methods. What's very important about these is that they help you more efficiently create gallery or browsing experiences in apps without having to manually open files and generate your own image from its contents. For one thing, this is somewhat difficult to do if the file itself is not already an image. In addition, it's horribly inefficient to open an image file as a whole just to generate a thumbnail.

For example, let's say you just want to show the images in the user's Pictures library. You can easily obtain its `StorageFolder`, enumerate all the `StorageFile` objects therein, pass each one to `URL.createObjectURL`, and store the result in a `WinJS.Binding.List` that you then provide to a ListView. Simple, yes! But—ouch!—take a look at the app in a memory profiler when your Pictures library contains a few hundred multi-megabyte images and you'll be wishing there was another way to do it. Run the app through other performance analyzers and you'll see that most of the time in your app is spent chewing on a bunch of potentially large images just to create small representations on the order of 150x150 pixels. This is one case where the obvious approach definitely does not yield the best results!

By using thumbnails, on the other hand, you take advantage of pre-cached image metadata, which also works for nonimage files (and folders) automatically, so you never need to make the distinction. Furthermore, thumbnails generally work for files even when `isAvailable` is `false` (unless there's simply no cached data), whereas the `URL.createObjectURL` approach—which opens the files and reads its contents!—will necessitate a download and thus also fail outright when `isAvailable` isn't set, regardless of existing caches.

Think also about your own pictures library and the typical images you probably have from your camera or phone. What are the native image resolutions? Many of mine are in the 2048x1536 or 3700x2400 range, and that's because I'm not too concerned with really great image quality. If you're a pixel junkie and have one of those 16+ megapixel cameras, your files are likely much larger. Now compare those sizes to your display resolutions—I have two 24" monitors in front of me right now, whose resolution is only 1920x1200. What this means is that most of the time, the images you display in an app—even when full screen—are essentially thumbnails of the original!

For the sake of memory efficiency and performance, then, the only time you should ever be loading up a full image file is when you need to display the raw pixels in a 1:1 mapping on the display, as when you're editing the image. Otherwise, essentially for all consumption scenarios, use a thumbnail.

(Remember that we did this to Here My Am! way back in Chapter 2, "Quickstart," after we initially used URL.createObjectURL. The Hilo sample app from Microsoft's Patterns & Practices group also used thumbnails exclusively.)

The two thumbnail methods, getThumbnailAsync and getScaledImageAsThumbnailAsync, both accomplish the same ends (a StorageItemThumbnail object). The difference between them is that the first *always* draws from a thumbnail cache, whereas the latter will go to the full file image as a fallback. For this reason, getScaledImageAsThumbnailAsync is primarily used for obtaining a large thumbnail—even one that's full screen—although you can also set it to use only cached images as well.

Both methods actually have three variants to accommodate some optional arguments. In all cases the required argument is a value from the ThumbnailMode enumeration that indicates the type of thumbnail you want (based on intended use) and the default size:

Mode	Use	Default Size
documentsView	Generic preview of folder contents	40x40 (square aspect)
musicView	Preview of music files, which will draw from album art if available	40x40 (square aspect)
picturesView	Image file previews	190x130 (wide aspect)
videosView	Preview of videos, using a still	190x130 (wide aspect)
listView	Generic ListView display	40x40 (square aspect)
singleItem	Preview of a single item	At least 256px on the longest side, using original aspect ratio of the file, if applicable

For in-depth discussion of these options with many examples, refer to Guidelines for thumbnails in the documentation.

The second optional argument is called *requestedSize*, an integer that indicates the pixel length of the thumbnail's longest edge (width for images that are more wide than tall, height for those that are more tall than wide). By default, this does not guarantee that the returned image will be exactly this size. Instead, the APIs use this to determine how best to use existing cached thumbnails, so you might get back an image that is larger or smaller.

Generally speaking, it's best to set *requestedSize* to one of the sizes that the system already caches. For the square aspect views in the table above, these sizes are 16, 32, 48, 96, 256, 1024, and 1600. (I don't honestly know why the defaults are 40x40, but there you are.) For wide aspects, the sizes are 190, 266, 342, 532, and 1026. You'll find these described on the topic Accessing the file system efficiently, whose first section is on thumbnails.

The third optional argument is one or more ThumbnailOptions values combined with the bitwise OR operator (|). These options affect the speed of the request and the resulting image quality:

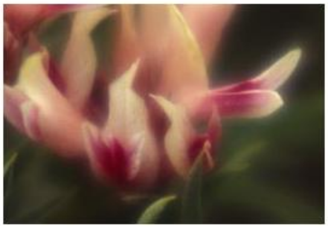
- none Use default behavior.
- resizeThumbnail Scale the thumbnail to match your *requestedSize*.
- useCurrentScale Increases *requestedSize* based on the pixel density of the display (that is, the scaling factor; see Chapter 8, "Layout and Views").

- returnOnlyIfCached Forces the API to fail if a thumbnail does not exist in the cache nor in the file itself. This prevents opening the full image file and potential downloads.

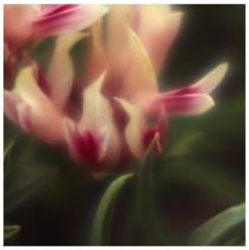
You can see some of these variations in action through the File and folder thumbnail sample. Most of the scenarios (1–5) use getThumbnailAsync for different libraries and ThumbnailMode settings and other options. In scenario 1, for example, you can choose from the picturesView, listView, and singleItem view mode (which is in the modes[modeselected] variable in the code below), toggle whether returnOnlyIfCached is set, and then choose an image for which to generate a ~200px thumbnail (js/scenario1.js):

```
var requestedSize = 200,
    thumbnailMode = modes[modeSelected],
    thumbnailOptions = Windows.Storage.FileProperties.ThumbnailOptions.useCurrentScale;

if (isFastSelected) {
    thumbnailOptions |= Windows.Storage.FileProperties.ThumbnailOptions.returnOnlyIfCached;
}

// File picker code omitted--choice is returned in 'file'
// Can also use getScaledImageAsThumbnailAsync here
file.getThumbnailAsync(thumbnailMode, requestedSize,
     thumbnailOptions).done(function (thumbnail) {
    if (thumbnail) {
        outputResult(file, thumbnail, modeNames[modeSelected], requestedSize);
    }

    // Error handling code omitted
});

function outputResult(item, thumbnailImage, thumbnailMode, requestedSize) {
    document.getElementById("picture-thumb-imageHolder").src =
        URL.createObjectURL(thumbnailImage, { oneTimeOnly: true });

    // Close the thumbnail stream once the image is loaded
    document.getElementById("picture-thumb-imageHolder").onload = function () {
        thumbnailImage.close();
    };
    document.getElementById("picture-thumb-modeName").innerText = thumbnailMode;
    document.getElementById("picture-thumb-fileName").innerText = "File used: " + item.name;
    document.getElementById("picture-thumb-requestedSize").innerText =
        "Requested size: " + requestedSize;
    document.getElementById("picture-thumb-returnedSize").innerText = "Returned size: "
        + thumbnailImage.originalWidth + "x" + thumbnailImage.originalHeight;
}
```

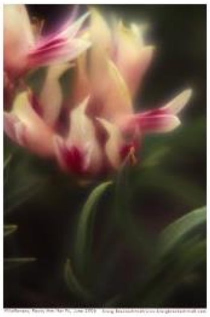
The output for each of the three modes (using the same original image) is as follows:



ThumbnailMode.picturesView
File used: wildflowers.jpg
Requested size: 200
Returned size: 266x182



ThumbnailMode.listView
File used: wildflowers.jpg
Requested size: 200
Returned size: 200x200



ThumbnailMode.singleItem
File used: wildflowers.jpg
Requested size: 200
Returned size: 168x256

You can see how the `picturesView` and `listView` modes automatically crop the image to maintain the aspect ratio implied by that mode. The `singleItem` mode, on the other hand, uses the original aspect ratio, so we see a full representation of the original (portrait) image.

Scenarios 2–5 are all variations on this same theme, showing, for example, how you get an icon thumbnail for something like an Excel document when using the `documentsView` mode. The one other bit to show is the use of `getScaledImageAsThumbnailAsync` in scenario 6 (js/scenario6.js), which effectively amounts to replacing `getThumbnailAsync` with `getScaledImageAsThumbnailAsync` in the code above. In fact, you can make this change throughout the sample and you'll see the same results for the most part—you just might get those results back more quickly, which is very helpful when retrieving thumbnails for a ListView.

What we do need to look at a bit more closely is the `StorageItemThumbnail` object we get as the result of these operations (in the `Windows.Storage.FileProperties` namespace). This is a rather rich object that contains quite a few methods and properties, because it's actually a derivative of `IRandomAccessStream`. The benefit of this is that you can toss this object to our old friend `URL.createObjectURL`, as shown in the code above, and it works as you expect. Otherwise, the members in which you're usually most interested (the thumbnail-specific ones) are:

- `close` Always call this when you're done using the thumbnail. Notice how the sample code calls this once the `img` element we're loading with the thumbnail has finished its work.
- `originalHeight`, `originalWidth` The nonscaled pixel dimensions of the thumbnail.
- `type` A value from `ThumbnailType` indicating whether it contains a thumbnail `image` or an `icon` representation.
- `returnedSmallerCachedSize` A Boolean indicating whether the returned thumbnail came from a cache with a size smaller than the *requestedSize*.

For the rest, refer to the `StorageItemThumbnail` documentation.

Tip As we saw in Chapter 7, "Collection Controls," the WinJS.UI.StorageDataSource object simplifies many of the details of setting up file queries over various libraries for use with a ListView control. Its loadThumbnail method also encapsulates many of the details we've seen here to help you easily load up a thumbnail for items in the collection. Refer to "A FlipView Using the Pictures Library" in Chapter 7 for more.

File Properties

Last but certainly not least is the StorageFile.properties property (say that ten times fast!), along with the getBasicPropertiesAsync method. To put it mildly, these are just the first of many doors that open up all kinds of deep information about files and media file in particular, as illustrated in Figure 11-11, along with the other direct properties and thumbnails. As you can see, alongside the direct properties of StorageFile, some extended properties are retrieved through getBasicPropertiesAsync and the media-specific methods of the properties object.

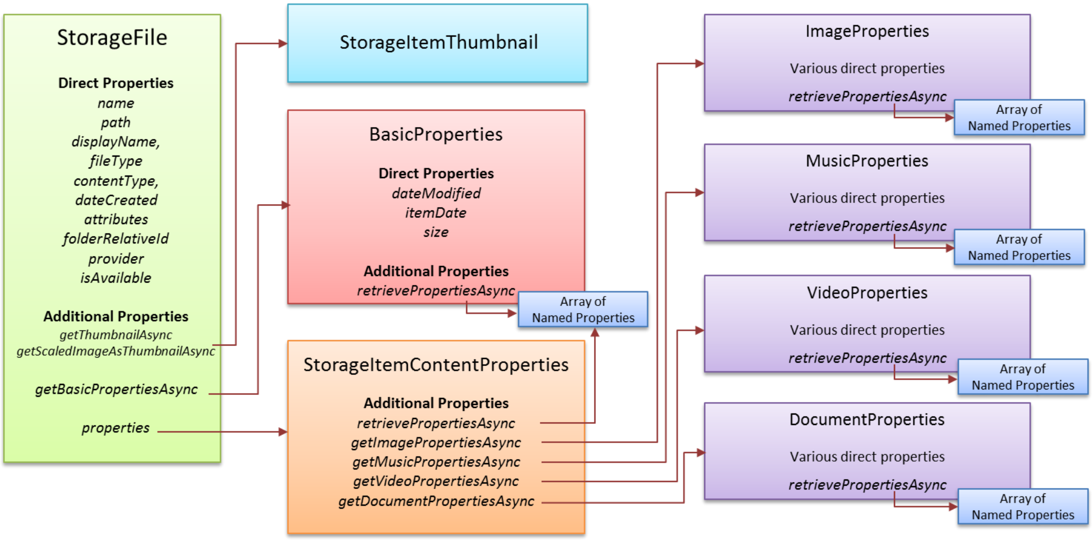


FIGURE 11-1 Relationships between the StorageFile object and metadata objects.

First, all of classes in Figure 11-11 come from the Windows.Storage.FileProperties namespace (except for StorageFile itself), so assume that is our context unless otherwise noted.

BasicProperties is the first one of interest, as we've already seen thumbnails in the previous section. This is what we get from StorageFile.getBasicPropertiesAsync, and it provides just three properties: dateModified is the last modified date to complement StorageFile.dateCreated, size is the size of the file, and itemDate contains the system's best attempt to find a relevant date for the file's *contents* based on other properties. For example, the relevant date for a picture or video is when it was taken; for music it's the release date. This is actually quite convenient because it relieves you from having to implement similar heuristics of your own and helps promote consistency across apps.

"But still," you might be saying to yourself, "all this a snoozer! An async call just to get an object with three properties?" Yes, it looks that way until you see that little retrievePropertiesAsync method—but let's come back to that in a moment because it shows up all over the place (as you can see in Figure 11-11) and is accompanied by another ubiquitous method, savePropertiesAsync (not shown in the figure).

StorageFile.properties contains a StorageItemContentProperties object that interestingly enough contains no direct properties! It only contains six methods—four of these retrieve media-specific properties, which will be described soon and which are especially helpful when creating gallery experiences over the user's media libraries or some other arbitrary folder. The other two methods are our friends retrievePropertiesAsync and savePropertiesAsync (through which you can access the same properties as the media-specific methods).

Every retrievePropertiesAsync method is capable of retrieving an array of name-value pairs for all kinds of other metadata related to files. The only argument you provide is an array of the property names you want where each name is a string that comes from a very extensive list of Windows Properties, such as *System.FileOwner* and *System.FileAttributes*. (Be aware that many of the listed properties don't apply to file system entities like *System.Devices.BatteryLife*.)

Tip To access the most common property names as strings, use the properties of the Windows.Storage.StorageProperties object, which has the benefit of providing auto-complete within Visual Studio.

An example of this is found in scenario 6 of the File access sample, which employs both getBasicPropertiesAsync and retrievePropertiesAsync to show the basic properties along with the last access date and the file owner (js/scenario6.js, where file is StorageFile):

```
var dateAccessedProperty = "System.DateAccessed";
var fileOwnerProperty    = "System.FileOwner";

file.getBasicPropertiesAsync().then(function (basicProperties) {
    outputDiv.innerHTML += "Size: " + basicProperties.size + " bytes<br />";
    outputDiv.innerHTML += "Date modified: " + basicProperties.dateModified + "<br />";

    // Get extra properties
    return file.properties.retrievePropertiesAsync([fileOwnerProperty, dateAccessedProperty]);
}).done(function (extraProperties) {
    var propValue = extraProperties[dateAccessedProperty];
    if (propValue !== null) {
        outputDiv.innerHTML += "Date accessed: " + propValue + "<br />";
    }
    propValue = extraProperties[fileOwnerProperty];
    if (propValue !== null) {
        outputDiv.innerHTML += "File owner: " + propValue;
    }
}
```

The result of retrievePropertiesAsync, in the extraProperties variable in the code above, is a simple Map collection. We met maps back in Chapter 6, in "Maps and Property Sets." A key point about a map is that you can access its members with the [ ] operator, as we see above, but a map is *not* an array and does not have array methods. If you want to iterate over a map, the best way is to pass the map to Object.keys and loop over that array. For example, the specific lookup code above could be replaced with the following:

```
Object.keys(extraProperties).forEach(function (key) {
    outputDiv.innerHTML += key + ": " + extraProperties[key] + "<br/>";
});
```

where you could use a separate map object to look up UI labels for the property name in key, of course.

> **Note** If a requested property is not available on the file, retrievePropertiesAsync will not include an entry for it in the map but the map's size property will still reflect the size of the original input array. For this reason, use Object.keys(<map>).length to determine the actual number of returned properties.

What's very useful about this is that the map from retrievePropertiesAsync is directly connected to the property story of the underlying file. This means you can modify its contents and add new entries, call savePropertiesAsync (with no arguments), and voila! You've just updated those properties on the file. This assumes, of course, that those properties are writeable and supported for the file type in question. If they're supported but read-only, my experience shows that savePropertiesAsync will ignore them. If they're not supported, on the other hand, the method will throw an exception with the message "the parameter is incorrect."

For example, the sample.dat file that is created in the File access sample doesn't support any writeable properties, so it's not a good test case. If you use an image file instead (like a JPEG), you can write properties such as *System.Keywords* or *System.Author*. (A good way to check what's writable is to right-click a file of some type in Windows Explorer, select Properties, and look at the Details tab to see what properties can be edited and which ones appear as read-only.)

To show a bit of code, assume that file here points to a JPEG, extraProperties came from a retrievePropertiesAsync call, and we want to add some keywords:

```
extraProperties.insert("System.Keywords", "sample keyword");
file.properties.savePropertiesAsync().done(function () {
    console.log("success");
});
```

If we'd made any other changes within extraProperties, those too would be saved. Alternately, we can pass savePropertiesAsync a Windows.Foundation.Collections.PropertySet object with those specific properties we want to set, in this case *System.Author*.

```
var propsToAdd = new Windows.Foundation.Collections.PropertySet()
propsToAdd.insert("System.Author", currentAuthor);
```

```
file.properties.savePropertiesAsync(propsToAdd).done(function () {
    console.log("success");
});
```

Here's another example showing how to make a file read-only through the *System.FileAttributes* property, where we OR in the value of 1 (FILE_ATTRIBUTE_READONLY in the Win32 file API):

```
var key = "System.FileAttributes";
var FILE_ATTRIBUTES_READONLY = 1;  //From the Win32 API
var file;

//Assign some StorageFile to the file variable

file.properties.retrievePropertiesAsync([key]).then(function (props) {
    if (props) {
        props[key] |= FILE_ATTRIBUTES_READONLY;
    } else {
        props = new Windows.Foundation.Collections.PropertySet();
        props.insert(key, FILE_ATTRIBUTES_READONLY);
    }

    return file.properties.savePropertiesAsync(props);
}).done(function () {
    //Any other action
});
```

We'll see a third example in the next section with an image file and the `ImageProperties` object.

Tip Avoid calling `savePropertiesAsync` when another save is still outstanding, such as running the second snippet above while the first has not yet completed. Doing so will throw an exception with the message "A method was called at an unexpected time." Instead, consolidate your property changes into a single call, or use a promise chain to run the async operations sequentially.

Media-Specific Properties

Alongside the `BasicProperties` class in `Windows.Storage.FileProperties` we also have those returned by the `StorageFile.properties.get*PropertiesAsync` methods: `ImageProperties`, `VideoProperties`, `MusicProperties`, and `DocumentProperties`. Though we've had to dig deep to find these, they each contain deeper treasure troves of information—and I do mean deep! The tables below summarize each of these in turn, and each object contains `retrievePropertiesAsync` and `savePropertiesAsync` methods, as we've seen, so that you can work with additional properties that aren't directly surfaced in the media-specific object.

Note that the links at the top of the table identify the most relevant groups of Windows properties.

ImageProperties	from StorageFile.properties.getImagePropertiesAsync	
Additional properties	System.Image, System.Photo, System.Media	
Property	**DataType**	**Applicable Windows Property**
title	String	System.Title
dateTaken	Date	System.Photo.DateTaken
latitude	Double (see below)	System.GPS.LatitudeDecimal, or combination of System.GPS.Latitude, System.GPS.LatitudeDenominator, System.GPS.LatitudeNumerator, and System.GPS.LatitudeRef
longitude	Double (see below)	System.GPS.LongitudeDecimal, or combination of System.GPS.Longitude, System.GPS.LongitudeDenominator, System.GPS.LongitudeNumerator, and System.GPS.LongitudeRef
cameraManufacturer	String	System.Photo.CameraManufacturer
cameraModel	String	System.Photo.CameraModel
width	Number in pixels	System.Image.HorizontalSize
height	Number in pixels	System.Image.VerticalSize
orientation	A FileProperties.PhotoOrientation object containing unspecified, normal, flipHorizontal, flipVertical, transpose, transverse, rotate90, rotate180, rotate270	System.Photo.Orientation
peopleNames	String vector	System.Photo.PeopleNames
keywords	String vector	System.Keywords
rating	Number (1-99 with 0 meaning "no rating")	System.Rating

VideoProperties	**from** StorageFile.properties.getVideoPropertiesAsync	
Additional properties	System.Video, System.Media, System.Image, System.Photo	
Property	**DataType**	**Applicable Windows Property**
title	String	System.Title
subtitle	String	System.Media.SubTitle
year	Number	System.Media.Year
publisher	String	System.Media.Publisher
rating	Number	System.Rating
width	Number in pixels	System.Video.FrameWidth
height	Number in pixels	System.Video.FrameHeight
orientation	A FileProperties.VideoOrientation object containing normal, rotate90, rotate180, rotate270	System.Photo.Orientation
duration	Number (in 100ns units, i.e., 1/10th milliseconds)	System.Media.Duration
bitrate	Number (in bits/second)	System.Video.TotalBitrate, System.Video.EncodingBitrate
directors	String vector	System.Video.Director
producers	String vector	System.Media.Producer
writers	String vector	System.Media.Writer
keywords	String vector	System.Keywords
latitude	Double (see below)	System.GPS.LatitudeDecimal, or combination of System.GPS.Latitude, System.GPS.LatitudeDenominator, System.GPS.LatitudeNumerator, and System.GPS.LatitudeRef

longitude	Double (see below)	System.GPS.LongitudeDecimal, or combination of System.GPS.Longitude, System.GPS.LongitudeDenominator, System.GPS.LongitudeNumerator, and System.GPS.LongitudeRef

MusicProperties	**from** StorageFile.properties.getMusicPropertiesAsync	
Additional properties	System.Music, System.Media	
Property	**DataType**	**Applicable Windows Property**
title	String	System.Title, System.Music.AlbumTitle
subtitle	String	System.Media.SubTitle
trackNumber	Number	System.Music.TrackNumber
year	Number	System.Media.Year
publisher	String	System.Media.Publisher
artist	String	System.Music.Artist, System.Music.DisplayArtist
albumArtist	String	System.Music.DisplayArtist (read), System.Music.AlbumArtist (write)
genre	String vector	System.Music.Genre
composers	String vector	System.Music.Composer
conductors	String vector	System.Music.Conductor
rating	Number (1-99 with 0 meaning "no rating")	System.Rating
duration	Number (in 100ns units, i.e., 1/10th milliseconds)	System.Media.Duration
bitrate	Number (in bits/second)	System.Video.TotalBitrate, System.Video.EncodingBitrate
producers	String vector	System.Media.Producer
writers	String vector	System.Media.Writer

DocumentProperties	**from** StorageFile.properties.getDocumentPropertiesAsync	
Additional properties	System	
Property	**DataType**	**Applicable Windows Property**
title	String	System.Title
Author	String vector	System.Author
keywords	String vector	System.Keywords
Comments	String	System.Comment

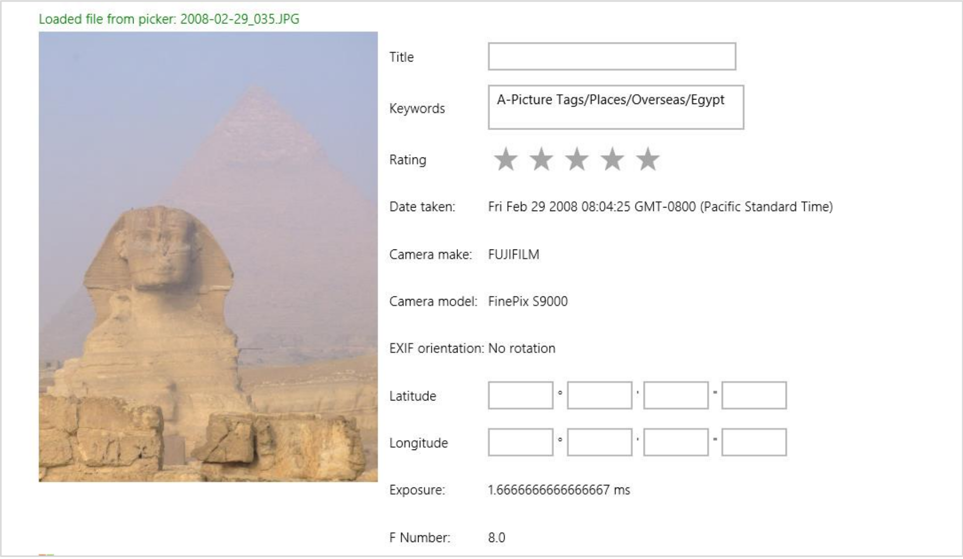
Note The latitude and longitude properties for images and video are double types but contain degrees, minutes, seconds, and a directional reference. The Simple imaging sample (in js/default.js) contains a helper function to extract the components of these values and convert them into a string:

```
"convertLatLongToString": function (latLong, isLatitude) {
    var reference;

    if (isLatitude) {
        reference = (latLong >= 0) ? "N" : "S";
    } else {
        reference = (latLong >= 0) ? "E" : "W";
    }
```

```
        latLong = Math.abs(latLong);
        var degrees = Math.floor(latLong);
        var minutes = Math.floor((latLong - degrees) * 60);
        var seconds = ((latLong - degrees - minutes / 60) * 3600).toFixed(2);

        return degrees + "°" + minutes + "\'" + seconds + "\"" + reference;
    }
```

To summarize, the sign of the value indicates direction. A positive value for latitude means North, negative means South; for longitude, positive means East, negative means West. The whole number portion of the value provides the degrees, and the fractional part contains the number of minutes expressed in base 60. Multiplying this value by 60 gives the whole minutes, with the remainder then containing the seconds. Although this floating-point value isn't all that convenient for UI output, as we see here, it's what you typically want for coordinate math and talking with various web services.

Speaking of the Simple imaging sample, it's one of a few samples in the Windows SDK that demonstrate working with these properties. Scenario 1 (js/scenario1.js) provides the most complete demonstration because you can choose an image file and it will load and display various properties, as shown in Figure 11-12. I can verify that the date, camera make/model, and exposure information are all accurate.



Loaded file from picker: 2008-02-29_035.JPG

Title

Keywords A-Picture Tags/Places/Overseas/Egypt

Rating ★ ★ ★ ★ ★

Date taken: Fri Feb 29 2008 08:04:25 GMT-0800 (Pacific Standard Time)

Camera make: FUJIFILM

Camera model: FinePix S9000

EXIF orientation: No rotation

Latitude ° ' "

Longitude ° ' "

Exposure: 1.6666666666666667 ms

F Number: 8.0

FIGURE 11-12 Image file properties in the Simple imaging sample, which is panned down to show the whole image and more of the property fields.

The sample's `openHandler` method is what retrieves these properties from the file using `StorageFile.properties.getImagePropertiesAsync` and `ImageProperties.retrievePropertiesAsync` for a couple of additional properties not already in `ImageProperties`. Then `getImagePropertiesForDisplay` coalesces these into a single object used by the sample's UI. Some lines are omitted in the code shown here:

```
var ImageProperties = {};

function openHandler() {
    // Keep data in-scope across multiple asynchronous methods.
    var file = {};

    Helpers.getFileFromOpenPickerAsync().then(function (_file) {
        file = _file;
        return file.properties.getImagePropertiesAsync();
    }).then(function (imageProps) {
        ImageProperties = imageProps;

        var requests = [
            "System.Photo.ExposureTime",       // In seconds
            "System.Photo.FNumber"            // F-stop values defined by EXIF spec
        ];

        return ImageProperties.retrievePropertiesAsync(requests);
    }).done(function (retrievedProps) {
        // Format the properties into text to display in the UI.
        displayImageUI(file, getImagePropertiesForDisplay(retrievedProps));
    });
}

function getImagePropertiesForDisplay(retrievedProps) {
    // If the specified property doesn't exist, its value will be null.
    var orientationText = Helpers.getOrientationString(ImageProperties.orientation);

    var exposureText = retrievedProps.lookup("System.Photo.ExposureTime") ?
        retrievedProps.lookup("System.Photo.ExposureTime") * 1000 + " ms" : "";

    var fNumberText = retrievedProps.lookup("System.Photo.FNumber") ?
        retrievedProps.lookup("System.Photo.FNumber").toFixed(1) : "";

    // Omitted: Code to convert ImageProperties.latitude and ImageProperties.longitude to
    // degrees, minutes, seconds, and direction

    return {
        "title": ImageProperties.title,
        "keywords": ImageProperties.keywords, // array of strings
        "rating": ImageProperties.rating, // number
        "dateTaken": ImageProperties.dateTaken,
        "make": ImageProperties.cameraManufacturer,
        "model": ImageProperties.cameraModel,
        "orientation": orientationText,
        // Omitted: lat/long properties
        "exposure": exposureText,
        "fNumber": fNumberText
    };
}
```

Most of the `displayImageUI` function to which these properties are passed just copies the data into various controls. It's good to note again, though, that displaying the picture itself is easily accomplished with our pal, `URL.createObjectURL`, but as we learned earlier in this chapter, it would be better to avoid loading the whole image file and instead use a thumbnail from StorageFile.getScaledImageThumbnailAsync. That is, change this line of code in displayImageUI (js/scenario1.js):

```
id("outputImage").src = window.URL.createObjectURL(file, { oneTimeOnly: true });
```

to the following:

```
var mode = Windows.Storage.FileProperties.ThumbnailMode.singleItem;
file.getScaledImageAsThumbnailAsync(mode, 500).done(function (thumb) {
    var img = id("outputImage");
    img.src = URL.createObjectURL(thumb, { oneTimeOnly: true });
    img.onload = function () {
        thumb.close();
    }
});
```

A bit more code to write, but definitely more efficient!

For MusicProperties a small example can be found in the Playlist sample and another in the Configure keys for media sample, both of which we'll see in Chapter 13. The latter especially shows how to use the music properties to obtain album art. As for VideoProperties and DocumentProperties, the SDK doesn't have samples for these, but working with them follows the same pattern as shown above for ImageProperties.

As for saving properties, the Simple Imaging sample delivers there as well, also in scenario 1. As the fields shown earlier in Figure 11-12 are editable, the sample provides an Apply button (panned off the top of the screen) that invokes the applyHandler function below to write them back to the file (js/scenario1.js):

```
function applyHandler() {
    ImageProperties.title = id("propertiesTitle").value;

    // Keywords are stored as an array of strings. Split the textarea text by newlines.
    ImageProperties.keywords.clear();
    if (id("propertiesKeywords").value !== "") {
        var keywordsArray = id("propertiesKeywords").value.split("\n");

        keywordsArray.forEach(function (keyword) {
            ImageProperties.keywords.append(keyword);
        });
    }

    var properties = new Windows.Foundation.Collections.PropertySet();

    // When writing the rating, use the "System.Rating" property key.
    // ImageProperties.rating does not handle setting the value to 0 (no stars/unrated).
    properties.insert("System.Rating", Helpers.convertStarsToSystemRating(
        id("propertiesRatingControl").winControl.userRating
        ));

    // Code omitted: convert discrete latitude/longitude values from the UI into the
    // appropriate forms needed for the properties, and do some validation; the end result
```

```
        // is to store these in the properties list
        properties.insert("System.GPS.LatitudeRef", latitudeRef);
        properties.insert("System.GPS.LongitudeRef", longitudeRef);
        properties.insert("System.GPS.LatitudeNumerator", latNum);
        properties.insert("System.GPS.LongitudeNumerator", longNum);
        properties.insert("System.GPS.LatitudeDenominator", latDen);
        properties.insert("System.GPS.LongitudeDenominator", longDen);

        // Write the properties array to the file
        ImageProperties.savePropertiesAsync(properties).done(function () {
            // ...
        }, function (error) {
            // Some error handling as some properties may not be supported by all image formats.
        });
    }
```

A few noteworthy features of this code include the following:

- It separates keywords in the UI control and separately appends each to the keywords vector.
- It creates a new Windows.Foundation.Collections.PropertySet, which is the expected input to savePropertiesAsync. Remember from Chapter 6 that the PropertySet is the only WinRT collection class that you can instantiate directly, as we must do here.
- The Helpers.convertStarsToSystemRating method (also in js/default.js) converts between 1–5 stars, as used in the WinJS.UI.Rating control, to the *System.Rating* value that uses a 1–99 range. The documentation for System.Rating specifically indicates this mapping.

In general, all the detailed information you want for any particular Windows property can be found on the reference page for that property. Again start at Windows Properties and drill down from there.

Folders and Folder Queries

Now that we've seen everything we can do with a StorageFile and file properties, it's time to turn our attention to the folders and libraries in which those files live and the StorageFolder and StorageLibrary objects that represent them.

As with StorageFile, we've already peeked at some of the basic StorageFolder methods in Chapter 10, such as createFileAsync, createFolderAsync, getFileAsync, getFolderAsync, getItemAsync, getFolderFromPathAsync, deleteAsync, and renameAsync. It also has many of the same properties as a file, including name, path, displayName, displayType, attributes, folderRelativeId, dateCreated, provider, and properties. It also shares a number of identical methods: isEqual, isOfType, getThumbnailAsync, getScaledImageAsThumbnailAsync, and getBasicPropertiesAsync. Everything about these is the same as for files, so please refer back to "StorageFile Properties and Metadata" for the details.

Be mindful, though, that the Windows properties you can retrieve and modify on a folder differ from those supported by files. For example, you can use StorageFolder.properties.retrievePropertiesAsync for the *System.FreeSpace* property and you'll actually get the free space on the drive where the folder lives. Pretty cool, eh?

And here are the unique aspects of StorageFolder:

- tryGetItemAsync Identical to getItemAsync (to retrieve a contained item) except that it results in null (to your completed handler) for the most common errors instead of calling your error handler. This can simplify app logic. getItemAsync, on the other hand, will succeed if the item exists invokes your error handler for all error cases. For a simple demonstration, see scenario 11 of the File access sample.
- getFilesAsync, getFoldersAsync, and getItemsAsync These are the basic methods to enumerate the immediate contents of a folder depending on whether you want files only, folders only, or both, respectively. Each method results in a vector of the appropriate object type—StorageFile, StorageFolder, or StorageItem—as discussed in "Simple Enumeration and Common Queries" below.
- Any *Query method or property All of these members deal with file queries, which is how you do deep enumerations of folder contents based on Windows properties, drawing on the power of the system indexer. This also includes several overloads of getFilesAsync and getFoldersAsync. We'll talk of all this under both "Simple Enumeration and Common Queries" and "Custom Queries."

Before that, however, let's spend a few minutes on the known folders and the StorageLibrary object, as well as working with removable storage—these are special cases of handling containers for files and folders. (The special Windows.Storage.DownloadsFolder mentioned at the beginning of this chapter in "The Big Picture of User Data" doesn't need any more explanation.)

Sidebar: Indexing App Content

The query capabilities that we'll be learning about here draw on the system indexer, which works automatically on file system content. Beyond this, the API in Windows.Storage.Search also gives you the ability to easily add Windows properties to files along with the means to index app content that isn't stored in files at all. Both topics are covered in Chapter 15, "Contracts," in the discussion on Search, specifically in "Indexing and Searching Content." For now, just check out the Indexer sample where these features are demonstrated.

KnownFolders and the StorageLibrary Object

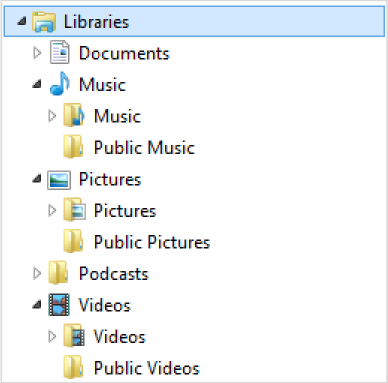
As you already know, `Windows.Storage.KnownFolders` gives you direct access to the `StorageFolder` objects for various user data locations. The `picturesLibrary`, `musicLibrary`, and `videosLibrary` are the obvious ones, but we haven't yet mentioned a number of others, some of which depend on the same capabilities in your manifest. All of them are summarized in the following table:

Folder	Required Capability	Description
`cameraRoll`	Pictures library.	The *Camera roll* folder in the user's Pictures library.
`documentsLibrary`	Documents library, which has additional requirements; see "The Big Picture of User Data" at the beginning of the chapter.	The user's local Documents folder.
`homeGroup`	Music library, Pictures library, or Videos library.	Container for the user's HomeGroup items. Refer to the HomeGroup app sample.
`mediaServerDevices`	Music library, Pictures library, or Videos library.	Container for connected media servers; see Chapter 13.
`musicLibrary`	Music library.	The user's root Music library.
`picturesLibrary`	Pictures library.	The user's root Pictures library.
`playlists`	Music library.	Default storage location for playlists; see Chapter 13.
`removableDevices`	Removable storage plus at least one file type association; see "Removable Storage" below.	A folder containing subfolders for each attached device.
`savedPictures`	Pictures library.	Same as the Pictures library.
`videosLibrary`	Videos library.	The user's root Videos library.

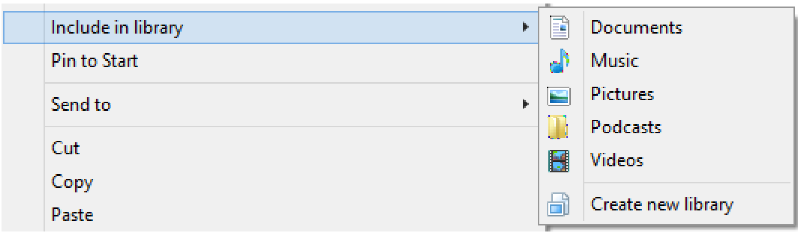
Again, only request specific library access if you're going to work within any of these libraries outside of the file picker—for example, when you want to create your own UI to show folder contents (which the file pickers do very well already).

As you will rightly expect, some of these folders do not support the creation of new files or folders within them—specifically, `homeGroup`, `mediaServerDevices`, and `removableDevices`—though in the latter you can generally create subfolders within the folder for any one device. As for the rest, they behave just like other local folders. The `cameraRoll` and `savedPictures` folders, for their part, are alternate routes into the Pictures library that come from the ongoing work to bring the Windows and Windows Phone platforms together. They don't otherwise have any special meaning.

With the three primary media libraries (and documents), you also have the ability to programmatically include other folders in those libraries. This is different, mind you, from creating a subfolder in that library directly—a media folder as Windows sees it is a list of any number of other folders on the file system that are then treated as a single entity. You see this in Windows Explorer under Libraries (ignore the legacy Podcasts artifact):



In Windows Explorer, if you right-click a folder and select Include In Library, you'll see a popup menu with these same choices:



The `Windows.Storage.StorageLibrary` object gives you access to these capabilities from within an app. To obtain a `StorageLibrary`, call the static `Windows.Storage.StorageLibrary.getLibraryAsync` method with a value from `Windows.Storage.KnownLibraryId` enumeration, as shown here in scenario 1 of the Library management sample (js/S1_AddFolder.js):

```
Windows.Storage.StorageLibrary.getLibraryAsync(Windows.Storage.KnownLibraryId.pictures)
    .then(function (library) {
    // ...
    });
}
```

where the available options in `KnownLibraryId` are `pictures`, `music`, `videos`, and `documents`. Once you have a `StorageLibrary` object, you have these members to play with:

- `folders`   A read-only but observable vector of `StorageFolder` objects in the library.
- `saveFolder`   The `StorageFolder` of the default save location for the library (read-only).
- `requestAddFolderAsync`   Invokes the folder picker UI that automatically shows only those locations eligible to add to the library (excluding other apps, for instance, and removable storage, but it *does* include OneDrive given that it has a local folder with at least placeholder files). Once the user selects a folder or cancels, the API will complete with the selected `StorageFolder` or `null`, respectively.

- requestRemoveFolderAsync Prompts the user to confirm removal of a given StorageFolder from the library; the Boolean result indicates whether the user consented to the action.
- definitionChanged Fired when the contents of the folders collection have been changed either through Windows Explorer or another app. Uses this to update your UI as needed.

The Library management sample shows all of these features except for saveFolder. To complete scenario 1, it calls requestAddFolderAsync (js/S1_AddFolder.js):

```
Windows.Storage.StorageLibrary.getLibraryAsync(Windows.Storage.KnownLibraryId.pictures)
    .then(function (library) {
        return library.requestAddFolderAsync();
    }).done(function (folderAdded) {
    // ...
    });
}
```

Scenario 3 does the opposite with requestRemoveFolderAsync (js/S3_RemoveFolder.js):

```
var folderToRemove =
    picturesLibrary.folders[document.getElementById("foldersSelect").selectedIndex];
picturesLibrary.requestRemoveFolderAsync(folderToRemove).done(function (folderRemoved) {
    // ...
});
```

where the foldersSelect control is populated from the folders collection (js/S3_RemoveFolder.js):

```
function fillSelect() {
    var select = document.getElementById("foldersSelect");
    select.options.length = 0;
    picturesLibrary.folders.forEach(function (folder) {
        var option = document.createElement("option");
        option.textContent = folder.displayName;
        select.appendChild(option);
    });
}
```

Nearly identical code is found in scenario 2 (js/S2_ListFolders.js), which just outputs the list of current folders to the display.

If you add and remove folders in this sample, also run the built-in Pictures app and you can see the contents of those folders appearing and disappearing. This is because the Pictures app is using the definitionChanged event to keep itself updated. The Library management sample does the same just to refresh its drop-down list in scenario 3 and refreshes its display in scenario 2 (js/S2_ListFolders.js):

```
Windows.Storage.StorageLibrary.getLibraryAsync(Windows.Storage.KnownLibraryId.pictures)
    .then(function (picturesLibrary) {
        picturesLibrary.addEventListener("definitionchanged", updateListAndHeader);
        updateListAndHeader();// Refresh the display
    });
```

You can test this by adding and removing folders to the Pictures library in Windows Explorer while the sample is running.

Hint Remember that `definitionChanged` is a WinRT event, so be sure to remove its listeners appropriately. Note that the Library management sample does *not* do this properly.

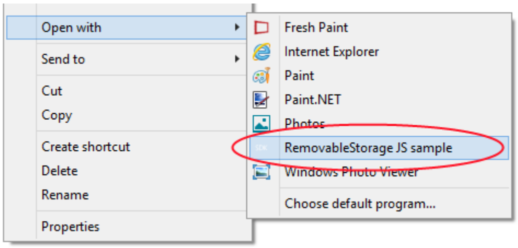
Removable Storage

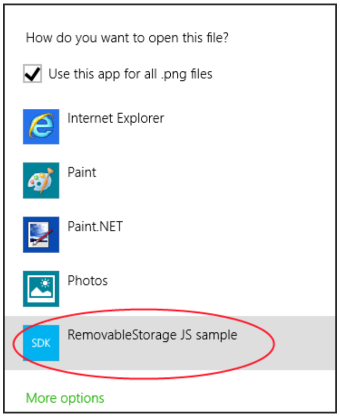
As with the media libraries, programmatic access to the user's removable storage devices is controlled by a capability declaration plus one or more file type associations. This means that you cannot simply enumerate the contents of these folders directly or write whatever files you want therein. Put another way, going directly to `Windows.Storage.KnownFolders.removableDevices` is only something you do when you're looking for a specific file type. For example, a Photo management app can look here for cameras or USB drives from which to import the image files it supports.

If you simply want to allow the user to open or save files on removable devices, the capability and the `removableDevices` folder isn't what you need: use the file picker instead. What you *don't* want to do is create a file type association that you don't support, because users will attempt to launch your app through those associations and will reflect their disappointment in your Store ratings!

Tip One reason you might think about removable storage is to let the user save files to an SD card. In this case, it's better for the user to directly configure PC Settings > PC and Devices > Devices > Default Save Locations, which is at the very bottom of the Devices page and is easy to overlook.

That said, let's assume that the capability is appropriate for what you want to do. If you load the Removable storage sample in Visual Studio and run it, you can see the effect of it associating itself with .gif, .jpg, and .png files. As a result, it shows up in Open With lists such as the context menu of Windows Explorer and the default program selector:




How do you want to open this file?

☑ Use this app for all .png files

How you create the appropriate declarations and handle file activation is a subject we'll return to later in "File Activation and Association." For now, assuming that you've done all that properly, your app will be able to get to the removableDevices folder. The sample (scenario 1) just uses the StorageFolder.getFoldersAsync method to list the connected devices. Scenarios 2 and 3 send an image to and retrieve an image from a selected device, where the device names themselves are obtained from APIs in the Windows.Devices namespace, as we'll see in Chapter 17, "Devices and Printing." You don't have to go to that extent, however, because the removableDevices folder already gives you access to the StorageFolder objects you need for the same purposes.

Scenario 4 demonstrates handling Auto Play activation, which we'll also return to in "File Activation and Association." In such cases you'll likely query the contents of the removable device in question to create a list of the files you care about, and for that we need to look at file queries.

Simple Enumeration and Common Queries

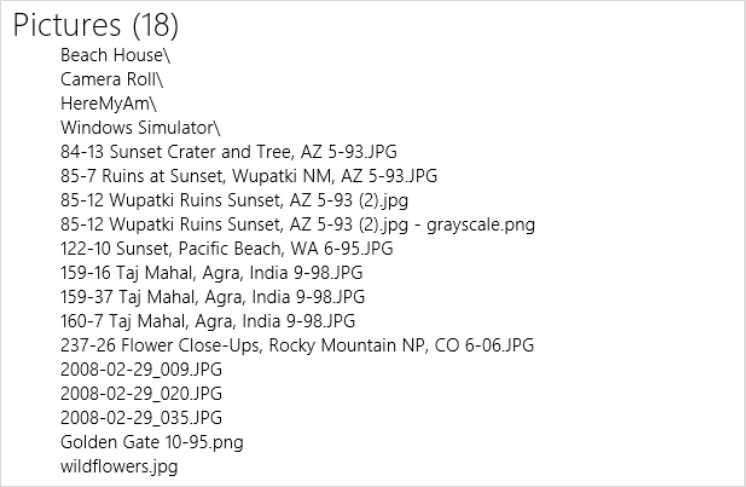
A simple or *shallow* enumeration of folder contents happens through the variants of getFilesAsync, getFoldersAsync, and getItemsAsync in the StorageFolder object, which take no arguments. Each method results in a vector of the appropriate item types: getFilesAsync enumerates files only and provides a vector of StorageFile objects; getFolderAsync enumerates only immediate child folders in a vector of StorageFolder objects; and getItemsAsync provides a vector of both together, each represented by a StorageItem object. Note that getItemsAsync has a variant, getItemsAsync-(*<startIndex>, <maxItemsToRetrieve>*), with which you can do a partial enumeration. This is especially helpful when dealing with folders that contains a few hundred items or more such that you can bring items into memory only when they come into view.

Once you receive one of the vectors, you can iterate over it as you would an array, as a vector is projected into JavaScript as such. With the StorageFile, StorageFolder, and StorageItem objects you can extract their display names and thumbnails to create a gallery view, present a more compact list to the user, or do whatever else you want. Note that the various permissions and manifest capabilities do not affect enumeration: if you were able to acquire the root StorageFolder, you have programmatic permission to enumerate its contents. (For removable storage and the documents library, however, enumeration *is* automatically limited to the file types you declare in the manifest.)

The Folder enumeration sample shows us these simple use cases. Scenario 1 calls getItemsAsync on the Pictures library:

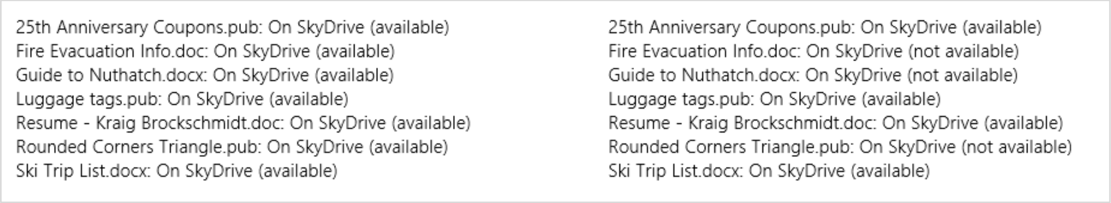
```
picturesLibrary.getItemsAsync().done(function (items) {
    // Output all contents under the library group
    outputItems(group, items);
});
```

where the outputItems function just iterates the resulting list and creates some DOM elements to show their names (the group variable just the header element with the count):



```
Pictures (18)
        Beach House\
        Camera Roll\
        HereMyAm\
        Windows Simulator\
        84-13 Sunset Crater and Tree, AZ 5-93.JPG
        85-7 Ruins at Sunset, Wupatki NM, AZ 5-93.JPG
        85-12 Wupatki Ruins Sunset, AZ 5-93 (2).jpg
        85-12 Wupatki Ruins Sunset, AZ 5-93 (2).jpg - grayscale.png
        122-10 Sunset, Pacific Beach, WA 6-95.JPG
        159-16 Taj Mahal, Agra, India 9-98.JPG
        159-37 Taj Mahal, Agra, India 9-98.JPG
        160-7 Taj Mahal, Agra, India 9-98.JPG
        237-26 Flower Close-Ups, Rocky Mountain NP, CO 6-06.JPG
        2008-02-29_009.JPG
        2008-02-29_020.JPG
        2008-02-29_035.JPG
        Golden Gate 10-95.png
        wildflowers.jpg
```

Skipping to scenario 4, we find an example of checking the `StorageFile.isAvailable` flag for enumerated items. This scenario lets you select a folder through the folder picker and then iterates the results to show the file's `displayName`, the `provider.displayName`, and the file's availability. Doing this for one of my OneDrive folders, I get the results below left when I have connectivity whereas I see those below right when I'm offline or on a metered network, as described by the table in "Availability" earlier, because those files are marked online-only:



```
25th Anniversary Coupons.pub: On SkyDrive (available)           25th Anniversary Coupons.pub: On SkyDrive (available)
Fire Evacuation Info.doc: On SkyDrive (available)               Fire Evacuation Info.doc: On SkyDrive (not available)
Guide to Nuthatch.docx: On SkyDrive (available)                 Guide to Nuthatch.docx: On SkyDrive (not available)
Luggage tags.pub: On SkyDrive (available)                       Luggage tags.pub: On SkyDrive (available)
Resume - Kraig Brockschmidt.doc: On SkyDrive (available)        Resume - Kraig Brockschmidt.doc: On SkyDrive (available)
Rounded Corners Triangle.pub: On SkyDrive (available)           Rounded Corners Triangle.pub: On SkyDrive (not available)
Ski Trip List.docx: On SkyDrive (available)                     Ski Trip List.docx: On SkyDrive (available)
```

I skipped ahead to scenario 4 first because it gives us our first taste of file and folder queries, specifically the function `StorageFolder.createFileQuery`, which does exactly the same thing as `getItemsAsync` when called with no arguments:

```
var query = folder.createFileQuery();
query.getFilesAsync().done(function (files) {
    files.forEach(function (file) {
    // ...
});
```

What comes back from `createFileQuery` is now a different object, a `StorageFileQueryResult` (in the `Windows.Storage.Search` namespace). Two parallel methods, `createFolderQuery` and `createItemQuery`, return a `StorageFolderQueryResult` and a `StorageItemQueryResult`, respectively.

As you can see in the code above, the StorageFileQueryResult has a getFilesAsync that behaves exactly like its namesake in StorageFolder (resulting in a StorageFile vector). StorageFolderQueryResult, for its part, has a getFoldersAsync (resulting in a StorageFolder vector) and StorageItemQueryResult has a getItemsAsync. And all these methods have variants to also return an index-based subset of results instead of the whole smash.

The three objects then have the rest of their members in common:

- folder A property containing the root StorageFolder in which the query was run.
- getItemCountAsync Retrieves the number of results from the query.
- contentsChanged An event that's fired when changes to the file system affect query results. This prompts an app displaying those contents to refresh itself. (If an app is suspended when this happens, all changes that take place during suspension are consolidated into a single event.)
- getCurrentQueryOptions, applyNewQueryOptions Retrieves and modifies the QueryOptions object used to define the query. Calling applyNewQueryOptions fires the optionsChanged event. See "Custom Queries" later on for using query options.
- findStartIndexAsync and getMatchingPropertiesWithRanges Used to identify exactly where matches in a query are located; also described in "Custom Queries."

So now our enumerations start to become more interesting! The whole idea of a query, after all, is to retrieve a subset of items based on certain criteria. Using the full query API that we'll see in the next section, the sky is really the limit here! But the designers of the WinRT API also understood that certain queries are the ones that most apps will care about—including deep rather than shallow queries—so they made is easy to execute them without having to ponder the fine details.

You do this through variants of the createFileQuery and createFolderQuery methods (no variant exists for items):

- createFileQuery(CommonFileQuery) Takes a value from the CommonFileQuery enumeration (in Windows.Storage.Search).
- createFolderQuery(CommonFolderQuery) Takes a value from the CommonFolderQuery enumeration (also in Windows.Storage.Search).

Both support an option called defaultQuery, which returns the same vector as getFilesAsync() and getFoldersAsync(), respectively, using a shallow enumeration. All other options, shown in the tables below, perform a deep enumeration. Furthermore, all queries can be run in any library or HomeGroup folder; CommonFileQuery.orderByName and orderBySearchRank can also be run in any folder. Refer also to Windows Properties for documentation on *System.ItemNameDisplay* and so forth.

CommonFileQuery	Windows Properties used in query
`orderbyName`	Sorted by the *System.ItemNameDisplay* property (this name is appropriate for use in UI).
`orderBySearchRank`	Sorted by the *System.Search.Rank* and then by *System.DateModified* properties.
`orderByTitle`	Sorted by the *System.Title* property.
`orderByDate`	Sorted by *System.ItemDate*, which varies with the type of file (e.g., *System.Photo.DateTaken* for images, *System.Media.DateReleased* for music).
`orderByMusicProperties`	Sorted by these four properties in this order: *System.Music.DisplayArtist*, *System.Music.AlbumTitle*, *System.Music.TrackNumber*, and *System.Title*.

CommonFolderQuery	Windows properties used in query
`groupByType`	Grouped *System.ItemTypeText*, with one virtual folder per type.
`groupByTag`	Grouped by *System.Keywords*, with one virtual folder per keyword; files with multiple tags will appear in multiple folders.
`groupByAuthor`	Grouped by *System.Author*, with one virtual folder per author; files with multiple authors will appear in multiple folders.
`groupByYear`	Grouped by the year in *System.ItemDate*, with one virtual folder per year.
`groupByMonth`	Grouped by the month in *System.ItemDate*, with one virtual folder per month.
`groupByPublishedYear`	Grouped by the year in *System.Media.Year*, with one virtual folder per year.
`groupByRating`	Grouped by *System.Rating* with one virtual folder per rating.
`groupByAlbum`	Grouped by *System.Music.AlbumTitle* with one virtual folder per title.
`groupByArtist`	Grouped by *System.Music.Artist* with one virtual folder per artist.
`groupByAlbumArtist`	Grouped by *System.Music.AlbumArtist* with one virtual folder per album artist.
`groupByComposer`	Grouped by *System.Music.Composer*, with one virtual folder per composer; files with multiple composers will appear in multiple folders.
`groupByGenre`	Grouped by *System.Music.Genre*, with one virtual folder per genre.

Clearly, many of the common folder queries (and one common file query) apply to music specifically, but others apply generally. However, *not every location supports every type of file query*—those that don't will throw a "parameter is incorrect" exception, which can be confusing if you're not prepared for it! (Query support is related to the indexing status of the location in question; libraries are indexed by default.)

The way you check for support is through `StorageFolder.isCommonFileQuerySupported` and `isCommonFolderQuerySupported`, to which you pass the desired `CommonFileQuery` and `CommonFolderQuery` value you want to test.

To help you play with this, I've modified scenario 4 of the Folder enumeration sample that's in this chapter's companion content. I've added a drop-down list through which you can select which `CommonFolderQuery` to run; once you've chosen a folder, it also checks if the query is supported:

```
var select = document.getElementById("selectQuery");
var selectedQuery = queries[select.selectedIndex];

if (!folder.isCommonFileQuerySupported(selectedQuery)) {
    //Show message in output if not supported and return.
    return;
}

//Run query as usual
```

Assuming the query is supported and succeeds when run, the result is just a flat list of files (a StorageFile vector) that you can easily iterate.

Folder queries are a little more complicated because the StorageFolderQueryResult.getFoldersAsync method gives you a StorageFolder vector, where each one is a *virtual folder* that's used to group files according to the query. That is, each folder does not represent an actual folder on whatever location was queried but is simply used to organize the results and help you present them in your UI. The displayName and other properties of each StorageFolder can be used to create group headings, and the files within each folder that you enumerate with StorageFolder.getFilesAsync are that group's contents.

Scenario 2 of the Folder enumeration sample demonstrates this with groupByMonth, groupByTag, and groupByRating queries on the Pictures library. It runs each query as follows, using the same processing code for the results (js/scenario2.js; I've shortened one variable name for brevity):

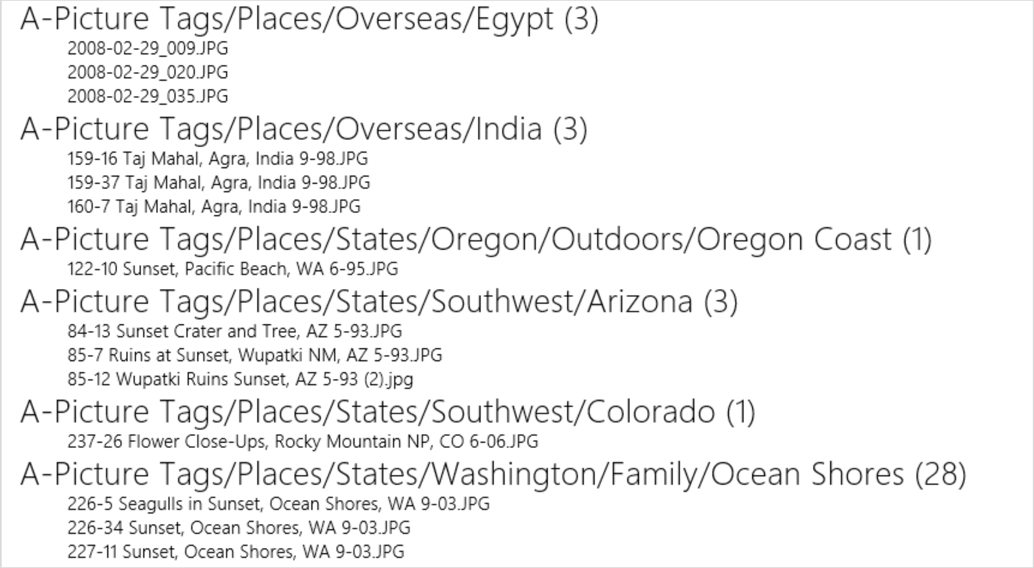
```
var pix = Windows.Storage.KnownFolders.picturesLibrary;
var query = pix.createFolderQuery(Windows.Storage.Search.CommonFolderQuery.groupByTag);
query.getFoldersAsync().done(outputFoldersAndFiles);
```

The query variable, to be clear, is the StorageFolderQueryResult object, and calling its getFolderAsync is what performs the actual enumeration. The outputFoldersAndFiles function—which is a completed handler—receives and iterates the resulting StorageFolder vectors, calling getFilesAsync for each and joining the resulting promises. It then processes each folder's results when they're all ready to show the output (js/scenario2.js):

```
function outputFoldersAndFiles(folders) {
    // Add all file retrieval promises to an array of promises
    var promises = folders.map(function (folder) {
        return folder.getFilesAsync();
    });
    // Aggregate the results of multiple asynchronous operations
    // so that they are returned after all are completed. This
    // ensures that all groups are displayed in order.
    WinJS.Promise.join(promises).done(function (folderContents) {
        for (var i in folderContents) {
            // Create a group for each of the file groups
            var group = outputResultGroup(folders.getAt(i).name);
            // Add the group items in the group
            outputItems(group, folderContents[i]);
        }
    });
}
```

This is a marvelously concise piece of code, so let me explain what's happening. First, folders is the StorageFolder vector, and because we can treat it as an array we can use folders.map to execute a function for each folder in the vector. The promises variable is an array of promises, one for each folder's getFilesAsync. Passing this to WinJS.Promise.join, if you remember from Chapter 3, "App Anatomy and Performance Fundamentals," returns a promise that is fulfilled when all the promises in the array are fulfilled, so this effectively waits for each one to complete.

The completed handler for `join` then receives an array that contains all the results of the individual promises. This handler then simply iterates that array, calling the sample's `outputResultGroup` and `outputItems` methods for each set, which just build up a DOM tree for the display. The results for my pictures library with `groupByTag` are as follows (revealing a small extent of my travels!):



```
A-Picture Tags/Places/Overseas/Egypt (3)
    2008-02-29_009.JPG
    2008-02-29_020.JPG
    2008-02-29_035.JPG
A-Picture Tags/Places/Overseas/India (3)
    159-16 Taj Mahal, Agra, India 9-98.JPG
    159-37 Taj Mahal, Agra, India 9-98.JPG
    160-7 Taj Mahal, Agra, India 9-98.JPG
A-Picture Tags/Places/States/Oregon/Outdoors/Oregon Coast (1)
    122-10 Sunset, Pacific Beach, WA 6-95.JPG
A-Picture Tags/Places/States/Southwest/Arizona (3)
    84-13 Sunset Crater and Tree, AZ 5-93.JPG
    85-7 Ruins at Sunset, Wupatki NM, AZ 5-93.JPG
    85-12 Wupatki Ruins Sunset, AZ 5-93 (2).jpg
A-Picture Tags/Places/States/Southwest/Colorado (1)
    237-26 Flower Close-Ups, Rocky Mountain NP, CO 6-06.JPG
A-Picture Tags/Places/States/Washington/Family/Ocean Shores (28)
    226-5 Seagulls in Sunset, Ocean Shores, WA 9-03.JPG
    226-34 Sunset, Ocean Shores, WA 9-03.JPG
    227-11 Sunset, Ocean Shores, WA 9-03.JPG
```

As for scenario 3 of the sample, that brings us into the matter of query options, which are discussed next.

Custom Queries

Now that we've seen the basics of enumerating folder contents with common queries and processing their results, we're ready to look at the full query capabilities offered by WinRT wherein you construct a custom query from scratch using a `QueryOptions` object. (The common queries are just shortcuts for typical cases to save you the trouble.) A custom query basically gives you all the capabilities that you have through the search features of the desktop Windows Explorer, using whatever Windows Properties you require, especially those you use to index your own app content.

As with the common queries, not all locations support custom queries, so once you've built the `QueryOptions` you want to call `StorageFolder.areQueryOptionsSupported` with that object. It if returns `true`, you can proceed. If you attempt to query a folder with unsupported options, expect a "parameter is incorrect" exception.

Something else you might want to know ahead of time is whether the folder itself has been indexed, which greatly affects the potential speed of the query. You do this through `StorageFolder.getIndexedStateAsync`. This results in an `IndexedState` value: `notIndexed`, `partiallyIndexed`, fullyIndexed, and unknown. Do note that a fully indexed folder could yet contain nonindexed folders, but generally speaking an indexed folder will produce results more quickly than the others.

The next step is to create the query by passing your query options to one of these StorageFolder methods: createFileQueryWithOptions, createFolderQueryWithOptions, and createItemQueryWithOptions. These result in StorageFileQueryResult, StorageFolderQueryResult, and StorageItemQueryResult objects, as already discussed. With these you can enumerate the results however you need, just as you do when you use a common query.

If you remember from earlier, the getCurrentQueryOptions and applyNewQueryOptions methods of Storage[File | Folder | Item]QueryResult work with the QueryOptions that were used to create the query. The *get* method retrieves the QueryOptions object, obviously, and calling the *apply* method will change the query mid-flight, firing the query result's optionsChanged event.

Ultimately, these queries involve what are known as Advanced Query Syntax (AQS) strings that are capable of identifying and describing as many specific criteria you desire. Each criteria is a Windows Property (again see Windows properties for the reference) such as *System.ItemDate*, *System.Author*, *System.Keywords*, *System.Photo.LightSource*, and so on.[85] Each property can contain a target value such as *System.Author(Patrick OR Bob)* and *System.ItemType: "mp3"*, and terms can be combined with AND (which is implicit), OR, and NOT operators, as in *System.Keywords: "needs review" AND (System.ItemType: ".doc" OR System.ItemType: ".docx")*.

Tip The AND, OR, and NOT operators must be in uppercase or else they are interpreted as a keyword. Also note that quotation marks aren't strictly necessary.

Simply said, an AQS string is exactly what you can type into the search control of Windows Explorer. You can also try out AQS strings through the Programmatic file search sample, which we'll see in a moment once we look at the QueryOptions object that is so central to this process (it's where you provide AQS strings).

First, there are three QueryOptions constructors:

- new QueryOptions() Creates an uninitialized object.
- new QueryOptions(<CommonFolderQuery>) Creates an object pre-initialized from a CommonFolderQuery.
- new QueryOptions(<CommonFileQuery>, <file_types>) Creates an object pre-initialized from a CommonFileQuery along with an array of file types—for example, ["*.jpg", "*.jpeg", "*png"]. If you specify null for <file_types>, it's the same as ["*"].

[85] Contrary to any examples in the documentation, queries should *always* use a full property name such as *System.ItemDate:* rather than the user-friendly shorthand *date:* because the latter will not work on localized builds of Windows.

Once constructed you then set any of the other properties you care about—all types and enumerations referred to below such as `FolderDepth` are found in `Windows.Storage.Search`:

Property	Description
`folderDepth`	Either `FolderDepth.shallow` (the default) or `deep`.
`fileTypeFilter`	A vector of strings that describe the desired file type extensions, as in `".mp3"`. The default is an empty list (no filtering), meaning `"*"`.
`sortOrder`	A vector of `SortEntry` structures that each contain a Boolean named `ascendingOrder` (false for descending order) and a `propertyName` string. Each entry in the vector defines a sort criterion; these are applied in the order they appear in the vector. An example of this will be given a little later.
`indexerOption`	A value from `IndexerOption`, which is one of `useIndexerWhenAvailable`, `onlyUseIndexer` (limit the search to indexed content only), and `doNotUseIndexer` (query the file system directly bypassing the indexer). As the latter is the default, you'll typically want to explicitly set this property to `useIndexerWhenAvailable`.
`userSearchFilter`	An AQS string for the query, which is combined with `applicationSearchFilter`.
`applicationSearchFilter`	An AQS string for the query, which is combined with `userSearchFilter`.
`language`	A string containing the BCP-47 language tag associated with the AQS strings.
`dateStackOption`	A read-only value from `DateStackOption` that can be set when creating the `QueryOptions` from a `CommonFolderQuery`.
`groupPropertyName`	A read-only string identifying the property used for grouping results with a common query. In `CommonFolderQuery.groupByYear`, for example, this is set to *System.ItemDate*.
`storageProviderIdFilter`	A read-only vector of provider strings, where you add any specific providers by calling `storageProviderIdFilter.append`, specifically to limit the results to those providers.

QueryOptions methods You can save a `QueryOptions` for later use and then reinitialize an instance from that saved state. The `QueryOptions.saveToString` returns a string representation for the query that you can save in your app data—and remember also to save the `StorageFolder` in the access cache! Later, when you want to reload the options, use `QueryOptions.loadFromString`. These methods are, in short, analogs to `JSON.stringify` and `JSON.parse`.

`QueryOptions` has two other methods—`setPropertyPrefetch` and `setThumbnailPrefetch`—which are discussed in "Metadata Prefetching with Queries."

Here's a brief snippet to do a `CommonFileQueryorderByTitle` file query for MP3s, where we remember to call `areQueryOptionsSupported`:

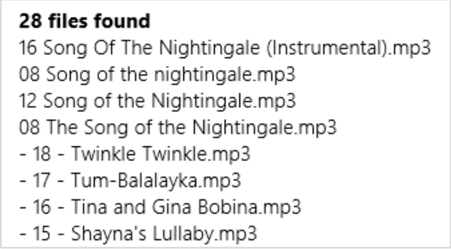
```
var musicLibrary = Windows.Storage.KnownFolders.musicLibrary;
var options = new Windows.Storage.Search.QueryOptions(
    Windows.Storage.Search.CommonFileQuery.orderByTitle, [".mp3"]);

if (musicLibrary.areQueryOptionsSupported(options)) {
    var query = musicLibrary.createFileQueryWithOptions(options);
    showResults(query.getFilesAsync());
}
```

I noted in the table that the application and user filter AQS strings here are combined within the query. What this means is that you can separately manage any filter you want to apply generally for your app (`applicationSearchFilter`) from user-supplied search terms (`userSearchFilter`). This way you can enforce some search filters without requiring the user to type them in and without always having to combine strings yourself. It's also helpful to separate locale-independent and locale-specific properties, as terms in `applicationSearchFilter` should always use locale-invariant property names (like *System.FileName* instead of *filename*) to make sure results come out as expected. For more on this, see Using Advanced Query Syntax Programmatically.

It's necessary to use one or both of these properties with the `CommonFileQuery.orderBySearchRank` query because text-based searches return ranked results to which this query applies. (The query sorts by *System.Search.Rank*, *System.DateModified*, and then *System.ItemDisplayName*.) Scenario 1 of the Programmatic file search sample shows a bit of this, where it uses this ordering along with the `userSearchFilter` property, whose value is set to whatever you enter in the sample's search box (js/scenario1.js):

```
var musicLibrary = Windows.Storage.KnownFolders.musicLibrary;
var options = new Windows.Storage.Search.QueryOptions(
    Windows.Storage.Search.CommonFileQuery.orderBySearchRank, ["*"]);
options.userSearchFilter = searchFilter;
var fileQuery = musicLibrary.createFileQueryWithOptions(options);
```

On my machine, where I have a number of songs with "Nightingale" in the title, as well as an album called "Nightingale Lullaby," a search using the string *Nightingale AND System.ItemType: mp3* in the above code gives me results that look like this in the sample (only partial results shown for brevity):[86]



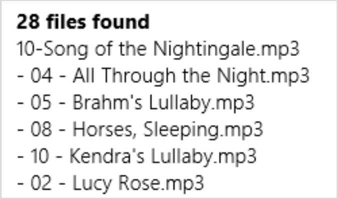
28 files found
16 Song Of The Nightingale (Instrumental).mp3
08 Song of the nightingale.mp3
12 Song of the Nightingale.mp3
08 The Song of the Nightingale.mp3
- 18 - Twinkle Twinkle.mp3
- 17 - Tum-Balalayka.mp3
- 16 - Tina and Gina Bobina.mp3
- 15 - Shayna's Lullaby.mp3

This shows that the search ranking favors songs with "Nightingale" directly in the title but also includes those from an album with that name.

A query like this lets us see the purpose of the other two methods on the `Storage[File | Folder | Item]QueryResult` objects that we haven't mentioned yet: `findStartIndexAsync` and `getMatchingPropertiesWithinRanges`.

[86] To reiterate the purpose of `applicationSearchFilter` and `userSearchFilter`, if my app was capable of working with *only* mp3 formats, I could store the constant term `"System.ItemType: 'mp3'"` in `applicationSearchFilter` and then put variable, user-provided terms like `"Nightingale"` in `userSearchFilter`.

findStartIndexAsync retrieves the StorageFile for the first item in the query results where the text argument you provide matches the first property used in the query. This is a bit tricky. In the sample above, which uses CommonFileQuery.orderBySearchRank, there's actually not much to compare to. If you use orderByTitle, on the other hand, you can compare against values of the System.Title property. For instance, using this and the same AQS string as before, I get these results:



28 files found
```
10-Song of the Nightingale.mp3
- 04 - All Through the Night.mp3
- 05 - Brahm's Lullaby.mp3
- 08 - Horses, Sleeping.mp3
- 10 - Kendra's Lullaby.mp3
- 02 - Lucy Rose.mp3
```

Calling findStartIndexAsync with "Horses" I get the result of 3 (zero-based). I can pass this to the query's getFilesAsync(index, 1) method to retrieve the StorageFile for that item:

```
fileQuery.findStartIndexAsync("Horses").done(function (index) {
    console.log("First item with 'Horses' at index: " + index);

    if (index != 4294967295) {
        fileQuery.getFilesAsync(index, 1).done(function (files) {
            files && console.log(files[0].displayName);
        });
    }
});
```

This prints "- 08 - Horses, Sleeping.mp3" to the console. Note that the return value of 4294967295 is a long integer -1, meaning "no index."

As for getMatchingPropertiesWithRanges, this works for only one StorageFile at a time and is typically called when going through the query results from getFilesAsync(). It returns a Map of property names, where the values are arrays (vectors) of objects that describe where matches occurred in that property. Each object has startPosition and length properties that pinpoint each location.

For a demonstration, play around with scenario 3 of the Semantic text query sample. (The other two scenarios are for searching in text content, which we'll mention in Chapter 15, "Contracts.") It runs a query against the Music library, as we've been doing, using the CommonFileQuery.orderBySearchRank plus whatever extra AQS string you type in (js/filePropertiesMatches.js):

```
var musicLibrary = Windows.Storage.KnownFolders.musicLibrary;
var options = new Windows.Storage.Search.QueryOptions(
    Windows.Storage.Search.CommonFileQuery.orderBySearchRank, ["*"]);
options.userSearchFilter = searchFilter;
options.setPropertyPrefetch(
    Windows.Storage.FileProperties.PropertyPrefetchOptions.musicProperties, []);

var fileQuery = musicLibrary.createFileQueryWithOptions(options);

fileQuery.getFilesAsync().done(function (files) {
    if (files.size > 0) {
        files.forEach(function (file) {
            var searchHits = fileQuery.getMatchingPropertiesWithRanges(file);

            // [Output code omitted to highlight System.FileName occurrences]
        });
    }
});
```

To keep things simple, I just ran a search on “Horses” that results in just the one file, “ - 08 - Horses, Sleeping.mp3”. The map that I got back in the `searchHits` variable contained the following (only one location per property):

Property	startPosition	Length
System.FileName	7	6
System.ItemNameDisplay	7	6
System.ItemPathDisplay	65	6
System.ParsingName	7	6
System.Title	0	6

Clearly, you can see that position 7 (zero-based) in the filename is where “Horses” begins. The *System.Title* of the track is just “Horses, Sleeping,” so its start position is appropriately 0.

In some cases, as with my “Nightingale” search before, the term might not occur in the filename or title at all. In such cases you’ll see a completely different list of properties in the map, such as *System.ItemFolderNameDisplay* and *System.Music.AlbumTitle*.

Metadata Prefetching with Queries

The `QueryOptions.setPropertyPrefetch` method allows you to indicate a group of file properties that you want to optimize for fast retrieval—they’re accessed through the same APIs as properties are otherwise, but they come back faster. This is very helpful when you’re displaying a collection of files in a ListView, using a custom data source with certain properties from enumerated files. In that case, you’d want to set those up for prefetch so that the control renders faster. Similarly, the `setThumbnailPrefetch` method tells Windows what kinds of thumbnails you want to include in the query—again, you can ask for these without setting the prefetch, but they come back faster when you do. This helps you optimize the display of a file collection.

Examples of this can be found in two samples. In scenario 3 of the Semantic text query sample, which we saw at the end of the previous section, it made sure to prefetch the music properties it planned to search (js/filePropertiesMatches.js):

```
options.setPropertyPrefetch(
    Windows.Storage.FileProperties.PropertyPrefetchOptions.musicProperties, []);
```

The values affected are determined by the first argument to `setPropertyPrefetch`, which comes from `PropertyPrefetchOptions`: `none`, `musicProperties`, `videoProperties`, `imageProperties`, `documentProperties`, and `basicProperties`. If something sounds familiar here, it’s because these exactly match the objects returned through methods of the StorageFile.properties object, such as retrieveMusicPropertiesAsync, as discussed in "Media-Specific Properties" earlier.

To that group of properties you can also specify a custom list in the second argument, with each property name in quotes, such as ["System.Copyright", "System.Image.ColorSpace"]; for no custom properties, just pass [ ]. Remember that you can also use strings from the SystemProperties object in Windows.Storage.

This, in fact, is exactly what you see in an example of the API, found in scenario 3 of the Folder enumeration sample that we've been looking at already (js/scenario3.js):

```
var search = Windows.Storage.Search;
var fileProperties = Windows.Storage.FileProperties;

// Create query options with common query sort order and file type filter.
var fileTypeFilter = [".jpg", ".png", ".bmp", ".gif"];
var queryOptions = new search.QueryOptions(search.CommonFileQuery.orderByName, fileTypeFilter);

// Set up property prefetch - use the PropertyPrefetchOptions for top-level properties
// and an array for additional properties.
var imageProperties = fileProperties.PropertyPrefetchOptions.imageProperties;
var copyrightProperty = "System.Copyright";
var colorSpaceProperty = "System.Image.ColorSpace";
var additionalProperties = [copyrightProperty, colorSpaceProperty];
queryOptions.setPropertyPrefetch(imageProperties, additionalProperties);

// Query the Pictures library.
var query = Windows.Storage.KnownFolders.picturesLibrary.
    createFileQueryWithOptions(queryOptions);
```

setThumbnailPrefetch is similar, where you specify a ThumbnailMode, a requested size, and options, all of which are the same as discussed in "Thumbnails" earlier. The same scenario of the Folder enumeration sample shows a use of this, before the call to createFileQueryWithOptions (js/scenario3.js):

```
var thumbnailMode = fileProperties.ThumbnailMode.picturesView;
var requestedSize = 190;
var thumbnailOptions = fileProperties.ThumbnailOptions.useCurrentScale;
queryOptions.setThumbnailPrefetch(thumbnailMode, requestedSize, thumbnailOptions);
```

Clearly, you'd often use a thumbnail prefetch when creating a gallery experience with a ListView control, where you'd typically be using a WinJS.UI.StorageDataSource object as well. In fact, the StorageDataSource has its own options that it uses to automatically set up the appropriate prefetching, in which case you don't use the QueryOptions directly. We'll see this in the next section.

Creating Gallery Experiences

In Chapter 7 we learned about the three components of a collection control: a data source, a layout, and templates. One of the most common uses of a collection control—especially the ListView—is to display a gallery of entities in the file system, organized in interesting ways. This section brings together much of what we've learned in this chapter where implementing such an experience is concerned:

- Unless you're working with a library for which a capability exists, you'll need to have the user choose folders to include in your gallery through the Folder Picker.[87] If that set of folders is unlikely to change often, it's best for the app to have a page where the user manages the included folders—adding new ones from the Folder Picker, and removing existing ones. When the user selects a new folder, be sure to save it in the `Windows.Storage.AccessCache` so that subsequent app sessions won't need to ask again.

- Always, always use thumbnails from the `StorageFile.getThumbnailAsync` and `getScaledImageAsThumbnailAsync` methods. The thumbnails can be passed directly to `URL.createObjectURL` to display in `img` elements in place of the full `StorageFile`. Using thumbnails will perform much better both in terms of speed (because thumbnails are typically drawn from existing caches) and memory (because you're not loading the whole image file).

- Differentiate file availability based on the `StorageFile.isAvailable` flag. You can use this to provide a textual indicator and also "ghost" unavailable items by setting their opacity to something like 40%. Remember that availability means that network conditions are such that the file's contents cannot be accessed, so you typically want to disable interactivity (like invocation) on unavailable items. Furthermore, some items might be available but are represented by only a local placeholder file. Be sure, then, to show an indeterminate `progress` control if it takes longer than a second or two to get a file's contents, and also provide a means to cancel the operation. This gives the user control on slow networks.

- Very often, your gallery will involve some kind of query against the file system to create its data source. Depending on your needs, you can use a data source built on a `WinJS.Binding.List`, which you populate with the results from your queries. Alternately, you can use the `WinJS.UI.StorageDataSource` object, which has built-in support for queries (more on this below). The caveat with `StorageDataSource` is that it doesn't support grouping.

For a full end-to-end demonstration of these and other patterns, I'll refer you to the JavaScript Hilo app put together by Microsoft's Patterns & Practices team. (Note: As of the current preview, the app has not yet been updated to Windows 8.1, but it shows many of these patterns nonetheless.)

[87] It's generally best to work with library content through one of the `KnownFolder` objects because these will reflect all the other folders that a user might have added to those libraries. This will not be the case if you have the user select only an individual local folder. That said, one point of user frustration (which I've encountered personally) is the inability to add a folder on removable storage to a library, in which case providing a way to include such folders directly is helpful.

With the StorageDataSource in particular, scenario 5 of the StorageDataSource and GetVirtualizedFilesVector sample offers a demonstration of working with queries. Back in Chapter 7 we just used one of the shortcuts like this:

```
myFlipView.itemDataSource = new WinJS.UI.StorageDataSource("Pictures",
    { requestedThumbnailSize: 480 });
```

The first argument to the constructor is actually a query object of some sort—"Pictures" here is again just a shortcut. But you can create any query you want and use it here. The sample, then, creates a QueryOptions from scratch, setting up its own sorting by *System.IsFolder* and *System.ItemName*. The purpose of having this query is that for some types of galleries, such as music, the folder hierarchy on the file system is irrelevant and you want to organize by metadata. This is why a query such as CommonFileQuery.orderByMusicProperties returns a flat list of files rather than a folder tree, because you'll typically group the results by criteria other than folder.

Other options we give to the StorageDataSource set up thumbnails, which are used to call the QueryOption.setThumbnailPrefetch method on our behalf. The result is a data source over a query that should perform well by default (js/scenario2ListView.js; code slightly edited):

```
function loadListViewControl() {
    // Build datasource from the pictures library
    var library = Windows.Storage.KnownFolders.picturesLibrary;
    var queryOptions = new Windows.Storage.Search.QueryOptions;

    // Shallow query to get the file hierarchy
    queryOptions.folderDepth = Windows.Storage.Search.FolderDepth.shallow;

    // Order items by type so folders come first
    queryOptions.sortOrder.clear();
    queryOptions.sortOrder.append({ascendingOrder: false, propertyName: "System.IsFolder"});
    queryOptions.sortOrder.append({ascendingOrder: true, propertyName: "System.ItemName"});
    queryOptions.indexerOption =
        Windows.Storage.Search.IndexerOption.useIndexerWhenAvailable;

    var fileQuery = library.createItemQueryWithOptions(queryOptions);
    var dataSourceOptions = {
        mode: Windows.Storage.FileProperties.ThumbnailMode.picturesView,
        requestedThumbnailSize: 190,
        thumbnailOptions: Windows.Storage.FileProperties.ThumbnailOptions.none
    };

    var dataSource = new WinJS.UI.StorageDataSource(fileQuery, dataSourceOptions);

    // Create the ListView using dataSource...
};
```

Within its storageRenderer function (the item renderer), the sample uses the StorageDataSource.loadThumbnail method to load up the prefetched thumbnails and display them in their img elements.

If you're interested, you can dig into the WinJS sources (the ui.js file) and see how StorageDataSource works with its queries and sets up an observable collection. Along the way, you'll run into one more set of WinRT APIs: Windows.Storage.BulkAccess. This was originally created for the StorageDataSource but is now considered deprecated. If you create your own data source or collection control, just use the enumeration and prefetch APIs we've already discussed.

File Activation and Association

As noted a number of times already, an app typically obtains StorageFile and StorageFolder objects either from locations where it already has programmatic access or through the picker APIs. But a third option exists: when an app is associated with a particular file type, the user and/or other apps can launch a file or files, which in turn launches an associated app to handle them. In the process, the app that's activated for this purpose receives those StorageFile objects in its activated handler and is automatically granted full programmatic access. This makes sense if you think about it: if the user activated the file(s) from Windows Explorer, they've implicitly given their consent in the process. And for another app to launch a file, it must first get to its StorageFile object, and that happens either through the file picker or some other means that has programmatic access.

Both sides of file activation are demonstrated in the Association launching sample. Let's start with scenarios 1 and 2 that show the launching part, as that's simple and straightforward.

To launch a given StorageFile, just call Windows.System.Launcher.launchFileAsync, the results of which is a Boolean indicating whether it was successful (js/launch-file.js):

```
// file is the StorageFile to launch
Windows.System.Launcher.launchFileAsync(file).done(
    function (success) {
        // success indicates whether the file was launched.
    }
});
```

Note The launcher blocks any file type that can contain executable code, such as .exe, .msi, .js, .etc.

Alternately, you can launch a URI by using the launchUriAsync method, which is typically used to launch a browser or an URI with a custom scheme (such as mailto:) that activates an app through protocol association. (We'll return to protocols in Chapter 15.) This is demonstrated in scenario 2 (js/launch-uri.js):

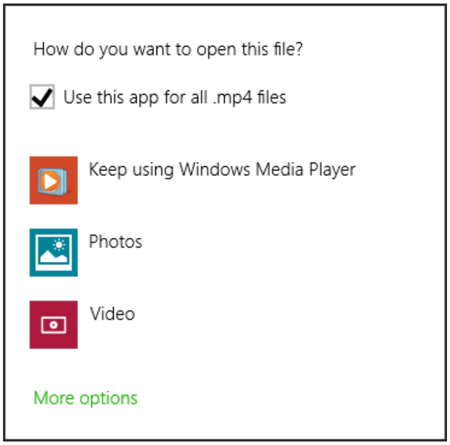
```
var uri = new Windows.Foundation.Uri(document.getElementById("uriToLaunch").value);

// Launch the URI.
Windows.System.Launcher.launchUriAsync(uri).done(
    function (success) {
        // success indicates whether the URI was launched.
    }
});
```

Note launchUriAsync does not recognize ms-appx, ms-appx-web, or ms-appdata URIs, because these already map to the current app. The app should just display such pages directly.

With both APIs you can control some aspects of the launching process by passing a LauncherOptions object as the second argument. Its properties are as follows:

Property	**Description**
contentType	The content type associated with a URI (launchUriAsync only).
desiredRemainingView	A value from Windows.UI.ViewManagement.ViewSizePreference (see Chapter 8), indicating how the calling app should appear after the launch: default, useLess, useHalf, useMore, useMinimum, or useNone. This helps when implementing cross-app scenarios, although the choice is not guaranteed, such as when the device is in portrait orientation where split views are not supported.
displayApplicationPicker	If true, displays the Open With dialog (see image below) instead of using the default association.
fallbackUri	An http: or https: URI to use if a custom scheme URI has no associated apps (launchUriAsync only). This is only allowed if the preferredApplication* properties are empty.
preferredApplicationDisplayName preferredApplicationPackageFamilyName	The display name and package ID of the recommended app in the Store if no app currently exists to handle a file type or URI. If you're using a custom URI scheme, this is an especially helpful way to get users to a companion app that handles that scheme. These cannot be used if you set fallbackUri.
treatAsUntrusted	If true, displays a warning to the user that the file or URI has not come from a trusted source. You should always set this flag if you're not certain about the origins of the content.
UI	A LauncherUIOptions object. For the launching app, lets you set an invocationPoint or selectionRect, and the preferredPlacement of the Open With dialog relative to that point or rectangle.

The Association launching sample lets you play with some of these options. Here's how the Open With dialog appears for an .mp4 file when using displayApplicationPicker:


How do you want to open this file?

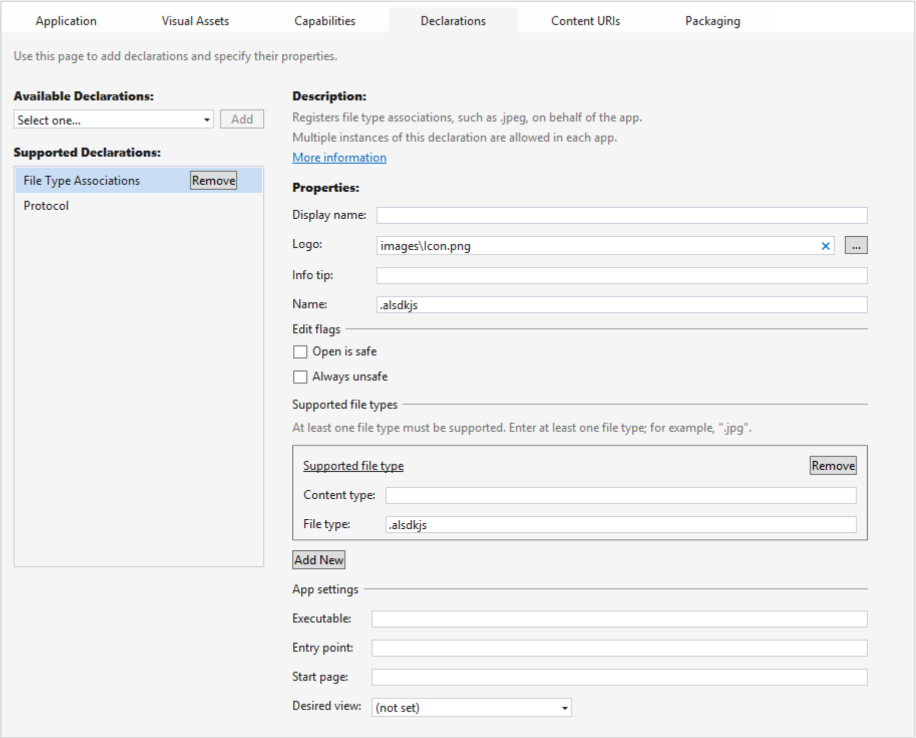
☑ Use this app for all .mp4 files

Let's switch now to the other side of the equation: an app that can be launched through a file association must first have at least one File Type Association on the Declarations tab of the manifest editor, as shown in Figure 11-13. Each file type can have multiple specific types (notice the Add New button under Supported File Types), such as a JPEG having .jpg and .jpeg file extensions. Note again that some file types are disallowed for apps; see How to handle file activation for the complete list.

Under Properties, the Display Name is the overall name for a group of file types (this is optional; not needed if you have only one type). The Name, on the other hand, is required—it's the internal identity for the file group and one that should remain consistent for the entire lifetime of your app across all updates. In a way, the Name/Display Name properties for the whole group of file types is like your real name, and all the individual file types are nicknames—any of them ultimately refer to the core file type and your app.

Info Tip is tooltip text for when the user hovers over a file of this type and the app is the primary association. The Logo is a little tricky; in Visual Studio here, you simply refer to a base name for an image file, like you do with other images in the manifest. In your actual project, however, you should have multiple files for the same image in different target sizes (not resolution scales): 16x16, 32x32, 48x48, and 256x256; you'll find a place to specify all of these under the Square 30x30 Logo section of the Visual Assets tab in the manifest editor. The Association launching sample uses such images with *targetsize-** suffixes in the filenames, as in *smallTile-sdk.targetSize-32.png*.[88] These various sizes help Windows provide the best user experience across many different types of devices, and you should be sure to provide them all.

[88] Ignore, however, the sample's use of *targetsize-** naming conventions for the app's tile images; this is an error because target sizes apply only to file and URI scheme associations.



Application | Visual Assets | Capabilities | Declarations | Content URIs | Packaging

Use this page to add declarations and specify their properties.

Available Declarations:

Select one... | Add

Supported Declarations:

File Type Associations | Remove

Protocol

Description:

Registers file type associations, such as .jpeg, on behalf of the app.

Multiple instances of this declaration are allowed in each app.

More information

Properties:

Display name:

Logo: images\Icon.png

Info tip:

Name: .alsdkjs

Edit flags

□ Open is safe

□ Always unsafe

Supported file types

At least one file type must be supported. Enter at least one file type; for example, ".jpg".

Supported file type | Remove

Content type:

File type: .alsdkjs

Add New

App settings

Executable:

Entry point:

Start page:

Desired view: (not set)

FIGURE 11-13 The Declarations > File Type Associations UI in the Visual Studio manifest designer.

The options under Edit Flags control whether an "Open" verb is available for a downloaded file of this type: checking Open Is Safe will enable the verb in various parts of the Windows UI; checking Always Unsafe disables the verb. Leaving both blank might enable the verb, depending on where the file is coming from and other settings within the system.

At the very bottom of this UI you can also set a discrete start page for handling activations, including a setting for the launched app's desired view. Typically, though, as shown in js/default.js of the sample, you'll just use your main activation handler:

```
function activated(e) {
    // If activated for file type or protocol, launch the appropriate scenario.
    // Otherwise navigate to either the first scenario or to the last running scenario
    // before suspension or termination.
    var url = null;
    var arg = null;

    if (e.detail.kind === Windows.ApplicationModel.Activation.ActivationKind.file) {
        url = scenarios[2].url;
        arg = e.detail.files;
    } else if (e.detail.kind === Windows.ApplicationModel.Activation.ActivationKind.protocol) {
        url = scenarios[3].url;
        arg = e.detail.uri;
    } else if (e.detail.kind === Windows.ApplicationModel.Activation.ActivationKind.launch) {
        url = WinJS.Application.sessionState.lastUrl || scenarios[0].url;
    }

    // ...

}
```

Here you can see some of the other possibilities in the ActivationKind enumeration. The file kind means the app was launched through file activation, in response to which it navigates to scenario 3 ([2] by array position). The protocol kind means it was activated through a custom URI scheme, as we'll again see in Chapter 15, which navigates to scenario 4. And of course launch is the default kind, when an app is launched from a tile.

Note Apps can be activated for these purposes when they're already running. Be sure to test that case in your own apps and verify that the appropriate app data and settings are loaded, including session state if eventArgs.detail.previousExecutionState is terminated.

With the activation kind of file, the eventArgs.detail is a WebUIFileActivatedEventArgs object. As with normal activation, this contains an activatedOperation object with a getDeferral method (in case you need to do async operations) along with previousExecutionState and splashScreen properties.

Three members are unique to file activation. First is the files property, which contains either the single StorageFile that the launching app provided to launchFileAsync or an array of StorageFile objects if the app was launched from Windows Explorer for a multiple selection. The detail.verb property will be "open". The sample, for its part, doesn't do anything with the files it receives except pass them onto the page control in js/receive-file.js that just outputs some information about those files. Your real apps, of course, will respond more intelligently by working with the files and their contents as appropriate.

Tip Because the files might have come from anywhere, the receiving app should always treat them as untrusted content. Avoid taking permanent actions based on the file contents.

The event args also contains a property called neighboringFilesQuery, which is a ready-made StorageFileQueryResult that allows you to retrieve sibling files even though they weren't actually selected or activated. This helps in creating gallery views for a folder that contains the activated file(s), and it's a good place to call the query's findStartIndexAsync, especially when using a semantic zoom control where you want to zoom to that index. In this case, the argument to findStartIndexAsync is one of the StorageFile objects within eventArgs.detail.files rather than a keyword.

Caveat With neighboringFilesQuery, the app still needs programmatic access to the folder in question, such as a library capabilities, otherwise you'll see access denied exceptions. Thus, the query is primarily useful for those libraries and not arbitrary folders.

Apps that declare the Removable Storage capability and at least one file type can also be launched with ActivationKind.device, which means the user selected to launch that app in response to an AutoPlay event when a device (like a camera) was attached to the system. An app will typically then do things like import images from that device. Scenario 4 of the Removable storage sample is activated for this launch kind, for example. The eventArgs here is WebUIDeviceActivatedEventArgs, the key property of which is eventArgs.detail.deviceInformationId that identifies the attached device. This sample passes that onto its page control in js/s4_autoplay.js (as arg) that then gets a virtual StorageFolder through Windows.Devices.Portable.StorageDevice.fromId and runs a file query to find images:

```
var storage = (typeof arg === "string" ?
    Windows.Devices.Portable.StorageDevice.fromId(arg) : arg);
if (storage) {
    var storageName = storage.name;

    // Construct the query for image files
    var queryOptions = new Windows.Storage.Search.QueryOptions(
        Windows.Storage.Search.CommonFileQuery.orderByName, [".jpg", ".png", ".gif"]);
    var imageFileQuery = storage.createFileQueryWithOptions(queryOptions);

    // Run the query for image files
    imageFileQuery.getFilesAsync().done(function (imageFiles) {
        // process results
    });
```

What We've Just Learned

- User data lives anywhere outside the app's own app data folders (and package). User data locations span the local file system, removable storage devices, local networks, and the cloud, and the StorageFile and StorageFolder objects hide the details about how those locations are referenced and their access models.
- Access to user data folders, such as media libraries, documents, and removable storage, is controlled by manifest capabilities. Such capabilities need be declared only if the app needs to access the file system in some way other than using the file picker.
- The file picker is the way that users can select files from any safe location in the file system, as well as files that are provided by other apps (where those files might be remote, stored in a database, or otherwise not present as file entities on the local file system). The ability to select files directly from other apps—including files that another app might generate on demand—is one of the most convenient and powerful features of Windows Store apps.
- Apps should always use the Windows.Storage.AccessCache to preserve programmatic access to files and folders for later sessions. The cache maintains two independent lists: one for recently

used items (limited to 25) and one for general purpose use (limited to 1000 items).

- OneDrive is deeply integrated into Windows such that there is always at least a local placeholder file that will automatically download its contents (if allowed by the current network) when the file is open. The `StorageFile.isAvailable` flag indicates whether the file's contents are currently accessible.
- The `StorageFile` object provide access to rich metadata, including thumbnails and media specific properties.
- Apps should always use thumbnails to show image contents in consumption scenarios to avoid loading full image data. Loading the image is necessary only in editing and full-image panning views.
- The `StorageLibrary` object supplies the ability to manage which folders are included in the user's media libraries.
- `StorageFolder` objects provide a very rich and extensive capability to enumerate its contents and to query for items that match certain criteria. WinRT provides common file and folder queries for typical scenarios, but you can also build custom queries with Advanced Query Syntax (AQS) strings. Custom queries also provide prefetching capabilities for properties and thumbnails.
- The `WinJS.UI.StorageDataSource` object provides a built-in means with query support to create a gallery experience in a ListView control. A gallery should always use thumbnails for images and use the `isAvailable` flag to differentiate items in its UI.
- To support file activation, an app associates itself with one or more file types in its manifest and then watches for the `ActivationKind.file` whose event args will contain the `StorageFile` objects for the files that were launched, as well as a `neighboringFilesQuery` that provides access to other files in the folder.
- Apps that support removable storage can also associate themselves with one or more file types and be launched for Auto Play events with `ActivationKind.device`.

Chapter 12

Input and Sensors

Touch is clearly one of the most exciting means of interacting with a computer and one that has finally come of age. Sure, we've had touch-sensitive devices for many years: I remember working with a touch-enabled screen in my college days, which I have to admit is almost an embarrassingly long time ago now! In that case, the touch sensor was a series of transparent wires embedded in a plastic sheet over the screen, with an overall touch resolution of around 60 wide by 40 high…and, to really date myself, the monitor itself was only a text terminal!

Now, touch screens are responsive enough for general purpose use (you don't have to stab them to register a point), built into high-resolution displays, relatively inexpensive, and capable of doing something more than replicating the mouse, such as supporting multitouch and sophisticated gestures.

Because great touch interaction is a fundamental feature of great apps, designing for touch means in many ways thinking through UI concerns anew. In your layout, for example, it means making hit targets a size that's suitable for a variety of fingers. In your content navigation, it means utilizing direct gestures such as swipes and pinches rather than relying on only item selection and navigation controls. Designing for touch also means thinking through how gestures might enrich the user experience—how to make most content directly interactive, and also how to provide for discoverability and user feedback that has generally relied on mouse-only events like hover.

All in all, approach your design as if touch is the *only* means of interaction your users will have. At the same time, it's important to remember that new methods of input seldom obsolete existing ones. Sure, punch cards did eventually disappear, but the introduction of the mouse did not make keyboards obsolete. The availability of speech and handwriting recognition has obsoleted neither mouse nor keyboard. And I think the same is true for touch: it's a complementary input method that has its own particular virtues but is unlikely to wholly supplant the others. As Bill Buxton of Microsoft Research has said, "Every modality, including touch, is best for something and worst for something else." I expect, in time, most consumers will find themselves using keyboard, mouse, and touch together, just as we learned to integrate the mouse in what was once a keyboard-only reality.

Windows is designed to work well with all forms of input—to work great with touch, to work great with mice, to work great with keyboards, and, well, to just work great on diverse hardware. (And Windows Store certification essentially requires this for apps as well.) For this reason, Windows provides a unified pointer-based input model with which you can distinguish the different types of input if you really need to, but otherwise it treats them equally. You can also focus more on higher-level gestures as well, which might arise from any input source, and not worry about raw pointer events. Indeed, the fact that we haven't even brought this subject up until now, midway through this book, gives testimony to just how transparently you work with all kinds of pointer input, especially when using controls that do the work for us. Handling such events ourselves thus arises primarily with custom controls and direct manipulation of noncontrol objects.

The keyboard also remains an important consideration, and this means both hardware keyboards and the on-screen "soft" keyboard. The latter has gotten more attention in recent years for touch-only devices but actually has been around for some time for accessibility purposes. In Windows, too, the soft keyboard includes a handwriting recognizer—something apps get for free. And when an app wants to work more closely with raw handwriting input—known as inking—those capabilities are present as well.

The other topic we'll cover in this chapter is sensors. It might seem an incongruous subject to place alongside input until you come to see that sensors, like touch screens themselves, *are* another form of input! Sensors tell an app what's happening to the device in its relationship to the physical world: how it's positioned in space (relative to a number of reference points), how it's moving through space, how it's being held relative to its "normal" orientation, and even how much light is shining on it. Thinking of sensors in this light (pun intended), we begin to see opportunities for apps to directly integrate with the world around a device rather than requiring users to tell the app about those relationships in some abstract way. And just to warn you, once you see just how easy it is to use the WinRT APIs for sensors, you might be shopping for a new piece of well-equipped hardware!

Let me also mention that the sensors we'll cover in this chapter are those for which specific WinRT APIs exist. There might be other peripherals that can also act as input devices, but we'll come back to those generally in Chapter 17, "Devices and Printing."

Touch, Mouse, and Stylus Input

Where pointer-based input is concerned—which includes touch, mouse, and pen/stylus input—the singular message from Microsoft has been and remains, "Design for touch and get mouse and stylus for free." This is very much the case, as we shall see, but I've heard that a phrase like "touch-first design," which sounds great to a consumer, can be a terrifying proposition for developers! With all the attention around touch, consumer expectations are often demanding, and meeting such expectations seems like it will take a lot of work.

Fortunately, Windows provides a unified framework for handling pointer input—from all sources—such that you don't actually need to think about the differences until there's a specific reason to do so. In this way, touch-first design *is* a design issue more than an implementation issue.

We'll talk more about designing for touch soon. What I wanted to discuss first is how you as a developer should approach implementing those designs once you have them:

- First, *use templates and standard controls* and you get lots of touch support for free, along with mouse, pen, stylus, and accessibility-ready keyboard support. If you build your UI with standard controls, set appropriate `tabindex` attributes for keyboard users, and handle standard DOM events like `click`, you're pretty much covered. Controls like Semantic Zoom already handle different kinds of input (as we saw in Chapter 7, "Collection Controls"), and other CSS styles like snap points and content zooming automatically handle various interaction gestures.

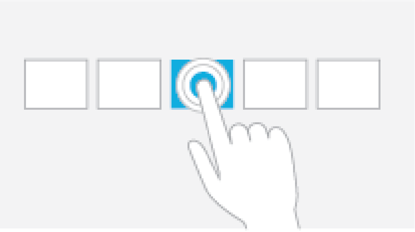
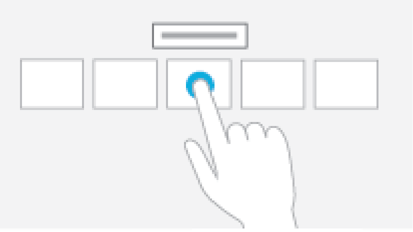
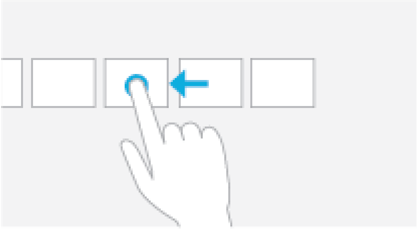
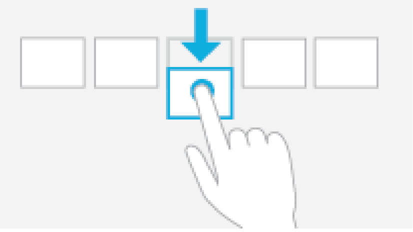
- Second, when you need to handle gestures yourself, as with custom controls or other elements with which the user will interact directly, *use the gesture events* like `MSGestureTap` and `MSGestureHold` along with event sequences for inertial gestures (`MSGestureStart`, `MSGestureChange`, and `MSGestureEnd`). Gestures are essentially higher-order interpretations of lower-level pointer events, meaning that you don't have to do such interpretation yourself. For example, a pointer down followed by a pointer up within a certain movement threshold (to account for wiggling fingers) becomes a single tap gesture. A pointer down followed by a short drag followed by a pointer up becomes a swipe that triggers a series of events, possibly including inertial events (ones that continue to fire even after the pointer, like a touch point, is physically released).
- Third, if you need to handle pointer events directly, *use the unified pointer events* like `pointerdown`, `pointermove`, and so forth. These are lower-level events than gestures, and they are primarily appropriate for apps that don't necessarily need gesture interpretation. For example, a drawing or inking app simply needs to trace different pointers with on-screen feedback, where concepts like swipe and inertia aren't meaningful. Pointer events also provide more specialized device data such as pressure, rotation, and tilt, which is surfaced through the pointer events. Still, it is possible to implement gestures directly with pointer events, as a number of the built-in controls do.
- Finally, an app can *work directly with the gesture recognizer* to provide its own interpretations of pointer events into gestures.

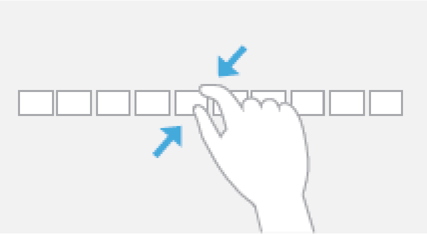
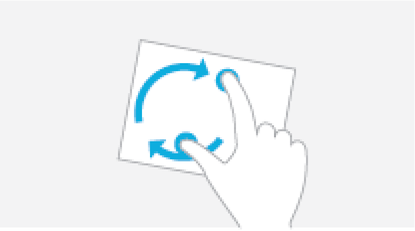
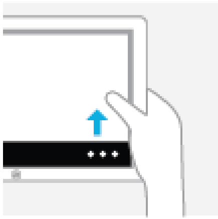
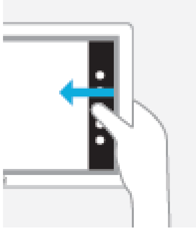
So, what about legacy DOM events that we already know and love, beyond `click`? Can you still work with the likes of `mousedown`, `mouseup`, `mouseover`, `mousemove`, `mouseout`, and `mousewheel`? The answer is yes, because pointer events from all input sources will be automatically translated into these legacy events. This translation takes a little extra processing time, however, so for new code you'll generally realize better responsiveness by using the gesture and pointer events directly. Legacy mouse events also assume a single pointer and will be generated only for the primary touch point (the one with the `isPrimary` property). As much as possible, use the gesture and pointer events in your code.

The Touch Language and Mouse/Keyboard Equivalents

On the Windows Developer Center, the rather extensive article on Touch interaction design is helpful for designers and developers alike. It discusses various ergonomic considerations, has some great diagrams on the sizes of human fingers, provides clear guidance on the proper size for touch targets given that human reality (falling between 30px and 50px), and outlines key design principles such as providing direct feedback for touch interaction (animation) and having content follow your finger.

Most importantly, the design guidance also describes the Windows Touch Language, which contains the eight core gestures that are baked into the system and the controls. The table below shows and describes the gestures and indicates what events appear in the app for them. The Touch interaction design topic also has videos of these gestures. And for design guidelines around this touch language, see Gestures, manipulations, and interactions.

Gesture	Meaning and Gesture Events	Description
One finger touches the screen and lifts up. 	Tap for primary action (commanding); appears as `click` and `MSGestureTap` events on the element.	Tapping on an element invokes its primary action, typically executing a command, checking a box, setting a rating, positioning a cursor, etc.
One finger touches the screen and stays in place. 	Press and hold to learn; appears as `contextmenu` and `MSGestureHold` events on the element.	This touch interaction displays detailed information or teaching visuals (for example, a tooltip or context menu) without a commitment to an action. Anything displayed this way should not prevent users from panning if they begin sliding their finger.
One or more fingers touch the screen and move in the same direction. 	Slide to pan (can be horizontal or vertical); automatically appears as scrolling events for scrollable regions. Also appears as a gesture series (`MSGestureStart`, `MSGestureChange`, `MSGestureEnd`, possibly with inertial gesture events signaled by `MSInertiaStart`, plus `pointer*` events).	Slide is used primarily for panning interactions but can also be used for moving, drawing, or writing. Slide can also be used to target small, densely packed elements by scrubbing (sliding the finger over related objects such as radio buttons).
One or more fingers touch the screen and move a short distance in the same direction. 	Swipe to select, command, and move (can be horizontal or vertical)—also called *cross-slide*; appears as a gesture series (`MSGestureStart`, `MSGestureChange`, `MSGestureEnd`, as well as `pointer*` events). The gesture recognizer doesn't distinguish this from vertical panning, however, so an app or control needs to implement that interpretation directly (a good reason to use controls like the ListView!).	Sliding the finger a short distance, perpendicular to the panning direction, selects objects in a list or grid; also implies displaying commands in an app bar relevant to the selection.

Two or more fingers touch the screen and move closer together or farther apart. 	Pinch and stretch to zoom; appears as a gesture series (`MSGestureStart`, `MSGestureChange`, `MSGestureEnd`), but apps can use the `-ms-content-zooming: zoom` and `touch-action: pinch-zoom` CSS styles to enable touch zooming automatically.	Can be used for optical zoom or resizing, as well as for Semantic Zoom where applicable.
Two or more fingers touch the screen and move in a clockwise or counter-clockwise arc. 	Turn to rotate; appears as a gesture series (`MSGestureStart`, `MSGestureChange`, `MSGestureEnd`).	Rotates an object or a view.
	Swipe from top or bottom edge for app commands; handled automatically through the AppBar control, though an app can also detect these events directly through `Windows.UI.Input.EdgeGesture`.	The bottom app bar contains app commands for the current page context; the top app bar provides for navigation, if applicable.
	Swipe from edge for system commands; handled automatically by the system with the app receiving events related to the selected charm, when applicable, as well as `focus` and `blur` events if the foreground app is changed when swiping from the left edge.	Swiping from the right displays the Charms bar; swiping from the left cycles through currently running apps; swiping from the top edge to the bottom closes the current app; swiping from the top edge to the left or right snaps the current app to one side of the screen.

You might notice in the table above that many of the gestures in the touch language, like pinch and rotate, don't have a single event associated with them but are instead represented by a *series* of gesture or pointer events. The reason for this is that these gestures, when used with touch, typically involve animation of the affected content while the gesture is happening. Swipes, for example, show linear movement of the object being panned or selected. A pinch or stretch movement will often be actively zooming the content. (Semantic Zoom is an exception, but then you just let the control handle the details.) And a rotate gesture should definitely give visual feedback. In short, handling these gestures with touch, in particular, means dealing with a series of events rather than just a single one.

This is one reason that it's so helpful (and time-saving!) to use the built-in controls as much as possible, because they already handle all the gesture details for you. The ListView control, for example, contains all the pointer/gesture logic to handling pans and swipes, along with taps. The Semantic Zoom control, like I said, implements pinch and stretch by watching `pointer*` events. If you look at the source code for these controls within WinJS, you'll start to appreciate just how much they do for you (and what it will look like to implement a rich custom control of your own, using the gesture recognizer!).

You can also save yourself a lot of trouble with the `touch-action` CSS properties described later under "CSS Styles That Affect Input." Using this has the added benefit of processing the touch input on a non-UI thread, thereby providing much smoother manipulation than could be achieved by handling pointer or gesture events.

On the theme of "write for touch and get other input for free," all of these gestures also have mouse and keyboard equivalents, which the built-in controls also implement for you. It's also helpful to know what those equivalents are, as shown in the table below. "Standard Keystrokes" later in this chapter also lists many other command-related keystrokes.

Touch	Keyboard	Mouse	Pen/Stylus
Press and hold (or tap on text selection)	Right-click button	Right button click	Press and hold
Tap	Enter	Left button click	Tap
Slide (short distance)	Arrow keys	Left button click and drag, click on scrollbar arrows, drag the scrollbar thumb, use the mouse wheel	Tap on scrollbar arrows, drag scrollbar thumb, tap and drag
Slide + inertia (long distance)	Page Up/Page Down	Left button click and drag, click on scrollbar track, drag the scrollbar thumb, use the mouse wheel	Tap on scrollbar track, drag scrollbar thumb, tap and drag
Swipe to select	Right-click button or spacebar	Right button click	Tap and drag
Pinch/Stretch	Ctrl+ and Ctrl-	Ctrl+mouse wheel or UI command	UI command or other hardware feature
Swipe from edge	Win+Z, Win+Tab, Win+C or Win+Shift+C	Clicking on corners of the screen; right-click shows app bar	Drag in from edge
Rotate	Ctrl+, and Ctrl+.	Ctrl+Shift+mouse wheel	UI command or other hardware feature

You might notice a conspicuous absence of double-click and/or double-tap gestures in this list. Does that surprise you? In early builds of Windows 8 we actually did have a double-tap gesture, but it turned out to not be all that useful, it conflicted with the zoom gesture, and it was sometimes very difficult for users to perform. I can say from watching friends over the years that double-clicking with the mouse isn't even all it's cracked up to be. People with not-entirely-stable hands will often move the mouse quite a ways between clicks, just as they might move their finger between taps. As a result, the reliability of a double-tap ends up being pretty low, and because it wasn't really needed in the touch language, it was simply dropped altogether.

Sidebar: Creating Completely New Gestures?

While the Windows touch language provides a simple yet fairly comprehensive set of gestures, it's not too hard to imagine other possibilities. The question is, when is it appropriate to introduce a new kind of gesture or manipulation?

First, avoid introducing new ways to do the same things, such as additional gestures that just swipe, zoom, etc. It's better to get more creative in how the app interprets an existing gesture. For example, a swipe gesture might pan a scrollable region but can also just move an object on the screen—no need to invent a new gesture.

Second, if you have controls placed on the screen where you want the user to give input, there's no need to think in terms of gestures at all: just apply the input from those controls appropriately.

Third, even when you do think a custom gesture is needed, the bottom-line recommendation is to make those interactions feel natural, rather than something you just invent for the sake of invention. Microsoft also recommends that gestures behave consistently with the number of pointers, velocity/time, and so on. For example, separating an element into three pieces with a three-finger stretch and into two pieces with a two-finger stretch is fine; having a three-finger stretch enlarge an element while a two-finger stretch zooms the canvas is a bad idea, because it's not very discoverable. Similarly, the speed of a horizontal or vertical flick can affect the velocity of an element's movement, but having a fast flick switch to another page and a slow flick highlight text is a bad idea. In this case, having different functions based on speed creates a difficult UI for your customers because they'll all have different ideas about what "fast" and "slow" mean and might also be limited by their physical abilities.

Finally, with any custom gesture, recognize that you are potentially introducing an inconsistency between apps. When a user starts interacting with a certain kind of app in a new way, he or she might start to expect the same from other apps and might become confused (or upset) when those apps don't behave identically, especially if the apps use a similar gesture for completely different purposes! Complex gestures, too, might be difficult for some, if not many, people to perform; might be limited by the kind of hardware in the device (number of touch points, responsiveness, etc.); and are generally not very discoverable. In most cases it's simpler to add an appbar command or a button on your app canvas to achieve the same goal.

Edge Gestures

As we saw in Chapter 9, "Commanding UI," you don't need to do anything special for commands on the app bar or navigation bar to appear: Windows automatically handles the edge swipe from the top and bottom of your app, along with right-click, Win+Z, and the context menu key on the keyboard. That said, you can detect when these events happen directly by listening for the `starting`, `completed`, and `canceled` events on the `Windows.UI.Input.EdgeGesture` object (which are all WinRT events, these are subject to the considerations in "WinRT Events and removeEventListener" in Chapter 3, "App

Anatomy and Performance Fundamentals"):

```
var edgeGesture = Windows.UI.Input.EdgeGesture.getForCurrentView();
edgeGesture.addEventListener("starting", onStarting);
edgeGesture.addEventListener("completed", onCompleted);
edgeGesture.addEventListener("canceled", onCanceled);
```

The `completed` event fires for all input types; `starting` and `canceled` occur only for touch. Within these events, the `eventArgs.kind` property contains a value from the [`EdgeGestureKind`] enumeration that indicates the kind of input that invoked the event. The `starting` and `canceled` events will always have the kind of `touch`, obviously, whereas `completed` can be any `touch`, `keyboard`, or `mouse`:

```
function onCompleted(e) {
    // Determine whether it was touch, mouse, or keyboard invocation
    if (e.kind === Windows.UI.Input.EdgeGestureKind.touch) {
        id("ScenarioOutput").innerText = "Invoked with touch.";
    }
    else if (e.kind === Windows.UI.Input.EdgeGestureKind.mouse) {
        id("ScenarioOutput").innerText = "Invoked with right-click.";
    }
    else if (e.kind === Windows.UI.Input.EdgeGestureKind.keyboard) {
        id("ScenarioOutput").innerText = "Invoked with keyboard.";
    }
}
```

The code above is taken from scenario 1 of the Edge gesture invocation sample (js/edgeGestureEvents.js). In scenario 2, the sample also shows that you can prevent the edge gesture event from occurring for a particular element by handling its `contextmenu` event and calling `eventArgs.preventDefault` in your handler. It does this for one element on the screen such that right-clicking that element with the mouse or pressing the context menu key when that element has the focus will prevent the edge gesture events:

```
document.getElementById("handleContextMenuDiv").addEventListener("contextmenu", onContextMenu);

function onContextMenu(e) {
    e.preventDefault();
    id("ScenarioOutput").innerText =
        "The ContextMenu event was handled. The EdgeGesture event will not fire.";
}
```

Note that this method has no effect on edge gestures via touch and does not affect the Win+Z key combination that normally invokes the app bar. It's primarily to show that if you need to handle the `contextmenu` event specifically, you usually want to prevent the edge gesture.

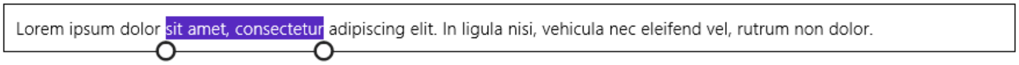
CSS Styles That Affect Input

While we're on the subject of input, it's a good time to mention a number of CSS styles that affect the input an app might receive. One style is `–ms-user-select`, which we've encountered a few times in Chapter 3 and Chapter 5, "Controls and Control Styling." This style can be set to one of the following:

- `none` Disables direct selection, though the element as a whole can be selected if its parent is selectable.
- `inherit` Sets the selection behavior of an element to match its parent.
- `text` Enables selection for text even if the parent is set to `none`.
- `element` Enables selection for an arbitrary element.
- `auto` (the default) May or may not enable selection depending on the control type and the styling of the parent. For a nontext element that does not have `contenteditable="true"`, it won't be selectable unless it's contained within a selectable parent.

If you want to play around with the variations, refer to the [Unselectable content areas with -ms-user-select CSS attribute sample](), which has the third longest JavaScript sample name in the entire Windows SDK!

A related style, but one not shown in the sample, is `-ms-touch-select`, which can be either `none` or `grippers`, the latter being the style that enables the selection control circles for touch:



Lorem ipsum dolor sit amet, consectetur adipiscing elit. In ligula nisi, vehicula nec eleifend vel, rutrum non dolor.

Selectable text elements automatically get this style, as do other textual elements with `contenteditable = "true"`— `-ms-touch-select` turns them off. To see the effect, try this with some of the elements in scenario 1 of the aforementioned sample with the really long name!

Tip The `document.onselectionchange` event will fire if the user changes the gripper positions.

In Chapter 8, "Layout and Views," we introduced the idea of snap points for panning, with the `-ms-scroll-snap*` styles, and those for zooming, namely `-ms-content-zooming` and the `-ms-content-zoom*` (refer to the [Touch: Zooming and Panning]() styles reference). The important thing is that `-ms-content-zooming: zoom` (as opposed to the default, `none`) enables automatic zooming with touch and the mouse wheel, provided that the element in question allows for overflow in both x and y dimensions. There are quite a number of variations here for panning and zooming and for how those gestures interact with WinJS controls. The [HTML scrolling, panning, and zooming sample]() explains the details.

Finally, the `touch-action` style provides for a number of options on an element:[89]

- `none` Disables default touch behaviors like pan and zoom on the element. You often set this on an element when you want to control the touch behavior directly.

[89] `double-tap-zoom` is not supported for Windows Store apps. Note also that the earlier vendor-prefixed variant of this style, `-ms-touch-action`, is deprecated in favor of `touch-action`.

- auto Enables usual touch behaviors.
- pan-x/pan-y The element permits horizontal/vertical touch panning, which is performed on the nearest ancestor that is horizontally/vertically scrollable, such as a parent div.
- pinch-zoom Enables pinch-zoom on the element, performed on the nearest ancestor that has -ms-content-zooming: zoom and overflow capability. For example, an img element by itself won't respond to the gesture with this style, but it will if you place it in a parent div with overflow set.
- manipulation Shorthand equivalent of pan-x pan-y pinch-zoom.

For an example of panning and zooming, try creating a simple app with markup like this (use whatever image you'd like):

```
<div id="imageContainer">
    <img id="image1" src="/images/flowers.jpg" />
</div>
```

and style the container as follows:

```
#imageContainer {
    overflow: auto;
    -ms-content-zooming: zoom;
    touch-action: manipulation;
}
```

What Input Capabilities Are Present?

The WinRT API in the Windows.Devices.Input namespace provides all the information you need about the capabilities that are available on the current device, specifically through these three objects:

- MouseCapabilities Properties are mousePresent (0 or 1), horizontalWheelPresent (0 or 1), verticalWheelPresent (0 or 1), numberOfButtons (a number), and swapButtons (0 or 1).
- KeyboardCapabilities Contains only a single property, keyboardPresent (0 or 1), to indicate the presence of a physical keyboard. It does not indicate the presence of the on-screen keyboard, which is always available.
- TouchCapabilities Properties are touchPresent (0 or 1) and contacts (a number). Where touch is concerned, you might also be interested in the Windows.UI.ViewManagement.UISettings.handPreference property, which indicates the user's right- or left-handedness.

To check whether touch is available, then, you can use a bit of code like this:

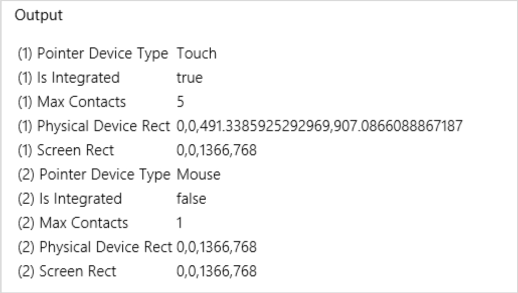
```
var tc = new Windows.Devices.Input.TouchCapabilities();
var touchPoints = 0;

if (tc.touchPresent) {
    touchPoints = tc.contacts;
}
```

Note In the web context where WinRT is not available, some information about capabilities can be obtained through the msPointerEnabled, msManipulationViewsEnabled, and msMaxTouchPoints properties that are hanging off DOM elements. These also work in the local context. The msPointerEnabled flag, in particular, tells you whether pointer* events are available for whatever hardware is available in the system. If those events are not supported, you'd use standard mouse events as an alternative.

You'll notice that the capabilities above don't say anything about a stylus or pen. For these and for more extensive information about all pointer devices, including touch and mouse, we have the Windows.Devices.Input.PointerDevice.getPointerDevices method. This returns an array of PointerDevice objects, each of which has these properties:

- pointerDeviceType A value from PointerDeviceType that can be touch, pen, or mouse.
- maxContacts The maximum number of contact points that the device can support—typically 1 for mouse and stylus and any other number for touch.
- isIntegrated true indicates that the device is built into the machine so that its presence can be depended upon; false indicates a peripheral that the user could disconnect.
- physicalDeviceRect This Windows.Foundation.Rect object provides the bounding rectangle as the device sees itself. Oftentimes, a touch screen's input resolution won't actually match the screen pixels, meaning that the input device isn't capable of hitting exactly one pixel. On one of my touch-capable laptops, for example, this resolution is reported as 968x548 for a 1366x768 pixel screen (as reported in screenRect below). A mouse, on the other hand, typically does match screen pixels one-for-one. This could be important for a drawing app that works with a stylus, where an input resolution smaller than the screen would mean there will be some inaccuracy when translating input coordinates to screen pixels.
- screenRect This Windows.Foundation.Rect object provides the bounding rectangle for the device on the screen, which is to say, the minimum and maximum coordinates that you should encounter with events from the device. This rectangle will take multimonitor systems into account, and it's adjusted for resolution scaling.
- supportedUsages An array of PointerDeviceUsage structures that supply what's called HID (human interface device) usage information. This subject is beyond the scope of this book, so I'll refer you to the HID Usages page on MSDN for starters.

The Input Device capabilities sample in the Windows SDK retrieves this information and displays it to the screen through the code in js/pointer.js. I won't show that code here because it's just a matter of iterating through the array and building a big HTML string to dump into the DOM. In the simulator, the output appears as follows—notice that the simulator reports the presence of touch and mouse both in this case:



```
Output

(1) Pointer Device Type  Touch
(1) Is Integrated        true
(1) Max Contacts         5
(1) Physical Device Rect 0,0,491.3385925292969,907.0866088867187
(1) Screen Rect          0,0,1366,768
(2) Pointer Device Type  Mouse
(2) Is Integrated        false
(2) Max Contacts         1
(2) Physical Device Rect 0,0,1366,768
(2) Screen Rect          0,0,1366,768
```

> **Curious Forge?** Interestingly, I ran this same sample in Visual Studio's Local Machine debugger on a laptop that is definitely not touch-enabled and yet a touch device was still reported as in the image above! Why was that? It's because I still had the Visual Studio simulator running, which adds a virtual touch device to the hardware profile. After closing the simulator completely (not just minimizing it), I got an accurate report for my laptop's capabilities. Be mindful of this if you're writing code to test for specific capabilities.

> **Tried remote debugging yet?** Speaking of debugging, testing an app against different device capabilities is a great opportunity to use remote debugging in Visual Studio. If you haven't done so already, it takes only a few minutes to set up and makes it far easier to test apps on multiple machines. For details, see Running Windows Store apps on a remote machine.

Unified Pointer Events

For any situation where you want to work directly with touch, mouse, and stylus input, perhaps to implement parts of the touch language in this way, use the standard `pointer*` events as adopted by the app host and most browsers. Art/drawing apps, for example, will use these events to track and respond to screen interaction. Remember again that pointers are a lower-level way of looking at input than gestures, which we'll see coming up. Which input model you use depends on the kind of events you're looking to work with.

> **Tip** Pointer events won't fire if the system is trying to do a manipulation like panning or zooming. To disable manipulations on an element, set the `-ms-content-zooming: none` or `-ms-touch-action: none`, and avoid using `-ms-touch-action` styles of `pan-x`, `pan-y`, `pinch-zoom`, and `manipulation`.

As with other events, you can listen to `pointer*` events[90] on whatever elements are relevant to you, remembering again that these are translated into legacy mouse events, so you should not listen to both. The specific events are described as follows, given in the order of their typical sequencing:

[90] The vendor-prefixed `MSPointer*` events still work in Windows 8.1 but are deprecated.

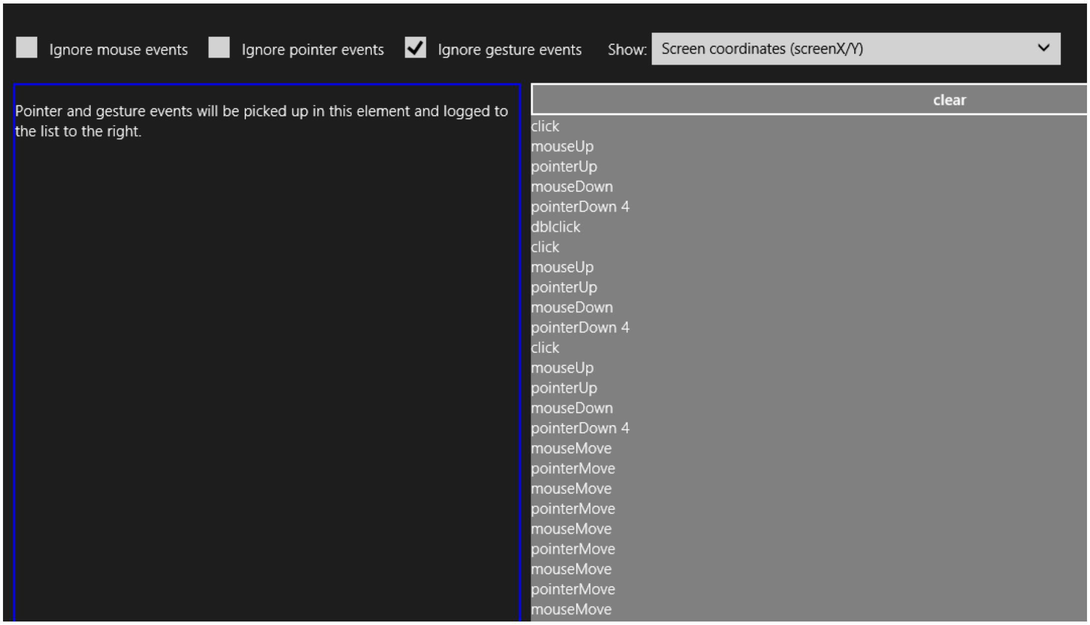
- pointerover, pointerenter Pointer moved into the bounds of the element from outside; pointerover precedes pointerenter.
- pointerdown Pointer down occurred on the element.
- pointermove Pointer moved across the element, where a positive button state indicates pen hover (this replaces the Windows 8 MSPointerHover event). This will precede both pointerover and pointerenter when a pointer moves into an element.
- pointerup Pointer was released over the element. (If an element previously captured the touch, msReleasePointerCapture is called automatically.) Note that if a pointer is moved outside of an element and released, it will receive pointerout but not pointerup.
- pointercancel The system canceled a pointer event.
- pointerout Pointer moved out of the bounds of the element, which also occurs with an up or cancel event.
- pointerleave Pointer moved out of the bounds of the element or one of its descendants, including as a result of a down event from a device that doesn't support hover. This will follow pointerout.
- gotpointercapture The pointer is captured by the element.
- lostpointercapture The pointer capture has been lost for the element.

These are the names you use with addEventListener; the equivalent property names are of the form onpointerdown, as usual. It should be obvious that some of these events might not occur with all pointer types—touch screens, for instance, generally don't provide hover events, though some that can detect the proximity of a finger are so capable.

Tip If for some reason you want to prevent the translation of a pointer* event into a legacy mouse event, call the eventArgs.preventDefault method within the appropriate event handler.

Tip Be mindful that the same pointer events are fired for all mouse buttons: the button property of the event args (as we'll see shortly) is what differentiates the left, middle, and right buttons.

The PointerEvents example provided with this chapter's companion content and shown in Figure 12-1 lets you see what's going on with all the mouse, pointer, and gesture events, selectively showing groups of events in the display.



Show: Screen coordinates (screenX/Y)

Pointer and gesture events will be picked up in this element and logged to the list to the right.

```
click
mouseUp
pointerUp
mouseDown
pointerDown 4
dblclick
click
mouseUp
pointerUp
mouseDown
pointerDown 4
click
mouseUp
pointerUp
mouseDown
pointerDown 4
mouseMove
pointerMove
mouseMove
pointerMove
mouseMove
pointerMove
mouseMove
pointerMove
mouseMove
```

FIGURE 12-1 The PointerEvents example display (screenshot cropped a bit to show detail).

Within the handlers for all of the `pointer*` events, the `eventArgs` object contains a whole roster of properties. One of them, `pointerType`, identifies the type of input: `"touch"`, `"pen"`, or `"mouse"`. This property lets you implement different behaviors for different input methods, if desired (and note that these changed from integer values in Windows 8 to strings in Windows 8.1). Each event object also contains a unique `pointerId` value that identifies a stroke or a path for a specific contact point, allowing you to correlate an initial `pointerdown` event with subsequent events. When we look at gestures, we'll also see how we use the `pointerId` of `pointerdown` to associate a gesture with a pointer.

The complete roster of properties that come with the event is actually far too much to show here, as it contains many of the usual DOM properties along with many pointer-related ones from an object type called `PointerEvent` (which is also what you get for `click`, `dblclick`, and `contextmenu` events starting with Windows 8.1). The best way to see what shows up is to run some code like the Input DOM pointer event handling sample (a `canvas` drawing app), set a breakpoint within a handler for one of the events, and examine the event object. The table below describes some of the properties (and a few methods) relevant to our discussion here.

> **Performance tips** Pointer events are best for quick responses to input, especially to touch, because they perform more quickly than gesture events. Also, avoid using an input event to render UI directly, because input events can come in much more quickly than screen refresh rates. Instead, use `requestAnimationFrame` to call your rendering function in alignment with screen refresh.

Properties	Description
`currentPoint`	A `Windows.UI.Input.PointerPoint` object. This contains many other properties such as `pointerDevice` (a `Windows.Input.Device.PointerDevice` object, as described in "What Input Capabilities Are Present" earlier in this chapter) and one just called `properties`, which is a `Windows.UI.Input.PointerPointProperties`.
`pointerType`	The source of the event, could be `"touch"` or `"pen"` or `"mouse"`. You can use this to make adjustments according to input type, if necessary.
`pointerId`	The unique identifier of the contact. This remains the same throughout the lifetime of the pointer. If desired, you can call `Windows.Devices.Input.getPointerDevice` with this id to obtain a `PointerDevice` that describes the input device's capabilities, as described earlier in "What Input Capabilities are Present?"
`type`	The name of the event, as in "pointerdown".
`x`, `screenX`, `y`, `screenY`	The x- and y-coordinates of the pointer's center point position relative to the screen.
`clientX`, `clientY`	The x- and y-coordinates of the pointer's center point position relative to the client area of the app.
`offsetX`, `offsetY`	The x- and y-coordinates of the pointer's center point position relative to the element.
`button`	Determines the button pressed by the user (on mice and other input devices with buttons). The left is 0, middle is 1, and right is 2; these values can be combined with the bitwise OR operator for chord presses (multiple buttons). This means that a mouse right-click will generate a `contextmenu` event along with a `pointerDown` with button set to 2. A mouse left-click will generate a `click` event along with a `pointerDown` with button set to 0.
`ctrlKey`, `altKey`, `shiftKey`	Indicates whether certain keys were depressed when the pointer event occurred.
`hwTimestamp`	The timestamp (in microseconds) at which the event was received from the hardware.
`relatedTarget`	Provides the element related to the current event, e.g., the `pointerout` event will provide the element to which the touch is moving. This can be `null`.
`isPrimary`	Indicates if this pointer is the primary one in a multitouch scenario (such as the pointer that the mouse would control).
Properties surfaced depending on hardware support (if not supported, these values will be 0)	
`width`, `height`	The contact width and height of the touch point specified by `pointerId`, in CSS pixels (note that these were screen pixels in Windows 8).
`pressure`	Pen pressure normalized in a decimal range of 0 to 1. This is emulated for nonsensitive hardware based on button states, returning 0.5 if buttons are down, 0 otherwise.
`rotation`	Clockwise rotation of the cursor around its own major axis in a range of 0 to 359.
`tiltX`	The left-right tilt away from the normal of a transducer (typically perpendicular to the surface) in a range of -90 (left) to 90 (right).
`tiltY`	The forward-back tilt away from the normal of a transducer (typically perpendicular to the surface) in a range of -90 (forward/away from user) to 90 (back/toward user).
Properties/Methods	
`currentPoint` `getCurrentPoint`	Providers the `Windows.UI.Input.PointerPoint` object for the current pointer relevant to the target element (`currentPoint`) or a given element (`getCurrentPoint`).
`intermediatePoints` `getIntermediatPoints`	Provides the `PointerPoint` history for the current pointer relative to the target element (`intermediatePoint`) or a given element (`getIntermediatePoints`).

It's very instructive to run the Input DOM pointer event handling sample on a multitouch device because it tracks each `pointerId` separately, allowing you to draw with multiple fingers simultaneously.

Pointer Capture

It's common with down and up events for an element to set and release a capture on the pointer. To support these operations, the following methods are available on each element in the DOM and apply to each pointerId separately:

Method	Description
setPointerCapture	Captures the pointerId for the element so that pointer events come to it and are not raised for other elements (even if you move outside the first element and into another). gotpointercapture will be fired on the element as well.
releasePointerCapture	Ends capture, triggering a lostpointercapture event. Note that this must be called through the element that has the capture, otherwise has no effect.

We see this in the Input DOM pointer event handling sample, where it sets capture within its pointerdown handler and releases it in pointerup (hs/canvaspaint.js):

```
this.pointerdown = function (evt) {
    canvas.setPointerCapture(evt.pointerId);
    // ...
};

this.pointerup = function (evt) {
    canvas.releasePointerCapture(evt.pointerId);
    // ...
};
```

Gesture Events

The first thing to know about all MSGesture* events is that they don't fire automatically like click and pointer* events, so you don't just add a listener and be done with it (that's what click is for!). Instead, you need to do a little bit of configuration to tell the system how exactly you want gestures to occur, and you need to use pointerdown to associate the gesture configurations with a particular pointerId. This small added bit of complexity (and a small cost in overall performance) makes it possible for apps to work with multiple concurrent gestures and keep them all independent, just as you can do with pointer events. Imagine, for example, a jigsaw puzzle app that allows multiple people sitting around a table-size touch screen to work with individual pieces as they will. (The sample described in "The Input Instantiable Gesture Sample" later on is a bit like this.) Using gestures, each person can be manipulating an individual piece (or two!), moving it around, rotating it, perhaps zooming in to see a larger view, and, of course, testing out placement. For Windows Store apps written in JavaScript, it's also helpful that manipulation deltas for configured elements—which include translation, rotation, and scaling—are given in the coordinate space of the parent element, meaning that it's fairly straightforward to translate the manipulation into CSS transforms and such to animate the element with the manipulation. In short, there is a great deal of flexibility here when you need it; if you don't, you can use gestures in a simple manner as well. Let's see how it all works.

Tip If you're observant, you'll notice that everything described here has no dependency on WinRT APIs. As a result, these gesture events work in both the local and web contexts.

The first step to receiving gesture events is to create an MSGesture object and associate it with the element for which you're interested in receiving events. In the PointerEvents example, that element is named *divElement*; you need to store that element in the gesture's `target` property and store the gesture object in the element's `gestureObject` property for use by `pointerdown`:

```
var gestureObject = new MSGesture();
gestureObject.target = divElement;
divElement.gestureObject = gestureObject;
```

With this association, you can then just add event listeners as usual. The example shows the full roster of the six gesture events:

```
divElement.addEventListener("MSGestureTap", gestureTap);
divElement.addEventListener("MSGestureHold", gestureHold);

divElement.addEventListener("MSGestureStart", gestureStart);
divElement.addEventListener("MSGestureChange", gestureChange);
divElement.addEventListener("MSGestureEnd", gestureEnd);
divElement.addEventListener("MSInertiaStart", inertiaStart);
```

We're not quite done yet, however. If this is all you do in your code, you still won't receive any of the events because each gesture has to be associated with a pointer. You do this within the `pointerdown` event handler:

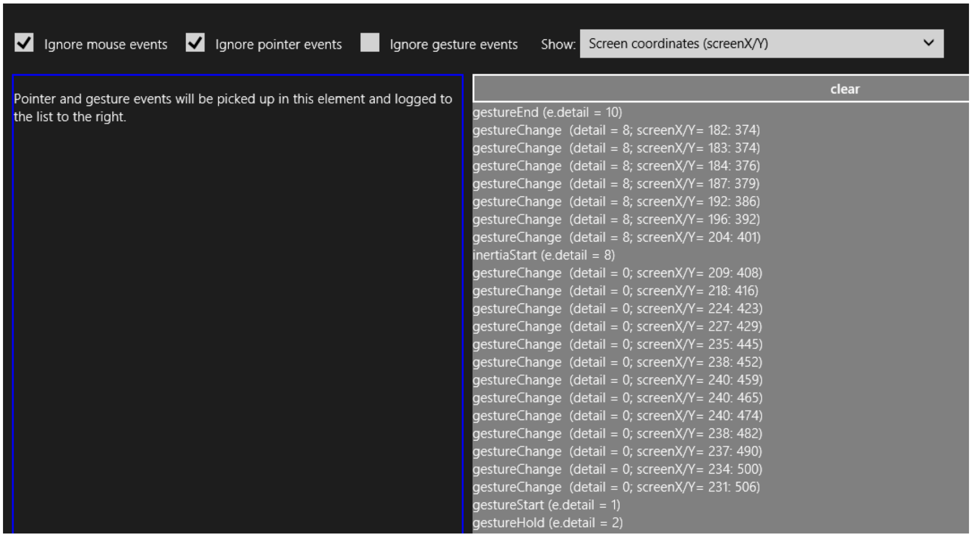
```
function pointerDown(e) {
    //Associate this pointer with the target's gesture
    e.target.gestureObject.addPointer(e.pointerId);
}
```

To enable rotation and pinch-stretch gestures with the mouse wheel (which you should do), add an event handler for the `wheel` event, set the `pointerId` for that event to 1 (a fixed value for the mouse wheel), and send it on to your `pointerdown` handler:

```
divElement.addEventListener("wheel", function (e) {
    e.pointerId = 1;   // Fixed pointerId for MouseWheel
    pointerDown(e);
});
```

Now gesture events will start to come in *for that element*. (Remember that the mouse wheel by itself means translate, Ctrl+wheel means zoom, and Shift+Ctrl+wheel means rotate.) What's more, if additional `pointerdown` events occur for the same element with different `pointerId` values, the `addPointer` method will include that new pointer in the gesture. This automatically enables pinch-stretch and rotation gestures that rely on multiple points.

If you run the PointerEvents example (checking Ignore Mouse Events and Ignore Pointer Events) and start doing taps, tap-holds, and short drags (with touch or mouse), you'll see output like that shown in Figure 12-2. The dynamic effect is shown in Video 12-1.



☑ Ignore mouse events ☑ Ignore pointer events □ Ignore gesture events Show: Screen coordinates (screenX/Y)

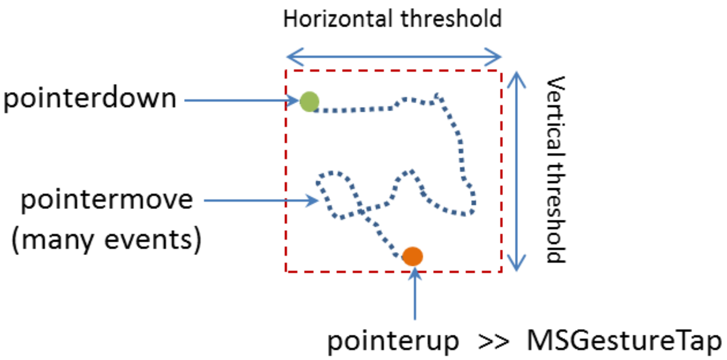
Pointer and gesture events will be picked up in this element and logged to the list to the right.

```
gestureEnd (e.detail = 10)
gestureChange  (detail = 8; screenX/Y= 182: 374)
gestureChange  (detail = 8; screenX/Y= 183: 374)
gestureChange  (detail = 8; screenX/Y= 184: 376)
gestureChange  (detail = 8; screenX/Y= 187: 379)
gestureChange  (detail = 8; screenX/Y= 192: 386)
gestureChange  (detail = 8; screenX/Y= 196: 392)
gestureChange  (detail = 8; screenX/Y= 204: 401)
inertiaStart (e.detail = 8)
gestureChange  (detail = 0; screenX/Y= 209: 408)
gestureChange  (detail = 0; screenX/Y= 218: 416)
gestureChange  (detail = 0; screenX/Y= 224: 423)
gestureChange  (detail = 0; screenX/Y= 227: 429)
gestureChange  (detail = 0; screenX/Y= 235: 445)
gestureChange  (detail = 0; screenX/Y= 238: 452)
gestureChange  (detail = 0; screenX/Y= 240: 459)
gestureChange  (detail = 0; screenX/Y= 240: 465)
gestureChange  (detail = 0; screenX/Y= 240: 474)
gestureChange  (detail = 0; screenX/Y= 238: 482)
gestureChange  (detail = 0; screenX/Y= 237: 490)
gestureChange  (detail = 0; screenX/Y= 234: 500)
gestureChange  (detail = 0; screenX/Y= 231: 506)
gestureStart (e.detail = 1)
gestureHold (e.detail = 2)
```

FIGURE 12-2 The PointerEvents example output for gesture events (screen shot cropped a bit to emphasize detail).

Again, gesture events are fired in response to a series of pointer events, offering higher-level interpretations of the lower-level pointer events. It's the process of interpretation that differentiates the tap/hold events from the start/change/end events, how and when the `MSInertiaStart` event kicks off, and what the gesture recognizer does when the `MSGesture` object is given multiple points.

Starting with a single pointer gesture, the first aspect of differentiation is a *pointer movement threshold*. When the gesture recognizer sees a `pointerdown` event, it starts to watch the `pointermove` events to see whether they stay inside that threshold, which is the effective boundary for tap and hold events. This accounts for and effectively ignores small amounts of jiggle in a mouse or a touch point as illustrated (or shall I say, exaggerated!) below, where a pointer down, a little movement, and a pointer up generates an `MSGestureTap`:



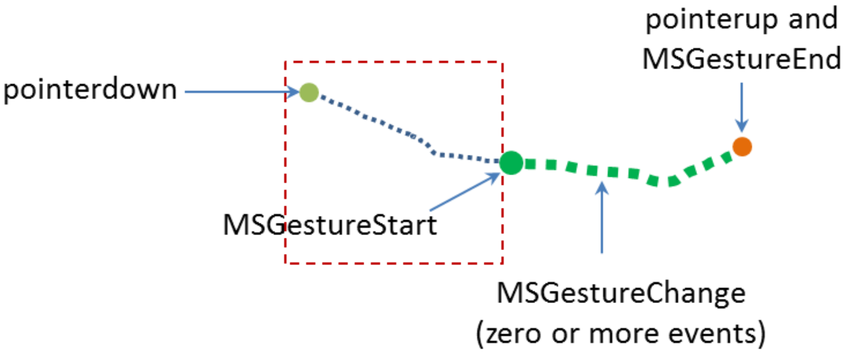
What then differentiates MSGestureTap and MSGestureHold is a *time threshold*:

- MSGestureTap occurs when pointerdown is followed by pointerup within the time threshold.
- MSGestureHold occurs when pointerdown is followed by pointerup outside the time threshold. MSGestureHold then fires once when the time threshold is passed with eventArgs.detail set to 1 (MSGESTURE_FLAG_BEGIN). Provided that the pointer is still within the movement threshold, MSGestureHold fires then again when pointerup occurs, with eventArgs.detail set to 2 (MSGESTURE_FLAG_END). You can see this detail included in the first two events of Figure 12-2 above.

The gesture flags in eventArgs.detail value is accompanied by many other positional and movement properties in the eventArgs object:

Properties	**Description**
screenX, screenY	The x- and y-coordinates of the gesture center point relative to the screen.
clientX, clientY	The x- and y-coordinates of the gesture center point relative to the client area of the app.
offsetX, offsetY	The x- and y-coordinates of the gesture center point relative to the element.
translationX, translationY	Translation along the x- and y-axes.
velocityX, velocityY	Velocity of movement along x- and y-axes.
scale	Scale factor for zoom (percentage change in the scale).
expansion	Diameter of the manipulation area (absolute change in size, in pixels).
velocityExpansion	Velocity of expanding manipulation area.
rotation	Rotation angle in radians.
velocityAngular	Angular velocity in radians.
detail	Contains the gesture flags that describe the gesture state of the event; these flags are defined as values in eventArgs itself: eventArgs.MSGESTURE_FLAG_NONE (0): Indicates ongoing gesture such as MSGestureChange where there is change in the coordinates. eventArgs.MSGESTURE_FLAG_BEGIN (1): The beginning of the gesture sequence. If the interaction contains single event such as MSGestureTap, both MSGESTURE_FLAG_BEGIN and MSGESTURE_FLAG_END flags will be set (detail will be 3). eventArgs.MSGESTURE_FLAG_END (2): The end of the gesture sequence. Again, if the interaction contains single event such as MSGestureTap, both MSGESTURE_FLAG_BEGIN and MSGESTURE_FLAG_END flags will be set (detail will be 3). eventArgs.MSGESTURE_FLAG_CANCEL (4): The gesture was cancelled. Always comes paired with MSGESTURE_FLAG_END, (detail will be 6). eventArgs.MSGESTURE_FLAG_INERTIA (8): The gesture is in an inertia state. The MSGestureChange event can be distinguished from direct interaction and timer driven inertia through this flag.
hwTimestamp	The timestamp of the pointer assigned by the system when the input was received from the hardware.

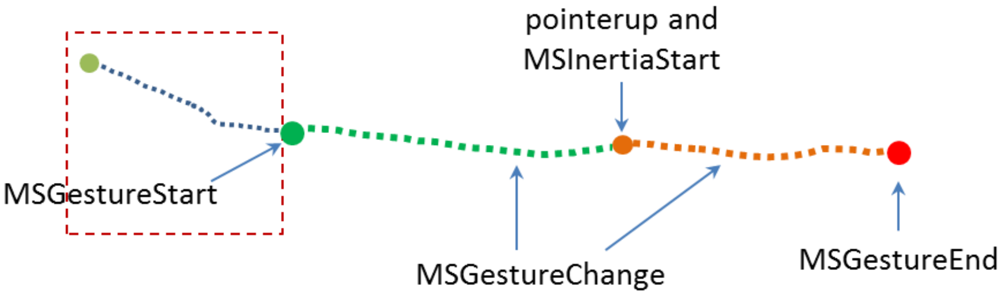
Many of these properties become much more interesting when a pointer moves *outside* the movement threshold, after which time you'll no longer see the tap or hold events. Instead, as soon as the pointer leaves the threshold area, MSGestureStart is fired, followed by zero or more MSGestureChange events (typically many more!), and completed with a single MSGestureEnd event:



Note that if a pointer has been held within the movement threshold long enough for the first MSGestureHold to fire with MSGESTURE_FLAG_BEGIN, but then the pointer is moved out of the threshold area, MSGestureHold will be fired a second time with MSGESTURE_FLAG_CANCEL | MSGESTURE_FLAG_END in eventArgs.detail (a value of 6), followed by MSGestureStart with MSGESTURE_FLAG_BEGIN. This series is how you differentiate a hold from a slide or drag gesture even if the user holds the item in place for a while.

Together, the MSGestureStart, MSGestureChange, and MSGestureEnd events define a *manipulation* of the element to which the gesture is attached, where the pointer remains in contact with the element throughout the manipulation. Technically, this means that the pointer was no longer moving when it was released.

If the pointer *was* moving when released, we switch from a manipulation to an *inertial* motion. In this case, an MSInertiaStart event gets fired to indicate that the pointer effectively continues to move even though contact was released or lifted. That is, you'll continue to receive MSGestureChange events until the movement is complete:



Conceptually, you can see the difference between a manipulation and an inertial motion, as illustrated in Figure 12-3. The curves shown here are not necessarily representative of actual changes between messages. If the pointer is moved along the green line such that it's no longer moving when released, we see the series of gesture that define a manipulation. If the pointer is released while moving, we see MSInertiaStart in the midst of MSGestureChange events and the event sequence follows the orange line.

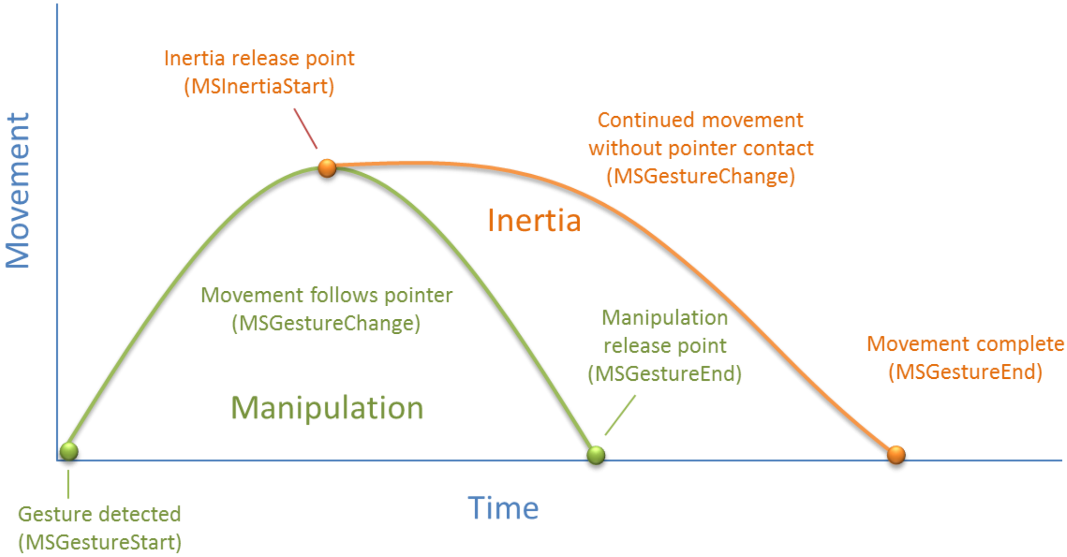


FIGURE 12-3 A conceptual representation of manipulation (green) and inertial (orange) motions.

Referring back to Figure 12-2, when the Show drop-down list (as shown!) is set to Velocity, the output for MSGestureChange events includes the eventArgs.velocity* values. During a manipulation, the velocity can change at any rate depending on how the pointer is moving. Once an inertial motion begins, however, the velocity will gradually diminish down to zero at which point MSGestureEnd occurs. The number of change events depends on how long it takes for the movement to slow down and come to a stop, of course, but if you're just moving an element on the display with these change events, the user will see a nice fluid animation. You can play with this in the PointerEvents example, using the Show drop-down list to also look at how the other positional properties are affected by different manipulations and inertial gestures.

Multipoint Gestures

What we've discussed so far has focused on a single point gesture, but the same is also true for multipoint gestures. When an MSGesture object is given multiple pointers through its addPointer event, it will also fire MSGestureStart, MSGestureChange, MSGestureEnd for rotations and pinch-stretch gestures, along with MSInertiaStart. In these cases, the scale, rotation, velocityAngular, expansion, and velocityExpansion properties in the eventArgs object become meaningful.

You can selectively view these properties for MSGestureChange events through the upper-right drop-down list in the PointerEvents example. You might notice is that if you do multipoint gestures in the Visual Studio simulator, you'll never see MSGestureTap events for the individual points. This is because the gesture recognizer can see that multiple pointerdown events are happening almost simultaneously (which is where the hwTimestamp property comes into play) and combines them into an MSGestureStart right away (for example, starting a pinch-stretch or rotation gesture).

Now I'm sure you're asking some important questions. While I've been speaking of pinch-stretch, rotation, and translation gestures as different things, how does one, in fact, differentiate these gestures when they're all coming into the app through the same MSGestureChange event? Doesn't that just make everything confusing? What's the strategy for translation, rotation, and scaling gestures?

Well, the answer is, you don't have to separate them! If you think about it for a moment, how you handle MSGestureChange events and the data each one contains depends on the kinds of manipulations you actually support in your UI:

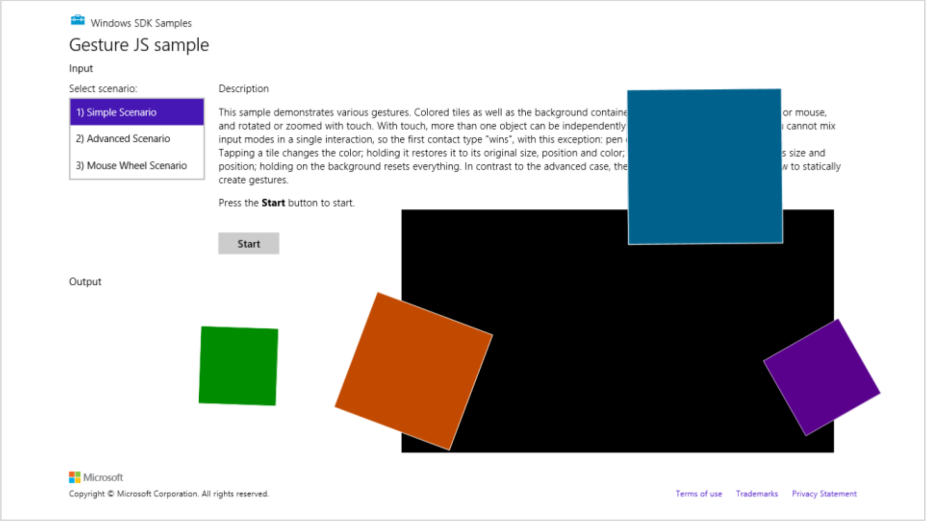
- If you're supporting only translation of an element, you'll simply never pay any attention to properties like scale and rotation and apply only those like translationX and translationY. This would be the expected behavior for selecting an item in a collection control, for example (or a control that allowed drag-and-drop of items to rearrange them).
- If you support only zooming, you'll ignore all the positional properties and work with scale, expansion, and/or velocityExpansion. This would be the sort of behavior you'd expect for a control that supported optical or Semantic Zoom.
- If you're interested in only rotation, the rotation and velocityAngular properties are your friends.

Of course, if you want to support multiple kinds of manipulations, you can simply apply all of these properties together, feeding them into combined CSS transforms. This would be expected of an app that allowed arbitrary manipulation of on-screen objects, and it's exactly what one of the gesture samples of the Windows SDK demonstrates.

The Input Instantiable Gesture Sample

While the PointerEvents example included with this chapter gives us a raw view of pointer and gesture events, what really matters to apps is how to apply these events to real manipulation of on-screen objects, which is to say, implementing parts of touch language such as pinch/stretch and rotation. For these we can turn to the Input Instantiable gestures sample.

This sample primarily demonstrates how to use gesture events on multiple elements simultaneously. In scenarios 1 and 2, the app simulates a simple example of a puzzle app, as mentioned earlier. Each colored box can be manipulated separately, using drag to move (with or without inertia), pinch-stretch gestures to zoom, and rotation gestures to rotate, as shown in Figure 12-4 and demonstrated in Video 12-2.



This sample demonstrates various gestures. Colored tiles as well as the background containe or mouse,
and rotated or zoomed with touch. With touch, more than one object can be independently cannot mix
input modes in a single interaction, so the first contact type "wins", with this exception: pen
Tapping a tile changes the color; holding it restores it to its original size, position and color; s size and
position; holding on the background resets everything. In contrast to the advanced case, the w to statically
create gestures.

Press the **Start** button to start.

Copyright © Microsoft Corporation. All rights reserved.

FIGURE 12-4 The Input Instantiable Gestures Sample after playing around a bit. The "instantiable" word comes from the need to instantiate an MSGesture object to receive gesture events.

In scenario 1 (js/instantiableGesture.js), an MSGesture object is created for each screen element along with one for the black background "table top" element during initialization (the initialize function). This is the same as we've already seen. Similarly, the pointerdown handler (onPointerDown) adds pointers to the gesture object for each element, adding a little more processing to manage z-index. This avoids having simultaneous touch, mouse, and stylus pointers working on the same element (which would be odd!):

```
function onPointerDown(e) {
    if (e.target.gesture.pointerType === null) {    // First contact
        e.target.gesture.addPointer(e.pointerId);   // Attaches pointer to element
        e.target.gesture.pointerType = e.pointerType;
    }
    else if (e.target.gesture.pointerType === e.pointerType) { // Contacts of similar type
        e.target.gesture.addPointer(e.pointerId);              // Attaches pointer to element
    }

    // ZIndex Changes on pointer down. Element on which pointer comes down becomes topmost
    var zOrderCurr = e.target.style.zIndex;
    var elts = document.getElementsByClassName("GestureElement");
    for (var i = 0; i < elts.length; i++) {
        if (elts[i].style.zIndex === 3) {
            elts[i].style.zIndex = zOrderCurr;
        }
        e.target.style.zIndex = 3;
    }
}
```

The MSGestureChange handler for each individual piece (onGestureChange) then takes all the translation, rotation, and scaling data in the eventArgs object and applies them with CSS. This shows how convenient it is that all those properties are already reported in the coordinate space we need:

```
function onGestureChange(e) {
    var elt = e.target;
    var m = new MSCSSMatrix(elt.style.msTransform);

    elt.style.msTransform = m.
        translate(e.offsetX, e.offsetY).
        translate(e.translationX, e.translationY).
        rotate(e.rotation * 180 / Math.PI).
        scale(e.scale).
        translate(-e.offsetX, -e.offsetY);
}
```

There's a little more going on in the sample, but what we've shown here are the important parts. Clearly, if you didn't want to support certain kinds of manipulations, you'd again simply ignore certain properties in the event args object.

Scenario 2 of this sample has the same output but is implemented a little differently. As you can see in its initialize function (js/gesture.js), the only events that are initially registered apply to the entire "table top" that contains the black background and a surrounding border. Gesture objects for the individual pieces are created and attached to a pointer within the pointerdown event (onTableTopPointerDown). This approach is much more efficient and scalable to a puzzle app that has hundreds or even thousands of pieces, because gesture objects are held only for as long as a particular piece is being manipulated. Those manipulations are also like those of scenario 1, where all the MSGestureChange properties are applied through a CSS transform. For further details, refer to the code comments in js/gesture.js, as they are quite extensive.

Scenario 3 of this sample provides another demonstration of performing translate, pinch-stretch, and rotate gestures using the mouse wheel. As shown in the PointerEvents example, the only thing you need to do here is process the wheel event, set eventArgs.pointerId to 1, and pass that on to your pointerdown handler that then adds the pointer to the gesture object:

```
elt.addEventListener("wheel", onMouseWheel, false);

function onMouseWheel(e) {
    e.pointerId = 1;  // Fixed pointerId for MouseWheel
    onPointerDown(e);
}
```

Again, that's all there is to it. (I love it when it's so simple!) As an exercise, you might try adding this little bit of code to scenarios 1 and 2 as well.

The Gesture Recognizer

With inertial gestures, which continue to send some number of MSGestureChange events after pointers are released, you might be asking this question: What, exactly, controls those events? That is, there is obviously a specific deceleration model built into those events, namely the one around which the look and feel of Windows is built. But what if you want a different behavior? And what if you want to interpret pointer events in different way altogether?

The agent that interprets pointer events into gesture events is called the gesture recognizer, which you can get to directly through the Windows.UI.Input.GestureRecognizer object. After instantiating this object with new, you set its gestureSettings properties for the kinds of manipulations and gestures you're interested in. The documentation for GestureSettings gives all the options here, which include tap, doubleTap, hold, holdWithMouse, rightTap, drag, translations, rotations, scaling, inertia motions, and crossSlide (swipe). For example, in the Input Gestures and manipulations with GestureRecognizer sample (js/dynamic-gestures.js) we can see how it configures a recognizer for tap, rotate, translate, and scale (with inertia):

```
gr = new Windows.UI.Input.GestureRecognizer();

// Configuring GestureRecognizer to detect manipulation rotation, translation, scaling,
// + inertia for those three components of manipulation + the tap gesture
gr.gestureSettings =
    Windows.UI.Input.GestureSettings.manipulationRotate |
    Windows.UI.Input.GestureSettings.manipulationTranslateX |
    Windows.UI.Input.GestureSettings.manipulationTranslateY |
    Windows.UI.Input.GestureSettings.manipulationScale |
    Windows.UI.Input.GestureSettings.manipulationRotateInertia |
    Windows.UI.Input.GestureSettings.manipulationScaleInertia |
    Windows.UI.Input.GestureSettings.manipulationTranslateInertia |
    Windows.UI.Input.GestureSettings.tap;

// Turn off UI feedback for gestures (we'll still see UI feedback for PointerPoints)
gr.showGestureFeedback = false;
```

The GestureRecognizer also has a number of properties to configure those specific events. With cross-slides, for example, you can set the crossSlideThresholds, crossSlideExact, and crossSlideHorizontally properties. You can set the deceleration rates (in pixels/ms^2) through inertiaExpansionDeceleration, inertiaRotationDeceleration, and inertiaTranslation-Deceleration.

Once configured, you then start passing pointer* events to the recognizer object, specific to its methods named processDownEvent, processMoveEvents, and processUpEvent (also processMouseWheelEvent, and processInertia, if needed). In response, depending on the configuration, the recognizer will then fire a number of its own events. First, there are discrete events like crossSliding, dragging, holding, rightTapped, and tapped. For all others it will fire a series of manipulationStarted, manipulationUpdated, manipulationInertiaStarting, and manipulationCompleted events. Note that all of these come from WinRT, so be sure to call removeEventListener as needed.

When you're using the recognizer directly, in other words, you'll be listening for `pointer*` events, feeding them to the recognizer, and then listening for and acting on the recognizer's specific events (as above) rather than the `MSGesture*` events that come out of the default recognizer configured by the `MSGesture` object.

Again, refer to the documentation on [GestureRecognizer]() for all the details and refer to the sample for some bits of code. As one extra example, here's a snippet to capture a small horizontal motion by using the `manipulationTranslateX` setting:

```
var recognizer = new Windows.UI.Input.GestureRecognizer();
recognizer.gestureSettings = Windows.UI.Input.GestureSettings.manipulationTranslateX;
var DELTA = 10;

myElement.addEventListener('pointerdown', function (e) {
    recognizer.processDownEvent(e.getCurrentPoint(e.pointerId));
});
myElement.addEventListener('pointerup', function (e) {
    recognizer.processUpEvent(e.getCurrentPoint(e.pointerId));
});
myElement.addEventListener('pointermove', function (e) {
    recognizer.processMoveEvents(e.getIntermediatePoints(e.pointerId));
});

// Remember removeEventListener as needed for this event
recognizer.addEventListener('manipulationcompleted', function (args) {
    var pt = args.cumulative.translation;
    if (pt.x < -DELTA) {
        // move right
    }
    else if (pt.x > DELTA) {
          // move left
    }
});
```

Beyond the recognizer, do note that you can always go the low-level route and do your own processing of `pointer*` events however you want, completely bypassing the gesture recognizer. This would be necessary if the configurations allowed by the recognizer object don't accommodate your specific need. At the same time, now is a good opportunity to re-read "Sidebar: Creating Completely New Gestures?" at the end of "The Touch Language and Mouse/Keyboard Equivalents" earlier. It addresses a few of the questions about when and whether custom gestures are needed.

Keyboard Input and the Soft Keyboard

After everything to do with touch and other forms of input, it seems almost anticlimactic to consider the humble keyboard, yet the keyboard remains utterly important for textual input, whether it's a physical key-board or the on-screen "soft" keyboard. It's especially important for accessibility as well, because some users are physically unable to use a mouse or other devices. In fact, the App certification requirements (section 6.13.4) requires that you disclose anything short of full keyboard support.

Fortunately, there is nothing special about handling keyboard input in a Windows Store app and a little goes a long way. Drawing from Implementing keyboard accessibility, here's a summary:

- Process keydown, keyup, and keypress events as you already know how to do, especially for implementing special key combinations. See "Standard Keystrokes" later for a quick run-down of typical mappings.
- Have tabindex attributes on interactive elements that should be tab stops. Avoid adding tabindex to noninteractive elements because this will interfere with screen readers.
- Have accesskey attributes on those elements that should have keyboard shortcuts. Try to keep these simple so that they're easier to use with the Sticky Keys accessibility feature.
- Call the DOM focus API on whatever element should be the default.
- Take advantage of the keyboard support that exists in built-in controls, such as the App Bar.

As an example, the Here My Am! app we've been working with in this book (in this chapter's companion content) now has full keyboard support. This was mostly a matter of adding tabindex to a few elements, setting focus to the image area, and picking up keydown events on the img elements for the Enter key and spacebar where we've already been handling click. Within those keydown events, note that it's helpful to use the WinJS.Utilities.Key enumeration for comparing key codes:

```
var Key = WinJS.Utilities.Key;
var image = document.getElementById("photo");

image.addEventListener("keydown", function (e) {
    if (e.keyCode == Key.enter || e.keyCode==Key.space) {
        image.click();
    }
});
```

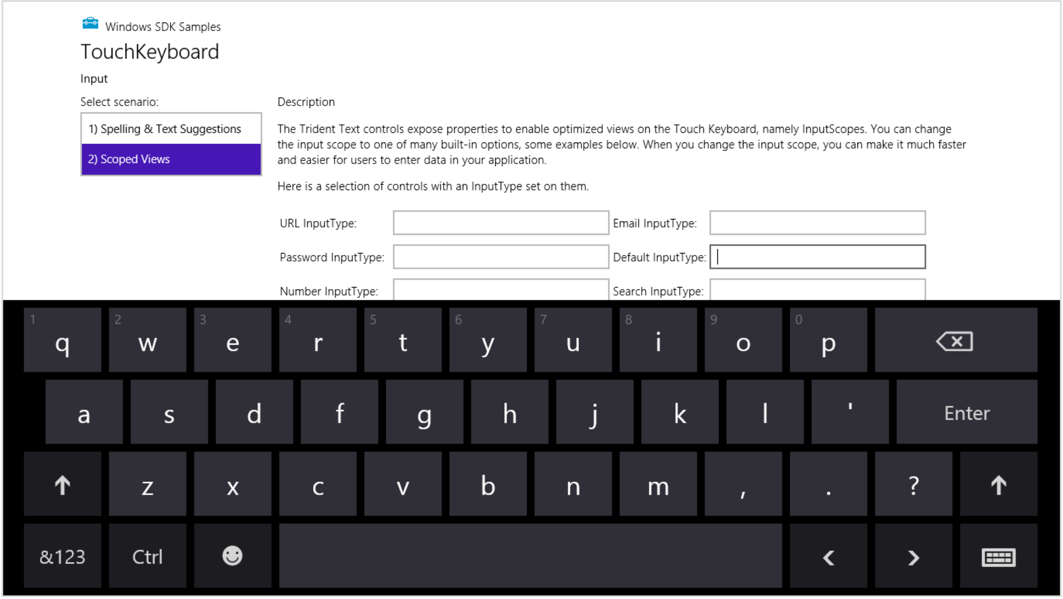
All this works for both the physical keyboard as well as the soft keyboard. Case closed? Well, not entirely. Two special concerns with the soft keyboard exist: how to make it appear, and the effect of its appearance on app layout. After covering those, I'll also provide a quick run-down of standard keystrokes for app commands.

Soft Keyboard Appearance and Configuration

The appearance of the soft keyboard happens for one reason and one reason only: the user *touches* a text input element or an element with the contenteditable="true" attribute (such as a div or canvas). There isn't an API to make the keyboard appear, nor will it appear when you click in such an element with the mouse or a stylus or tab to it with a physical keyboard.

The configuration of the keyboard is also sensitive to the type of input control. We can see this through scenario 2 of the Input Touch keyboard text input sample, where html/ScopedViews.html contains a bunch of input controls (surrounding table markup omitted), which appear as shown in Figure 12-5:

```
<input type="url" name="url" id="url" size="50" />
<input type="email" name="email" id="email" size="50" />
<input type="password" name="password" id="password" size="50" />
<input type="text" name="text" id="text" size="50" />
<input type="number" name="number" id="number" />
<input type="search" name="search" id="search" size="50" />
<input type="tel" name="tel" id="tel" size="50" />
```



Windows SDK Samples

TouchKeyboard

Input

Select scenario:

1) Spelling & Text Suggestions

2) Scoped Views

Description

The Trident Text controls expose properties to enable optimized views on the Touch Keyboard, namely InputScopes. You can change the input scope to one of many built-in options, some examples below. When you change the input scope, you can make it much faster and easier for users to enter data in your application.

Here is a selection of controls with an InputType set on them.

URL InputType:

Email InputType:

Password InputType:

Default InputType:

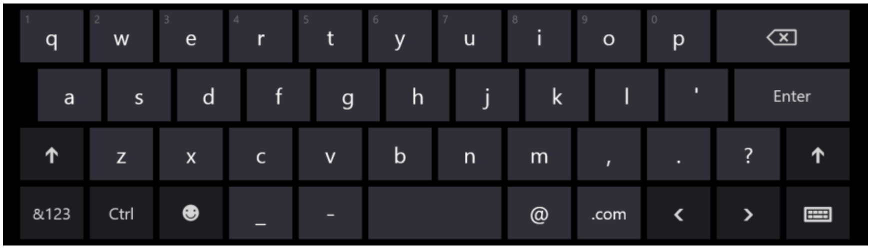
Number InputType:

Search InputType:

1 q 2 w 3 e 4 r 5 t 6 y 7 u 8 i 9 o 0 p
a s d f g h j k l ' Enter
z x c v b n m , . ?
&123 Ctrl

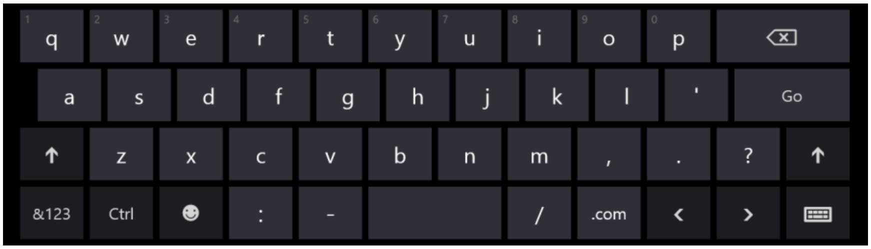
FIGURE 12-5 The soft keyboard appears when you touch an input field, as shown in the Input Touch keyboard text input sample (scenario 2). The exact layout of the keyboard changes with the type of input field.

What's shown in Figure 12-5 is the default keyboard. If you tap in the Search field, you get pretty much the same view as Figure 12-5 except the Enter key turns into Search . For the Email field, it's much like the default view except you get @ and .com keys near the spacebar:

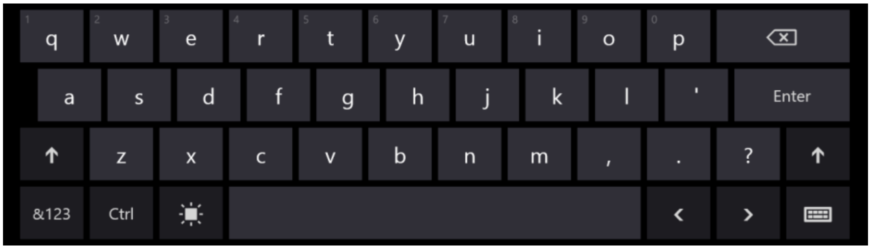


1 q 2 w 3 e 4 r 5 t 6 y 7 u 8 i 9 o 0 p
a s d f g h j k l ' Enter
z x c v b n m , . ?
&123 Ctrl _ - @ .com

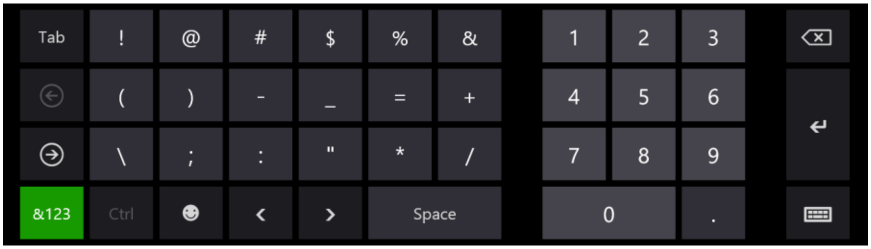
The URL keyboard is the same except a few keys change and Enter turns into Go:


1 q 2 w 3 e 4 r 5 t 6 y 7 u 8 i 9 o 0 p ⌫
a s d f g h j k l ' Go
↑ z x c v b n m , . ? ↑
&123 Ctrl ☻ : - / .com < > ⌨

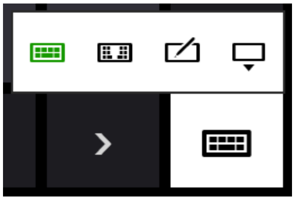
For passwords you get a key to hide keypresses (below, to the left of the spacebar), which prevents a visible animation from happening on the screen—a very important feature if you're recording videos!


1 q 2 w 3 e 4 r 5 t 6 y 7 u 8 i 9 o 0 p ⌫
a s d f g h j k l ' Enter
↑ z x c v b n m , . ? ↑
&123 Ctrl < > ⌨

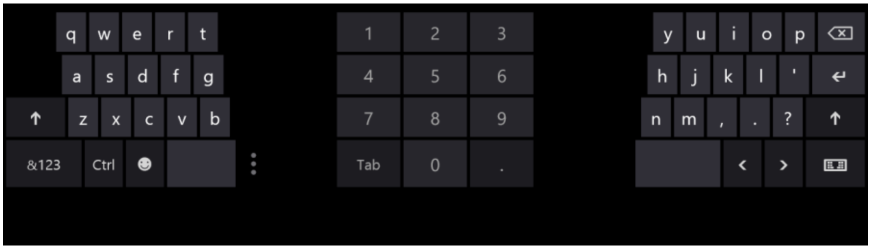
And finally, the Number and Telephone fields bring up a number-oriented view:

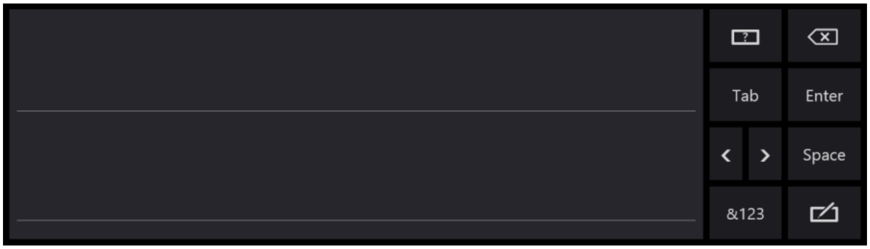


In all of these cases, the key on the lower right (whose icon looks a bit like a keyboard) lets you switch to other keyboard layouts:



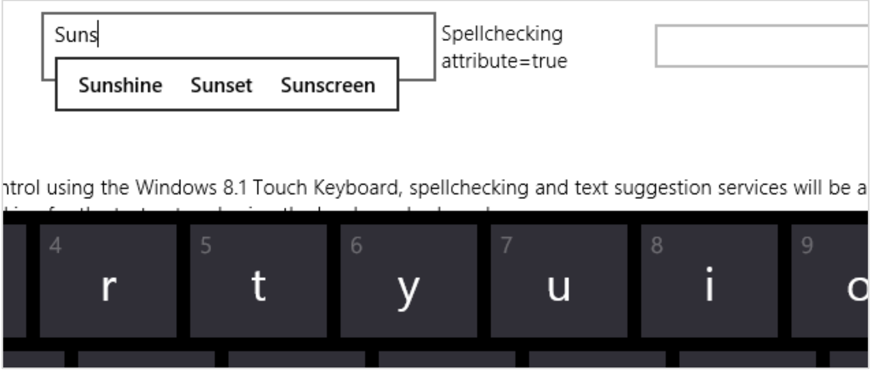
The options here are the normal (wide) keyboard, the split keyboard, a handwriting recognition panel, and a key to dismiss the soft keyboard entirely. Here's what the default split keyboard and handwriting panels look like:





This handwriting panel for input is simply another mode of the soft keyboard: you can switch between the two, and your selection sticks across invocations. (For this reason, Windows does not automatically invoke the handwriting panel for a pen pointer, because the user may prefer to use the soft keyboard even with the stylus.)

And although the default keyboard appears for text input controls, those controls also provide text suggestions for touch users. This is demonstrated in scenario 1 of the sample and shown below:


Suns

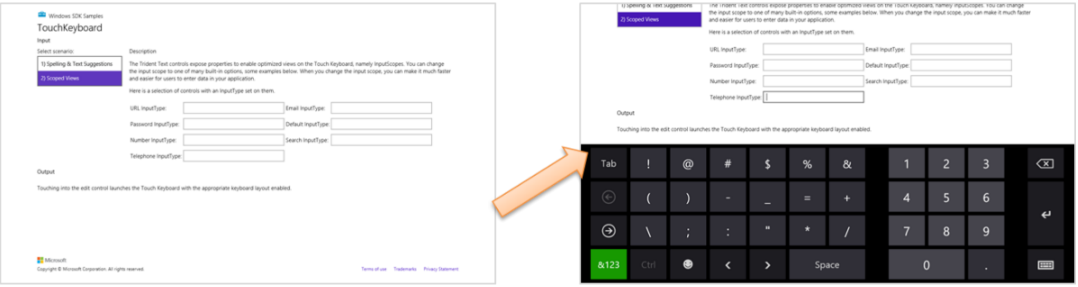
Spellchecking attribute=true

ntrol using the Windows 8.1 Touch Keyboard, spellchecking and text suggestion services will be a

Adjusting Layout for the Soft Keyboard

The second concern with the soft keyboard (no, I didn't forget!) is handling layout when the keyboard might obscure the input field with the focus.

When the soft keyboard or handwriting panel appears, the system tries to make sure the input field is visible by scrolling the page content if it can. This means that it just sets a negative vertical offset to your entire page equal to the height of the soft keyboard. For example, on a 1366x768 display (as in the simulator), touching the Telephone Input Type field in scenario 2 of the Input Touch keyboard text input sample will slide the whole page upward, as shown in Figure 12-6 and also Video 12-3.



Windows SDK Samples

TouchKeyboard

Input

Select scenario:

1) Spelling & Text Suggestions

2) Scoped Views

Description

The Trident Text controls expose properties to enable optimized views on the Touch Keyboard, namely InputScopes. You can change the input scope to one of many built-in options, some examples below. When you change the input scope, you can make it much faster and easier for users to enter data in your application.

Here is a selection of controls with an InputType set on them.

URL InputType:
Email InputType:
Password InputType:
Default InputType:
Number InputType:
Search InputType:
Telephone InputType:

Output

Touching into the edit control launches the Touch Keyboard with the appropriate keyboard layout enabled.

Microsoft

Copyright © Microsoft Corporation. All rights reserved.

FIGURE 12-6 When the soft keyboard appears, Windows will automatically slide the app page up to make sure the input field isn't obscured.

Although this can be the easiest solution for this particular concern, it's not always ideal. Fortunately, you can do something more intelligent if you'd like by listening to the `hiding` and `showing` events of the `Windows.UI.ViewManagement.InputPane` object and adjust your layout directly. Code for doing this can be found in the—are you ready for this one?—Responding to the appearance of the on-screen keyboard sample.[91] Adding listeners for these events is simple (see the bottom of js/keyboardPage.js, which also removes the listeners properly):

```
var inputPane = Windows.UI.ViewManagement.InputPane.getForCurrentView();
inputPane.addEventListener("showing", showingHandler, false);
inputPane.addEventListener("hiding", hidingHandler, false);
```

Within the `showing` event handler, the `eventArgs.occludedRect` object (a `Windows.Foundation.Rect`) gives you the coordinates and dimensions of the area that the soft keyboard is covering. In response, you can adjust whatever layout properties are applicable and set the `eventArgs.ensuredFocusedElementInView` property to `true`. This tells Windows to bypass its automatic offset behavior:

```
function showingHandler(e) {
    if (document.activeElement.id === "customHandling") {
        keyboardShowing(e.occludedRect);

        // Be careful with this property. Once it has been set, the framework will
        // do nothing to help you keep the focused element in view.
        e.ensuredFocusedElementInView = true;
    }
```

[91] And although you might think this is a strong contender for the longest JavaScript sample name in the Windows SDK, it runs a mere sixth. The top three are the Unselectable content areas with -ms-user-select CSS attribute sample, which wins the bronze; AppContainer, mobile broadband pin, connection, and device management sample (Appendix C), which grabs the silver; and, in first place, Windows Runtime in-process component authoring with proxy\stub generation sample (Chapter 18, "WinRT Components")! I don't mind such long names, however—I'm delighted that we have such an extensive set of great samples to draw from.

```
}
```

The sample shows both cases. If you tap on the aqua-colored *defaultHandling* element on the bottom left of the app, as shown in Figure 12-7, this `showingHandler` does nothing, so the default behavior occurs. See the dynamic effect in Video 12-4.

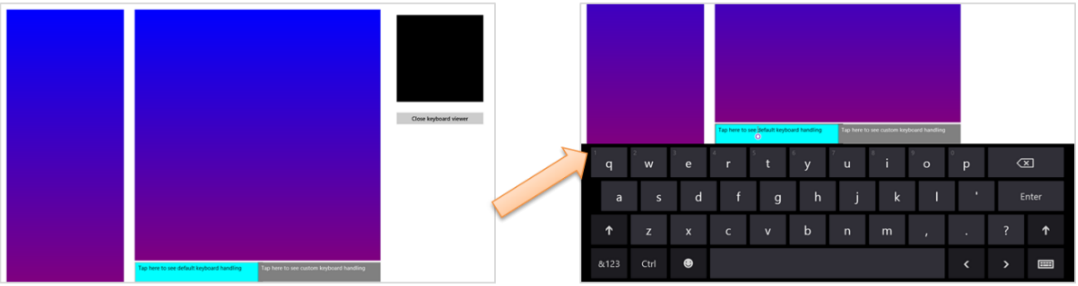

Close keyboard viewer
Tap here to see default keyboard handling
Tap here to see custom keyboard handling
Tap here to see default keyboard handling
Tap here to see custom keyboard handling
q w e r t y u i o p
a s d f g h j k l '
Enter
z x c v b n m , . ?
&123
Ctrl
&lt;
>

FIGURE 12-7 Tapping on the left *defaultHanding* element at the bottom shows the default behavior when the keyboard appears, which offsets other page content vertically.

If you tap the *customHandling* element (on the right), it calls its `keyboardShowing` routine to do layout adjustment:

```
function keyboardShowing(keyboardRect) {
    // Some code omitted...

    var elementToAnimate = document.getElementById("middleContainer");
    var elementToResize = document.getElementById("appView");
    var elementToScroll = document.getElementById("middleList");

    // Cache the amount things are moved by. It makes the math easier
    displacement = keyboardRect.height;
    var displacementString = -displacement + "px";

    // Figure out what the last visible things in the list are
    var bottomOfList = elementToScroll.scrollTop + elementToScroll.clientHeight;

    // Animate
    showingAnimation = KeyboardEventsSample.Animations.inputPaneShowing(elementToAnimate,
        { top: displacementString, left: "0px" }).then(function () {

        // After animation, layout in a smaller viewport above the keyboard
        elementToResize.style.height = keyboardRect.y + "px";

        // Scroll the list into the right spot so that the list does not appear to scroll
        elementToScroll.scrollTop = bottomOfList - elementToScroll.clientHeight;
        showingAnimation = null;
    });
}
```

The code here is a little involved because it's animating the movement of the various page elements. The layout of affected elements—namely the one that is tapped—is adjusted to make space for the keyboard. Other elements on the page are otherwise unaffected. The result is shown in Figure 12-8.

Again, the dynamic effect is shown in Video 12-4 in contrast to the default effect.

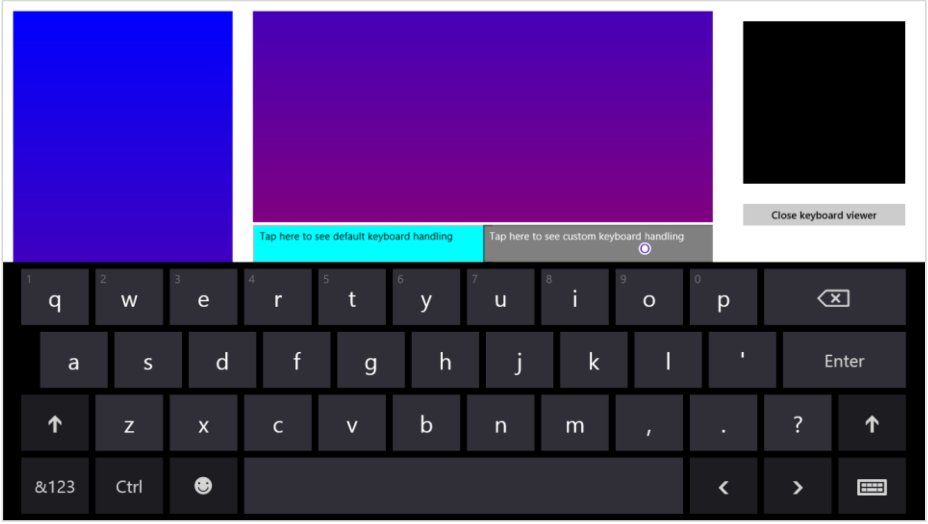

Close keyboard viewer
Tap here to see default keyboard handling
Tap here to see custom keyboard handling
q w e r t y u i o p
a s d f g h j k l ' Enter
z x c v b n m , . ?
&123
Ctrl

FIGURE 12-8 Tapping the gray *customHandling* element on the right shows custom handling for the keyboard's appearance.

Standard Keystrokes

The last piece I wanted to include on the subject of the keyboard is a table of command keystrokes you might support in your app. These are in addition to the touch language equivalents, and you're probably accustomed to using many of them already. They're good to review because, again, apps should be fully usable with just the keyboard and implementing keystrokes like these goes a long way toward fulfilling that requirement and enabling more efficient use of your app by keyboard users.

Action or Command	**Keystroke**
Move focus	Tab
Back (navigation)	Back button on special keyboards; backspace if not in a text field; Alt+left arrow
Forward (navigation)	Alt+right arrow
Up	Alt+up arrow
Cancel/Escape from mode	ESC
Walk through items in a list	Arrow keys (plus Tab)
Jump through items in a list to next group if selection doesn't automatically follow focus	Ctrl+arrow keys
Zoom (semantic and optical)	Ctrl+ and Ctrl-
Jump to something in a named collection	Start typing
Jump far	Page up/down (should work in panning UI, in either horizontal or vertical directions)
Next tab or group	Ctrl+Tab
Previous tab or group	Ctrl+Shift+Tab
*N*th tab or group	Ctrl+*N* (1-9)

Open app bar (Windows handles this automatically)	Win+Z
Context menu	Context menu key
Open additional flyout/select menu item	Enter
Navigate into/activate	Enter (on a selection)
Select	Space
Select contiguous	Shift+arrow keys
Pin this	Ctrl+Shift+!
Save	Ctrl+S
Find	Ctrl+F
Print	Ctrl+P (call `Windows.Graphics.Printing.PrintManager.showPrintUIAsync`)
Copy	Ctrl+C
Cut	Ctrl+X
Paste	Ctrl+V
New Item	Ctrl+N
Open address	Ctrl+L or Alt+D
Rotate	Ctrl+, and Ctrl+.
Play/Pause	Ctrl+P (media apps only)
Next item	Ctrl+F (conflict with Find)
Previous item	Ctrl+B
Rewind	Ctrl+Shift+B
Fast forward	Ctrl+Shift+F

Inking

Beyond the built-in soft keyboard/handwriting pane, an app might also want to provide a surface on which it can directly accept pointer input as *ink*. By this I mean more than just having a `canvas` element and processing `pointer*` events to draw on it to produce a raster bitmap. Ink is a data structure that maintains the complete input stream—including pressure, angle, and velocity if the hardware supports it—which allows for handwriting recognition and other higher-level processing that isn't possible with raw pixel data. In other words, ink remembers *how* an image was drawn, not just the final image itself, and it works with all types of pointer input.

Ink support in WinRT is found in the `Windows.UI.Input.Inking` namespace. This API doesn't depend on any particular presentation framework, nor does it provide for rendering: it deals only with managing the data structures that an app can process however it wants or simply render to a drawing surface such as a `canvas`. Here's how inking works:

- Create an instance of the manager object with `new Windows.UI.Input.Inking.InkManager()`.
- Assign any drawing attributes by creating an `InkDrawingAttributes` object and settings attributes like the ink `color`, `fitToCurve` (as opposed to the default straight lines), `ignorePressure`, `penTip` (`PenTipShape.circle` or `rectangle`), and `size` (a

`Windows.Foundation.Size` object with `height` and `width`).

- For the input element, listen for the `pointerdown`, `pointermove`, and `pointerup` events, which you generally need to handle for display purposes anyway. The `eventArgs.currentPoint` is a `Windows.UI.Input.PointerPoint` object that contains a pointer id, point coordinates, and properties like pressure, tilt, and twist.
- Pass that `PointerPoint` object to the ink manager's `processPointerDown`, `processPointerUpdate`, and `processPointerUp` methods, respectively.
- After `processPointerUp`, the ink manager will create an `InkStroke` object for that path. Those strokes can then be obtained through the ink manager's `getStrokes` method and rendered as desired.
- Higher-order gestures can be also converted into `InkStroke` objects directly and given to the manager through its `addStroke` method. Stroke objects can also be deleted with `deleteStroke`.

The ink manager also provides methods for performing handwriting recognition with its contained strokes, saving and loading the data, and handling different modes like draw and erase. For a complete demonstration, check out the Input Ink sample that is shown in Figure 12-9. This sample lets you see the full extent of inking capabilities, including handwriting recognition.

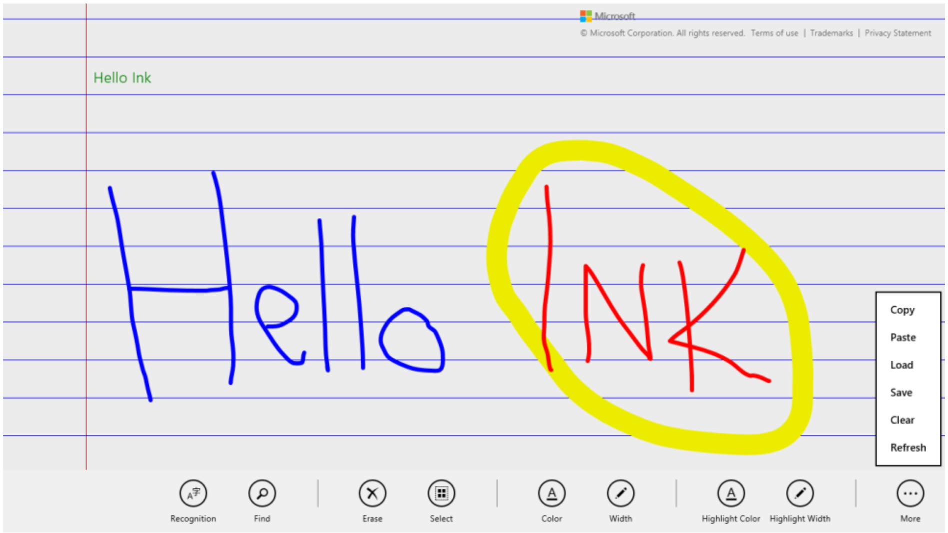


FIGURE 12-9 The Input Ink sample with many commands on its app bar. The small, green "Hello Ink" text in the upper left was generated by tapping the Recognition command.

The SDK also includes the Input Simplified ink sample to demonstrate a more focused handwriting recognition scenario, as shown in Figure 12-10. You should know that this is one sample that *doesn't* support touch at all—it's strictly for mouse or stylus, and it uses keystrokes for various commands instead of an app bar. Look at the `keydown` function in simpleink.js for a list of the Ctrl+key commands; the spacebar performs recognition of your strokes, and the backspace key clears the canvas. As you can see in the figure, I think the handwriting recognition is quite good! (It tells me that the handwriting samples I gave to an engineering team at Microsoft somewhere in the early 1990s must have ma de a valuable contribution.)

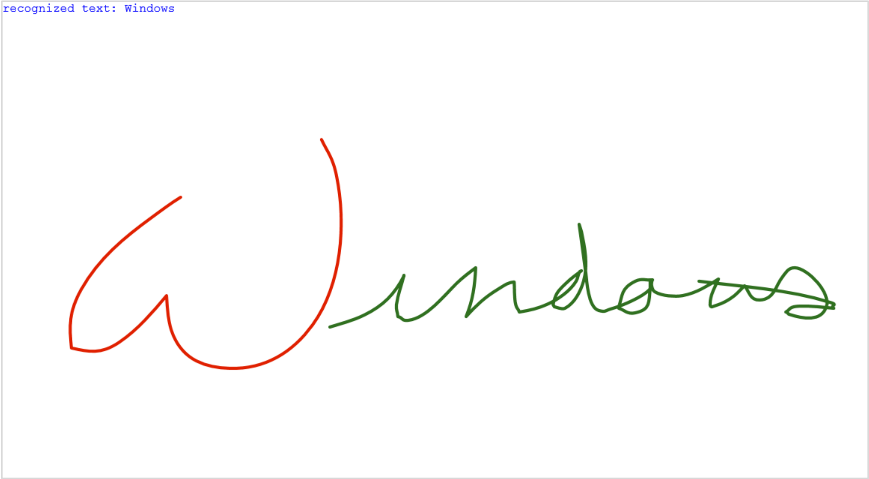

recognized text: Windows

FIGURE 12-10 The Input Simplified Ink sample doing a great job recognizing my sloppy mouse-based handwriting.

Geolocation

Before we explore sensors more generally, I want to separately call out the geolocation capabilities for Windows Store apps because its API is structured differently from other sensors. We've already used this since Chapter 2, "Quickstart," in the Here My Am! app, but we need the more complete story of this highly useful capability.

Unlike all other sensors, in fact, geolocation is the *only* one that has an associated capability you must declare in the manifest. Where you are on the earth is an absolute measure, if you will, and is therefore classified as a piece of personal information. So, users must give their consent before an app can obtain that information, and to pass Windows Store certification your app must also provide a Privacy Statement about how it will use that information. Other sensor data, in contrast, is relative—you cannot, for example, really know anything about a *person* from how a device is tilted, how it's moving, or how much light is shining on it. Accordingly, you can use those others sensors without declaring any specific capabilities.

As you might know, geolocation can be obtained in two different ways. The primary and most precise way, of course, is to get a reading from an actual GPS radio that is talking to geosynchronous satellites some hundreds of miles up in orbit. The other reasonably useful means, though not always accurate, is to attempt to find one's position through the IP address of a wired network connection or to triangulate from the position of available Wi-Fi hotspots. Whatever the case, WinRT will do its best to give you a decent reading.

To access geolocation readings, you must first create an instance of the WinRT geolocator, `Windows.Devices.Geolocation.Geolocator`. With that in hand, you can then call its `getGeopositionAsync` method, whose result, delivered to your completed handler, is a `Geoposition` object (in the same `Windows.Devices.Geolocation` namespace, as everything here is unless noted). Here's the code as it appears in Here My Am!:

```
//Make sure this variable stays in scope while getGeopositionAsync is happening.
var locator = new Windows.Devices.Geolocation.Geolocator();

locator.getGeopositionAsync().done(function (position) {
    var position = geocoord.coordinate.point.position;

    //Save for share
    app.sessionState.lastPosition =
        { latitude: position.latitude, longitude: position.longitude };
}
```

Tip As suggested by the code comment here, the variable that holds the `Geolocator` object must presently stay in scope while the `getGeopositionAsync` call is in process, otherwise that call is canceled. For this reason, the `locator` variable in Here My Am! (the first line of code above) is declared outside the function that calls `getGeopositionAsync`.

The `getGeopositionAsync` method also has a variation where you can specify two parameters: a maximum age for a cached reading (which is to say, how stale you can allow a reading to be) and a timeout value for how long you're willing to wait for a response. Both values are in milliseconds.

A `Geoposition` contains two properties:

- `coordinate` A `Geocoodinate` object that provides `accuracy` (meters), `altitudeAccuracy` (meters), `heading` (degrees relative to true north), `point` (a `Geopoint` that contains the coordinates, altitude, and some other detailed data), `positionSource` (a value from `PositionSource` identifying how the location was obtained, e.g. `cellular`, `satellite`, `wiFi`, `ipAddress`, and `unknown`), `satelliteData` (a `GeocoordinateSatelliteData` object), `speed` (meters/sec), and a `timestamp` (a `Date`).
- `civicAddress` A `CivicAddress` object, which might contain `city` (string), `country` (string, a two-letter ISO-3166 country code), `postalCode` (string), `state` (string), and `timestamp` (Date) properties, if the geolocation provider supplies such data.[92]

You can indicate the accuracy you're looking for through the Geolocator's `desiredAccuracy` property, which is either `PositionAccuracy.default` or `PositionAccuracy.high`. The latter, mind

[92] That is, the `civicAddress` property might not be available or might be empty. An alternate means to obtain it is to use the Bing Maps API, specifically the MapAddress class, to convert coordinates into an address.

you, will be much more radio- or network-intensive. This might incur higher costs on metered broadband connections and can shorten battery life, so set this to high only if it's essential to your user experience. You can also be more specific by using Geolocator.desiredAccuracyInMeters, which will override desiredAccuracy.

The Geolocator also provides a locationStatus property, which is a value from the PositionStatus enumeration: ready, initializing, noData, disabled, notInitialized, or notAvailable. It should be obvious that you can't get data from a Geolocator that's in any state other than ready. To track this, listen to the Geolocator's statuschanged event, where eventArgs.status in your handler contains the new PositionStatus; this is helpful when you find that a GPS device might take a couple seconds to provide a reading. For an example of using this event, see scenario 1 of the Geolocation sample in the Windows SDK (js/scenario1.js):

```
geolocator = new Windows.Devices.Geolocation.Geolocator();
geolocator.addEventListener("statuschanged", onStatusChanged);  //Remember to remove later

function onStatusChanged(e) {
    switch (e.status) {
    // ...
    }
}
```

PositionStatus and statuschanged reflect both the readiness of the GPS device as well as the Location permission for the app, as set through the Settings charm or through PC Settings > Privacy > Location (status is disabled if permission is denied). You can use this event, therefore, to detect changes to permissions while the app is running and to respond accordingly. Of course, it's possible for the user to change permission in PC Settings while your app is suspended, so you'll typically want to check Geolocator status in your resuming event handler as well.

The other two interesting properties of the Geolocator are movementThreshold, a distance in meters that the device can move before another reading is triggered (which can be used for geo-fencing scenarios), and reportInterval, which is the number of milliseconds between attempted readings. Be conservative with the latter, setting it to what you really need, again to minimize network or radio activity. In any case, when the Geolocator takes another reading and finds that the device has moved beyond the movementThreshold, it will fire a positionchanged event, where the eventArgs.position property is a new Geoposition object. This is also shown in scenario 1 of the Geolocation sample (js/sceario2.js):

```
geolocator.addEventListener("positionchanged", onPositionChanged);

function onPositionChanged(e) {
    var coord = e.position.coordinate;

    document.getElementById("latitude").innerHTML = coord.point.position.latitude;
    document.getElementById("longitude").innerHTML = coord.point.position.longitude;
    document.getElementById("accuracy").innerHTML = coord.accuracy;
}
```

With movementThreshhold and reportInterval, really think through what your app needs based on the accuracy and/or refresh intervals of the data you're using in relation to the location. For example, weather data is regional and might be updated only hourly. Therefore, movementThreshold might be set on the scale of miles or kilometers and reportInterval at 15, 30, or 60 minutes, or longer. A mapping or real-time traffic app, on the other hand, works with data that is very location-sensitive and will thus have a much smaller threshold and a much shorter interval.

For similar purposes you can also use the more power-efficient *geofencing* capabilities, which we'll talk about very soon.

Where battery life is concerned, it's best to simply take a reading when the user wants one, rather than following the position at regular intervals. But this again depends on the app scenario, and you could also provide a setting that lets the user control geolocation activity.

It's also very important to note that apps won't get positionchanged or statuschanged events while suspended unless you register a time trigger background task for this purpose and the user adds the app to the lock screen. We'll talk more of this in Chapter 16, "Alive with Activity," and you can also see how this works in scenario 3 of the Geolocation sample. If, however, you don't use a background task or the user doesn't place you on the lock screen and you still want to track the user's position, be sure to handle the resuming event and refresh the position there.

On the flip side, some geolocation scenarios, such as providing navigation, need to also keep the display active (preventing automatic screen shutoff) even when there's no user activity. For this purpose you can use the Windows.System.Display.DisplayRequest class, namely its requestActive and releaseRelease methods that you would call when starting and ending a navigation session. Of course, because keeping the display active consumes more battery power, only use this capability when necessary—as when specifically providing navigation—and avoid simply making the request when your app starts. Otherwise your app will probably gain a reputation in the Windows Store as being power-hungry!

Sidebar: HTML5 Geolocation

An experienced HTML/JavaScript developer might wonder why WinRT provides a Geolocation API when HTML5 already has one: window.navigator.geolocation and its getCurrent-Position method that returns an object with coordinates. The reason for the overlap is that other languages like C#, Visual Basic, and C++ don't have another API to draw from, which leaves HTML/JavaScript developers a choice. Under the covers, the HTML5 API hooks into the same data as the WinRT API, requires the same manifest capability (*Location*), and is subject to the same user consent, so for the most part the two APIs are almost equivalent in basic usage. The WinRT API, however, also supports the movementThreshold option, which helps the app cooperate with power management, along with geofencing. Like all other WinRT APIs, however, Windows.Devices.Geolocation is available only in local context pages in a Windows Store app. In web context pages you can use the HTML5 API.

Geofencing

A *geofence* is defined as a virtual perimeter around a real-world geographic location, such that entering into or leaving that perimeter will trigger events. Perimeters can be established anywhere and can be static or dynamic—it doesn't matter. And you can really do anything with geofencing events, such as providing guidebook information for the present location, coupons for nearby merchants, reminders for bus/train stops, shopping lists for the store you just walked into, automatic check-ins to social media, and so on. In short, although geolocation tells you where the device is located on the earth, simple positions in terms of latitude and longitude are not all that meaningful to human beings. Geofencing lets you define zones that do have meaning to an individual user, and it lets you know exactly when the device—presumably with that user!—has crossed into or out of those zones so that your app can take equally meaningful action.

The Geolocator's `movementThreshhold` and `reportInterval` properties can help you implement some basic types of geofencing. The drawbacks, however, are many. For one, you'd have to constantly calculate the difference between the current location and the coordinates of your geofencing zones, which can be cumbersome. The app and/or its background tasks would also need to be running quite a lot, which drains battery power. It would also need to be watching for changes in the Geolocator's status, especially when a device switches the underlying provider.

Fortunately, the APIs in `Windows.Devices.Geolocation.Geofencing` encapsulates all of this to spare you such details. After all, what you want to do in an app is concentrate on helping the user set up their meaningful zones and then on bringing up appropriate content, rather than messing with geospatial mathematics! And because the WinRT API can provide these services to multiple apps simultaneously, it can do so more efficiently and thus conserve power.

The Geofencing API provides the means to dynamically create geofences (zones) and fires events on entry and exit from a geofence. It also lets you set up time windows during which those geofences are active, set *dwell times* for those geofences (how long the device must be in or out of a zone before raising an event), and set up background tasks with a *location trigger* to specifically watch for geofence events when the app isn't active.

Each geofence you want to monitor is based on a `Geocircle` object, whose `center` property is a `BasicGeoposition` containing latitude and longitude and whose `radius` property defines a circle around that center (the units depend on the `altitudeReferenceSystem` property, nominally in meters). Clearly, then, only circular zones are supported at present.

Taking the `Geocircle`, you create a `Geofence` object from it along with an identity key, a mask that defines the events of interest (`MonitoredGeofenceStates`), the dwell time, the start time, and the duration. Here's how it's done in the Geolocation sample (js/scenario4UIHandlers.js):

```
Function generateGeofence() {
    var geofence = null;
    try {
        var fenceKey = nameElement.value;
        var position = {
            latitude: decimalFormatter.parseDouble(latitude.value),
            longitude: decimalFormatter.parseDouble(longitude.value),
            altitude: 0
        };
        var radiusValue = decimalFormatter.parseDouble(radius.value);

        // the geofence is a circular region
        var geocircle = new Windows.Devices.Geolocation.Geocircle(position, radiusValue);

        var singleUse = false;

        if (geofenceSingleUse.checked) {
            singleUse = true;
        }

        // want to listen for enter geofence, exit geofence and remove geofence events
        var mask = 0;

        mask = mask | Windows.Devices.Geolocation.Geofencing.MonitoredGeofenceStates.entered;
        mask = mask | Windows.Devices.Geolocation.Geofencing.MonitoredGeofenceStates.exited;
        mask = mask | Windows.Devices.Geolocation.Geofencing.MonitoredGeofenceStates.removed;

        var dwellTimeSpan = new Number(parseTimeSpan(dwellTimeField, defaultDwellTimeSeconds));
        var durationTimeSpan = null;
        if (durationField.value.length) {
            durationTimeSpan = new Number(parseTimeSpan(durationField, 0));
        } else {
            durationTimeSpan = new Number(0); // duration required if start time is set
        }
        var startDateTime = null;
        if (startTimeField.value.length) {
            startDateTime = new Date(startTimeField.value);
        } else {
            startDateTime = new Date(); // default is 1/1/1601
        }

        geofence = new Windows.Devices.Geolocation.Geofencing.Geofence(fenceKey, geocircle,
            mask, singleUse, dwellTimeSpan, startDateTime, durationTimeSpan);
    } catch (ex) {
        WinJS.log && WinJS.log(ex.toString(), "sample", "error");
    }

    return geofence;
}
```

What you now do with the Geofence object depends on whether you want foreground or background monitoring. The best way to think about this is that *background* monitoring should be your default choice. The reasons for this are twofold. First, background monitoring also works when the app is running, thereby allowing you to have just one piece of code to handle monitoring events. Second, if you set up event handlers for the running app alongside a background task, both will pick up geofencing events but the order isn't guaranteed. Thus, you'd choose foreground monitoring only for specific scenarios that would not need background monitoring at all.

Either way, the first step is to add your Geofence object to the GeofenceMonitor (in Windows.Devices.Geolocation.Geofencing), specifically through its geofences vector. To end monitoring of that geofence, just remove that instance from the vector. Be mindful that the GeofenceMonitor is a system object that you access directly (not using new). So, if you have a Geofence in the variable *geofence*, you start monitoring like so:

```
var monitor = Windows.Devices.Geolocation.Geofencing.GeofenceMonitor.current;
monitor.geofences.push(geofence);
```

This is demonstrated again in scenario 4 of the sample (see js/scenario4.js), but there's a bunch more code going around to manage the UI, so what I'm showing above is the simplest form. Note also that scenario 5 of the sample, which sets up the background task, requires you to first create a geofence in scenario 4. What's different between the two scenarios is how we pick up geofencing events.

For background monitoring, you don't assign any explicit event handlers. Instead, the running app must create and register a background task with the location trigger, the general pattern for which we'll be talking about in Chapter 16, and you can refer to scenario 5 in the sample again for details. Background monitoring requires *Location* background task declaration in the manifest, and setting up the task means that the app has to be run at least once. Registration of the background task on first run will prompt the user for consent, because background tasks like this affect battery life.

Here's the short version of the code that registers the task (js/scenario5.js):

```
var sampleBackgroundTaskName = "SampleGeofencingBackgroundTask";
var sampleBackgroundTaskEntryPoint = "js\\geofencebackgroundtask.js";

var builder = new Windows.ApplicationModel.Background.BackgroundTaskBuilder();
builder.name = sampleBackgroundTaskName;
builder.taskEntryPoint = sampleBackgroundTaskEntryPoint;
builder.setTrigger(new Windows.ApplicationModel.Background.LocationTrigger(
    Windows.ApplicationModel.Background.LocationTriggerType.geofence));
```

Once registered, the trigger essentially acts like the event and the background task as a whole is the handler: when the trigger occurs, the JavaScript code file that you assign to the task gets executed (as a web worker). For details on how to simulate the trigger for testing, see Testing and debugging your geofencing apps on the Windows App Builder's blog. The process involves using the Visual Studio simulator, in which you can set the position returned by the Geolocator. The trick is that you need to locally deploy and run the app once to add it to the lock screen, because doing so isn't supported through the simulator.

In that task (refer to js/geofencebackgroundtask.js) you then obtain the GeofenceMonitor object (as shown earlier) and call its readReports method. This returns a vector view of GeofenceStateChangeReport objects, which can be empty if nothing has changed since the last call. Each report contains a geofence property (the Geofence that changed), the geoposition of that geofence, and the newState of that geofence. There is also a removalReason property if the state change is because monitoring ended.

What you're most interested in is the value of newState, which is a value from the GeofenceState enumeration: none, entered, exited, and removed. You can then take the appropriate action. (The sample just takes this information and outputs it to the display; see the getGeofenceStateChanged-Reports function in js/geofencebackgroundtask.js. I hope your apps will do something much more interesting!) And if you want to communicate any information to the running app, remember that you can use local app data for this purpose.

If the app is running when all this happens, it will probably want to know that the background task completes its work and exits. To do this, you assign a handler for the background task's completed event after registering that task. In that handler you can check for whatever app data the background task saved and then take any further actions desired. For an additional example of this, refer to the Creating smarter apps with geofencing post also on the Windows App Builder blog.

If you really want to do only foreground monitoring, you can listen to the GeofenceMonitor.-ongeofencestatuschanged event:

```
monitor.addEventListener("geofencestatechanged", onGeofenceStateChanged);
```

In your handler, the eventArgs.target will contain the GeofenceMonitor object, on which you can call readReports and process as needed. The sample does it this way in scenario 4 (js/scenario4.js):

```
function onGeofenceStateChanged(args) {
    args.target.readReports().forEach(processReport);
}
```

Additional information on this can be found on Guidelines for geofencing apps, but be aware that no JavaScript-specific version of this topic exists at present.

Sensors

As I wrote in this chapter's introduction, I like to think of sensors as another form of input. It makes a lot of sense because every device that is now wholly integrated into our computer systems—such that we take them for granted—was at one point a kind of human-interface peripheral. In time, I suspect that many of the sensors that are new to us today will be standard equipment just about everywhere.

Sensors, again, are a way of understanding the relationship of a device to the physical world around it, and this constitutes input because you, as a human being, can affect that relationship primarily by moving the device around in physical space or otherwise changing its environment. Sensors can also be used as direct input to cause motion on the screen rather than relying on some form of abstract input like the keyboard or mouse. For example, instead of using keystrokes to abstractly tilt a game board, you can, with sensors, just tilt the device. Shaking, in fact, is becoming a well-known physical gesture that can be wired to a command of some kind like *Retry Now, darn you! Why aren't you doing what I want?* Haven't we for years been shaking or smacking our computers when they aren't behaving properly? Well, with sensors the computer can now actually respond!

Here, then, is what the various sensors tell us:

- **Location** The device's position on the earth (as we covered earlier in "Geolocation").
- **Compass and orientation** The direction the device is pointing, relative to the earth's magnetic poles or relative to the device's inherent sense of position (both simple and complex orientation).
- **Inclinometer** The static pitch, roll, and yaw of the device in 3D space.
- **Gyrometer** The angular velocity/rotational motion of the device in 3D space.
- **Accelerometer** The linear G-force acceleration of the device within 3D space (x, y, z).
- **Ambient light** The amount of light shining on the surface of the device with the sensor.

These are the sensors that are represented in the WinRT API,[93] some of which are created in software through *sensor fusion*. This means taking raw data from one or more hardware sensors and combining, interpreting, and presenting it all in a form that's more directly useful to apps. Just as with pointers, you can still get to raw data if you want it, but oftentimes it's unnecessary. For example, the Simple Orientation sensor provides a simple interpretation of how the device is oriented in relation to its default position, rounding everything off, as it were, to the nearest 90-degree quadrant. The full Orientation sensor combines gyrometer, accelerometer, and compass data to provide an exact 3D orientation matrix that is much more precise but much more oriented (if I might make the pun!) to advanced scenarios than simply needing to know whether the device is upside down or right-side up.

Because all of these sensors are very similar in how they work (which is intentional, with the exception of the Simple Orientation sensor, which is intentionally dissimilar!), I want to show the general pattern of the sensor APIs rather than explicit examples for each. Such demonstrations are readily available in these SDK samples: Accelerometer, Compass, Gyrometer, Inclinometer, Light Sensor, and OrientationSensor. A device like the Microsoft Surface Pro is a good one for all of these, because it is fully equipped and capable of running Visual Studio directly.

The usage pattern is as follows, with the particulars summarized in the table that follows:

- Obtain a sensor object via `Windows.Devices.Sensors.<sensor>.getDefault()`.
- Call that object's `getCurrentReading` to obtain a one-time reading.
- For ongoing readings, configure the object's `minimumReportInterval` and `reportInterval` properties (both in milliseconds) and listen to the object's `readingchanged` event. Your handler will receive a reading object of an appropriate type in response. As with geolocation, setting these values wisely will help optimize battery life by avoiding excess electrons flying through the sensors!

[93] There is also the proximity sensor for near-field communications (NFC) that tells us when devices are near one another or make contact, but this is more a networking handshake than a sensor like the others. We'll see this in Chapter 17.

Sensor Name (Windows.Devices.Sensors.)	**Added Members**	**Reading Type (Windows.Devices.Sensors)**	**Reading Properties (timestamp is a Date; all others are Numbers unless noted)**
Accelerometer	Event: shaken (event args contains only a timestamp property)	AccelerometerReading	accelerationX (G's), accelerationY, accelerationZ, timestamp
Compass	n/a	CompassReading	headingMagneticNorth (degrees), headingTrueNorth, headingAccuracy, timestamp
Gyrometer	n/a	GyrometerReading	angularVelocityX (degrees/sec), angularVelocityY, angularVelocityZ, timestamp
Inclinometer	n/a	InclinometerReading	pitchDegrees (degrees), rollDegrees (degrees), yawDegrees (degrees), yawAccuracy, timestamp
LightSensor	n/a	LightSensorReading	illuminenceInLux (lux), timestamp
OrientationSensor	n/a	OrientationSensorReading	quaternion, (SensorQuaternion containing w, x, y, and z properties) rotationMatrix (Sensor-RotationMatrix containing m11, m12, m13, m21, m22, m23, m31, m32, m33 properties), yawAccuracy, timestamp

Here's an example of such code from the Gyrometer sample (js/scenario1.js):

```
gyrometer = Windows.Devices.Sensors.Gyrometer.getDefault();

var minimumReportInterval = gyrometer.minimumReportInterval;
var reportInterval = minimumReportInterval > 16 ? minimumReportInterval : 16;
gyrometer.reportInterval = reportInterval;

gyrometer.addEventListener("readingchanged", onDataChanged);    // Remember to remove as needed

function onDataChanged(e) {
    var reading = e.reading;

    document.getElementById("eventOutputX").innerHTML = reading.angularVelocityX.toFixed(2);
    document.getElementById("eventOutputY").innerHTML = reading.angularVelocityY.toFixed(2);
    document.getElementById("eventOutputZ").innerHTML = reading.angularVelocityZ.toFixed(2);
}
```

The samples for the compass, inclinometer, and orientation sensor also have scenarios for calibration because each one has a limited degree of accuracy.

Relative axes With all the directional sensors, the values they report through their readings are relative to the device orientation (portrait or landscape), rather than being absolute. If you allow different rotations, you'll need to take these into account, which is especially important for *portrait-first* devices like smaller 7" or 8" tablets. The specific mathematics for these cases is beyond the scope of this book, however, so refer to Aligning sensors with your app's orientation on the Windows App Builder blog for more details.

With the Orientation Sensor, a *quaternion* can be most easily understood as a rotation of a point [x,y,z] about a single arbitrary axis. This is different from a rotation matrix, which represents rotations around three axes. The mathematics behind quaternions is fairly exotic because it involves the geometric properties of complex numbers and mathematical properties of imaginary numbers, but working with them is simple and frameworks like DirectX support them. See the OrientationSensor sample for more.

Speaking of orientation, I mentioned that the `SimpleOrientationSensor` works a little differently. Its purpose is to supply *quadrant* orientation rather than exact orientation, which is perhaps all you need. For example, a star chart app would need to know if a slate device is upside down so that it can adjust its display (along with a compass reading) to match the sky itself.

To summarize this sensor's usage:

- Call `Windows.Devices.Sensors.SimpleOrientation.getDefault` to obtain the object.
- Call the `getCurrentOrientation` to obtain a reading.
- The `orientationChanged` event provides for ongoing readings, where `eventArgs` contains `orientation` (a reading) and `timestamp` properties.
- The reading is a `SimpleOrientation` value whose meaning is relative to the native device orientation:
  - `notRotated`, `rotated90DegreesCounterclockwise`, `rotated90DegreesCounterclockwise`, `rotated270DegreesCounterclockwise`. Note that these are entirely different from view orientations like landscape and portrait that you'd pick up in media queries and so forth. A portrait-first device in its native state, for example, will report a portrait view orientation but `notRotated` as its `SimpleOrientation`.
  - `faceup`, `facedown` (tablet devices only).

For a demonstration, see the SimpleOrientationSensor sample.

What We've Just Learned

- "Design for touch, get mouse and stylus for free" is a message that holds true, because working with pointer and gesture input from a variety of input devices doesn't require you to differentiate between the forms of input.
- Using built-in controls is the easiest way to handle input, but you can also handle `pointer*` events and `MSGesture*` events directly, when needed. You can also feed `pointer*` events into a custom gesture recognizer that then issues its own events.
- The Windows touch language includes tap, press and hold, slide/pan, cross-slide (to select), pinch-stretch, rotate, and edge gestures (from top/bottom and from the sides). A tap is typically handled with a `click` event, whereas the others require the creation of an `MSGesture` object, association of that object with a pointer, and handling of `MSGesture*` event sequences, which provide for manipulations and inertial motions together.
- The touch language also has mouse, stylus, and keyboard equivalents. For mouse and stylus, there is very little work an app needs to do (such as sending mouse `wheel` events to the gesture object). Keyboard support must be implemented separately but simply uses the standard HTML/JavaScript events.
- Keyboard support also includes accommodating the soft (on-screen) keyboard, which appears automatically for text input fields and other content-editable elements. It automatically adjusts its appearance according to input type, and it will slide the app contents up if necessary to avoid having the keyboard overlap the input control. An app can also handle visibility events directly to provide a better experience than the default.
- The Inking API provides apps with the means to record, save, and render an entire series of pointer activities, where the strokes can also be fed into a handwriting recognizer.
- The Geolocation API in WinRT provides apps with access to GPS data as well as events when the device has moved past a specified threshold, including support for geofencing.
- The WinRT API represents a number of sensors that can also be used as input to an app. In addition to geolocation, the sensors are compass, orientation, simple orientation (quadrant-based), inclinometer, gyrometer, accelerometer, and ambient light.
- Most sensors follow the same usage pattern: acquire the sensor object, get a current reading, and possibly listen to the `readingchanged` event. They are very easy to work with, leaving much of your energy to apply them creatively!

Chapter 13

Media

To say that media is important to apps—and to culture in general—is a terrible understatement. Ever since the likes of Edison made it possible to record a performance for later enjoyment, and the likes of Marconi made it possible to widely broadcast and distribute such performances, humanity's worldwide appetite for media—graphics, audio, and video—has probably outpaced the appetite for automobiles, electricity, and even junk food. In the early days of the Internet, graphics and images easily accounted for the bulk of network traffic. Today, streaming video even from a single source like Netflix holds top honors for pushing the capabilities of our broadband infrastructure! (It certainly holds true in my own household with my young son's love of Tintin, Bob the Builder, Looney Tunes, and other such shows.)

Incorporating some form of media is likely a central concern for most Windows Store apps. Simple ones, even, probably use at least a few graphics to brand the app and present an attractive UI, as we've already seen on a number of occasions. Many others, especially games, will certainly use graphics, video, and audio together. In the context of this book, all of this means using the `img`, `svg` (Scalable Vector Graphics), `canvas`, `audio`, and `video` elements of HTML5.

Of course, working with media goes well beyond just presentation because apps might also provide any of the following capabilities:

- Organize and edit media files, including those in the pictures, music, and videos media libraries.
- Playback of custom audio and video formats.
- Transcode (convert) media files, possibly applying various filters.
- Organize and edit playlists.
- Capture audio and video from available devices.
- Edit or modify media directly in the rendering pipeline through media stream sources.
- Stream media from a server to a device, or from a device to a Play To target, perhaps also applying digital rights management (DRM).

These capabilities, for which many WinRT APIs exist, along with the media elements of HTML5 and their particular capabilities within the Windows environment, will be our focus for this chapter.

Note As is relevant to this chapter, a complete list of audio and video formats that are natively supported for Windows Store apps can be found on Supported audio and video formats.

The Media Hub sample In the Windows SDK you'll find the Media Hub sample, which provides an rich, end-to-end sample for many of the individual features that we'll talk about in this chapter, including media playback, media capture, effects, system media transport controls, background audio, 3D video, and Play To. I won't be drawing from this sample here, however, as it has its own documentation on the MediaHub sample app page.

Sidebar: Performance Tricks for Faster Apps

Various recommendations in this chapter come from two great //build talks: 50 Performance Tricks to Make Your HTML5 Apps and Sites Faster and Fast Apps and Sites with JavaScript. While some tricks are specifically for web applications running in a browser, many of them are wholly applicable to Windows Store apps written in JavaScript because they run on top of the same infrastructure as Internet Explorer.

Creating Media Elements

Certainly the easiest means to incorporate media into an app is what we've already been doing for years: simply use the appropriate HTML element in your layout and *voila!* there you have it. With `img`, `audio`, and `video` elements, in fact, you're completely free to use content from just about any location. That is, the `src` attributes of these elements can be assigned http:// or https:// URIs for remote content, ms-appx:/// and ms-appdata:/// URIs for local content, or URIs from `URL.createObjectURL` for any content represented by a `StorageFile` object. Remember with bitmap images that it's more memory efficient to use the `StorageFile` thumbnail APIs and pass the thumbnail to `URL.createObjectURL` instead of opening the whole image file. The `img` element can also use an SVG file as a source.

There are three ways to create a media element in a page or page control.

First is to include the element directly in declarative HTML. Here it's often useful to use the `preload="auto"` attribute for remote audio and video to increase the responsiveness of controls and other UI that depend on those elements. (Doing so isn't really important for local media files since they are, well, already local!) Oftentimes, media elements are placed near the top of the HTML file, in order of priority, so that downloading can begin while the rest of the document is being parsed.

On the flip side, if the user can wait a short time to start a video, use a preview image in place of the video and don't start the download until it's actually necessary. Code for this is shown later in this chapter in the "Video Playback and Deferred Loading" section. You can also consider using the background transfer APIs, as we discussed in Chapter 4, "Web Content and Services," to save media files locally for later playback.

Playback for a declarative element can be automatically started with the `autoplay` attribute, through the built-in UI if the element has the `controls` attribute, or by calling `<element>.play()` from JavaScript.

The second method is to create an HTML element in JavaScript via `document.createElement` and add it to the DOM with `<parent>.appendChild` and similar methods. Here's an example using media files in this chapter's companion content, though you'll need to drop the code into a new project of your own in a media folder:

```
//Create elements and add to DOM, which will trigger layout
var picture = document.createElement("img");
picture.src = "/media/wildflowers.jpg";
picture.width = 300;
picture.height = 450;
document.getElementById("divShow").appendChild(picture);

var movie = document.createElement("video");
movie.src = "/media/ModelRocket1.mp4";
movie.autoplay = false;
movie.controls = true;
document.getElementById("divShow").appendChild(movie);

var sound = document.createElement("audio");
sound.src = "/media/SpringyBoing.mp3";
sound.autoplay = true;  //Play as soon as element is added to DOM
sound.controls = true;  //If false, audio plays but does not affect layout

document.getElementById("divShow").appendChild(sound);
```

Unless otherwise hidden by styles, adding image and video elements to the DOM, plus audio elements with the `controls` attribute, will trigger re-rendering of the document layout. An audio element *without* that attribute will not cause re-rendering. As with declarative HTML, setting `autoplay` to `true` will cause video and audio to start playing as soon as the element is added to the DOM.

Finally, for audio, apps can create an `Audio` object in JavaScript to play sounds or music without any effect on UI. More on this later. JavaScript also has the `Image` class, and the `Audio` class can be used to load video:

```
//Create objects (preloading), then set other DOM object sources accordingly
var picture = new Image(300, 450);
picture.src = "http://www.kraigbrockschmidt.com/downloads/media/wildflowers.jpg";
document.getElementById("image1").src = picture.src;

//Audio object can be used to preload (but not render) video
var movie = new Audio("http://www.kraigbrockschmidt.com/downloads/media/ModelRocket1.mp4");
document.getElementById("video1").src = movie.src;

var sound = new Audio("http://www.kraigbrockschmidt.com/downloads/media/SpringyBoing.mp3");
document.getElementById("audio1").src = sound.src;
```

Creating an `Image` or `Audio` object from code does not create elements in the DOM, which can be a useful trait. The `Image` object, for instance, has been used for years to preload an array of image sources for use with things like image rotators and popup menus, and you can use the same trick for preloading image thumbnails. For remote sources, preloading means that the images have been downloaded and cached. This way, assigning the same URI to the `src` attribute of an element that *is* in the DOM, as shown above, will make that image appear immediately. The same is true for preloading video and audio, but again, this is primarily helpful with remote media because files on the local file system will load relatively quickly as is. Still, if you have large local images and want them to appear quickly when needed, preloading their thumbnails is a useful strategy.

Of course, you might want to load media only when it's needed, in which case the same type of code can be used with existing elements, or you can just create an element and add it to the DOM as shown earlier.

Graphics Elements: Img, Svg, and Canvas (and a Little CSS)

I know you're probably excited to get to sections of this chapter on video and audio, but we cannot forget that images have been the backbone of web applications since the beginning and remain a huge part of any app's user experience. Indeed, it's helpful to remember that video itself is conceptually just a series of static images sequenced over time! Fortunately, HTML5 has greatly expanded an app's ability to incorporate image data by adding SVG support and the `canvas` element to the tried-and-true `img` element. Furthermore, applying CSS animations and transitions (covered in detail in Chapter 14, "Purposeful Animations") to otherwise static image elements can make them appear very dynamic.

Speaking of CSS, it's worth noting that many graphical effects that once required the use of static images can be achieved with *just* CSS, especially CSS3:

- Borders, background colors, and background images
- Folder tabs, menus, and toolbars
- Rounded border corners, multiple backgrounds/borders, and image borders
- Transparency
- Embeddable fonts
- Box shadows
- Text shadows
- Gradients

In short, if you've ever used `img` elements to create small visual effects, create gradient backgrounds, use a nonstandard font, or provide some kind of graphical navigation structure, there's probably a way to do it in pure CSS. For details, see the great overview of CSS3 by Smashing Magazine as well as the CSS specs at http://www.w3.org/. CSS also provides the ability to declaratively handle some events and visual states using pseudo-selectors of `hover`, `visited`, `active`, `focus`, `target`, `enabled`, `disabled`, and `checked`. For more, see http://css-tricks.com/ as well as another Smashing Magazine tutorial on pseudo-classes.

That said, let's review the three primary HTML5 elements for graphics:

- `img` is used for raster data. The PNG format is generally preferred over other formats, especially for text and line art, though JPEG makes smaller files for photographs. GIF is generally considered outdated, as the primary scenarios where GIF produced a smaller file size can probably be achieved with CSS directly. Where scaling is concerned, Windows Store apps need to consider pixel density, as we saw in Chapter 8, "Layout and Views," and provide separate image files for each scale the app might encounter. This is where the smaller size of JPEGs can reduce the overall size of your app package in the Windows Store.

- SVGs are best used for smooth scaling across display sizes and pixel densities. SVGs can be declared inline, created dynamically in the DOM, or maintained as separate files and used as a source for an `img` element (in which case all the scaling characteristics are maintained). As we saw in Chapter 8, preserving the aspect ratio of an SVG is often important, for which you employ the `viewBox` and `preserveAspectRatio` attributes of the `svg` tag.

- The `canvas` element provides a drawing surface and API for creating graphics with lines, rectangles, arcs, text, and so forth, including 3D graphics via WebGL (starting in Windows 8.1). The `canvas` ultimately generates raster data, which means that once created, a `canvas` scales like a bitmap. (An app, of course, will typically redraw a `canvas` with scaled coordinates when necessary to avoid pixelation.) The `canvas` is also very useful for performing pixel manipulation, even on individual frames of a video while it's playing.

Apps often use all three of these elements, drawing on their various strengths. I say this because when `canvas` first became available, developers seemed so enamored with it that they seemed to forget how to use `img` elements and they ignored the fact that SVGs are often a better choice altogether! (And did I already say that CSS can accomplish a great deal by itself as well?)

In the end, it's helpful to think of all the HTML5 graphics elements as ultimately producing a bitmap that the app host simply renders to the display. You can, of course, programmatically animate the internal contents of these elements in JavaScript, as we'll see in Chapter 14, but for our purposes here it's helpful to think of these as essentially static.

What differs between the elements is how image data gets into the element to begin with. `Img` elements are loaded from a source file, `svg`s are defined in markup, and `canvas` elements are filled through procedural code. But in the end, as demonstrated in scenario 1 in the HTML Graphics example for this chapter and shown in Figure 13-1, each can produce identical results.

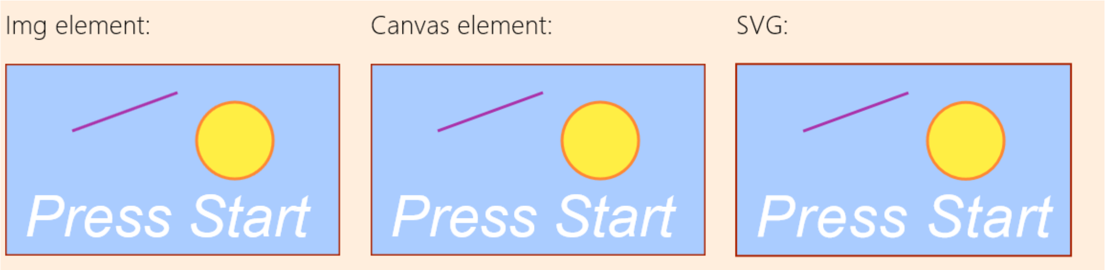


FIGURE 13-1 Image, canvas, and svg elements showing identical results.

In short, there are no fundamental differences as to what *can* be rendered through each type of element (though WebGL in a `canvas` has much richer 3D capabilities). However, they do have differences that become apparent when we begin to manipulate those elements as with CSS. Because each element is just a node in the DOM, plain and simple, they are treated like all other nongraphic elements: CSS doesn't affect the internals of the element, just how it ultimately appears on the page. Individual parts of SVGs declared in markup can, in fact, be separately styled so long as they can be identified with a CSS selector. In any case, such styling affects only presentation, so if new styles are applied, they are applied to the original contents of the element.

What's also true is that graphics elements can overlap with each other and with nongraphic elements (as well as video), and the rendering engine automatically manages transparency according to the `z-index` of those elements. Each graphic element can have clear or transparent areas, as is built into image formats like PNG. In a `canvas`, any areas cleared with the `clearRect` method that aren't otherwise affected by other API calls will be transparent. Similarly, any area in an SVG's rectangle that's not affected by its individual parts will be transparent.

Scenario 2 in the HTML Graphics example allows you to toggle a few styles (with a check box) on the same elements shown earlier. In this case, I've left the background of the canvas element transparent so that we can see areas that show through. When the styles are applied, the `img` element is rotated and transformed, the `canvas` gets scaled, and individual parts of the `svg` are styled with new colors, as shown in Figure 13-2.

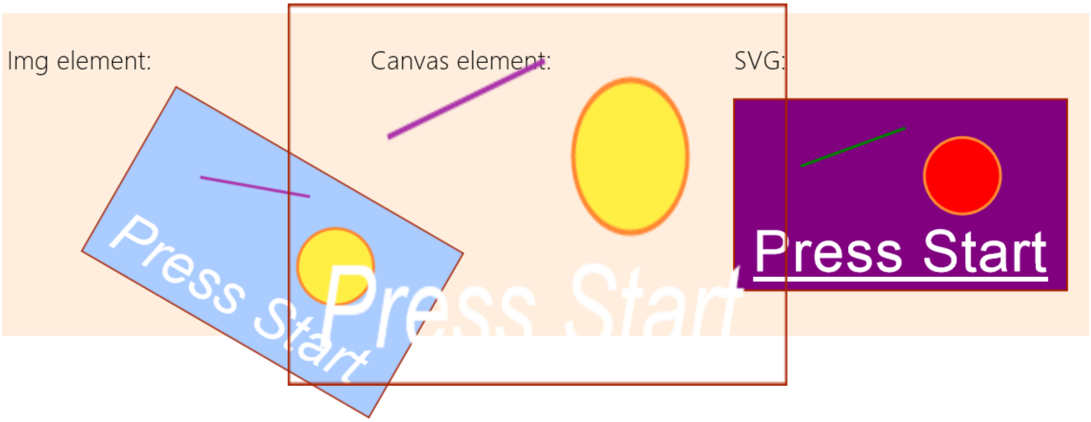


FIGURE 13-2 Styles applied to graphic elements; individual parts of the SVG can be styled if they are accessible through the DOM.

The styles in css/scenario2.css are simple:

```
.transformImage {
    transform: rotate(30deg) translateX(120px);
}

.scaleCanvas {
    transform: scale(1.5, 2);
}
```

as is the code in js/scenario2.js that applies them:

```
function toggleStyles() {
    var applyStyles = document.getElementById("check1").checked;

    document.getElementById("image1").className = applyStyles ? "transformImage" : "";
    document.getElementById("canvas1").className = applyStyles ? "scaleCanvas" : "";

    document.getElementById("r").style.fill = applyStyles ? "purple" : "";
    document.getElementById("l").style.stroke = applyStyles ? "green" : "";
    document.getElementById("c").style.fill = applyStyles ? "red" : "";
    document.getElementById("t").style.fontStyle = applyStyles ? "normal" : "";
    document.getElementById("t").style.textDecoration = applyStyles ? "underline" : "";
}
```

The other thing you might have noticed when the styles are applied is that the scaled-up canvas looks rasterized, like a bitmap would typically be. This is expected behavior, as shown in the following table of scaling characteristics. These are demonstrated in scenarios 3 and 4 of the HTML Graphics example.

Element	Scaling	Handling layout changes for best appearance
`img`	rasterized	Change `src` attribute for different scales (or just use an SVG file as a source).
`canvas`	rasterized	Redraw canvas using scaled dimensions; this is often best done by calling `<context>.scale` (2D) or `<context>.uniformMatrix3fv` according to the needed display size instead of changing the coordinates used in the code.
`svg`	smooth	Not needed. Use `viewBox` and `preseveAspectRatio` for proportional scaling.

Additional Characteristics of Graphics Elements

There are a few additional characteristics to be aware of with graphics elements. First, different kinds of operations will trigger a re-rendering of the element in the document. Second is the mode of operation of each element. Third are the relative strengths of each element. These are summarized in the following table:

Element	Trigger for re-rendering	Mode	Strengths
`img`	Change `src` attribute Change of styling via JavaScript	Pixel	Fast to render and transform Great for static elements and static/repeating backgrounds Sprite animation by changing `src` attribute
`canvas`	Calls to context API Change of styling via JavaScript Note: re-rendering happens only when code returns control to the host and unblocks the UI thread; there are no visible changes while the code is manipulating the canvas.	Immediate: API calls are rendered to pixels and forgotten.	Fine-grained dynamic content Fast to render after being drawn Pixel-level manipulation Excellent for fine-grained dynamic/interactive content with frequent computation
`svg`	Change to element structure Change of styling via JavaScript	Retained: all shapes exist as DOM elements (unless used as `img` src).	Smooth scaling Fine-grained control over individual (retained) elements Shape-level manipulation Excellent for interactive graphics, detailed and scalable styling, and dynamic per-shape attributes

Sidebar: Using Media Queries to Show and Hide SVG Elements

Because SVGs generate elements in the DOM, those elements can be individually styled. You can use this fact with media queries to hide different parts of the SVG depending on its size. To do this, add different classes to those SVG elements. Then, in CSS, add or remove the `display: none` style for those classes within media queries like `@media (min-width:300px) and (max-width:499px)`. You may need to account for the size of the SVG relative to the app window, but it means that you can effectively remove detail from an SVG rather than allowing those parts to be rendered with too few pixels to be useful.

In the end, HTML5 includes all three of these elements because all three are really needed. All of them benefit from full hardware acceleration, just as they do in Internet Explorer, since apps written in HTML and JavaScript run on the same rendering engine as the browser.

The best practice in app design is to explore the appropriate use of each type of elements. Each element can have transparent areas, so you can easily achieve some very fun effects. For example, if you have data that maps video timings to caption or other text, you can use an interval handler (with the interval set to the necessary granularity like a half-second) to take the video's `currentTime` property, retrieve the appropriate text for that segment, and render the text to an otherwise transparent canvas that sits on top of the video. Titles and credits can be done in a similar manner, eliminating the need to re-encode the video.

Some Tips and Tricks

Working with the HTML graphics elements is generally straightforward, but knowing some details can help when working with them inside a Windows Store app.

> **General tip** To protect any content of an app view from screen capture, obtain the `ApplicationView` object from `Windows.UI.ViewManagement.ApplicationView.getForCurrentView()` and set its `isScreenCaptureEnabled` property to `false`. This is demonstrated in the Disable screen capture sample in the Windows SDK. You would do this, for example, when rendering content obtained from a rights-protected source.

Img Elements

- When possible, avoid loading an entire image file by using the `StorageFile` thumbnail APIs, `getThumbnailAsync` and `getScaledImageAsThumbnailAsync`, as described in Chapter 11, "The Story of State, Part 2." You can pass a thumbnail to `URL.createObjectURL` as you would a `StorageFile`. Of course, if you're using remote resources directly with `http[s]://` URIs, you won't be able to intercept the rendering to do this.

- Use the `title` attribute of `img` for tooltips, not the `alt` attribute. You can also use a `WinJS.UI.Tooltip` control, as described in Chapter 5, "Controls and Control Styling."

- To create an image from an in-memory stream, see `MSApp.createBlobFromRandomAccessStream` (introduced in Chapter 10, "The Story of State, Part 1"), the result of which can be then given to `URL.createObjectURL` to create an appropriate URI for a `src` attribute. We'll encounter this elsewhere in this chapter, and we'll need it when working with the Share contract in Chapter 15, "Contracts." The same technique also works for audio and video streams, including those partially downloaded from the web.

- When loading images from `http://` or other remote sources, you run the risk of having the element show a red X placeholder image. To prevent this, catch the `img.onerror` event and supply your own placeholder:

```
var myImage = document.getElementById('image');
myImage.onerror = function () { onImageError(this);}

function onImageError(source) {
    source.src = “placeholder.png”;
    source.onerror = "";
}
```

- Supported image formats for the `img` element are listed at the bottom of the img element documentation. Note that as of Windows 8.1, the `img` element supports the Direct Draw Surface (DDS) file format for in-package content. DDS files are commonly used for game assets and benefit from full hardware acceleration and very short image decoding time. A demonstration of using these can be found in the Block compressed images sample.
- Want to do optical character recognition? Check out the Bing OCR control available from http://www.bing.com/dev/en-us/ocr, which is free to use for up to 5,000 transactions per month.

Svg Elements

- `<script>` tags are not supported within `<svg>`.
- If you have an SVG file in your package (or appdata), you can load it into an `img` element by pointing at the file with the `src` attribute, but this doesn't let you traverse the SVG in the DOM. What you can do instead is load the SVG file by using the simple `XMLHttpRequest` method or the `WinJS.xhr` wrapper (see Appendix C, "Additional Networking Topics"), and then insert the marking directly into the DOM as a child of some other element. This lets you traverse the SVG's content and style it with CSS without having to place the SVG directly in your HTML files. Scenario 2 of the HTML Graphics example in the companion content shows this (js/scenario2.js):

```
WinJS.xhr({ url: "/html/graphic.svg", responseType: "text" }).done(function (request) {
    //setInnerHTMLUnsafe is OK because we know the content is coming from our package.
    WinJS.Utilities.setInnerHTMLUnsafe(document.getElementById("svgPlaceholder"),
        request.response);
});
```

- PNGs and JPEGs generally perform better than SVGs, so if you don't technically need an SVG or have a high-performance scenario, consider using scaled raster graphics. Or you can dynamically create a scaled static image from an SVG so as to use the image for faster rendering later:

```
<!-- in HTML-->
<img id="svg" src="somesvg.svg" style="display: none;" />
<canvas id="canvas" style="display: none;" />

// in JavaScript
var c = document.getElementById("canvas").getContext("2d");
c.drawImage(document.getElementById("svg"),0,0);
var imageURLToUse = document.getElementById("canvas").toDataURL();
```

- Two helpful SVG references (JavaScript examples): http://www.carto.net/papers/svg/samples/ and http://srufaculty.sru.edu/david.dailey/svg/.
- A number of tools are available to create SVGs: see 4 useful commercial SVG tools and 5 useful open source SVG tools (both on the IDR solutions blog).

Canvas Elements

As you probably know, and as demonstrated in the HTML Graphics example, you obtain a 2D context for a canvas with code like this:

```
var c = document.getElementById("canvas").getContent("2d");
```

To obtain a 3D WebGL context (as can be done starting with Windows 8.1), the argument to `getContext` must be *experimental-webgl*:

```
var c = document.getElementById("canvas").getContent("experimental-webgl");
```

From that point you can use the supported WebGL APIs as documented in WebGL APIs for Internet Explorer. In this book I won't go into any of the details about the API itself, as it quickly gets complicated. Besides, there are plenty of tutorials on the web.

WebGL aside, here are other tips and tricks for the `canvas` (note that all the methods named here are found on the context object):

- Remember that a `canvas` element needs specific `width` and `height` *attributes* (in JavaScript, `canvas.width` and `canvas.height`), not styles. It does not accept px, em, %, or other units.
- Despite its name, the `closePath` method is *not* a direct complement to `beginPath`. `beginPath` is used to start a new path that can be stroked, clearing any previous path. `closePath`, on the other hand, simply connects the two endpoints of the current path, as if you did a `lineTo` between those points. It does *not* clear the path or start a new one. This seems to confuse programmers quite often, which is why you sometimes see a circle drawn with a line to the center!
- A call to `stroke` is necessary to render a path; until that time, think of paths as a pencil sketch of something that's not been inked in. Note also that stroking implies a call to `beginPath`.
- When animating on a canvas, doing `clearRect` on the entire canvas and redrawing every frame is generally easier to work with than clearing many small areas and redrawing individual parts of the canvas. The app host eventually has to render the entire canvas in its entirety with every frame anyway to manage transparency, so trying to optimize performance by clearing small rectangles isn't an effective strategy except when you're doing only a small number of API calls for each frame.
- Rendering canvas API calls is accomplished by converting them to the equivalent DirectX calls in the GPU. This draws shapes with automatic antialiasing. As a result, drawing a shape like a 2D

circle in a color and drawing the same circle with the background color does *not* erase every pixel. To effectively erase a shape, use `clearRect` on an area that's slightly larger than the shape itself. This is one reason why clearing the entire canvas and redrawing every frame often ends up being easier.

- To set a background image in a canvas (so that you don't have to draw each time), you can use the `canvas.style.backgroundImage` property with an appropriate URI to the image.
- Use the `msToBlob` method on a `canvas` object to obtain a `blob` for the canvas contents.
- When using `drawImage`, you may need to wait for the source image to load using code such as

```
var img = new Image();
img.onload = function () { myContext.drawImage(myImg, 0, 0); }
myImg.src = "myImageFile.png";
```

- The context's `msImageSmoothingEnabled` property (a Boolean) determines how images are resized on the `canvas` when rendered with `drawImage` or pattern-filling through `fill`, `stroke`, or `fillText`. By default, smoothing is enabled (`true`), which uses a bilinear smoothing method. When this flag is false, a nearest-neighbor algorithm is used instead, which is appropriate for the retro-graphics look of 1980s video games.
- Although other graphics APIs see a circle as a special case of an ellipse (with x and y radii being the same), the canvas `arc` function works with circles only. Fortunately, a little use of scaling makes it easy to draw ellipses, as shown in the utility function below. Note that we use `save` and `restore` so that the `scale` call applies *only* to the `arc`; it does not affect the `stroke` that's used from `main`. This is important, because if the scaling factors are still in effect when you call `stroke`, the line width will vary instead of remaining constant.

```
function arcEllipse(ctx, x, y, radiusX, radiusY, startAngle, endAngle, anticlockwise) {
   //Use the smaller radius as the basis and stretch the other
   var radius = Math.min(radiusX, radiusY);
   var scaleX = radiusX / radius;
   var scaleY = radiusY / radius;

   ctx.save();
   ctx.scale(scaleX, scaleY);

   //Note that centerpoint must take the scale into account
   ctx.arc(x / scaleX, y / scaleY, radius, startAngle, endAngle, anticlockwise);
   ctx.restore();
}
```

- There's no rule that says you have to do everything on a single `canvas` element. It can be very effective to layer multiple elements directly on top of one another to optimize rendering of different parts of your display, especially where game animations are concerned. See to Optimize HTML5 canvas rendering with layering (IBM developerWorks).

- By copying pixel data from a video, it's possible with the canvas to dynamically manipulate a video (without affecting the source, of course). This is a useful technique, even if it's processor-intensive (which means it might not work well on low-power devices).

Here's an example of frame-by-frame video manipulation, the technique for which is nicely outlined in a Windows team blog post, Canvas Direct Pixel Manipulation.[94] In the VideoEdit example for this chapter, default.html contains a `video` and `canvas` element in its main body:

```
<video id="video1" src="Rocket01.mp4" muted style="display: none"></video>
<canvas id="canvas1" width="640" height="480"></canvas>
```

In code (js/default.js), we call `startVideo` from within the activated handler. This function starts the video and uses `requestAnimationFrame` to do the pixel manipulation for every video frame:

```
var video1, canvas1, ctx;
var colorOffset = { red: 0, green: 1, blue: 2, alpha: 3 };

function startVideo() {
    video1 = document.getElementById("video1");
    canvas1 = document.getElementById("canvas1");
    ctx = canvas1.getContent("2d");

    video1.play();
    requestAnimationFrame(renderVideo);
}

function renderVideo() {
    //Copy a frame from the video to the canvas
    ctx.drawImage(video1, 0, 0, canvas1.width, canvas1.height);

    //Retrieve that frame as pixel data
    var imgData = ctx.getImageData(0, 0, canvas1.width, canvas1.height);
    var pixels = imgData.data;

    //Loop through the pixels, manipulate as needed
    var r, g, b;

    for (var i = 0; i < pixels.length; i += 4) {
        r = pixels[i + colorOffset.red];
        g = pixels[i + colorOffset.green];
        b = pixels[i + colorOffset.blue];

        //This creates a negative image
        pixels[i + colorOffset.red] = 2–5 - r;
        pixels[i + colorOffset.green] = 2–5 - g;
        pixels[i + colorOffset.blue] = 2–5 - b;
    }

    //Copy the manipulated pixels to the canvas
    ctx.putImageData(imgData, 0, 0);
```

[94] See also http://beej.us/blog/2010/02/html5s-canvas-part-ii-pixel-manipulation/.

```
    //Request the next frame
    requestAnimationFrame(renderVideo);
}
```

Here the page contains a hidden video element (`style="display: none"`) that is told to start playing once the document is loaded (`video1.play()`). In a `requestAnimationFrame` loop, the current frame of the video is copied to the canvas (`drawImage`) and the pixels for the frame are copied (`getImageData`) into the `imgData` buffer. We then go through that buffer and negate the color values, thereby producing a photographically negative image (an alternate formula to change to grayscale is also shown in the code comments, omitted above). We then copy those pixels back to the canvas (`putImageData`) so that when we return, those negated pixels are rendered to the display.

Again, this is processor-intensive because it's not generally a GPU-accelerated process, and it might perform poorly on lower-power devices. (Be sure, however, to run a Release build outside the debugger when evaluating performance.) It's much better to write a video effect DLL where possible, as discussed in "Applying a Video Effect" later in this chapter. Nevertheless, it is a useful technique to know. What's really happening is that instead of drawing each frame with API calls, we're simply using the video as a data source. So we could, if we like, embellish the canvas in any other way we want before returning from the `renderVideo` function. An example of this that I enjoy is shown in Manipulating video using canvas on Mozilla's developer site, which dynamically makes green-screen background pixels transparent so that an `img` element placed underneath the video shows through as a background. The same could even be used to layer two videos so that a background video is used instead of a static image. Again, be mindful of performance on low-power devices; you might consider providing a setting through which the user can disable such extra effects.

Rendering PDFs

In addition to the usual image formats, you may need to load and display a PDF into an `img` element (or a `canvas` by using its `drawImage` function). Aside from third-party libraries, you can use the WinRT APIs in `Windows.Data.Pdf` for this purpose. Here you'll find a `PdfDocument` object that represents a document as a whole, along with a `PdfPage` object that represents a single page within a document.

Note Although WinRT offers a means to load and display PDFs, it does not have an API for generating PDFs. You'll still need third-party libraries for that.

There are two ways to load a PDF into a `PdfDocument`:

- Given a `StorageFile` object (from the local file system, the file picker, removable storage, etc.), call the static method `PdfDocument.loadFromFileAsync` (which has two variants, the second of which takes a password if that's necessary).
- Given some kind of random access stream object (from a partial HTTP request operation, for instance; refer to "Q&A on Files, Streams, Buffers, and Blobs" in Chapter 10), call the static method `PdfDocument.loadFromStreamAsync` (which again has a password variant).

In both cases the load* methods return a promise that's fulfilled with a PdfDocument instance. Here's an example from scenario 1 of the PDF viewer sample where the file in question (represented by pdfFileName) is located in the app package (js/scenario1.js):

```
var pdfLib = Windows.Data.Pdf;

Windows.ApplicationModel.Package.current.installedLocation.getFileAsync(pdfFileName)
  .then(function loadDocument(file) {
    return pdfLib.PdfDocument.loadFromFileAsync(file);
}).then(function setPDFDoc(doc) {
    renderPage(doc, pageIndex, renderOptions);
});
```

The file variable from the first promise is just a StorageFile, so you can substitute any other code that results in such an object before the call to loadFromFileAsync. The setPDFDoc completed handler, as it's named here, receives the PdfDocument, whose isPasswordProtected and pageCount properties provide you with some obvious information.

The next thing to do is then render one or more pages of that document, or portions of those pages. The API is specifically set up to render one page at a time, so if you want to provide a multipage view you'll need to render multiple pages and display them in side-by-side img elements (using a Repeater control, perhaps), display them in a ListView control, or render those pages into a large canvas. More on this in a bit.

To get a PdfPage object for any given page, call PdfDocument.getPage with the desired (zero-based index), as shown here from within the renderPage function of the sample (js/scenario1.js):

```
var pdfPage = pdfDocument.getPage(pageIndex);
```

At this point the page's properties will be populated. These include the following:

- index The zero-based position of the page in the document.
- preferredZoom The preferred magnification factor (a number) for the page.
- rotation A value from the PdfPageRotation enumeration, one of normal, rotate90, rotate180, and rotate270.
- dimensions A PdfPageDimensions object containing artBox, bleedBox, cropBox, mediaBox, and trimBox, each of which is a Windows.Foundation.Rect. All of these represent intentions of the PDF's author; for specific definitions, refer to the Abode PDF Reference.
- size A Windows.Foundation.Size object the page's basic width and height based on the dimensions.cropBox, dimensions.mediaBox, and rotation properties.

To render the page, call its renderToStreamAsync, which, as its name implies, requires a random access stream that receives the rendering. You can create an in-memory stream, a file-based stream, or perhaps a stream to some other data store entirely, again using the APIs discussed in Chapter 10, depending on where you want the rendering to end up. Generally speaking, if you want to render just a single page for display, create an in-memory stream like the sample (js/scenario1.js):

```
var pageRenderOutputStream = new Windows.Storage.Streams.InMemoryRandomAccessStream();
```

If, on the other hand, you want to render a whole document and don't want to goggle up so much memory that you kick out every other suspended app, you should definitely render each page into a temporary file instead. This is demonstrated in the other SDK sample for PDFs, the PDF showcase viewer sample, whose code contains a more sophisticated mechanism to build a data source for pages that are then displayed in a ListView. (This sample also has its own documentation on the PDF viewer end-to-end sample page.) Once it opens a `PdfDocument`, it iterates all the pages and calls the following `loadPage` method (which also allows for in-memory rendering; js/pdflibrary.js):

```
loadPage: function (pageIndex, pdfDocument, pdfPageRenderingOptions, inMemoryFlag, tempFolder) {
    var filePointer = null;

    var promise = null;
    if (inMemoryFlag) {
        promise = WinJS.Promise.wrap(new Windows.Storage.Streams.InMemoryRandomAccessStream());
    } else {
        // Creating file on disk to store the rendered image for a page on disk
        // This image will be stored in the temporary folder provided during VDS init
        var filename = this.randomFileName() + ".png";
        var file = null;
        promise = tempFolder.createFileAsync(filename,
            Windows.Storage.CreationCollisionOption.replaceExisting).then(function (filePtr) {
            filePointer = filePtr;
            return filePointer.openAsync(Windows.Storage.FileAccessMode.readWrite);
        }, function (error) {
            // Error while opening a file
            filePointer = null;
        }, function (error) {
            // Error while creating a file
        });
    }
    return promise.then(function (imageStream) {
        var pdfPage = pdfDocument.getPage(pageIndex);
        return pdfPage.renderToStreamAsync(imageStream, pdfPageRenderingOptions)
        .then(function () {
            return imageStream.flushAsync();
        })

    // ...
```

Either way, your stream object must get to `PdfPage.renderToStreamAsync`, which has two variants. One just takes a stream, and the other takes the stream plus a `PdfPageRenderingOptions` object that controls finer details: `backgroundColor`, `destinationHeight`, `destinationWidth`, `sourceRect`, `isIgnoringHighContrast`, and `bitmapEncoderId`. With these options, as shown in the first PDF viewer sample, you can render a whole page, a zoomed-in page, or a portion of a page (js/scenario1.js):

```
var pdfPage = pdfDocument.getPage(pageIndex);
var pdfPageRenderOptions = new Windows.Data.Pdf.PdfPageRenderOptions();
var renderToStreamPromise;
var pagesize = pdfPage.size;

switch (renderOptions) {
    case RENDEROPTIONS.NORMAL:
       renderToStreamPromise = pdfPage.renderToStreamAsync(pageRenderOutputStream);
       break;
    case RENDEROPTIONS.ZOOM:
       // Set pdfPageRenderOptions.'destinationwidth' or 'destinationHeight' to take
       // zoom factor into effect
       pdfPageRenderOptions.destinationHeight = pagesize.height * ZOOM_FACTOR;
       renderToStreamPromise = pdfPage.renderToStreamAsync(pageRenderOutputStream,
          pdfPageRenderOptions);
       break;
    case RENDEROPTIONS.PORTION:
       // Set pdfPageRenderOptions.'sourceRect' to the rectangle containing portion to show
       pdfPageRenderOptions.sourceRect = PDF_PORTION_RECT;
       renderToStreamPromise = pdfPage.renderToStreamAsync(pageRenderOutputStream,
          pdfPageRenderOptions);
       break;
};
```

The promise that comes back from renderToStreamAsync doesn't have any results, because the rendering will be contained in the stream. If the operation succeeds, your completed handler will be called and you can then pass the stream onto MSApp.createBlobFromRandomAccessStream, followed by our old friend URL.createObjectURL, whose result you can assign to an img.src. If the operation fails, your error handler is called, of course. Be mindful to call the stream's flushAsync first thing before getting the URL and to close the stream (through its close method or blob.msClose). Here's the whole process from the sample (js/scenario1.js):

```
renderToStreamPromise.then(function Flush() {
    return pageRenderOutputStream.flushAsync();
}).then(function DisplayImage() {
    if (pageRenderOutputStream !== null) {
        var blob = MSApp.createBlobFromRandomAccessStream("image/png", pageRenderOutputStream);
        var picURL = URL.createObjectURL(blob, { oneTimeOnly: true });
        scenario1ImageHolder1.src = picURL;
        pageRenderOutputStream.close();
        blob.msClose();  // Closes the stream
    };
},
function error() {
    if (pageRenderOutputStream !== null) {
        pageRenderOutputStream.close();

    }
});
```

If you're using file-based streams, as in the PDF showcase viewer sample, you can just hold onto a collection of StorageFile objects. When you need to render any particular page, you can grab a thumbnail from the StorageFile and pass it to URL.createObjectURL. Alternately, if you use the PdfPageRenderOptions to generate renderings that match your screen size, you can just pass those StorageFile objects to URL.createObjectURL directly. This is what the PDF showcase viewer sample does. Its data source, again, manages a bunch of `StorageFile` objects (or in-memory streams). To show that flow, we can see that each item in the data source is an object with *pageIndex* and *imageSrc* properties (js/pdfLibrary.js):

```
loadPage: function (pageIndex, pdfDocument, pdfPageRenderingOptions, inMemoryFlag, tempFolder) {
    // ... all code as shown earlier

    return promise.then(function (imageStream) {
        var pdfPage = pdfDocument.getPage(pageIndex);
        return pdfPage.renderToStreamAsync(imageStream, pdfPageRenderingOptions)
        .then(function () {
            return imageStream.flushAsync();
        })
        .then(function closeStream() {
            var picURL = null;
            if (inMemoryFlag) {
                var renderStream = Windows.Storage.Streams.RandomAccessStreamReference
                    .createFromStream(imageStream);
                return renderStream.openReadAsync().then(function (stream) {
                    imageStream.close();
                    pdfPage.close();
                    return { pageIndex: pageIndex, imageSrc: stream };
                });
            } else {
                imageStream.close();
                pdfPage.close();
                return { pageIndex: pageIndex, imageSrc: filePointer };
            }
        });
    });
},
```

In default.html, the app's display is composed of nothing more than two ListView controls inside a Semantic Zoom control:

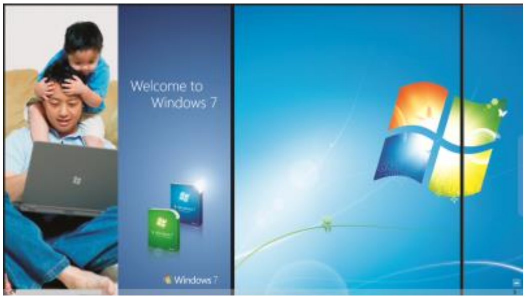
```
<div id="pdfViewTemplate" data-win-control="WinJS.Binding.Template">
    <div id="pdfitemmainviewdiv" data-win-control="WinJS.UI.ViewBox">
        <img src="/images/placeholder.jpg" alt="PDF page"
         data-win-bind="src: imageSrc blobUriFromStream" style="width: 100%; height: 100%;" />
    </div>
</div>

<div id="pdfSZViewTemplate" data-win-control="WinJS.Binding.Template" style="display: none">
    <div>
        <img src="/images/placeholder.jpg" alt="PDF page thumbnail"
            data-win-bind="src: imageSrc blobUriFromStream"/>
    </div>
</div>

<div id="semanticZoomDiv" data-win-control="WinJS.UI.SemanticZoom"
    data-win-options="{zoomedInItem: window.zoomedInItem, zoomedOutItem: window.zoomedOutItem }"
    style="height: 100%; width: 100%">
    <!-- zoomed-in view. -->
    <div id="zoomedInListView" data-win-control="WinJS.UI.ListView"
```

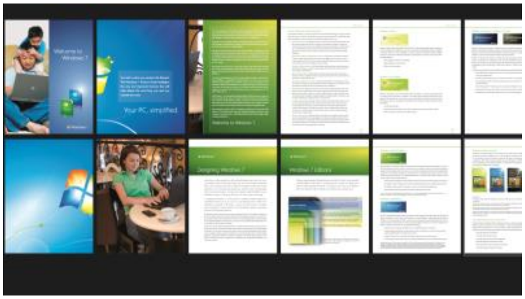
```
        data-win-options="{ itemTemplate: pdfViewTemplate, selectionMode: 'none',
            tapBehavior: 'invokeOnly', swipeBehavior: 'none' ,
            layout: {type: WinJS.UI.GridLayout, maxRows: 1}, }">
    </div>

    <!--- zoomed-out view. -->
    <div id="zoomedOutListView" data-win-control="WinJS.UI.ListView"
        data-win-options="{ itemTemplate: pdfSZViewTemplate, selectionMode: 'none',
            tapBehavior: 'invokeOnly', swipeBehavior: 'none',
            layout: {type: WinJS.UI.GridLayout} }">
    </div>
</div>
```

The last piece that glues it all together is the blobUriFromStream initializer in the data-win-bind statements of the templates. The code for this is hiding out at the bottom of js/default.js and is where the imageSrc from the data source—a StorageFile or stream—gets sent to URL.createObjectURL:

```
window.blobUriFromStream = WinJS.Binding.initializer(function (source, sourceProp,
    dest, destProp) {
    if (source[sourceProp] !== null) {
        dest[destProp] = URL.createObjectURL(source[sourceProp], { oneTimeOnly: true });
    }
});
```

The results of all this are shown in two views below, the zoomed-in view (left) and the zoomed-out view (right), revealing a curious advertisement for Windows 7!





Video Playback and Deferred Loading

Let's now talk about video playback. As we've already seen, simply including a video element in your HTML or creating an element on the fly gives you playback ability. In the code below, the video is sourced from an in-package file, starts playing by itself, loops continually, and provides controls:

```
<video src="/media/ModelRocket1.mp4" controls loop autoplay></video>
```

As with other standards we've discussed, I'm not going to rehash the details (properties, methods, and events) that are available in the W3C spec for the video and audio tags, found on http://www.w3.org/TR/html51/ in sections 4.8.6 to 4.8.10. Especially note the event summary in section 4.8.10.15 and that most of the properties and methods for both are found in section 4.8.10.

Note that the track element for subtitles is supported for both video and audio; you can find an example of using it in scenario 4 of the HTML media playback sample, which includes a WebVTT file (a simple text file; see media/sample-subtitle-en.vtt in the sample) that contains entries like the following to describe when a give subtitle should appear:

```
00:00:05.242 --> 00:00:08.501
My name is Jason Weber, and my job is to make Internet Explorer fast.
```

This file is then referenced in the track element in its src attribute (html/Subtitles.html):

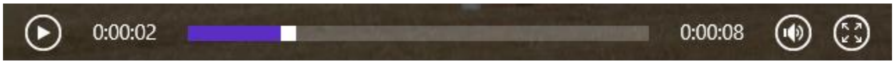
```
<video id="subtitleVideo" style="position: relative; z-index: auto; width: 50%;"
    src="http://ie.microsoft.com/testdrive/Videos/BehindIE9AllAroundFast/Video.mp4"
    poster="images/Win8MediaLogo.png" loop controls>
    <track id="scenario3entrack" src="/media/sample-subtitle-en.vtt" kind="subtitles"
        srclang="en" default>
</video>
```

Another bit that's helpful to understand is that video and audio are closely related, because they're part of the same spec. In fact, if you want to play just the audio portion of a video, you can use the Audio object in JavaScript:

```
//Play just the audio of a video
var movieAudio = new Audio("http://www.kraigbrockschmidt.com/downloads/media/ModelRocket1.mp4");
movieAudio.load();
movieAudio.play();
```

You can also have a video's audio track play in the background depending on the value assigned to the element's msAudioCategory attribute, as we'll see later under "Playback Manager and Background Audio." The short of it is that if you use the value ForegroundOnlyMedia for this attribute, the video will be muted when in the background, and you can also use this condition to automatically pause the video (again, see "Playback Manager and Background Audio"). If you use instead use BackgroundCapableMedia for the attribute, the video's soundtrack can play in the background provided that you've done the other necessary work for background audio. I, for one, appreciate apps that take trouble to make this work—I'll often listen to the audio for conference talks in the background and then watch only the most important video segments.

For any given video element, you can set the width and height to control the playback size (as to 100% for full-screen). This is important when your app view changes size, and you'll likely have CSS styles for video elements in your various media queries. Also, if you have a control to play full screen, simply make the video the size of the viewport. In addition, when you create a video element with the controls attribute, it will automatically have a full-screen control on the far right that does exactly what you expect within a Windows Store app:



In short, you don't need to do anything special to make this work, although you can employ the :-ms-fullscreen pseudo-class in CSS for full-screen styling. When the video is full screen, a similar button (or the ESC key) returns to the normal app view. If there's a problem going to full screen, the video element will fire an MSFullScreenError event.

Note In case you're wondering, the audio and video elements don't provide any CSS pseudo-selectors for styling the controls bar. As my son's preschool teacher would say (in reference to handing out popsicles, but it works here too), "You get what you get and you don't throw a fit and you're happy with it." If you'd like to do something different with these controls, you'll need to turn off the defaults (set the controls attribute to false) and provide controls of your own that would call the element methods appropriately.

When implementing your own controls, be sure to set a timeout to make the controls disappear (either hiding them or changing the z-index) when they're not being used. This is especially important because *whenever the video is partly obstructed by other controls, even by a single pixel, playback decoding will switch from the GPU to the CPU and thus consume more power and other system resources*. So be sure to hide those controls after a short time or size the video so that there's no overlap. Your customers will greatly appreciate it! I, for one, have been impressed with how power-efficient video is with GPU playback on ARM devices such as the Microsoft Surface. In years past, video playback was a total battery killer, but now it's no more an impact than answering emails.

You can use the various events of the video element to know when the video is played and paused through the controls, among other things (though there is not an event for going full-screen), but you should also respond appropriately when hardware buttons for media control are used. For this purpose, listen for the buttonpressed event coming from the Windows.Media.SystemMediaTransport-Controls object.[95] (This is a WinRT object event, so call removeEventListener as needed.) Refer to the System media transport controls sample for a demonstration; the process is basically add a listener for buttonpressed and then enable the buttons for which you want to receive that event (js/scenario1.js):

```
systemMediaControls = Windows.Media.SystemMediaTransportControls.getForCurrentView();
systemMediaControls.addEventListener("buttonpressed", systemMediaControlsButtonPressed, false);
systemMediaControls.isPlayEnabled = true;
systemMediaControls.isPauseEnabled = true;
systemMediaControls.isStopEnabled = true;
systemMediaControls.playbackStatus = Windows.Media.MediaPlaybackStatus.closed;
```

We'll talk more of these later under "The Media Transport Control UI" because they very much apply to audio playback where you might not have any other controls available.

I also mentioned that you might want to defer loading a video (called *lazy loading*) until it's needed and show a preview image in its place. This is accomplished with the poster attribute, whose value is the image to use, and then later setting the src attribute and calling the element's load method:

```
<video id="video1" poster="/media/rocket.png" width="640" height="480"></video>
```

[95] The SystemMediaTransportControls class replaces the deprecated Windows.Media.MediaControl class of Windows 8.

```
var video1 = document.getElementById("video1");
var clickListener = video1.addEventListener("click", function () {
    video1.src = "http://www.kraigbrockschmidt.com/downloads/media/ModelRocket1.mp4";
    video1.load();

    //Remove listener to prevent interference with video controls
    video1.removeEventListener("click", clickListener);

    video1.addEventListener("click", function () {
        video1.controls = true;
        video1.play();
    });
});
```

In this case I'm not using `preload="true"` or even providing a `src` value so that nothing is transferred until the video is started with a click or tap. Then that listener is removed, the video's own controls are turned on, and playback is started. This, of course, is a more roundabout method; often you'll use `preload="true" controls src="..."` directly in the `video` element, as the `poster` attribute will handle the preview image.

Streaming video Windows Store apps can certainly take advantage of streaming media, a subject that we'll return to in "Streaming Media and Play To" at the end of this chapter.

Sidebar: Source Attributes and Custom Formats

In web applications, video (and audio) elements can use HTML5 `source` attributes to provide alternate formats in case a client system doesn't have the necessary codec for the primary source. Given that the list of supported formats in Windows is well known (refer again to [Supported audio and video formats]()), this isn't much of a concern for Windows Store apps. However, `source` is still useful because it can identify the specific codecs for the source:

```
<video controls loop autoplay>
    <source src="video1.vp8" type="video/webm" />
</video>
```

This is important when you need to provide a custom codec for your app through `Windows.Media.MediaExtensionManager`, outlined in the "Handling Custom Audio and Video Formats" section later in this chapter, because the codec identifies the extension to load for decoding. I show WebM as an example here because it's not directly available to Windows Store apps (though it is in Internet Explorer). When the app host running a Windows Store app encounters the `video` element above, it will look for a matching decoder for the specified `type`.

Alternately, the `Windows.Media.Core.MediaStreamSource` object makes it possible for you to handle audio, video, and image formats that aren't otherwise supported in the platform, including plug-in free decryption of protected content. We'll also talk about this in the "Handling Custom Audio and Video Formats" section.

Disabling Screen Savers and the Lock Screen During Playback

When playing video, especially full-screen, it's important to disable any automatic timeouts that would blank the display or lock the device. This is done through the Windows.System.Display.DisplayRequest object. Before starting playback, create an instance of this object and call its requestActive method.

```
var displayRequest = new Windows.System.Display.DisplayRequest();
if (displayRequest) {
    displayRequest.requestActive();
}
```

If this call succeeds, you'll be guaranteed that the screen will stay active despite user inactivity. When the video is complete, be sure to call requestRelease:

```
displayRequest.releaseRequest();
```

See to the simple Display power state sample for a reference project.

Note that Windows will automatically deactivate such requests when your app is moved to the background, and it will reactivate them when the user switches back.

Tip As with image content, if you have a rights-protected video for which you want to disable screen capture, call Windows.UI.ViewManagement.ApplicationView.getForCurrentView() and set the resulting object's isScreenCaptureEnabled property to false. This is again demonstrated in the Disable screen capture sample.

Video Element Extension APIs

Beyond the HTML5 standards for video elements, the app host adds some additional properties and methods, as shown in the following table and documented on the video element page. Also note the references to the HTML media playback sample where you can find some examples of using these.

Properties	Description
msHorizontalMirror	A Boolean that controls whether the playback is flipped horizontally. This is particularly useful when sourcing the video element from a camera to make sure the user sees the proper orientation. See the notes on the enclosureLocation property in "Selecting a Media Capture Device" later on.
msZoom	A Boolean that indicates whether to allow the video element to fit inside its display space by trimming the top/bottom or left/right (when true). This allows apps to give users control over videos whose aspect ratio differs from that of its given display area—that is, to remove letterboxing or pillar boxes. For a demonstration, refer to scenario 3 of the HTML media playback sample.
msIsLayoutOptimalForPlayback (onMSVideoOptimalLayoutChanged)	A Boolean that indicates whether a video will have the best playback based on its layout. When this changes the onMSVideoOptimalLayoutChanged event fires. For details, see How to optimize video rendering and Audio and Video Performance.
msIsStereo3D	A Boolean that indicates whether the system considers the video element's source to be 3D (based on metadata in the video itself). Whether the *system* is itself capable can be determined through Windows.Graphics.Display.DisplayInformation.stereoEnabled. Apps can also listen for DisplayInformation.stereoEnabledChanged (a WinRT event) to

	know when the capabilities change. For details on this and other Stereo 3D concerns, refer to How to enable stereo video playback and scenario 5 of the HTML media playback sample.
msStereo3DRenderMode	Can be mono (the default) or stereo so that apps can control playback. (See above for references.)
msStereo3DPackingMode	Can be none (2D default), topbottom, or sidebyside; this is an adjustment available to apps when the video metadata doesn't clearly indicate which orientation to use. (See above for references.)
msRealtime	Enables the media to reduce initial latency as much as possible for playback. This is important for two-way communication apps, for example, as well as gaming chat, but should be used carefully. For details, refer to How to enable low-latency playback and the Real-time communications sample.
msPlayToDisabled msPlayToPrimary msPlayToSource	Properties related to the Play To feature in Windows. See the "Play To" section at the end of this chapter. Note that these are available on img and audio elements as well.
msAudioTracks	An array of audio track descriptions to support additional languages or other tracks (e.g., commentary). Set msAudioTracks.selectedTrack to the desired index to change the playback audio. For details, refer to How to select audio tracks in different languages as well as scenario 2 of the HTML media playback sample.
msAudioCategory	Identifies the kind of audio being played in the video; see "Playback Manager and Background Audio" later for the specific values. Note that setting this to "Communications" will also set the device type to "Communications" and force msRealtime to true. Typically a video should use "ForegroundOnlyMedia" so that it's muted when the playback app is switched to the background. The app will, in this case, receive an event indicating that the video has been muted; the app can use the mute event to pause the video as well.
msAudioDeviceType	Specifies the output devices that audio will be sent to; see "Audio Element Extension APIs."
Methods	**Description**
msFrameStep (onMSVideoFrameStepCompleted)	Steps the video by one frame forward or backward. The onMSVideoFrameStepCompleted event fires when the step is complete.
msInsertVideoEffect msInsertAudioEffect msClearEffects	Adds or removes effects during playback (see below). All are available on video; msInsertVideoEffect is not available on audio elements.
msSetMediaProtectionManager	Used for DRM with both audio and video; see "Streaming from a Server and Digital Rights Management" toward the end of this chapter.
msSetVideoRectangle	Sets the dimension of a subrectangle within a video.
onMSVideoFrameStepCompleted (event)	Occurs when the video format changes.

Sidebar: Zooming Video for Smaller Screens

With video playback on small devices, it's a good idea to provide a control that sets the msZoom property to true for full-screen playback. By default, full-screen video that doesn't exactly match the aspect ratio of the display will have pillar boxes. On a very small screen—such as 7" or 8" tablets, this might cause the video to be shrunk down to a size that's hard to see. By setting msZoom to true, you remove those pillar boxes automatically. If you want to go further, you can also do full-screen playback by default and even stretch the video element to be larger than the display size, effectively zooming in even further.

Applying a Video Effect

The earlier table shows that video elements have `msInsertVideoEffect` and `msInsertAudioEffect` methods on them. WinRT provides a built-in video stabilization effect that is easily applied to an element. This is demonstrated in scenario 3 of the Media extensions sample, which plays the same video with and without the effect, so the stabilized one is muted:

```
vidStab.msClearEffects();
vidStab.muted = true;
vidStab.msInsertVideoEffect(Windows.Media.VideoEffects.videoStabilization, true, null);
```

Custom effects, as demonstrated in scenario 4 of the sample, are implemented as separate dynamic-link libraries (DLLs*)* written in C++ and are included in the app package because a Windows Store app can install a DLL only for its own use and not for systemwide access. With the sample you'll find DLL projects for a grayscale, invert, and geometric effects, where the latter has three options for fisheye, pinch, and warp. In the js/CustomEffect.js file you can see how these are applied, with the first parameter to `msInsertVideoEffect` being a string that identifies the effect as exported by the DLL (see, for instance, the InvertTransform.idl file in the InvertTransform project):

```
vid.msInsertVideoEffect("GrayscaleTransform.GrayscaleEffect", true, null);

vid.msInsertVideoEffect("InvertTransform.InvertEffect", true, null);
```

The second parameter to `msInsertVideoEffect`, by the way, indicates whether the effect is required, so it's typically `true`. The third is a parameter called *config*, which just contains additional information to pass to the effect. In the case of the geometric effects in the sample, this parameter specifies the particular variation:

```
var effect = new Windows.Foundation.Collections.PropertySet();
effect["effect"] = effectName;
vid.msClearEffects();
vid.msInsertVideoEffect("PolarTransform.PolarEffect", true, effect);
```

where `effectName` will be either "Fisheye", "Pinch", or "Warp".

To be more specific, the *config* argument is a `PropertySet` that you can use to pass any information you need to the effect object. It can also communicate information back: if the effect writes information into the `PropertySet`, it will fire its `mapchanged` event.

Audio effects, not shown in the sample, are applied the same way with `msInsertAudioEffect` (with the same parameters). Do note that each element can have at most two effects per media stream. A `video` element can have two video effects and two audio effects; an `audio` element can have two audio effects. If you try to add more, the methods will throw an exception. This is why it's a good idea to call `msClearEffects` before inserting any others.

For additional discussion on effects and other media extensions, see Using media extensions.

Browsing Media Servers

Many households, including my own, have one or more media servers available on the local network from which apps can play media. Getting to these servers is the purpose of the one other property in `Windows.Storage.KnownFolders` that we haven't mentioned yet: `mediaServerDevices`. As with other known folders, this is simply a `StorageFolder` object through which you can then enumerate or query additional folders and files. In this case, if you call its `getFoldersAsync`, you'll receive back a list of available servers, each of which is represented by another `StorageFolder`. From there you can use file queries, as discussed in Chapter 11, to search for the types of media you're interested in or apply user-provided search criteria. An example of this can be found in the Media Server client sample.

Audio Playback and Mixing

The `audio` element in HTML5 has many things in common with `video`. For one, the `audio` element provides its own playback abilities, including controls, looping, and autoplay:

```
<audio src="media/SpringyBoing.mp3" controls loop autoplay></audio>
```

The same W3C spec applies to both `video` and `audio` elements, so the same code to play just the audio portion of a video is exactly what we use to play an audio file:

```
var sound = new Audio("media/SpringyBoing.mp3");
sound1.msAudioCategory = "SoundEffect";
sound1.load();  //For preloading media
sound1.play();  //At any later time
```

As mentioned earlier in this chapter, creating an `Audio` object without controls and playing it has no effect on layout, so this is what's generally used for sound effects in games and other apps.

As with video, it's important for many audio apps to respond appropriately to the `buttonpressed` event coming from the `Windows.Media.SystemMediaTransportControls` object[96] so that the user can control playback with hardware buttons. This is not a concern with audio such as game sounds, however, where playback control is not needed.

Speaking of which, an interesting aspect of audio is mixing multiple sounds together, as games generally require. Here it's important to understand that each `audio` element can be playing one sound: it has only one source file and one source file alone. However, multiple `audio` (and `video`) elements can be playing at the same time with automatic intermixing depending on their assigned `msAudioCategory` attributes. (See "Playback Manager and Background Audio" below.) In the following example, some background music plays continually (`loop` is set to `true`, and the volume is halved) while another sound is played in response to taps (see the AudioPlayback example with this chapter's

[96] This again replaces the deprecated `Windows.Media.MediaControl` object.

companion content):[97]

```
var sound1 = new Audio("/media/SpringyBoing.mp3");
sound1.msAudioCategory = "SoundEffects";  //Set this before setting src if possible
sound1.load();  //For preloading media

//Background music
var sound2 = new Audio();
sound2.msAudioCategory = "ForegroundOnlyMedia";  //Set this before setting src
sound2.src = "http://www.kraigbrockschmidt.com/mp3/WhoIsSylvia_PortlandOR_5-06.mp3";
sound2.loop = true;
sound2.volume = 0.5; //50%;
sound2.play();

document.getElementById("btnSound").addEventListener("click", function () {
    //Reset position in case we're already playing
    sound1.currentTime = 0;
    sound1.play();
});
```

By loading the tap sound when the object is created, we know we can play it at any time. When initiating playback, it's a good idea to set the `currentTime` to 0 so that the sound always plays from the beginning.

The question with mixing, especially in games, is a matter of managing many different sounds without knowing ahead of time how they will be combined. You may need, for instance, to overlap playback of the same sound with different starting times, but it's impractical to declare three audio elements with the same source. The technique that's emerged is to use "rotating channels," as described on HTML5 Audio Tutorial: Rotating Channels (Ajaxian website) and demonstrated in the AudioPlayback example in this chapter's companion content. To summarize:

15. Declare `audio` elements for each *sound* (with `preload="auto"`), and make sure they aren't showing controls so that they aren't part of your layout..
16. Create a pool (array) of `Audio` *objects* for however many simultaneous channels you need.
17. To play a sound:
    a. Obtain an available `Audio` object from the pool.
    b. Set its `src` attribute to one that matches a preloaded `audio` *element*.
    c. Call that pool object's `play` method.

As sound designers in the movies have discovered, it *is* possible to have too much sound going on at the same time, because it gets really muddied. You might not need more than a couple dozen

[97] And yes, I am playing the guitar and singing the lead part in this live recording, along with my friend Ted Cutler. The song, *Who is Sylvia?*, was composed by another friend, J. Donald Walters, using lyrics of Shakespeare.

channels at most.

> **Hint** Need some sounds for your app? Check out http://www.freesound.org.

> **Custom formats** The Windows.Media.Core.MediaStreamSource object enables you to work with audio formats that don't have native support in the platform. See "Handling Custom Audio and Video Formats" later in this chapter.

Audio Element Extension APIs

As with the video element, a few extensions are available on audio elements as well, namely those to do with effects (msInsertAudioEffect; see "Applying a Video Effect" earlier for a general discussion), DRM (msSetMediaProtectionManager), Play To (msPlayToSource, etc.), msRealtime, and msAudioTracks, as listed earlier in "Video Element Extension APIs." In fact, every extension API for audio exists on video, but two of them have primary importance for audio:

- msAudioDeviceType Allows an app to determine which output device audio will render to: "Console" (the default) and "Communications". This way an app that knows it's doing communication (like chat) doesn't interfere with media audio.

- msAudioCategory Identifies the type of audio being played (see table in the next section), which determines how it will mix with other audio streams. This is also very important to identify audio that can continue to play in the background (thereby preventing the app from being suspended), as described in the next section. Note that you should *always* set this property before setting the audio's src and that setting this to "Communications" will also set the device type to "Communications" and force msRealtime to true.

Do note that despite the similarities between the values in these properties, msAudioDeviceType is for selecting an output device whereas msAudioCategory identifies the *nature* of the audio that's being played through whatever device. A communications category audio could be playing through the console device, for instance, or a media category could be playing through the communications device. The two are separate concepts.

One other capability that's available for audio is *effects discovery*, which means an app can enumerate effects that are being used in the audio processing chain on any given device. I won't go into details here, but refer to the Windows.Media.Effects namespace and the Audio effects discovery sample in the SDK.

Playback Manager and Background Audio

To explore different kinds of audio playback (including the audio track of videos), let's turn our attention to the Playback Manager msAudioCategory sample. I won't show a screen shot of this because, doing nothing but audio, there isn't much to show! Instead, let me outline the behaviors of its different scenarios—which align to msAudioCategory values—in the following table, as well as list those categories that aren't represented in the sample but that can be used in your own app. In each scenario you need to first select an audio file through the file picker.

Scenario	msAudioCategory	Description
1	`BackgroundCapableMedia`	Plays the selected audio when the app is both visible and in the background, including when the user is on the desktop, the Start screen, and the lock screen. The app will not be suspended when in the background, which you can confirm through Task Manager. This is typically used for playing local playlists, local or streaming media files, music videos, etc. Using this requires a declaration in the manifest and handlers for media transport control buttons.
2	`Communications`	Like `BackgroundCapableMedia`, this will also continue to play the selected audio when the app is in the background. Use this for peer-to-peer chat, VoIP, etc.
3	`Other` (the default for `audio` elements)	Plays the selected audio when the app is in the foreground, mixing with background audio; the audio is paused when the app is in the background.
4	`ForegroundOnlyMedia`	Plays the selected audio when the app is in the foreground; the audio is paused and muted when the app is in the background (video will continue to play, however, in which case you use the mute event to also pause the video). When audio of this category is played, background audio will be muted.
5	`Alert`	Plays the selected audio when the app is in the foreground and attenuates background audio. This is used for app notifications like ringtones as well as system alerts.
n/a	`GameMedia`	Used for ambient music in a game.
n/a	`GameEffects`	Used for game sound effects intended to mix with existing audio (all nonmusic sounds).
n/a	`SoundEffects`	Other sound effects (outside of games) intended to mix in with existing audio, such as brief dings, beeps, boinks, and blurps that indicate activity but don't otherwise qualify as alerts.

Where a single audio stream is concerned, there isn't always a lot of difference between most of these categories. Yet as the table indicates, different categories have different effects on other concurrent audio streams. For this purpose, the Windows SDK does an odd thing by providing a second identical sample to the first, the Playback Manager companion sample. This allows you run these apps at the same time (side by side, or one or both in the background) and play audio with different category settings to see how they combine.

How different audio streams combine is a subject that's discussed in the Audio Playback in a Windows Store App whitepaper. However, you don't have direct control over mixing—instead, *the important thing is that you assign the most appropriate category to any particular audio stream*. These categories help the playback manager perform the right level of mixing between audio streams according to user expectations, both with multiple streams in the same app, and streams coming from multiple apps (with limits on how many background audio apps can be going at once). For example, users will expect that alarms, being an important form of notification, will temporarily attenuate other audio streams (just like the GPS system in my car attenuates music when it gives directions). Similarly, users expect that an audio stream of a foreground app takes precedence over a stream of the same category of audio playing in the background.

As a developer, then, avoid playing games with the categories or trying to second guess the mixing algorithms, because you'll end up creating an inconsistent user experience. Just assign the most appropriate category to your audio stream and let the playback manager deliver a consistent systemwide experience with audio from all sources.

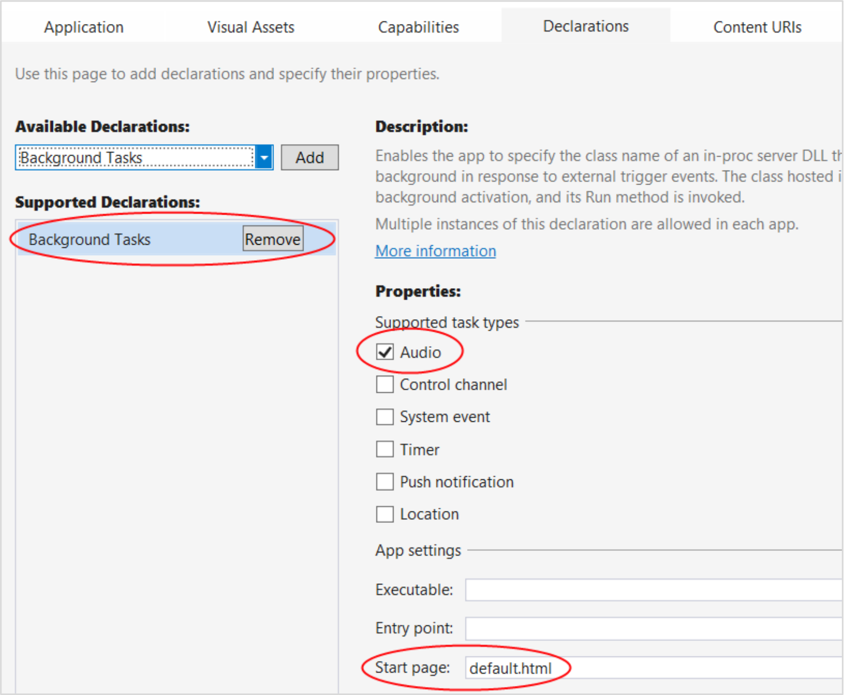
Setting an audio category for any given `audio` element is a simple matter of setting its `msAudioCategory` attribute. Every scenario in the sample does the same thing for this, making sure to set the category before setting the `src` attribute (shown here from js/backgroundcapablemedia.js):

```
audtag = document.createElement('audio');
audtag.setAttribute("msAudioCategory", "BackgroundCapableMedia");
audtag.setAttribute("src", fileLocation);
```

You could accomplish the same thing through `audtag.msAudioCategory` property, as seen in the previous section, as well as in markup:

```
<audio id="audio1" src="song.mp3" msAudioCategory="BackgroundCapableMedia"></audio>
<audio id="audio2" src="voip.mp3" msAudioCategory="Communications"></audio>
<audio id="audio3" src="lecture.mp3" msAudioCategory="Other"></audio>
```

With `BackgroundCapableMedia` and `Communications`, however, simply setting the category isn't sufficient: you also need to declare an audio background task extension in your manifest. This is easily accomplished by going to the Declarations tab in the manifest designer:



Use this page to add declarations and specify their properties.

Available Declarations: Background Tasks

Supported Declarations: Background Tasks

Description:

Enables the app to specify the class name of an in-proc server DLL th
background in response to external trigger events. The class hosted i
background activation, and its Run method is invoked.

Multiple instances of this declaration are allowed in each app.

Properties:

Supported task types

☑ Audio
☐ Control channel
☐ System event
☐ Timer
☐ Push notification
☐ Location

App settings

Executable:

Entry point:

Start page: default.html

First, select Background Tasks from the Available Declarations drop-down list and click Add. Then check Audio under Supported Task Types, and identify a Start Page under App Settings. The start page isn't really essential for background audio (because the app will never be launched for this purpose), but you need to provide something to make the manifest editor happy.

These declarations appear as follows in the manifest XML, should you care to look:

```
<Application Id="App" StartPage="default.html">
  <!-- ... -->
  <Extensions>
    <Extension Category="windows.backgroundTasks" StartPage="default.html">
      <BackgroundTasks>
        <Task Type="audio" />
      </BackgroundTasks>
    </Extension>
  </Extensions>
</Application>
```

Furthermore, background audio apps *must* do a few things with the Windows.Media.SystemMediaTransportControls object that we've already mentioned so that the user can control background audio playback through the media control UI (see the next section):

- Set the object's isPlayEnabled and isPauseEnabled properties to true.
- Listen to the buttonpressed event and handle play and pause cases in your handler by starting and stopping the audio playback as appropriate.

These requirements also make it possible for the playback manager to control the audio streams as the user switches between apps. If you fail to provide these listeners, your audio will always be paused and muted when the app goes into the background. (You can also optionally listen to the propertychanged event that is triggered for sound level changes.)

How to do this is shown in the Playback Manager sample for all its scenarios; the following is from js/backgroundcapablemedia.js (some code omitted), and note that the propertychanged event handler is not required for background audio:

```
var systemMediaControls = Windows.Media.SystemMediaTransportControls.getForCurrentView();

systemMediaControls.addEventListener("propertychanged", mediaPropertyChanged, false);
systemMediaControls.addEventListener("buttonpressed", mediaButtonPressed, false);
systemMediaControls.isPlayEnabled = true;
systemMediaControls.isPauseEnabled = true;


// audtag variable is the global audio element for the page
audtag.setAttribute("msAudioCategory", "BackgroundCapableMedia");
audtag.setAttribute("src", fileLocation);
audtag.addEventListener("playing", audioPlaying, false);
audtag.addEventListener("pause", audioPaused, false);


function mediaButtonPressed(e) {
    switch (e.button) {
        case Windows.Media.SystemMediaTransportControlsButton.play:
            audtag.play();
            break;

        case Windows.Media.SystemMediaTransportControlsButton.pause:
            audtag.pause();
            break;

        default:
            break;
    }
}

function mediaPropertyChanged(e) {
    switch (e.property) {
        case Windows.Media.SystemMediaTransportControlsProperty.soundLevel:
            // Catch SoundLevel notifications and determine SoundLevel state.  If it's muted,
            // we'll pause the player. If your app is playing media you feel that a user should
            // not miss if a VOIP call comes in, you may want to consider pausing playback when
            // your app receives a SoundLevel(Low) notification. A SoundLevel(Low) means your
            // app volume has been attenuated by the system (likely for a VOIP call).
            var soundLevel = e.target.soundLevel;

            switch (soundLevel) {
                case Windows.Media.SoundLevel.muted:
                    log(getTimeStampedMessage("App sound level is: Muted"));
                    break;
                case Windows.Media.SoundLevel.low:
                    log(getTimeStampedMessage("App sound level is: Low"));
                    break;
                case Windows.Media.SoundLevel.full:
                    log(getTimeStampedMessage("App sound level is: Full"));
                    break;
            }

            appMuted();  // Typically only call this for muted and perhaps low levels.
            break;

        default:
            break;
    }
}

function audioPlaying() {
    systemMediaControls.playbackStatus = Windows.Media.MediaPlaybackStatus.playing;
}

function audioPaused() {
    systemMediaControls.playbackStatus = Windows.Media.MediaPlaybackStatus.paused;
}

function appMuted() {
    if (audtag) {
        if (!audtag.paused) {
            audtag.pause();
        }
    }
}
```

Note Using the propertychanged event to detect a SoundLevel.muted on a video is the condition you typically use to pause a foreground-only video.

Given that WinRT events are involved here, the page control's unload handler makes sure to clear everything out (js/backgroundcapablemedia.js):

```
if (systemMediaControls) {
    systemMediaControls.removeEventListener("buttonpressed", mediaButtonPressed, false);
    systemMediaControls.removeEventListener("propertychanged", mediaPropertyChanged, false);
    systemMediaControls.isPlayEnabled = false;
    systemMediaControls.isPauseEnabled = false;
    systemMediaControls.playbackStatus = Windows.Media.MediaPlaybackStatus.closed;
    systemMediaControls = null;
}
```

Again, setting the media control object's isPlayEnabled and isPauseEnabled properties to true, make sure that the play/pause button is clickable in the UI and that the system controls also respond to hardware events, such as the buttons on my keyboard. For example, my keyboard also has next, previous, and stop buttons, but unless the app sets isNextEnabled, isPreviousEnabled, and isStopEnabled and handles those cases in the buttonpressed event, they won't have any effect. We'll see more in the next section.

Note The SystemMediaTransportControls.isEnabled property affects the entire control panel.

The other very important part to making the UI work properly is setting the playbackStatus value, otherwise the actual audio playback will be out of sync with the system controls. Take a look at the code again and you'll see that the playing and pause events of the audio element are wired to functions named audioPlaying and audioPaused. Those functions then set the playbackStatus to the appropriate value from the Windows.Media.MediaPlaybackStatus enumeration, whose values are playing, paused, stopped, closed, and changing.

In short, the buttonpressed event is how an app responds to system control events. Setting playbackStatus is how you then affect the system controls in response to *app* events.

A few additional notes about background audio:

- If the audio is paused, a background audio app will be suspended like any other, but if the user presses a play button, the app will be resumed and audio will then continue playback.
- The use of background audio is carefully evaluated with apps submitted to the Windows Store. If you attempt to play an inaudible track as a means to avoid being suspended, the app will fail Windows Store certification.

- A background audio app should be careful about how it uses the network for streaming media to support the low-power state called connected standby. For details, refer to Writing a power savvy background media app.

Now let's see the UI that Windows displays in response to hardware buttons.

The Media Transport Control UI

As mentioned in the previous section, handling the `buttonpressed` event from the `SystemMediaTransportControls` object is required for background audio so that the user can control the audio through hardware buttons (built into many devices, including keyboards and remote controls) without needing to switch to the app. This is especially important because background audio continues to play not only when the user switches to another app but also when the user switches to the Start screen, switches to the desktop, or locks the device. Furthermore, the system controls also integrate automatically with Play To, meaning that they act as a remote control for the remote Play To device.

The default media control UI appears in the upper left of the screen, as shown in Figure 13-3, regardless of what is on the screen at the time. Tapping anywhere outside the specific control buttons will switch to the app.

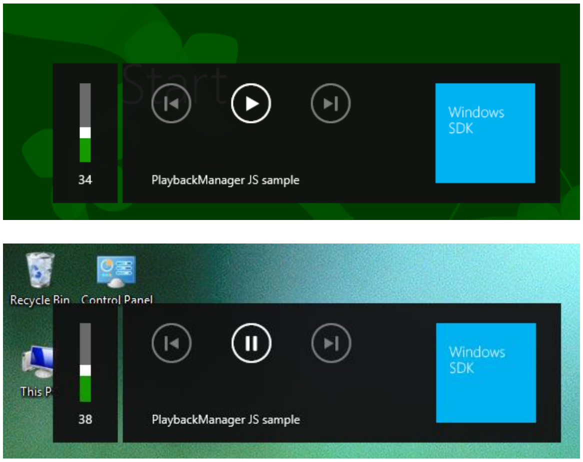


FIGURE 13-3 The system media control UI appearing above the Start screen (top) and the desktop (bottom). It will also show on the lock screen and on top of other Windows Store apps.

Setting the control object's `isPreviousEnabled` and `isNextEnabled` properties to true will, as you'd expect, enable the other two buttons you see in Figure 13-3. This is demonstrated in the System media transport controls sample, in whose single scenario you can select multiple files for playback. When you have multiple files selected, it will play them in sequence, enabling and disabling the buttons depending on the position of the track in the list, as shown in Figure 13-4. (The AudioPlayback example in the companion content shows this as well—see the next section.)

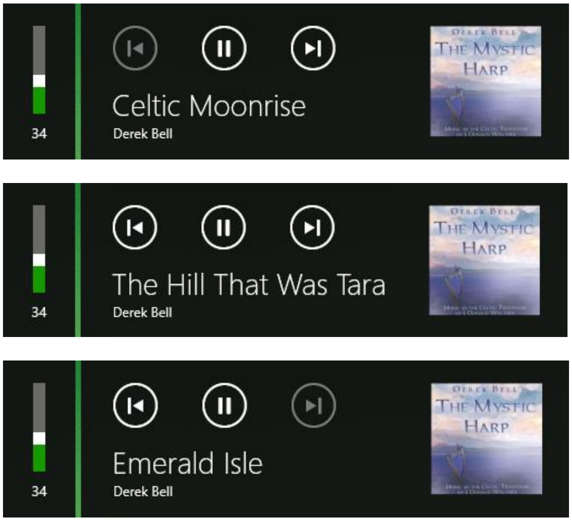


FIGURE 13-4 The system media control UI with different states of the previous and next buttons. Note that the gap between the volume control and the other controls is transparent and just shows whatever is underneath.

Notice the significant difference between Figure 13-3 and Figure 13-4: in the first case we just see the app's Display Name from its manifest along with its tile logo, where in the latter case we see album art along with the track title and artist name. Where do the system controls get this information?

This is done through the `SystemMediaTransportControls.displayUpdater` object, which is of class [`SystemMediaTransportControlsDisplayUpdater`](). Simply said, you populate whichever properties you need within this object and then call its `update` method to send them to the UI.

To populate the properties, you can either extract metadata from a `StorageFile` or set the properties manually. The first way is done with [`displayUpdater.copyFromFileAsync`](), which is how the SDK sample does it (js/scenario1.js; code here is condensed):

```
function updateSystemMediaControlsDisplayAsync(mediaFile) {
    var updatePromise;

    // This is a helper function to return a MediaPlaybackType value
    var mediaType = getMediaTypeFromFileContentType(mediaFile);

    var updatePromise = systemMediaControls.displayUpdater.copyFromFileAsync(
        mediaType, mediaFile);

    return updatePromise.then(function (isUpdateSuccessful) {
        if (!isUpdateSuccessful) {
            // Clear the UI if we couldn't get the metadata
            systemMediaControls.displayUpdater.clearAll();
        }

        systemMediaControls.displayUpdater.update();
    });
}
```

The AudioPlayback example for this chapter has another example, though the MP3s in question don't have associated album art. That is, Windows will automatically retrieve album art from a central service for published recordings and save it as part of the StorageFile properties.

Of course, sometimes you'll be working with unpublished audio (like the AudioPlayback example). You might also be drawing from a source that doesn't readily provide a metadata-equipped StorageFile, or you simply want more control over the process. In all these cases you can set displayUpdater properties individually. First, you *must* set the displayUpdater.type property to a value from the MediaPlaybackType enumeration (music, image, video, or unknown). Failure to do so will cause exceptions in the next step!

Depending on the type—other than unknown—you then populate fields of one of the following groups (where all strings must be less than 128 characters):

Group	Class (in Windows.Media)	Properties
displayUpdater.musicProperties	MusicDisplayProperties	artist, albumArtist, and title
displayUpdater.imageProperties	ImageDisplayProperties	title, subtitle
displayUpdater.videoProperties	VideoDisplayProperties	title, subtitle

Finally, set the displayUpdater.thumbnail property to a RandomAccessStreamReference for the image you want to display—that is, to the result that comes back from one of its static creation methods: createFromFile, createFromStream, or createFromUri.

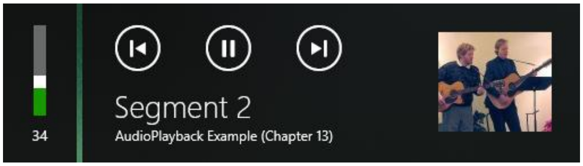
Here's how the AudioPlayback example with this chapter does it. When you first start playback of the music segments, it sets up the invariant parts of the UI (js/default.js):

```
var du = sysMediaControls.displayUpdater;
du.type = Windows.Media.MediaPlaybackType.music;
du.musicProperties.artist = "AudioPlayback Example (Chapter 13)";
var thumbUri = new Windows.Foundation.Uri("ms-appx:///media/albumArt.jpg");
du.thumbnail = Windows.Storage.Streams.RandomAccessStreamReference.createFromUri(thumbUri);
du.update();
```

and then whenever it switches to a different track, it updates the track title (js/default.js):

```
du.musicProperties.title = "Segment " + (curSong + 1);
du.update();
```

The result is as follows:



Playing Sequential Audio

An app that's playing audio tracks (such as music, an audio book, or recorded lectures) will often have a list of tracks to play sequentially, especially while the app is running in the background. In this case it's important to start the next track quickly because Windows will otherwise suspend the app 10 seconds after the current audio is finished. For this purpose, listen for the `audio` element's `ended` event and set `audio.src` to the next track. A good optimization in this case is to create a second `Audio` object and set its `src` attribute after the first track starts to play. This way that second track will be preloaded and ready to go immediately, thereby avoiding potential delays in playback between tracks. This is shown in the AudioPlayback example for this chapter, where I've split the one complete song into four segments for continuous playback. It also shows again how to handle the next and previous button events, along with setting the segment number as the track name:

```
var sysMediaControls = Windows.Media.SystemMediaTransportControls.getForCurrentView();
var playlist = ["media/segment1.mp3", "media/segment2.mp3", "media/segment3.mp3",
    "media/segment4.mp3"];
var curSong = 0;
var audio1 = null;
var preload = null;

document.getElementById("btnSegments").addEventListener("click", playSegments);
audio1 = document.getElementById("audioSegments");
preload = document.createElement("audio");

audio1.addEventListener("playing", function () {
    sysMediaControls.playbackStatus = Windows.Media.MediaPlaybackStatus.playing;
});

audio1.addEventListener("pause", function () {
    sysMediaControls.playbackStatus = Windows.Media.MediaPlaybackStatus.paused;
});

//Starts playback of sequential segments
function playSegments() {
    e.target.disabled = true;  //Prevent reentrancy

    curSong = 0;

    //Pause the other music
    document.getElementById("musicPlayback").pause();

    sysMediaControls.isPlayEnabled = true;
    sysMediaControls.isPauseEnabled = true;
    sysMediaControls.isNextEnabled = true;
    sysMediaControls.isPreviousEnabled = false;

    //Remember to remove this WinRT event listener if it goes out of scope
    sysMediaControls.addEventListener("buttonpressed", function (e) {
        var wmb = Windows.Media.SystemMediaTransportControlsButton;

        switch (e.button) {
            case wmb.play:
```

```
            audio1.play();
            break;

         case wmb.pause:
         case wmb.stop:
            audio1.pause();
            break;

         case wmb.next:
            playNext();
            break;

         case wmb.previous:
            playPrev();
            break;

         default:
            break;
      }
   });

   //Set invariant metadata [omitted--code is in previous section]


   //Show the element (initially hidden) and start playback
   audio1.style.display = "";
   audio1.volume = 0.5; //50%;
   playCurrent();

   //Preload the next track in readiness for the switch
   var preload = document.createElement("audio");
   preload.setAttribute("preload", "auto");
   preload.src = playlist[1];

   //Switch to the next track as soon as one had ended or next button is pressed
   audio1.addEventListener("ended", playNext);
}

function playCurrent() {
   audio1.src = playlist[curSong];
   audio1.play();

   //Update metadata title [omitted]
}

function playNext() {
   curSong++;

   //Enable previous button if we have at least one previous track
   sysMediaControls.isPreviousEnabled = (curSong > 0);

   if (curSong < playlist.length) {
      playCurrent();       //playlist[curSong] should already be loaded
```

```
        //Set up the next preload
        var nextTrack = curSong + 1;

        if (nextTrack < playlist.length) {
            preload.src = playlist[nextTrack];
        } else {
            preload.src = null;

            //Disable next if we're at the end of the list.
            sysMediaControls.isNextEnabled = false;
        }
    }
}

function playPrev() {
    //Enable Next unless we only have one song in the list
    sysMediaControls.isNextEnabled = (curSong != playlist.length - 1);

    curSong--;

    //Disable previous button if we're at the beginning now
    sysMediaControls.isPreviousEnabled = (curSong != 0);

    playCurrent();
    preload.src = playlist[curSong + 1]; //This should always work
}
```

When playing sequential tracks like this from an app written in JavaScript and HTML, you might notice brief gaps between the tracks, especially if the first track flows directly into the second. This is a present limitation of the platform given the layers that exist between the HTML `audio` element and the low-level XAudio2 APIs that are ultimately doing the real work. You can mitigate the effects to some extent—for example, you can crossfade the two tracks or crossfade a third overlay track that contains a little of the first and a little of the second track. You can also use a negative time offset to start playing the next track slightly before the previous one ends. But if you want a truly seamless transition, you'll need to bypass the `audio` element and use the XAudio2 APIs from a WinRT component for direct playback. How to do this is discussed in the Building your own Windows Runtime components to deliver great apps post on the Windows developer blog.

Playlists

The AudioPlayback example in the previous section is clearly contrived because an app wouldn't typically have an in-memory playlist. More likely an app would load an existing playlist or create one from files that a user has selected.

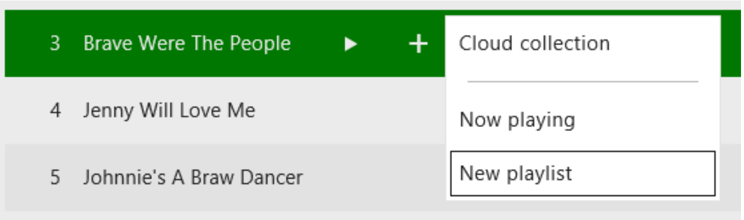
WinRT supports these actions through a simple API in `Windows.Media.Playlists`, which supports the WPL (Windows Media Player), ZPL (Zune), and M3U formats. The Playlist sample in the Windows

SDK[98] shows how to perform various tasks with the API. Scenario 1 lets you choose multiple files with the file picker, creates a new Playlist object, adds those files to its files list (a StorageFile vector), and saves the playlist with its saveAsAsync method (this code from js/create.js is simplified and reformatted a bit):

```
function pickAudio() {
    var picker = new Windows.Storage.Pickers.FileOpenPicker();
    picker.suggestedStartLocation = Windows.Storage.Pickers.PickerLocationId.musicLibrary;
    picker.fileTypeFilter.replaceAll(SdkSample.audioExtensions);

    picker.pickMultipleFilesAsync().done(function (files) {
        if (files.size > 0) {
            SdkSample.playlist = new Windows.Media.Playlists.Playlist();

            files.forEach(function (file) {
                SdkSample.playlist.files.append(file);
            });

            SdkSample.playlist.saveAsAsync(Windows.Storage.KnownFolders.musicLibrary,
                "Sample", Windows.Storage.NameCollisionOption.replaceExisting,
                Windows.Media.Playlists.PlaylistFormat.windowsMedia)
                .done();
        }
}
```

Notice that saveAsAsync takes a StorageFolder and a name for the file (along with an optional format parameter). This accommodates a common use pattern for playlists where a music app has a single folder in which it stores playlists and provides users with a simple means to name them and/or select them. In this way, playlists aren't typically managed like other user data files where one always goes through a file picker to do a Save As into an arbitrary folder. You could use FileSavePicker, get a StorageFile, and use its path property to get to the appropriate StorageFolder, but more likely you'll save playlists in one place and present them as entities that appear only within the app itself.

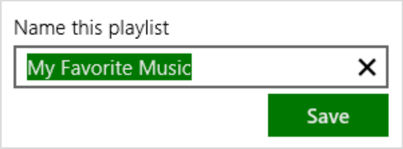
For example, the Music app that comes with Windows allows you create a new playlist when you're viewing tracks of some album:



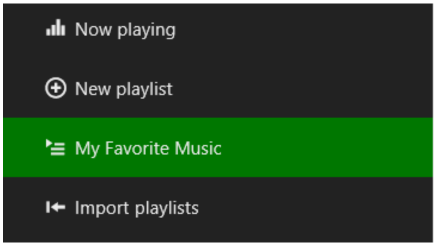
Or you can use the New Playlist command on the app's left control pane. In either case, selecting New

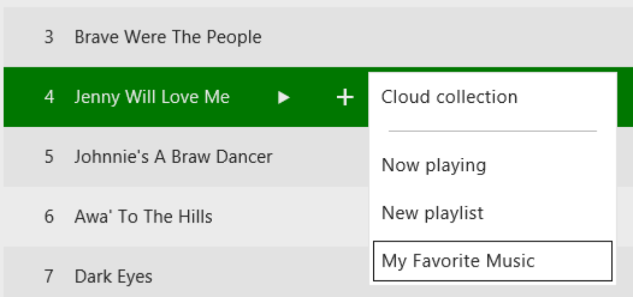
[98] This sample almost wins the prize for the *shortest* name in the Windows SDK, but it's beaten by the AtomPub sample, the Indexer sample, and the grand champion of conciseness—the Print sample!

Playlist displays a flyout in which you provide a name:



After this, the playlist will appear both on the left-side controls pane (below left), which makes it playable like an album, and in the track menu (below right):





In other words, though playlists might be saved in discrete files, they aren't necessarily presented that way to the user, and the API reflects that usage pattern.

Loading a playlist uses the `Playlist.loadAsync` method given a `StorageFile` for the playlist. This might be a `StorageFile` obtained from a file picker or from the enumeration of the app's private playlist folder. Scenario 2 of the Playlist sample (js/display.js) demonstrates the former, where it then goes through each file and requests their music properties (refer back to Chapter 11 in the section "Media Specific Properties" for information on media file properties and the applicable APIs):

```
function displayPlaylist() {
    var picker = new Windows.Storage.Pickers.FileOpenPicker();
    picker.suggestedStartLocation = Windows.Storage.Pickers.PickerLocationId.musicLibrary;
    picker.fileTypeFilter.replaceAll(SdkSample.playlistExtensions);

    var promiseCount = 0;

    picker.pickSingleFileAsync()
        .then(function (item) {
            if (item) {
                return Windows.Media.Playlists.Playlist.loadAsync(item);
            }
            return WinJS.Promise.wrapError("No file picked.");
        })
        .then(function (playlist) {
            SdkSample.playlist = playlist;
            var promises = {};

            // Request music properties for each file in the playlist.
            playlist.files.forEach(function (file) {
                promises[promiseCount++] = file.properties.getMusicPropertiesAsync();
            });

            // Print the music properties for each file. Due to the asynchronous
            // nature of the call to retrieve music properties, the data may appear
            // in an order different than the one specified in the original playlist.
            // To guarantee the ordering, we use Promise.join with an associative array
            // passed as a parameter, containing an index for each individual promise.
            return WinJS.Promise.join(promises);
        })
        .done(function (results) {
            var output = "Playlist content:\n\n";

            var musicProperties;
            for (var resultIndex = 0; resultIndex < promiseCount; resultIndex++) {
                musicProperties = results[resultIndex];
                output += "Title: " + musicProperties.title + "\n";
                output += "Album: " + musicProperties.album + "\n";
                output += "Artist: " + musicProperties.artist + "\n\n";
            }

            if (resultIndex === 0) {
                output += "(playlist is empty)";
            }

        }, function (error) {
            // ...
        });
}
```

The other method for managing a playlist is `PlayList.saveAsync`, which takes a single `StorageFile`. This is what you'd use if you've loaded and modified a playlist and simply want to save those changes (typically done automatically when the user adds or removes items from the playlist). This is demonstrated in scenarios 3, 4, and 5 of the sample (js/add.js, js/remove.js, and js/clear.js), which just use methods of the `Playlist.files` vector like `append`, `removeAtEnd`, and `clear`, respectively.

Playback of a playlist depends, of course, on the type of media involved, but typically you'd load a playlist and sequentially take the next `StorageFile` object from its `files` vector, pass it to `URL.createObjectURL`, and then assign that URI to the `src` attribute of an `audio` or `video` element. You could also use playlists to manage lists of videos and images for sequential showing as well.

Sidebar: Background Downloading

Now it's not always the case that everything you want to play is already on the local file system: you might want to be downloading the next track at the same time you're streaming the current one. This is a great opportunity to use the background downloader API that we talked about in Chapter 4 and, more specifically, the high priority and unconstrained download features that the API provides. Mind you, this would be for scenarios where you want to retain a copy of the media on the local file system after playback is done—you wouldn't need to worry about this if you're in a streaming-only situation.

Anyway, let's say that you want to download and play an album in its entirety. For this you'd use a strategy like the following:

- Start a download operation for the first track at high priority.
- Start additional downloads for one or more subsequent tracks at normal priority.
- As soon as the first track is transferred, begin playback and change the next track's download priority to high.
- Repeat the process of starting additional downloads as necessary, always setting the next track's priority to high so that it gets transferred soonest.

Alongside setting priorities, you might also configure these as unconstrained downloads if you'd like to allow the device to go into connected standby and continue to download and play, which would be very important for a series of videos where each file transfer could be quite large. Each request is subject to user consent, of course, but the capability is there so that the user can enjoy a continuous media experience without having everything in the system continue to run as it normally would.

More details on this subject can be found in Writing a power savvy background media app.

Text to Speech

Before we delve into the next round of media topics in this chapter, it's a good time to take a little break and look at a different set of APIs that are very much related to audio but don't rely on any preexisting media files: text to speech. These APIs are found in the `Windows.Media.SpeechSynthesis` namespace, specifically in the `SpeechSynthesizer` class.

The basic "hear me now" usage is straightforward:

- Create an `Audio` object to handle playback.
- Create an instance of the `SpeechSynthesizer` object.
- Call its `synthesizeTextToStreamAsync` method, which results in a `SpeechSynthesisStream` object.
- Pass that stream object to `MSApp.createBlobFromRandomAccessStream` to get a blob.
- Pass the blob to `URL.createObjectURL`.
- Assign the resulting URL to the audio's `src` property and then call its `play` method when you're ready for playback.

Here's how it's done in scenario 1 of the Speech synthesis sample, where the *Data* element is a text box in which you can type whatever you want (js/SpeakText.js):

```
var synth = new Windows.Media.SpeechSynthesis.SpeechSynthesizer();
var txtData = document.getElementById("Data");
var audio = new Audio();

synth.synthesizeTextToStreamAsync(txtData.value).then(function (markersStream) {
    var blob = MSApp.createBlobFromRandomAccessStream(markersStream.ContentType, markersStream);
    audio.src = URL.createObjectURL(blob, { oneTimeOnly: true });
    markersStream.seek(0); //start at beginning when speak is hit
    audio.AutoPlay = Boolean(true);
    audio.play();
});
```

The same scenario also offers an option to save the speech stream into a WAV file rather than playing it back. Here, `file` is a `StorageFile` from a `FileSavePicker`, and the code is just the same business of playing with the necessary files, buffers, and streams to get the job done (js/SpeakText.js):

```
synth.synthesizeTextToStreamAsync(txtData.value).then(function (markerStream) {
    var buffer = new Windows.Storage.Streams.Buffer(markerStream.size);
    file.openAsync(Windows.Storage.FileAccessMode.readWrite).then(function (writeStream) {
        var outputStream = writeStream.getOutputStreamAt(writeStream.size);
        var dataWriter = new Windows.Storage.Streams.DataWriter(outputStream);

        markerStream.readAsync(buffer, markerStream.size,
            Windows.Storage.Streams.InputStreamOptions.none).then(function () {
            dataWriter.writeBuffer(buffer);
            // close the data file streams
            dataWriter.storeAsync().then(function () {
                outputStream.flushAsync().then(function () {
                });
            });
        });
    });
})
```

Scenario 2 is exactly the same except that it works with the `synthesizeSsmlToStreamAsync` method, which supports the use of Speech Synthesis Markup Language (SSML) instead of just plain text. SSML is a W3C standard that enables you to encode much more subtlety and accurate pronunciations into your source text. For example, the phoneme tag lets you spell out the exact phonetic syllables for a word like *whatchamacallit* (html/SpeakSSML.html):

```
<phoneme alphabet='x-microsoft-ups' ph='S1 W AA T . CH AX . M AX . S2 K AA L . IH T'>
    whatchamacallit
</phoneme>
```

Give the sample a try if you don't know what that word sounds like!

The one other option you have with speech synthesis is to choose from a variety of *voices* that support different languages. For the complete list of 17 options covering almost as many languages, see the documentation for the `SpeechSynthesizer.voice` property. Note, however, that voices get installed on a device only as part of a set of locale-specific language resources, so only a smaller subset is typically available.[99] That list is available in the `SpeechSynthesizer.allVoices` vector with the default one in `defaultVoice`. You can enumerate the contents of `allVoices` to create a list of options for the user, if you want, or you can programmatically select one according to your preferences. (The sample does this to populate a drop-down list.)

To select a voice, simply copy one of the elements from `allVoices` into the `voice` property (js/SpeakText.js):

```
// voicesSelect is a drop-down element
var allVoices = Windows.Media.SpeechSynthesis.SpeechSynthesizer.allVoices;
var selectedVoice = allVoices[voicesSelect.selectedIndex];
synth.voice = selectedVoice;
```

And that's really all there is to it! I suspect by now you want to give it a try if you haven't already, and I'm sure you can think of creative ways to employ this API in your own projects, especially for teaching aids, accessibility features, and perhaps in early childhood educational apps.

Sidebar: OK, What About Speech Recognition?

Although WinRT has an API for speech synthesis, it does not at present have one for speech recognition. Fortunately, Bing provides a speech recognition control for Windows and Windows Phone that you can learn more about on http://www.bing.com/dev/en-us/speech.

Loading and Manipulating Media

So far in this chapter we've seen how to display images and play audio and video by using their respective HTML elements: `img`, `audio`, and `video`. We also covered a number of related topics in Chapter 11, including:

- Programmatically accessing the user's media libraries through `Windows.Storage.KnownFolders` with the appropriate manifest capabilities, as well as removable storage.
- Using thumbnails to display images or video placeholders instead of loading up the entire file contents.
- Retrieving and modifying file properties, including those specific to media, through the `getImagePropertiesAsync`, `getVideoPropertiesAsync`, and `getMusicPropertiesAsync` methods of `StorageFile.properties`.
- Using the `Windows.Storage.StorageLibrary` object to add folders to and remove folders

[99] To install a voice, go to PC Settings > Time and Language > Region and Language. First click "+ Add a Language" to select a language to activate, and then when it appears in the list on this page, select it, click Options, and that will take you to a page where you can download the language pack that includes the voice.

from the media libraries.

- Enumerating folder contents using queries.

In short, what we've covered to this point in the book is how to *consume* media for the purposes of display and playback, as well as creating gallery views of library content. We turn our attention now to *manipulating* media, namely the concerns of transcoding and editing. (Simply changing file-level properties, like title and author, are covered in Chapter 11.) For this we'll be looking at the core APIs that make this possible, including those that give you access to the raw media stream. What you then *do* with that raw stream is up to you—we'll see some basic examples here, but we won't delve into anything more specific than that because the subject can quickly become intricate and complicated. For that reason you'll probably find it helpful to refer to some of the topics in the documentation, such as Processing image files, Transcoding (audio and video), and Using media extensions.

Image Manipulation and Encoding

To do something more with an image than just loading and displaying it (where again you can apply various CSS transforms for effect), you need to get to the actual pixels by means of a *decoder*. This already happens under the covers when you assign a URI to an `img.src`, but to have direct access to pixels means decoding manually. On the flip side, saving pixels back out to an image file means using an encoder.

WinRT provides APIs for both in the `Windows.Graphics.Imaging` namespace, namely in the `BitmapDecoder`, `BitmapTransform`, and `BitmapEncoder` classes. Loading, manipulating, and saving an image file often involves these three classes in turn, though the `BitmapTransform` object is focused on rotation and scaling, so you won't use it if you're doing other manipulations.

One demonstration of this API can be found in scenario 2 of the Simple imaging sample. I'll leave it to you to look at the code directly, however, because it gets fairly involved—up to 11 chained promises to save a file! It also does all decoding, manipulation, and encoding within a single function such as `saveHandler` (js/scenario2.js). Here's the process it follows:

1. Open a file with `StorageFile.openAsync`, which provides a stream.
2. Pass that stream to the static method `BitmapDecoder.createAsync`, which provides a specific instance of `BitmapDecoder` for the stream.
3. Pass that decoder to the static method `BitmapEncoder.createForTranscodingAsync`, which provides a specific `BitmapEncoder` instance. This encoder is created with a new instance of `Windows.Storage.Streams.InMemoryRandomAccessStream`.
4. Set properties in the encoder's `bitmapTransform` property (a `BitmapTransform` object) to configure scaling and rotation. This creates the transformed graphic in the in-memory stream.
5. Create a property set (`Windows.Graphics.Imaging.BitmapPropertySet`) that includes *System.Photo.Orientation* and use the encoder's `bitmapProperties.setPropertiesAsync` to

save it.

6. Copy the in-memory stream to the output file stream by using `Windows.Storage.Stream.RandomAccessStream.copyAsync`.
7. Close both streams with their respective `close` methods (this is what closes the file).

As comprehensive as this scenario is, it's helpful to look at different stages of the process separately, for which purpose we have the ImageManipulation example in this chapter's companion content. This lets you pick and load an image, convert it to grayscale, and save that converted image to a new file. Its output is shown in Figure 13-5. It also gives us an opportunity to see how we can send decoded image data to an HTML `canvas` element and save that canvas's contents to a file.

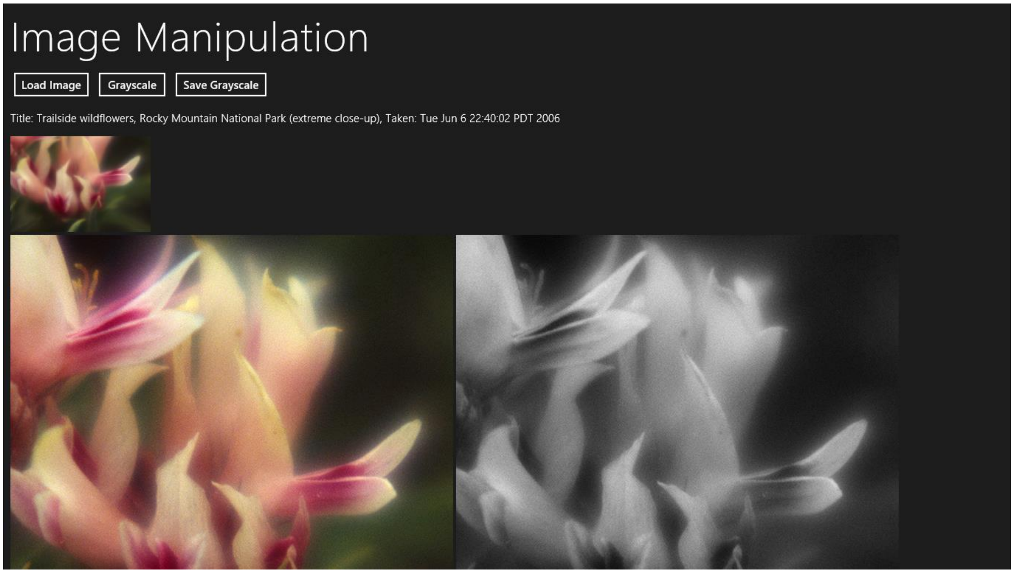


Image Manipulation

Title: Trailside wildflowers, Rocky Mountain National Park (extreme close-up), Taken: Tue Jun 6 22:40:02 PDT 2006

FIGURE 13-5 Output of the ImageManipulation example in the chapter's companion content.

The handler for the Load Image button (`loadImage` in js/default.js) provides the initial display. It lets you select an image with the file picker, displays the full-size image (not a thumbnail) in an `img` element with `URL.createObjectURL`, calls `StorageFile.properties.getImagePropertiesAsync` to retrieve the `title` and `dateTaken` properties, and uses `StorageFile.getThumbnailAsync` to provide the thumbnail at the top. We've seen all of these APIs in action already.

Clicking the Grayscale button enters the `setGrayscale` handler where the interesting work happens. We call `StorageFile.openReadAsync` to get a stream, call `BitmapDecoder.createAsync` with that to obtain a decoder, cache some details from the decoder in a local object (`encoding`), and call `BitmapDecoder.getPixelDataAsync` and copy those pixels to a canvas (and only three chained async operations here!):

```
var Imaging = Windows.Graphics.Imaging;  //Shortcut
var imageFile;                          //Saved from the file picker
var decoder;                            //Saved from BitmapDecoder.createAsync
var encoding = {};                       //To cache some details from the decoder

function setGrayscale() {
    //Decode the image file into pixel data for a canvas

    //Get an input stream for the file (StorageFile object saved from opening)
    imageFile.openReadAsync().then(function (stream) {
        //Create a decoder using static createAsync method and the file stream
        return Imaging.BitmapDecoder.createAsync(stream);
    }).then(function (decoderArg) {
        decoder = decoderArg;

        //Configure the decoder if desired. Default is BitmapPixelFormat.rgba8 and
        //BitmapAlphaMode.ignore. The parameterized version of getPixelDataAsync can also
        //control transform, ExifOrientationMode, and ColorManagementMode if needed.

        //Cache these settings for encoding later
        encoding.dpiX = decoder.dpiX;
        encoding.dpiY = decoder.dpiY;
        encoding.pixelFormat = decoder.bitmapPixelFormat;
        encoding.alphaMode = decoder.bitmapAlphaMode;
        encoding.width = decoder.pixelWidth;
        encoding.height = decoder.pixelHeight;

        return decoder.getPixelDataAsync();
    }).done(function (pixelProvider) {
        //detachPixelData gets the actual bits (array can't be returned from
        //an async operation)
        copyGrayscaleToCanvas(pixelProvider.detachPixelData(),
               decoder.pixelWidth, decoder.pixelHeight);
    });
}
```

The decoder's `getPixelDataAsync` method comes in two forms. The simple form, shown here, decodes using defaults. The full-control version lets you specify other parameters, as explained in the code comments above. A common use of this is doing a transform using a `Windows.Graphics.Imaging.BitmapTransform` object (as mentioned before), which accommodates scaling (with different interpolation modes), rotation (90-degree increments), cropping, and flipping.

Either way, what you get back from the `getPixelDataAsync` is not the actual pixel array, because of a limitation in the WinRT language projection mechanism whereby an asynchronous operation cannot return an array. Instead, the operation returns a `PixelDataProvider` object whose singular super-exciting synchronous method called `detachPixelData` gives you the array you want. (And that method can be called only once and will fail on subsequent calls, hence the "detach" in the method name.) In the end, though, what we have is exactly the data we need to manipulate the pixels and display the result on a canvas, as the `copyGrayscaleToCanvas` function demonstrates. You can, of course, replace this kind of function with any other manipulation routine:

```
function copyGrayscaleToCanvas(pixels, width, height) {
    //Set up the canvas context and get its pixel array
    var canvas = document.getElementById("canvas1");
    canvas.width = width;
    canvas.height = height;
    var ctx = canvas.getContext("2d");

    //Loop through and copy pixel values into the canvas after converting to grayscale
    var imgData = ctx.createImageData(canvas.width, canvas.height);
    var colorOffset = { red: 0, green: 1, blue: 2, alpha: 3 };
    var r, g, b, gray;
    var data = imgData.data;  //Makes a huge perf difference to not dereference
                              //imgData.data each time!

    for (var i = 0; i < pixels.length; i += 4) {
        r = pixels[i + colorOffset.red];
        g = pixels[i + colorOffset.green];
        b = pixels[i + colorOffset.blue];

        //Calculate brightness value for each pixel
        gray = Math.floor(30 * r + 55 * g + 11 * b) / 100;

        data[i + colorOffset.red] = gray;
        data[i + colorOffset.green] = gray;
        data[i + colorOffset.blue] = gray;
        data[i + colorOffset.alpha] = pixels[i + colorOffset.alpha];
    }

    //Show it on the canvas
    ctx.putImageData(imgData, 0, 0);

    //Enable save button
    document.getElementById("btnSave").disabled = false;
}
```

This is a great place to point out that JavaScript isn't necessarily the best language for working over a pile of pixels like this, though in this case the performance of a Release build running outside the debugger is actually quite good. Such routines may be better implemented as a WinRT component in a language like C# or C++ and made callable by JavaScript. We'll take the opportunity to do exactly this in Chapter 18, "WinRT Components," where we'll also see limitations of the `canvas` element that require us to take a slightly different approach.

Saving this canvas data to a file then happens in the `saveGrayscale` function, where we use the file picker to get a `StorageFile`, open a stream, acquire the `canvas` pixel data, and hand it off to a `BitmapEncoder`:

```
function saveGrayscale() {
    var picker = new Windows.Storage.Pickers.FileSavePicker();
    picker.suggestedStartLocation =
        Windows.Storage.Pickers.PickerLocationId.picturesLibrary;
    picker.suggestedFileName = imageFile.name + " - grayscale";
    picker.fileTypeChoices.insert("PNG file", [".png"]);
    var imgData, fileStream = null;

    picker.pickSaveFileAsync().then(function (file) {
        if (file) {
            return file.openAsync(Windows.Storage.FileAccessMode.readWrite);
        } else {
            return WinJS.Promise.wrapError("No file selected");
        }
    }).then(function (stream) {
        fileStream = stream;
        var canvas = document.getElementById("canvas1");
        var ctx = canvas.getContext("2d");
        imgData = ctx.getImageData(0, 0, canvas.width, canvas.height);

        return Imaging.BitmapEncoder.createAsync(
            Imaging.BitmapEncoder.pngEncoderId, stream);
    }).then(function (encoder) {
        //Set the pixel data--assume "encoding" object has options from elsewhere.
        //Conversion from canvas data to Uint8Array is necessary because the array type
        //from the canvas doesn't match what WinRT needs here.
        encoder.setPixelData(encoding.pixelFormat, encoding.alphaMode,
            encoding.width, encoding.height, encoding.dpiX, encoding.dpiY,
            new Uint8Array(imgData.data));

        //Go do the encoding
        return encoder.flushAsync();
    }).done(function () {
        fileStream.close();
    }, function () {
        //Empty error handler (do nothing if the user canceled the picker)
    });
}
```

Note how the `BitmapEncoder` takes a codec identifier in its first parameter. We're using `pngEncoderId`, which is, as you can see, defined as a static property of the `Windows.Graphics.Imaging.BitmapEncoder` class; other values are `bmpEncoderId`, `gifEncoderId`, `jpegEncoderId`, `jpegXREncoderId`, and `tiffEncoderId`. These are the formats supported by the API. You can set additional properties of the `BitmapEncoder` before setting pixel data, such as its `BitmapTransform`, which will then be applied during encoding.

One gotcha to be aware of here is that the pixel array obtained from a `canvas` element (a DOM `CanvasPixelArray`) is not directly compatible with the WinRT byte array required by the encoder. This is the reason for the `new Uint8Array` call down there in the last parameter.

Note Scenario 3 of the SDK's Simple imaging sample performs a different manipulation on `canvas` contents—applying artistic strokes—so you can refer to that for another demonstration. Its process of saving canvas contents is pretty much the same thing as shown above.

Transcoding and Custom Image Formats

In the previous section we mostly saw the use of a BitmapEncoder created with that class's static createAsync method to write a new file. That's all well and good, but you might want to know about a few of the encoder's other capabilities.

First is the BitmapEncoder.createForTranscodingAsync method that was mentioned briefly in the context of the Simple imaging sample. This specifically creates a new encoder that is initialized from an existing BitmapDecoder. This is primarily used to manipulate some aspects of the source image file while leaving the rest of the data intact. To be more specific, you can first change those aspects that are expressed through the encoder's setPixelData method: the pixel format (rgba8, rgba16, and bgra8; see BitmapPixelFormat), the alpha mode (premultiplied, straight, or ignore; see BitmapAlphaMode), the image dimensions, the image DPI, and, of course, the pixel data itself. Beyond that, you can change other properties through the encoder's bitmapProperties.setPropertiesAsync method. In fact, if all you need to do is change a few properties and you don't want to affect the pixel data, you can use BitmapEncoder.createForInPlacePropertyEncodingAsync instead (how's that for a method name!). This encoder allows calls to only bitmapProperties.setPropertiesAsync, bitmapProperties.getPropertiesAsync, and flushAsync, and since it can assume that the underlying data in the file will remain unchanged, it executes much faster than its more flexible counterparts and it has less memory overhead.

An encoder from createForTranscodingAsync does *not* accommodate a change of image file format (e.g., JPEG to PNG); for that you need to use createAsync wherein you can specify the specific kind of encoding. As we've already seen, the first argument to createAsync is a codec identifier, for which you normally pass one of the static properties of BitmapEncoder such as pngEncoderId. What I haven't mentioned is that you can also specify custom codecs in this first parameter and that the createAsync call also supports an optional third argument in which you can provide options for the particular codec in question. However, there are complications and restrictions here.

Let me address options first. The present documentation for the BitmapEncoder codec values (like pngEncoderId) lacks any details about available options. For that you need to instead refer to the docs for the Windows Imaging Component (WIC), specifically the Native WIC Codecs that are what WinRT surfaces to Windows Store apps. If you go into the page for a specific codec, you'll then see a section on "Encoder Options" that tells you what you can use. For example, the JPEG codec supports properties like ImageQuality (a value between 0.0 and 1.0), as well as built-in rotations. The PNG codec supports properties like FilterOption for various compression optimizations.

To provide these properties, you need to create a new BitmapPropertySet and insert an entry in that set for each desired option. If, for example, you have a variable named quality that you want to apply to a JPEG encoding, you'd create the encoder like this:

```
var options = new Windows.Graphics.Imaging.BitmapPropertySet();
options.insert("ImageQuality", quality);
var encoderPromise = Imaging.BitmapEncoder.createAsync(Imaging.BitmapEncoder.jpegEncoderId,
    stream, options);
```

You use the same `BitmapPropertySet` for any properties you might pass to an encoder's `bitmapProperties.setPropertiesAsync` call. Here's we're just using the same mechanism for encoder options.

As for custom codecs, this simply means that the first argument to `BitmapEncoder.createAsync` (as well as `BitmapDecoder.createAsync`) is the GUID (a class identifier or CLSID) for that codec, the implementation of which must be provided by a DLL. Details on how to write one of these is provided in How to Write a WIC-Enabled Codec. The catch is that including custom image codecs in your package is not presently supported. If the codec is already on the system (that is, installed via the desktop), it will work. However, the Windows Store policies do not allow apps to be dependent on other apps, so it's unlikely that you can even ship such an app unless it's preinstalled on some specific OEM device and the DLL is part of the system image. (An app written in C++ can do more here, but that's beyond the scope of this book.)

In short, for apps written in JavaScript and HTML, you're really limited, for all practical purposes, to image formats that are inherently supported in the system unless you're willing to write your own decoder for file data and do in-place conversions to a supported format.

Do note that these restrictions do *not* exist for custom audio and video codecs. The Media extensions sample shows how to do this with a custom video codec, as we'll see in "Custom Decoders/Encoders and Scheme Handlers" later..

Manipulating Audio and Video

As with images, if all you want to do is load the contents of a `StorageFile` into an audio or video element, you can just pass that `StorageFile` to `URL.createObjectUrl` and assign the result to a `src` attribute. Similarly, if you want to get at the raw data, we can just use the `StorageFile.openAsync` or `openReadAsync` methods to obtain a file stream.

To be honest, opening the file is probably the farthest you'd ever go in JavaScript with raw audio or video, if even that. While chewing on an image is a marginally acceptable process in the JavaScript environment, churning on audio and especially video is really best done in a highly performant C++ DLL. In fact, many third-party, platform-neutral C/C++ libraries for such manipulations are readily available, which you should be able to directly incorporate into such a DLL. In this case you might as well just let the DLL open the file itself!

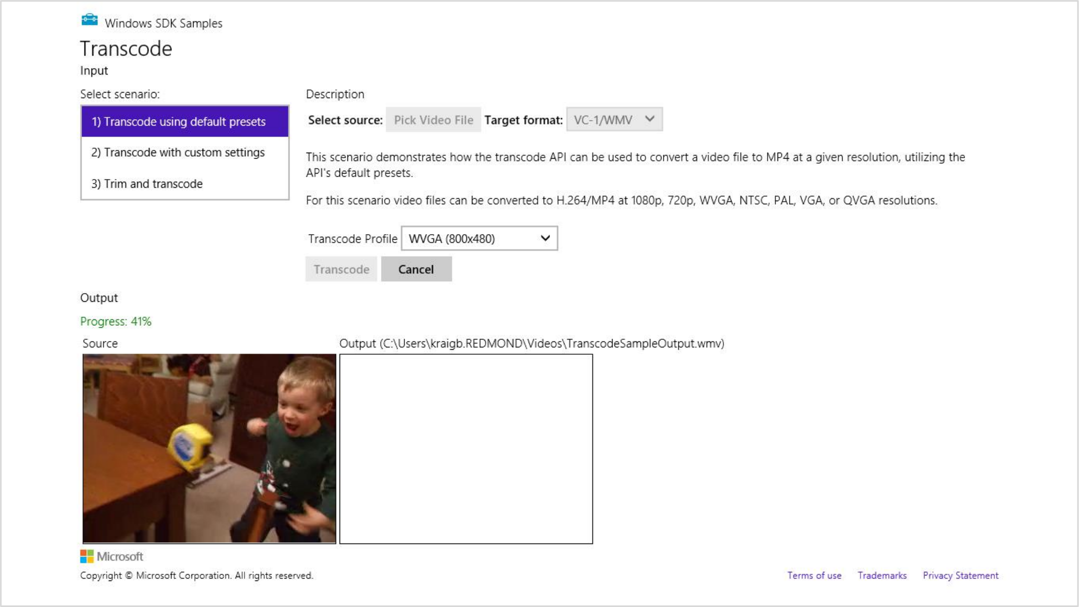
That said, WinRT (which is written in C++!) *does* provide for transcoding (converting) between different media formats, and it provides an extensibility model for custom codecs, effects, and scheme handlers. In fact, we've already seen how to apply custom video effects through the Media extensions sample (see "Applying a Video Effect" earlier in the chapter), and the same DLLs can also be used within an encoding process, where all that the JavaScript code really does is glue the right components together (which it's very good at doing). Let's see how this works with transcoding video first and then with custom codecs.

Transcoding

Transcoding both audio and video is accomplished through the Windows.Media.Transcoding.MediaTranscoder class, which supports output formats of mp3, wav, and wma for audio, and mp4, wmv, avi, and m4a for video. The transcoding process also allows you to apply effects and to trim start and end times.

Transcoding happens either from one StorageFile to another or one RandomAccessStream to another, and in each case it happens according to a MediaEncodingProfile. To set up a transcoding operation, you call the MediaTranscoder prepareFileTranscodeAsync or prepareStreamTranscodeAsync method, which returns back a PrepareTranscodeResult object. This represents the operation that's ready to go, but it won't happen until you call that result's transcodeAsync method. In JavaScript, each result is a promise, allowing you to provide completed and progress handlers for a single operation but also allowing you to combine operations with WinJS.Promise.join. This allows them to be set up and started later, which is useful for batch processing and doing automatic uploads to a service like YouTube while you're sleeping! (And at times like these I've actually pulled ice packs from my freezer and placed them under my laptop as a poor-man's cooling system….)

The Transcoding media sample provides us with a couple of transcoding scenarios. In scenario 1 (js/presets.js) we can pick a video file, pick a target format, select a transcoding profile, and turn the machine loose to do the job (with progress being reported), as shown in Figure 13-6.



Select source: Pick Video File Target format: VC-1/WMV

This scenario demonstrates how the transcode API can be used to convert a video file to MP4 at a given resolution, utilizing the API's default presets.

For this scenario video files can be converted to H.264/MP4 at 1080p, 720p, WVGA, NTSC, PAL, VGA, or QVGA resolutions.

Transcode Profile WVGA (800x480)

Progress: 41%

Output (C:\Users\kraigb.REDMOND\Videos\TranscodeSampleOutput.wmv)

Copyright © Microsoft Corporation. All rights reserved.

FIGURE 13-6 The Transcoding media sample cranking away on a video of my then two-year-old son discovering the joys of a tape measure.

The code that's executed when you press the Transcode button is as follows (js/presets.js, with some bits omitted; this sample happens to use nested promises, which again isn't recommended for proper error handling unless you want, as this code would show, to eat any exceptions that occur pri or to the `transcode-Async` call):

```
function onTranscode() {
    // Create transcode object.
    var transcoder = null;
    transcoder = new Windows.Media.Transcoding.MediaTranscoder();

    // Get transcode profile.
    getPresetProfile(id("profileSelect"));

    // Create output file and transcode.
    var videoLib = Windows.Storage.KnownFolders.videosLibrary;
    var createFileOp = videoLib.createFileAsync(g_outputFileName,
        Windows.Storage.CreationCollisionOption.generateUniqueName);

    createFileOp.done(function (ofile) {
        g_outputFile = ofile;
        g_transcodeOp = null;
        var prepareOp = transcoder.prepareFileTranscodeAsync(g_inputFile, g_outputFile,
            g_profile);

        prepareOp.done(function (result) {
            if (result.canTranscode) {
                g_transcodeOp = result.transcodeAsync();
                g_transcodeOp.done(transcodeComplete, transcoderErrorHandler,
                    transcodeProgress);
            } else {
                transcodeFailure(result.failureReason);
            }
        }); // prepareOp.done
        id("cancel").disabled = false;
    }); // createFileOp.done
}
```

The `getPresetProfile` method retrieves the appropriate profile object according to the option selected in the app. For the selections shown in Figure 13-6 (WMV and WVGA), we'd use these parts of that function:

```
function getPresetProfile(profileSelect) {
    g_profile = null;
    var mediaProperties = Windows.Media.MediaProperties;
    var videoEncodingProfile;

    switch (profileSelect.selectedIndex) {
        // other cases omitted
        case 2:
            videoEncodingProfile = mediaProperties.VideoEncodingQuality.wvga;
            break;
    }
    if (g_useMp4) {
        g_profile = mediaProperties.MediaEncodingProfile.createMp4(videoEncodingProfile);
    } else {
        g_profile = mediaProperties.MediaEncodingProfile.createWmv(videoEncodingProfile);
    }
}
```

In scenario 2, the sample always uses the WVGA encoding but allows you to set specific values for the video dimensions, the frame rate, the audio and video bitrates, audio channels, and audio sampling. It applies these settings in `getCustomProfile` (js/custom.js) simply by configuring the profile properties after the profile is created:

```
function getCustomProfile() {
    if (g_useMp4) {
        g_profile = Windows.Media.MediaProperties.MediaEncodingProfile.createMp4(
            Windows.Media.MediaProperties.VideoEncodingQuality.wvga);
    } else {
        g_profile = Windows.Media.MediaProperties.MediaEncodingProfile.createWmv(
            Windows.Media.MediaProperties.VideoEncodingQuality.wvga);
    }

    // Pull configuration values from the UI controls
    g_profile.audio.bitsPerSample = id("AudioBPS").value;
    g_profile.audio.channelCount = id("AudioCC").value;
    g_profile.audio.bitrate = id("AudioBR").value;
    g_profile.audio.sampleRate = id("AudioSR").value;
    g_profile.video.width = id("VideoW").value;
    g_profile.video.height = id("VideoH").value;
    g_profile.video.bitrate = id("VideoBR").value;
    g_profile.video.frameRate.numerator = id("VideoFR").value;
    g_profile.video.frameRate.denominator = 1;
}
```

And to finish off, scenario 3 is like scenario 1, but it lets you set start and end times that are then saved in the transcoder's `trimStartTime` and `trimStopTime` properties (see js/trim.js):

```
transcoder = new Windows.Media.Transcoding.MediaTranscoder();
transcoder.trimStartTime = g_start;
transcoder.trimStopTime = g_stop;
```

Though not shown in the sample, you can apply effects to a transcoding operation by using the transcoder's `addAudioEffect` and `addVideoEffect` methods.

Handling Custom Audio and Video Formats

Although Windows supports a variety of audio and video formats in-box, there are clearly many more formats that your app might want to work with. You can do this a couple of ways. First, you can use custom bytestream objects, media sources, codecs, and effects. Second, you can obtain what's called a media stream source with which you can do in-place decoding. We'll look at both in this section.

Custom Decoders/Encoders and Scheme Handlers

To support custom audio and video formats beyond those that Windows supports in-box, WinRT provides some extensibility mechanisms. I should warn you up front that this subject will take you into some pretty vast territory around the entire Windows Media Foundation (WMF) SDK. What's in WinRT is just a wrapper, so knowledge of WMF is essential and not for the faint of heart! It's also important to note that all such extensions are available to *only* the app itself and are not available to other apps on the system. Furthermore, Windows will always prefer in-box components over a custom one, which means don't bother wasting your time creating a new mp3 decoder or such since it will never actually be used. Your primary reference here is the Media extensions sample, which demonstrates an MPEG1 decoder along with grayscale, invert, and polar transform (pinch, warp, and fisheye) effects. Each of these is implemented as a C++ extension DLL. To enable these, the app must declare them in its manifest as follows (note that the manifest editor in Visual Studio does not surface these parts of the manifest, so you have to edit the XML directly):

```
<Extension Category="windows.activatableClass.inProcessServer">
   <InProcessServer>
      <Path>MPEG1Decoder.dll</Path>
      <ActivatableClass ActivatableClassId="MPEG1Decoder.MPEG1Decoder"
         ThreadingModel="both" />
   </InProcessServer>
</Extension>
```

The `ActivatableClassId` is how an extension is identified when calling the WinRT APIs, which is clearly mapped in the manifest to the specific DLL that needs to be loaded.

Depending, then, on the use of the extension, you might need to register it with WinRT through the methods of `Windows.Media.MediaExtensionManager`: `registerAudio[Decoder | Encoder]`, `registerByteStreamHandler` (media sinks), `registerSchemeHandler` (media sources/file containers), and `registerVideo[Decoder | Encoder]`. In scenario 1 of the Media extensions sample (js/LocalDecoder.js), we can see how to set up a custom decoder for video playback:

```
var page = WinJS.UI.Pages.define("/html/LocalDecoder.html", {
   extensions: null,
   MFVideoFormat_MPG1: { value: "{3147504d-0000-0010-8000-00aa00389b71}" },
   NULL_GUID: { value: "{00000000-0000-0000-0000-000000000000}" },

   ready: function (element, options) {
      if (!this.extensions) {
         // Add any initialization code here
         this.extensions = new Windows.Media.MediaExtensionManager();
         // Register custom ByteStreamHandler and custom decoder.
         this.extensions.registerByteStreamHandler("MPEG1Source.MPEG1ByteStreamHandler",
            ".mpg", null);
         this.extensions.registerVideoDecoder("MPEG1Decoder.MPEG1Decoder",
            this.MFVideoFormat_MPG1, this.NULL_GUID);
      }

   // ...
```

where the *MPEG1Source.MPEG1ByteStreamHandler* CLSID (class identifier) is implemented in one DLL (see the MPEG1Source C++ project in the sample's solution) and the *MPEG1Decoder.MPEG1.Decoder* CLSID is implemented in another (the MPEG1Decoder C++ project).

Scenario 2, for its part, shows the use of a custom scheme handler, where the handler (in the GeometricSource C++ project) generates video frames on the fly. Fascinating stuff, but beyond the scope of this book.

Effects, as we've seen, are quite simple to use once you have one implemented: just pass their CLSID to methods like msInsertVideoEffect and msInsertAudioEffect on video and audio elements. You can also apply effects during the transcoding process in the MediaTranscoder class's addAudioEffect and addVideoEffect methods. The same is also true for media capture, as we'll see shortly.

Media Stream Sources

If you're going to do heavy-duty encoding and decoding of a custom audio or video format, it's a good investment in the long time to create a custom media extension as outlined in the previous section. For other scenarios, however, it's sheer overkill. For all these reasons that is another way to handle custom formats: the Windows.Media.Core.MediaStreamSource API. This basically allows an app to inject its own code into the media pipeline that's used in playback, transcoding, and streaming alike. It operates like this:

- Create an instance of MediaStreamSource using a descriptor for the stream, and then set other desired properties. This object is what delivers samples into the media pipeline.
- Add listeners to the MediaStreamSource object's samplerequested, starting, and closed events. These are WinRT events to be sure to call removeEventListener when the object goes out of scope. You can also listen to the paused event if you're need to manage a read-ahead buffer.
- Pass the MediaStreamSource to URL.createObjectURL and assign the return to the media element's src attribute, after which you can start playback for audio and video.

The important work then occurs within your three events handlers:

- The starting event is fired when rendering begins. Here you open the source stream (e.g., open a file to obtain a RandomAccessStream) and adjust the starting position if needed.
- When the media engine is ready to render a frame or sample, it fires a samplerequested event. Here you read and parse data from the stream, manipulate it however you want, and return the sample. You can also decompress or decrypt your source's data, combine data from multiple sources, or even generate samples on the fly. In short, this is where you provide the raw bytes into the next stage of the pipeline, and you're in complete control.
- When rendering is complete, the closed event fires. Here you close whatever stream you opened in starting. This is also a place where you can remove event listeners.

The MediaStreamSource streaming sample demonstrates this process for an MP3 file that you select using the file picker, handling playback through a simple audio element (html/S1_StreamMP3.html):

```
<audio id="mediaPlayer" width="610" controls="controls" ></audio>
```

The selected file from the picker ends up in a variable called inputMP3File, after which the sample calls its function initializeMediaStreamSource in js/S1_StreamMP3.js. Let's follow though that code, which starts by retrieving various properties from the file:

```
function initializeMediaStreamSource() {
   byteOffset = 0;
   timeOffset = 0;

   // get the MP3 file properties

   getMP3FileProperties().then(function () {
      return getMP3EncodingProperties();
   }).then(function () {
```

The getMP3FileProperties function initializes variables called title and songDuration using the StorageFile.properties.getMusicProperties async method; the getMP3EncodingProperties function initialized variables called sampleRate, channelCount, and bitrate using StorageFile.properties.retrievePropertiesAsync for the associated *System.Audio** properties.

Note The original sample doesn't properly chain the get* functions as shown above and available in the modified sample in the companion content. Without this correction, the sample can crash if it attempts to continue executing the code before variables like songDuration have been initialized.

With these properties in hand, we can now create the descriptor for the stream:

```
var audioProps = Windows.Media.MediaProperties.AudioEncodingProperties.createMp3(
   sampleRate, channelCount, bitrate);

var audioDescriptor = new Windows.Media.Core.AudioStreamDescriptor(audioProps);
```

Then we can create the MediaStreamSource with that descriptor and set some of its properties like canSeek, duration, and musicProperties (or videoProperties and thumbnail for video):

```
MSS = new Windows.Media.Core.MediaStreamSource(audioDescriptor);
MSS.canSeek = true;
MSS.musicProperties.title = title;
MSS.duration = songDuration;
```

Now we're ready to attach event listeners, assign the stream to the media element, and start playback:

```
MSS.addEventListener("samplerequested", sampleRequestedHandler, false);
MSS.addEventListener("starting", startingHandler, false);
MSS.addEventListener("closed", closedHandler, false);

mediaPlayer.src = URL.createObjectURL(MSS, { oneTimeOnly: true });
```

```
        mediaPlayer.play();
    });
}
```

When play is called, Windows needs to start rendering the audio, so it first fires the starting event, where eventArgs.request is a MediaStreamSourceStartingRequest object containing a startPosition property, a setActualStartPosition method (so that you can make adjustments), and a getDeferral method. The deferral is necessary—as we've seen in other cases—because playback begins as soon as your event handler returns. The deferral allows you to do async work within the handler, as shown in the sample (js/S1_StreamMP3.js):

```
function startingHandler(e) {
    var request = e.request;

    if ((request.startPosition !== null) && request.startPosition <= MSS.duration) {
        var sampleOffset = Math.floor(request.startPosition / sampleDuration);
        timeOffset = sampleOffset * sampleDuration;
        byteOffset = sampleOffset * sampleSize;
    }

    // Create the RandomAccessStream for the input file for the first time
    if (mssStream === undefined){
        var deferral = request.getDeferral();
        try{
            inputMP3File.openAsync(Windows.Storage.FileAccessMode.read).then(function(stream) {
                mssStream = stream;
                request.setActualStartPosition(timeOffset);
                deferral.complete();
            });
        }
        catch (exception){
            MSS.notifyError(Windows.Media.Core.MediaStreamSourceErrorStatus.failedToOpenFile);
            deferral.complete();
        }
    }
    else {
        request.setActualStartPosition(timeOffset);
    }
}
```

As you can see, the sample simply opens the inputMP3File file to obtain a stream, which it stores in mssStream. At the top, it also sets timeOffset and byteOffset variables that are used within the samplerequested handler. When opening a new file, these end up being zero. Note also the use of MediaStreamSource.notifyError to communicate error conditions to the media pipeline.

You can also see that this event handler works when the stream has already been initialized. This happens if you start playback in the UI and then pause it, which will raise a paused event (not handled in the sample). When you resume playback, the starting event will fire again, but in that case we just need to update the timeOffset instead of opening the stream again.

Once the starting event handler returns, the samplerequested handler starts getting called. Its eventArgs.request is a MediaStreamSourceSampleRequest object that contains the streamDescriptor related to the request, a getDeferral method for the usual purposes, a reportSampleProgress method to call when the sample can't be delivered right away, and a sample property in which the sample data is stored before the handler returns (js/S1_StreamMP3.js):

```
function sampleRequestedHandler(e){
    var request = e.request;

    // Check if the sample requested byte offset is within the file size
    if (byteOffset + sampleSize <= mssStream.size)
    {
        var deferral = request.getDeferral();
        var inputStream = mssStream.getInputStreamAt(byteOffset);

        // Create the MediaStreamSample and assign to the request object.
        // You could also create the MediaStreamSample using createFromBuffer(...)
        Windows.Media.Core.MediaStreamSample.createFromStreamAsync(
            inputStream, sampleSize, timeOffset).then(function(sample) {
                sample.duration = sampleDuration;
                sample.keyFrame = true;

                // Increment the time and byte offset
                byteOffset = byteOffset + sampleSize;
                timeOffset = timeOffset + sampleDuration;
                request.sample = sample;
                deferral.complete();
            });
    }
}
```

The sample itself is represented by a MediaStreamSample object, which can be created in two ways. One is through the static MediaStreamSample.createFromStreamAsync method, as shown in the code above, where you indicate what portion of the stream to use. The other is the static method MediaStreamSample.createFromBuffer, where the buffer can contain any data that you've generated dynamically or data that you've read from your original input stream and then manipulated. See the sidebar on the next page.

Once created, be sure to set the duration property of the sample so that the rest of the media pipeline knows how much of the playback stream it has to work with. In addition, you can set these other properties:

- keyFrame If true, indicates that the sample can be independently decoded from other samples.
- discontinuous If true, indicates that the previous sample in the sequence was missing, such as when you drop video frames or audio samples because of network latency.
- protection A MediaStreamSampleProtectionProperties object for handling digital rights management. Refer to the MediaStreamSource.mediaProtectionManager property and

addProtectionKey method. We'll talk a bit more of DRM later in this chapter under "Streaming Media and Play To."

- decodeTimestamp By default, this is the same as the sample's read-only timestamp property, but some formats might require a different value for decoding, in which case you use this to override that default.

In any case, when you've created the necessary sample, assign it to the request.sample property and then return. If you want to know when the media pipeline has finished with the sample, you can listen to the MediaStreamSample.processed event. This is most useful to know when you can reuse its buffer.

If you've reached the end of the stream, on the other hand, simply return from the event handler without setting request.sample, which will signal the media pipeline that playback has finished. At that point the closed event will be fired, in which you do your cleanup (js/S1_StreamMP3.js):

```
function closedHandler(e){
    if (mssStream){
        mssStream.close();
        mssStream = undefined;
    }

    e.target.removeEventListener("starting", startingHandler, false);
    e.target.removeEventListener("samplerequested", sampleRequestedHandler, false);
    e.target.removeEventListener("closed", closedHandler, false);

    if (e.target === MSS) {
        MSS = null;
    }
}
```

Earlier I briefly mentioned the stream source's paused event. This again indicates that playback has been paused through the media element's UI. This is helpful if you're managing a read-ahead buffer in your stream (such as when you're downloading from a network). If you are, you also need to call the MediaStreamSource.setBufferedRange method to tell the media pipeline how much data you have. Windows uses this to determine when you've buffered the whole stream (from offset 0 to duration), which means it can enter a lower power mode.

By default as well, the media pipeline will be making sample requests three seconds ahead of the playback offset. This can be changed through the MediaStreamSource.bufferTime property (expressed in milliseconds, so the default value is 3000).

The other feature that I alluded to earlier is that you can have a stream source read from multiple streams simultaneously. For example, you might want to play an alternate audio track along with a video, in which case you would have separate descriptors for each and would use the second MediaStreamSource constructor that takes two descriptors instead of one. If you have more than two descriptors, you can also call the source's addStreamDescriptor after it's been created, but this has to be done before playback or any other operation has begun.

When dealing with multiple streams, you must clearly pay attention to the request.streamDescriptor property in the samplerequested event so that you return the correct sample! The MediaStreamSource also fires a switchStreamsRequested event to tell you that it's switching streams if you have other work you need to do in preparation for sample, but this is wholly optional.

Sidebar: Generating Sine Wave Audio

As an example of generating a dynamic stream, I've added a second scenario to the modified MediaStreamSource streaming sample in the companion content. For this I create a buffer ahead of time that contains data for a 110Hz sine wave, and then I set up the MediaStreamSource with an 8-bit PCM audio descriptor (PCM is the basis for the WAV file format). The samplerequested handler then uses MediaStreamSample.createFromBuffer to generate the audio sample from the buffer (special thanks to Anders Klemets for the code):

```
//Include approximately 100 ms of audio data in the buffer
var cyclesPerBuffer = Math.floor(sampleRate / 10 / samplesPerCycle);
var samplesPerBuffer = samplesPerCycle * cyclesPerBuffer;

var sampleLength = Math.floor(samplesPerBuffer * 1000 / sampleRate);


//sineBuffer if populated with 110Hz sine wave data
//sampleLength is the duration of the data in the buffer, in ms
//timeOffset is initially set to 0

function sampleRequestedHandler(e) {
   var sample = Windows.Media.Core.MediaStreamSample.createFromBuffer(
      sineBuffer, timeOffset);
   sample.duration = sampleLength;
   timeOffset = (timeOffset + sample.duration);
   e.request.sample = sample;
}
```

Media Capture

There are times when we can really appreciate the work that people have done to protect individual privacy, such as making sure I know when my computer's camera is being used since I am often using it in the late evening, sitting in bed, or in the early pre-shower mornings when I have, in the words of my father-in-law, "pineapple head" or I bear a striking resemblance to the character of Heat Miser in *The Year Without a Santa Claus*.

And there are times when we want to turn on a camera or a microphone and record something: a picture, a video, or audio. Of course, an app cannot know ahead of time what exact camera and microphones might be on a system. A key step in capturing media, then, is determining which device to use—something that the Windows.Media.Capture APIs provide for nicely, along with the process of doing the capture itself into a file, a stream, or some other custom "sink," depending on how an app wants to manipulate or process the capture.

Back in Chapter 2, "Quickstart," we learned how to use WinRT to easily capture a photograph in the Here My Am! app. To quickly review, we only needed to declare the *Webcam* capability in the manifest and add a few lines of code (this is from HereMyAm2a, the first version of the app):

```
function capturePhoto() {
    var captureUI = new Windows.Media.Capture.CameraCaptureUI();
    var that = this;

    captureUI.photoSettings.format = Windows.Media.Capture.CameraCaptureUIPhotoFormat.png;
    captureUI.photoSettings.croppedSizeInPixels =
        { width: that.clientWidth, height: that.clientHeight };

    captureUI.captureFileAsync(Windows.Media.Capture.CameraCaptureUIMode.photo)
        .done(function (capturedFile) {
            //Be sure to check validity of the item returned; could be null
            //if the user canceled.
            if (capturedFile) {
                lastCapture = capturedFile;  //Save for Share
                that.src = URL.createObjectURL(capturedFile, {oneTimeOnly: true});
            }
        }, function (error) {
            console.log("Unable to invoke capture UI: " + error.message);
        });
}
```

The UI that Windows brings up through this API provides for cropping, retakes, and adjusting camera settings. Another example of taking a photo can also be found in scenario 1 of the [CameraCaptureUI Sample](), along with an example of capturing video in scenario 2. In this latter case (js/capturevideo.js) we configure the capture UI object for a video format and indicate a video mode in the call to `captureFileAsync`. The resulting `StorageFile` can be passed straight along to a `video.src` property through our good friend `URL.createObjectURL`:

```
function captureVideo() {
    var dialog = new Windows.Media.Capture.CameraCaptureUI();
    dialog.videoSettings.format = Windows.Media.Capture.CameraCaptureUIVideoFormat.mp4;

    dialog.captureFileAsync(Windows.Media.Capture.CameraCaptureUIMode.video)
        .done(function (file) {
            if (file) {
                var videoBlobUrl = URL.createObjectURL(file, {oneTimeOnly: true});
                document.getElementById("capturedVideo").src = videoBlobUrl;
            } else {
                //...
            }
        });
}
```

It should be noted that the *Webcam* capability in the manifest applies only to the image or video side of camera capture. If you want to capture audio, be sure to also select the *Microphone* capability on the Capabilities tab of the manifest editor.

If you look in the `Windows.Media.Capture.CameraCaptureUI` object, you'll also see many other options you can configure. Its `photoSettings` property, a `CameraCaptureUIPhotoCaptureSettings` object, lets you indicate cropping size and aspect ratio, format, and maximum resolution. Its `videoSettings` property, a `CameraCaptureUIVideoCaptureSettings` object, lets you set the format, set the maximum duration and resolution, and indicate whether the UI should allow for trimming. All useful stuff! You can find discussions of some of these in the docs on Capturing or rendering audio, video, and images, including coverage of managing calls on a Bluetooth device.

Flexible Capture with the MediaCapture Object

Of course, the default capture UI won't necessarily suffice in every use case. For one, it always sends output to a file, but if you're writing a communications app, for example, you'd rather send captured video to a stream or send it over a network without any files involved at all. You might also want to preview a video before any capture actually happens, or simply show capture UI in place rather than in using the built-in full-screen overlay. Furthermore, you may want to add effects during the capture, apply rotation, and perhaps apply a custom encoding.[100]

All of these capabilities are available through the `Windows.Media.Capture.MediaCapture` class:

Properties	**Description (classes are in the Windows.Media.Capture namespace unless noted)**
`audioDeviceController`	An `AudioDeviceController` that controls volume and provides the ability to manage other arbitrary properties that affect the audio stream.
`mediaCaptureSettings`	A `MediaCaptureSettings` that contains device IDs and mode settings, and lets you set the source (audio, videoPreview, photo).
`videoDeviceController`	A `VideoDeviceController` that controls picture properties (brightness, hue, pan/tilt, zoom, etc.), provides adjustments for backlight and AC power frequency, and provides the ability to manage other arbitrary properties that affect the video stream.
Events	**Description**
`failed`	Fired when an error occurs during capture.
`recordLimitationExceeded`	Fired when the user tried to record video or audio past the allowable duration.
Methods[101]	**Description**
`initializeAsync`	Initialize the `MediaCapture` object (with defaults or with a `MediaCaptureInitialization-Settings` object that contains the same stuff as `MediaCaptureSettings`).
`addEffectAsync`	Applies an effect.
`clearEffectsAsync`	Clears all current effects.

[100] On this subject you might be interested in Dave Rousset's blog series, Using WinJS & WinRT to build a fun HTML5 Camera Application for Windows.

[101] Note that there are a few additional methods in the documentation that are not projected into JavaScript and thus aren't shown here, such as `startPreviewToCustomSinkAsync`. In JavaScript, you can just pass a preview to `URL.createObjectURL`, assign the result to `video.src`, and then call `video.play()` to preview.

capturePhotoToStorageFileAsync capturePhotoToStreamAsync	Captures an image to a storage file or a random access stream. Both take an instance of ImageEncodingProperties to control format (JPEG or PNG), type, dimensions, and other arbitrary [Windows Properties](), as described in Chapter 11.
getEncoderProperty setEncoderProperty setEncodingPropertiesAsync	Manages specific encoder properties.
startRecordToStorageFileAsync startRecordToStreamAsync stopRecordAsync	Starts and stops recording to a storage file or random access stream, a MediaEncodingProfile that determines the audio/video format, along with bitrate, quality, video dimensions, etc.
getRecordRotation setRecordRotation	For videos, these manage a VideoRotation value (90-degree increments) to apply to the recording. These do not affect audio.
startRecordToCustomSinkAsync	Starts recording into a custom sink that's described either by an implementation of Windows.Media.IMediaExtension or by an ID plus a property set of settings.
startPreviewAsync stopPreviewAsync getPreviewRotation setPreviewRotation	Same as recording but works for previews. In this case, if you call URL.createObjectURL and pass the MediaCapture object as the first parameter, the result can be assigned to the src attribute of a video element and the preview shows in that element when you call the video.play method.
getPreviewMirroring setPreviewMirroring	Controls preview mirroring, which means to flip the preview horizontally; this accounts for differences in camera direction, which can be in the same direction as the user (rear-mounted camera as on a tablet computer) or the opposite direction (camera mounted on a monitor or built into a laptop display). See the next section, "Selecting a Media Capture Device."
prepareLowLagPhotoCaptureAsync prepareLowLagPhotoSequeceCaptureAsync prepareLowLagRecordToStorageFileAsync prepareLowLagRecordToStreamAsync	Works with capture devices that are have low lag shutters, meaning that they can capture a series of images in a short period of time.

For a very simple demonstration of previewing video in a video element, we can look at the [CameraOptionsUI sample]() in js/showoptionsui.js. When you tap the Start Preview button, it creates and initializes a MediaCapture object as follows:

```
function initializeMediaCapture() {
    mediaCaptureMgr = new Windows.Media.Capture.MediaCapture();
    mediaCaptureMgr.initializeAsync().done(initializeComplete, initializeError);
}
```

where the initializeComplete handler calls into startPreview:

```
function startPreview() {
    document.getElementById("previewTag").src = URL.createObjectURL(mediaCaptureMgr);
    document.getElementById("previewTag").play();
    startPreviewButton.disabled = true;
    showSettingsButton.style.visibility = "visible";
    previewStarted = true;
}
```

The other little bit shown in this sample is invoking the Windows.Media.Capture.CameraOptionsUI, which happens when you tap its Show Settings button; see Figure 13-7. This is just a system-provided flyout with options that are relevant to the current media stream being captured:

```
function showSettings() {
    if (mediaCaptureMgr) {
        Windows.Media.Capture.CameraOptionsUI.show(mediaCaptureMgr);
    }
}
```

By the way, if you have trouble running a sample like this in the Visual Studio simulator—specifically, you see exceptions when trying to turn on the camera—try running on the local machine or a remote machine instead.

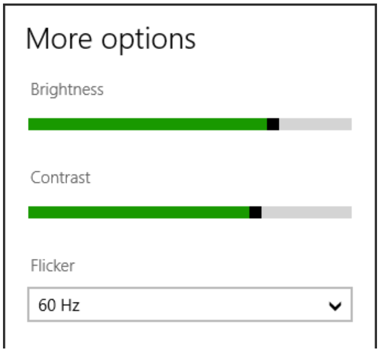


FIGURE 13-7 The Camera Options UI, as shown in the CameraOptionsUI sample (empty bottom is cropped).

More complex scenarios involving the MediaCapture class (and a few others) can be found now in the [Media capture using capture device sample](), such as previewing and capturing video, changing properties dynamically (scenario 1), selecting a specific media device (scenario 2), recording just audio (scenario 3), and capturing a photo sequence (scenario 4).

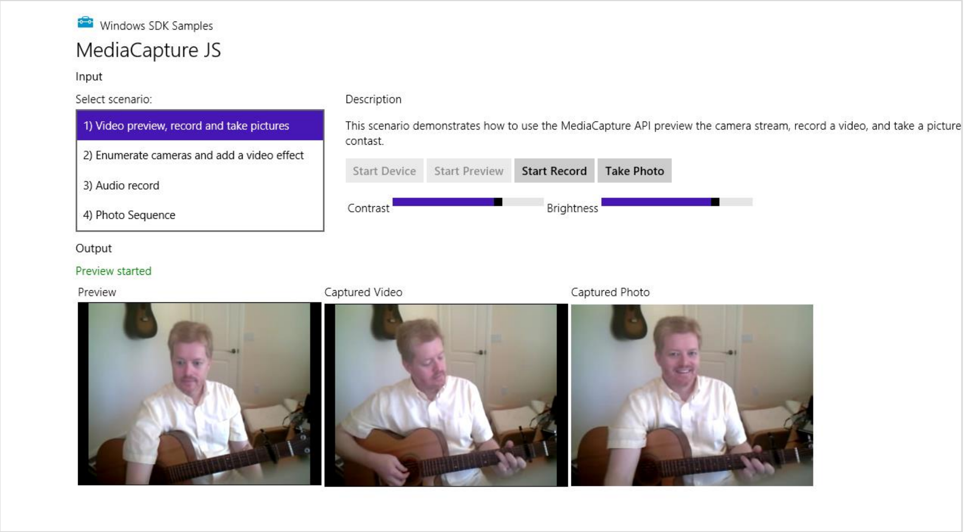
Starting with scenario 3 (js/AudioCapture.js, the simplest), here's the core code to create and initialize the MediaCapture object for an audio stream (see the streamingCaptureMode property in the initialization settings), where that stream is directed to a file in the music library via startRecordToStorageFileAsync (some code omitted and condensed for brevity):

```
var mediaCaptureMgr = null;
var captureInitSettings = null;
var encodingProfile = null;
var storageFile = null;

// This is called when the page is loaded
function initCaptureSettings() {
    captureInitSettings = new Windows.Media.Capture.MediaCaptureInitializationSettings();
    captureInitSettings.audioDeviceId = "";
    captureInitSettings.videoDeviceId = "";
    captureInitSettings.streamingCaptureMode =
        Windows.Media.Capture.StreamingCaptureMode.audio;
}
```

```
function startAudioCapture() {
    mediaCaptureMgr = new Windows.Media.Capture.MediaCapture();

    mediaCaptureMgr.initializeAsync(captureInitSettings).done(function (result) {
        // ...
    });
}

function startRecord() {
    // ...
    // Start recording.
    Windows.Storage.KnownFolders.videosLibrary.createFileAsync("cameraCapture.m4a",
        Windows.Storage.CreationCollisionOption.generateUniqueName)
        .done(function (newFile) {
            storageFile = newFile;
            encodingProfile = Windows.Media.MediaProperties
                .MediaEncodingProfile.createM4a(Windows.Media.MediaProperties
                .AudioEncodingQuality.auto);
            mediaCaptureMgr.startRecordToStorageFileAsync(encodingProfile,
                storageFile).done(function (result) {
                    // ...
                });
        });
}

function stopRecord() {
    mediaCaptureMgr.stopRecordAsync().done(function (result) {
        displayStatus("Record Stopped.  File " + storageFile.path + "  ");

        // Playback the recorded audio (using a video element)
        var video = id("capturePlayback" + scenarioId);
        video.src = URL.createObjectURL(storageFile, { oneTimeOnly: true });
        video.play();
    });
}
```

Scenario 1 is essentially the same code but captures a video stream as well as photos, with results shown in Figure 13-8. This variation is enabled through these properties in the initialization settings (see js/BasicCapture.js within `initCaptureSettings`):

```
captureInitSettings.photoCaptureSource =
    Windows.Media.Capture.PhotoCaptureSource.auto;
captureInitSettings.streamingCaptureMode =
    Windows.Media.Capture.StreamingCaptureMode.audioAndVideo;
```



Windows SDK Samples

MediaCapture JS

Input

Select scenario:

1) Video preview, record and take pictures

2) Enumerate cameras and add a video effect

3) Audio record

4) Photo Sequence

Description

This scenario demonstrates how to use the MediaCapture API preview the camera stream, record a video, and take a picture contast.

Output

Preview started

FIGURE 13-8 Previewing and recording video with the default device in the Media capture sample, scenario 1. (The output is cropped because I needed to run the app using the Local Machine option in Visual Studio and I didn't think you needed to see a 1920x1200 screen shot with lots of whitespace!)

Notice the Contrast and Brightness controls in Figure 13-8. Changing these will change the preview video, along with the recorded video. The sample does this through the `MediaCapture.videoDeviceController` object's `contrast` and `brightness` properties, showing that these (and any others in the controller) can be adjusted dynamically. Refer to the `getCameraSettings` function in js/BasicCapture.js that basically wires the slider `change` events into a generic anonymous function to update the desired property.

Selecting a Media Capture Device

Looking now at scenario 2 (js/AdvancedCapture.js), it's more or less like scenario 1 but it allows you to select the specific input device. Until now, everything we've done has simply used the default device, but you're not limited to that, of course. This is a good thing because my current laptop's default camera is usually covered when the machine is in its docking station—without the ability to choose another camera, I'd just get a bunch of black images!

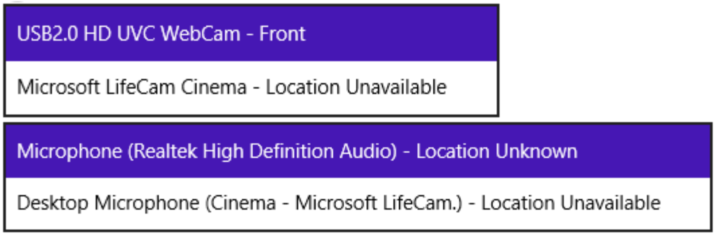
You use the [`Windows.Devices.Enumeration`] API to retrieve a list of devices within a particular device interface class, namely the [`DeviceInformation.findAllAsync`] method. To this you give it a value from the `DeviceClass` enumeration or another capture selector. The sample, for instance, uses `DeviceClass.videoCapture` to enumerate cameras (js/AdvancedCapture.js):

```
function enumerateCameras() {
    var cameraSelect = id("cameraSelect");
    cameraList = new Array();

    // Enumerate cameras and add them to the list
    var deviceInfo = Windows.Devices.Enumeration.DeviceInformation;
    deviceInfo.findAllAsync(Windows.Devices.Enumeration.DeviceClass.videoCapture)
        .done(function (cameras) {
            // ...
        });
    // ...
}
```

and a selector for microphones:

```
function enumerateMicrophones() {
    var microphoneSelect = id("microphoneSelect");
    var microphoneDeviceId = 0;
    microphoneList = new Array();

    // Enumerate microphones and add them to the list
    var microphoneDeviceInfo = Windows.Devices.Enumeration.DeviceInformation;
    microphoneDeviceInfo.findAllAsync(
        Windows.Media.Devices.MediaDevice.getAudioCaptureSelector(), null)
        .done(function (deviceInformation) {
            // ...
        }
    // ...
}
```

In both cases the result of the enumeration (`DeviceInformation.findAllAsync`) is a `DeviceInformationCollection` object that contains some number of `DeviceInformation` objects. These collections are used to populate the drop-down lists, as shown below. Here you can see that I have two cameras (one in my laptop lid and the other hanging off my secondary monitor) and two microphones (on the same devices):



USB2.0 HD UVC WebCam - Front

Microsoft LifeCam Cinema - Location Unavailable

Microphone (Realtek High Definition Audio) - Location Unknown

Desktop Microphone (Cinema - Microsoft LifeCam.) - Location Unavailable

The selected device's ID is then copied within to the `videoDeviceId` and audioDeviceId properties of the `MediaCaptureInitializationSetting` object:

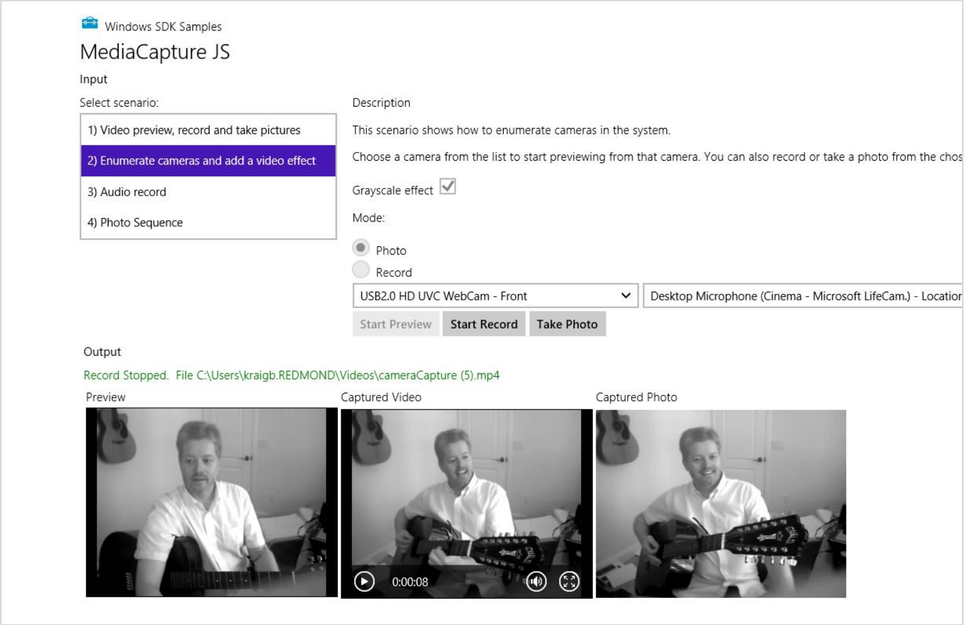
```
captureInitSettings = new Windows.Media.Capture.MediaCaptureInitializationSettings();

var selectedIndex = id("cameraSelect").selectedIndex;
var deviceInfo = deviceList[selectedIndex];
captureInitSettings.videoDeviceId = deviceInfo.id;

var selectedIndex = id("microphoneSelect").selectedIndex;
var microphoneDeviceInfo = microphoneList[selectedIndex];
captureInitSettings.audioDeviceId = microphoneDeviceInfo.id;
```

By the way, you can retrieve the default device ID at any time through the methods of the Windows.Media.Devices.MediaDevice object and listen to its events for changes in the default devices. It's also important to note that DeviceInformation (in the deviceInfo variable above) includes a property called enclosureLocation:

```
if (deviceInfo.enclosureLocation) {
    cameraLocation = deviceInfo.enclosureLocation.panel;
}
```

The enclosureLocation property is an EnclosureLocation object that has inDock, inLid, and panel properties; the latter is a value from Windows.Devices.Enumeration.Panel, whose values are front, back, top, bottom, left, right, and unknown. This tells you whether a camera is forward or backward facing, which you can use to rotate the video or photo as appropriate for the user's perspective (also taking the device orientation into account).

The other bit that scenario 2 demonstrates is using the MediaCapture.addEffectAsync with a grayscale effect, shown in Figure 13-9, that's implemented in a C++ DLL (the GrayscaleTransform project in the sample's solution). This works exactly as it did with transcoding, and you can refer to the addRemoveEffect and addEffectToImageStream functions in js/AdvancedCapture.js for the details. You'll notice there that these functions do a number of checks using the MediaCaptureSettings.videoDeviceCharacteristic value to make sure that the effect is added in the right place.



Windows SDK Samples

MediaCapture JS

Input

Select scenario:

1) Video preview, record and take pictures
2) Enumerate cameras and add a video effect
3) Audio record
4) Photo Sequence

Description

This scenario shows how to enumerate cameras in the system.

Choose a camera from the list to start previewing from that camera. You can also record or take a photo from the chos

Grayscale effect ☑

Mode:

● Photo
○ Record

USB2.0 HD UVC WebCam - Front

Desktop Microphone (Cinema - Microsoft LifeCam.) - Locatior

Output

Record Stopped. File C:\Users\kraigb.REDMOND\Videos\cameraCapture (5).mp4

FIGURE 13-9 Scenario 2 of the Media capture sample, in which one can select a specific device and apply an effect. (The output here is again cropped from a larger screen shot.) Were you also paying attention enough to notice that I switched guitars?

The last piece of the sample, scenario 4, demonstrates capture features related to low shutter lag–capable cameras. Some cameras, that is, have a long lag between each captured image. If a camera is low-lag–capable, this enables a capture mode called a *photo sequence*, which takes a continuous series of photos in a short period of time. This is configured through the `MediaCapture.videoDeviceController.lowLagPhotoSequence` property, a [`LowLagPhotoSequenceControl`]() object whose `supported` property tells you if your hardware is suitably capable of this feature. But I'll let you look at scenario 4 for all the details involving this and methods like `MediaCapture.prepareLowLagPhotoCaptureAsync`.

Streaming Media and Play To

To say that streaming media is popular is certainly a gross understatement. As mentioned in this chapter's introduction, Netflix alone consumes a large percentage of today's Internet bandwidth (including that of my own home). YouTube, Amazon Instant Video, and Hulu certainly do their part as well—so your app might as well contribute to the cause!

Streaming media from a server to your app is easily the most common case, and it happens automatically when you set an audio or video `src` attribute to a remote URI. To improve on this, Microsoft also has a Smooth Streaming Client SDK that helps you build media apps with a number of rich features, including live playback and PlayReady content protection. I won't be covering that SDK in this book, so I wanted to make sure you were aware of it along with the tutorial, Building Your First HTML5 Smooth Streaming Player.

What we'll focus on here, in the few pages we have left before my editors at Microsoft Press pull the plug on this chapter, are considerations for digital rights management and streaming not from a network but *to* a network (for example, audio/video capture in a communications app), as well as streaming media from an app to a Play To device.

Streaming from a Server and Digital Rights Management

Again, streaming media from a server is what you already do whenever you're using an `audio` or `video` element with a remote URI. The details just happen for you. Indeed, much of what a great media client app does is talking to web services, retrieving metadata and the catalog, helping the user navigate all of that information, and ultimately getting to a URI that can be dropped in the `src` attribute of a `video` or `audio` element. Then, once the app receives the `canplay` event, you can call the element's `play` method to get everything going.

Of course, media is often protected with DRM, otherwise the content on paid services wouldn't be generating much income for the owners of those rights! So there needs to be a mechanism to acquire and verify rights somewhere between setting the element's `src` and receiving `canplay`. Fortunately, there's a simple means to do exactly that:

8. Before setting the src attribute, create an instance of Windows.Media.Protection.MediaProtectionManager and configure its properties.

9. Listen to this object's serviceRequested event, the handler for which performs the appropriate rights checks and sets a completed flag when all is well. (Two other events, just to mention them, are componentloadfailed and rebootneeded.)

10. Assign the protection manager to the audio/video element with the msSetMediaProtectionManager extension method.

11. Set the src attribute. This will trigger the serviceRequested event to start the DRM process, which will prevent canplay until DRM checks are completed successfully.

12. In the event of an error, the media element's error event will be fired. The element's error property will then contain an msExtendedCode with more details.

You can refer to How to use pluggable DRM and How to handle DRM errors for additional details, but here's a minimal and hypothetical example of all this in code:

```
var video1 = document.getElementById("video1");

video1.addEventListener('error', function () {
    var error = video1.error.msExtendedCode;
    //...
}, false);

video1.addEventListener('canplay', function () {
    video1.play();
}, false);

var cpm = new Windows.Media.Protection.MediaProtectionManager();
cpm.addEventListener('servicerequested', enableContent, false);  //Remove this later
video1.msSetContentProtectionManager(cpm);
video1.src = "http://some.content.server.url/protected.wmv";

function enableContent(e) {
    if (typeof (e.request) != 'undefined') {
        var req = e.request;
        var system = req.protectionSystem;
        var type = req.type;

        //Take necessary actions based on the system and type
    }

    if (typeof (e.completion) != 'undefined') {
        //Requested action completed
        var comp = e.completion;
        comp.complete(true);
    }
}
```

How you specifically check for rights, of course, is particular to the service you're drawing from—and not something you'd want to publish in any case!

For a more complete demonstration of handling DRM, check out the PlayReady sample, which will require that you download and install the Microsoft PlayReady Client SDK. PlayReady, if you aren't familiar with it yet, is a license service that Microsoft provides so that you don't have to create one from scratch. The PlayReady Client SDK provides additional tools and framework support for apps wanting to implement both online and offline media scenarios, such as progressive download, download to own, rentals, and subscriptions. Plus, with the SDK you don't need to submit your app for DRM Conformance testing. In any case, here's how the PlayReady sample sets up its content protection manager, just to give an idea of how the WinRT APIs are used with specific DRM service identifiers:

```
mediaProtectionManager = new Windows.Media.Protection.MediaProtectionManager();
mediaProtectionManager.properties[“Windows.Media.Protection.MediaProtectionSystemId”] =
    ‘{F4637010-03C3-42CD-B932-B48ADF3A6A54}’

var cpsystems = new Windows.Foundation.Collections.PropertySet();
cpsystems[“{F4637010-03C3-42CD-B932-B48ADF3A6A54}”] =
  “Microsoft.Media.PlayReadyClient.PlayReadyWinRTTrustedInput”;
mediaProtectionManager.properties[
   “Windows.Media.Protection.MediaProtectionSystemIdMapping”] = cpsystems;
```

Streaming from App to Network

The next case to consider is when an app is the source of streaming media rather than the consumer, which means that client apps elsewhere are acting in that capacity. In reality, in this scenario—audio or video communications and conferencing—it's usually the case that the app plays both roles, streaming media to other clients and consuming media from them. This is the case with Skype and other such utilities, along with apps like games that include a chat feature.

Here's how such apps generally work:

1. Set up the necessary communication channels over the network, which could be a peer-to-peer system or could involve a central service of some kind.

2. Capture audio or video to a stream using the WinRT APIs we've seen (specifically `MediaCapture.startRecordToStreamAsync`) or capturing to a custom sink.

3. Do any additional processing to the stream data. Note, however, that effects are plugged into the capture mechanism (`MediaCapture.addEffectAsync`) rather than something you do in post-processing.

4. Encode the stream for transmission however you need.

5. Transmit the stream over the network channel.

6. Receive transmissions from other connected apps.

7. Decode transmitted streams and convert to a blob by using MSApp.createBlobFromRandomAccessStream.

8. Use URL.createObjectURL to hook an audio or video element to the stream.

To see such features in action, check out the Real-time communications sample that implements video chat in scenario 2 and demonstrates working with different latency modes in scenario 1. The last two steps in the list above are also shown in the PlayToReceiver sample that is set up to receive a media stream from another source.

Play To

The final case of streaming is centered on the Play To capabilities that were introduced in Windows 7. Simply said, Play To is a means through which an app can connect local playback/display for audio, video, and img elements to a remote device.

The details happen through the Windows.Media.PlayTo APIs along with the extension methods added to media elements. If, for example, you want to specifically start a process of streaming immediately to a Play To device, invoking the selection UI directly, you'd do the following:

1. Call Windows.Media.PlayTo.PlayToManager APIs:

    a. getForCurrentView returns the object.

    b. showPlayToUI invokes the flyout UI where the user selects a receiver.

    c. sourceRequested event is fired when user selects a receiver.

2. In the sourceRequested handler:

    a. Get PlayToSource object from audio, video, or img element (msPlayToSource property) and pass to e.setSource.

    b. Set PlayToSource.next property to the msPlayToSource of another element for continual playing.

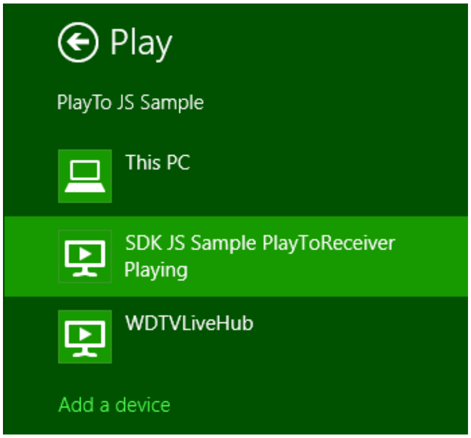
3. Pick up the media element's ended event to stage additional media.

You can find a demonstration in the Media Play To sample, where it displays a recommended glyph to programmatically display the Play To flyout using showPlayToUI:

You can also start media playback locally and then let the user choose a Play To receiver from the Devices > Play charm. In this case you don't need to do anything special with the Play To API at all because Windows will pick up the current playback element and direct it accordingly. But the app can listen to the `statechanged` event of the element's `msPlayToSource.connection` object (a PlayToConnection) that will fire when the user selects a receiver and when other changes happen.

One of the limitations of playing media from one machine to a Play To receiver is that DRM-protected playback isn't presently possible. Fortunately, there's another way to do it called *play by reference*. This means that your app just sends a URI to cloud-based media to the receiver such that the receiver can stream it directly. This is done through the `PlayToSource.preferredSourceUri` property or through an `audio` or `video` element's `msPlayToPreferredSourceUri` property (in JavaScript) or `x-ms-playToPreferredSourceUri` attribute (in HTML).

Generally speaking, Play To is primarily intended for streaming to a media receiver device that's probably connected to a TV or other large screen. This way you can select local content on a Windows device and send it straight to that receiver. But it's also possible to make a software receiver—that is, an app that can receive streamed content from a Play To source. The PlayToReceiver sample does exactly this, and when you run it *on another device* on your local network, it will show up in the Devices charm's Play group of your first machine as follows (shown here alongside my other hardware receiver):



Note that you might need to add the device through the Add a Device command at the bottom of the list (or through PC Settings > PC and Devices > Devices > Add a Device).

You can also run the receive app from your primary machine by using the remote debugging tools of Visual Studio, allowing you to step through the code of both source and receiver apps at the same time! Another option is to run Windows Media Player on one machine and check its Stream > Allow Remote Control Of My Player menu option. This should make that machine appear in the Play To target list. (If it doesn't, you might need to again add it through PC Settings > PC and Devices > Devices.)

To be a receiver, an app will generally want to declare some additional networking capabilities in the manifest—namely, *Internet (Client & Server)* and *Private Networks (Client & Server)*—otherwise it won't see much action! It then creates an instance of `Windows.Media.PlayTo.PlayToReceiver`, as shown in the Play To Receiver sample's `startPlayToReceiver` function (js/audiovideoptr.js):

```
function startPlayToReceiver() {
    if (!g_receiver) {
        g_receiver = new Windows.Media.PlayTo.PlayToReceiver();
    }
```

Next you'll want to wire up handlers for the element that will play the media stream:

```
var dmrVideo = id("dmrVideo");
dmrVideo.addEventListener("volumechange", g_elementHandler.volumechange, false);
dmrVideo.addEventListener("ratechange", g_elementHandler.ratechange, false);
dmrVideo.addEventListener("loadedmetadata", g_elementHandler.loadedmetadata, false);
dmrVideo.addEventListener("durationchange", g_elementHandler.durationchange, false);
dmrVideo.addEventListener("seeking", g_elementHandler.seeking, false);
dmrVideo.addEventListener("seeked", g_elementHandler.seeked, false);
dmrVideo.addEventListener("playing", g_elementHandler.playing, false);
dmrVideo.addEventListener("pause", g_elementHandler.pause, false);
dmrVideo.addEventListener("ended", g_elementHandler.ended, false);
dmrVideo.addEventListener("error", g_elementHandler.error, false);
```

along with handlers for events that the receiver object will fire:

```
g_receiver.addEventListener("playrequested", g_receiverHandler.playrequested, false);
g_receiver.addEventListener("pauserequested", g_receiverHandler.pauserequested, false);
g_receiver.addEventListener("sourcechangerequested",
    g_receiverHandler.sourcechangerequested, false);
g_receiver.addEventListener("playbackratechangerequested",
g_receiverHandler.playbackratechangerequested, false);
g_receiver.addEventListener("currenttimechangerequested",
g_receiverHandler.currenttimechangerequested, false);
g_receiver.addEventListener("mutechangerequested",
     g_receiverHandler.mutedchangerequested, false);
g_receiver.addEventListener("volumechangerequested",
     g_receiverHandler.volumechangerequested, false);
g_receiver.addEventListener("timeupdaterequested",
     g_receiverHandler.timeupdaterequested, false);
g_receiver.addEventListeer("stoprequested", g_receiverHandler.stoprequested, false);
g_receiver.supportsVideo = true;
g_receiver.supportsAudio = true;
g_receiver.supportsImage = false;
g_receiver.friendlyName = 'SDK JS Sample PlayToReceiver';
```

The last line above, as you can tell from the earlier image, is the string that will show in the Devices charm for this receiver once it's made available on the network. This is done by calling `startAsync`:

```
// Advertise the receiver on the local network and start receiving commands
g_receiver.startAsync().then(function () {
    g_receiverStarted = true;

    // Prevent the screen from locking
    if (!g_displayRequest) {
        g_displayRequest = new Windows.System.Display.DisplayRequest();
    }
    g_displayRequest.requestActive();
});
```

Of all the receiver object's events, the critical one is sourcechangerequested where eventArgs.stream contains the media we want to play in whatever element we choose. This is easily accomplished by creating a blob from the stream and then a URI from the blob that we can assign to an element's src attribute:

```
sourcechangerequested: function (eventIn) {
    if (!eventIn.stream) {
        id("dmrVideo").src = "";
    } else {
        var blob = MSApp.createBlobFromRandomAccessStream(eventIn.stream.contentType,
            eventIn.stream);
        id("dmrVideo").src = URL.createObjectURL(blob, {oneTimeOnly: true});
    }
}
```

All the other events, as you can imagine, are primarily for wiring together the source's media controls to the receiver such that pressing a pause button, switching tracks, or acting on the media in some other way at the source will be reflected in the receiver—it's what enables all the system transport controls (including hardware buttons on keyboards and such) to act as a remote control for the receiver. There might be a lot of events, but handling them is quite simple, as you can see in the sample. Also note that if the user switches to a different Play To device, the current receiver is stopped and everything gets picked up by the new receiver.

What We Have Learned

- Creating media elements can be done in markup or code by using the standard img, svg, canvas, audio, and video elements.
- The three graphics elements—img, svg, and canvas—can all produce essentially the same output, only with different characteristics as to how they are generated and how they scale. All of them can be styled with CSS, however.
- The Windows.Data.Pdf API provides a means to render PDF documents into files or streams.
- The Windows.System.Display.DisplayRequest object allows for disabling screen savers and the lock screen during video playback (or any other appropriate scenario).
- Both the audio and video elements provide a number of extension APIs (properties, methods, and events) for working with various platform-specific capabilities in Windows, such as horizontal mirroring, zooming, playback optimization, 3D video, low-latency rendering, Play To, playback management of different audio types or categories, effects (generally provided as DLLs in the app package), and digital rights management.
- Background audio is supported for several categories given the necessary declarations in the manifest and handlers for media control events (so that the audio can be appropriately paused and played). Media control events are important to support the system transport control UI.

- The `Windows.Media.SpeechSynthesis` API provides a built-in means to generate audio streams from plain text as well as Speech Synthesis Markup Language (SSML).
- The WinRT APIs provide for decoding and encoding of media files and streams, through which the media can be converted or the properties changed. This includes support for custom codecs as well as custom media stream sources that allow an app to inject processing code directly in the media rendering pipeline, even to dynamically generate media if desired.
- WinRT provides a rich API for media capture (photo, video, and audio), including a built-in capture UI, along with the ability to provide your own and yet still easily enumerate and access available devices.
- Streaming media is supported from a server (with and without DRM, including PlayReady), between apps (inbound and outbound), and from apps to Play To devices. An app can also be configured as a Play To receiver.

Chapter 14

Purposeful Animations

In the early 1990s, the wonderful world of multimedia first became prevalent on Windows PCs. Before that time it was difficult for such machines to play audio and video, access compact discs (remember those?), and otherwise provide the rich experience we take for granted today. The multimedia experience was new and exciting, and many people jumped in wholeheartedly, including the group of developer support engineers at Microsoft specializing in this area. Though my team (specializing in UI) sat more than 100 feet away from their area, we could clearly hear—for most of the day!—the various chirps and bleeps emitting from their speakers, against the background of a soft Amazon basin rainfall.

At that time too, many consumers of Windows were having fun attaching all kinds of crazy sounds to every mouse click, window transition, email arrival, and every other system event they could think of. Yet after a month or two of this sensual overload—not unlike being at a busy carnival—most people started to remove quite a few of those sounds, if not disable them altogether. I, for one, eventually turned off all my sounds. Simply said, I got tired of the extra noise.

Along these same lines, you may remember that when DVDs first appeared in their full glory, just about every title had fancy menus with clever transitions. No more: most consumers, I think, got tired of waiting for all this to happen and just want to get on with the business of watching the movie as quickly as possible.

Today we're reliving this same experience with fluid animations. Now that most systems have highly responsive touch screens and GPUs capable of providing very smooth graphical transitions, it's tempting to use animations superfluously. However, unless the animations actually add meaning and function to an app, consumers will likely tire of them like they did with DVD menus, especially if they end up interfering with a workflow by making one constantly wait for the animations to finish. I'll bet that every reader of this book has, at least once, repeatedly hit the Menu button on a DVD remote to no avail….

This is why Windows Store app design speaks of *purposeful* animations: if there's no real purpose behind an animation in your app, ask yourself, "Why am I wanting to use this?" Take a moment, in fact, to use Windows and some of the built-in apps to explore how animations are both used and *not* used. Notice how many animations are specifically to track or otherwise give immediate feedback for touch interactions, which purposefully help users know that their input is being registered by the system. Other animations, such as when items are added or removed from a list, are intended to draw attention to the change, soften its visual impact, and give it a sense of fluidity. In other cases, you may find apps that overdo animations, simply using them because they're available or trying too hard to emulate physical motion where it's simply not necessary. In this way, excessive animations constitute a kind of "chrome" with the same effect as other chrome: distracting the user from the content they really care about. (If you can't resist the temptation to add little effects that are like this, consider at least providing a setting to turn them off.)

Purposeful animations serve the needs of people rather than just showing off technology. Let me put it another way. When thinking about animations, ask yourself, "What do they communicate?" Animations are a form of communication, a kind of visual language. I would even venture to say (as I am venturing now) that animations really only say one or a combination of three things:

- "Thanks, I heard you," as when something on the screen moves naturally in response to a user gesture. Without this communication, the user might think that their gesture didn't register and will almost certainly poke at the app again.
- "Hello" and "Goodbye," as when items appear or disappear from view, or transition one to another. Without this communication, changes that happen to on-screen elements can be as jarring as Bilbo Baggins in *Lord of the Rings* slipping on the Ring of Power and instantly vanishing. This is not to say that most consumers are incredulous hobbits, of course, but you get the idea.
- "Hey, look at me!" as when something moves to only gain attention or look cute.

If I were to assign percentages to these categories to represent how often they would or should be used, I'd probably put them at 80%, 15%, and 5%, respectively (although some animations will serve multiple purposes). Put another way, the first bit of communication is really about listening and responding, which is what an app should be doing most of the time. The second bit is about courtesy, which is another good quality to express within reason—courtesy can, like handshakes, hugs, bows, and salutes, be overused to the point of annoyance. The third bit, for its part, can be helpful when there's a real and sincere reason to raise your hand or offer a friendly wave, but otherwise can easily become just another means of showing off.

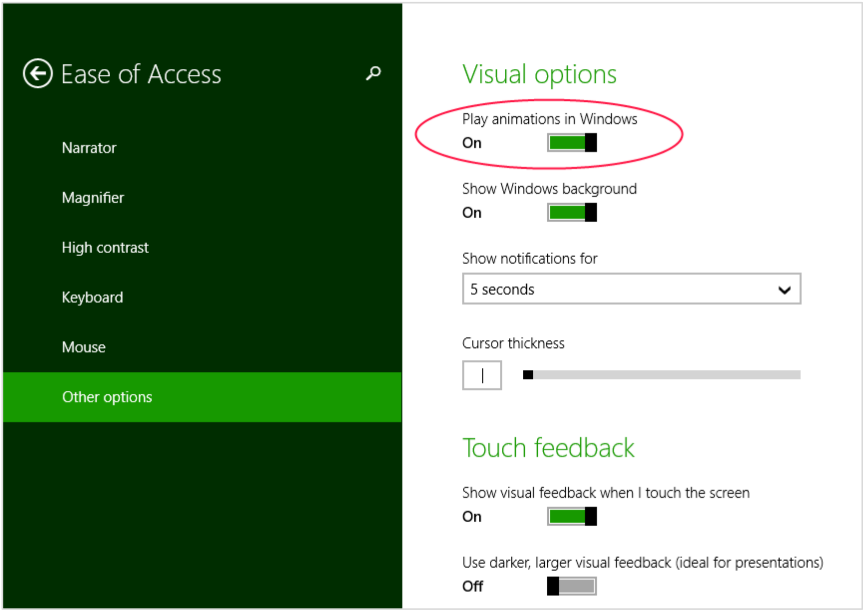
There's another good reason to be judicious about the use of animations and really make them count: power consumption. No matter how it's accomplished, via GPU or CPU, animation is an expensive process. Every watt of juice in a consumer's batteries should be directed toward fulfilling their goals with their device rather than scattered to the wind. Again, this is why this chapter is called "*Purposeful* Animations" and not just "Animations"!

In any case, you and your designers are the ultimate arbiters of how and when you'll use animations. Let me emphasize here that animations should be part of an app's design, not just an implementation detail. Animations are very much part of the overall user experience of an app. Oftentimes app designs focus on static wireframes and static mockups, neither of which indicate dynamic elements like animations and transitions. Animations are also tightly coupled to the app's layout and should be designed alongside that layout from the earliest stages of design.

In this uncommonly short chapter, then, we'll first look at what's provided for you in the WinJS Animations Library, a collection of animations built on CSS that already embody the Windows look and feel for many common operations. After that we'll review the underlying CSS capabilities that you can, of course, use directly. In fact, aside from games and other apps whose primary content consists of animated objects, you can probably use CSS for most other animation needs. This is a good idea because the CSS engine is very much optimized to take advantage of hardware acceleration, something that isn't true when doing frame-by-frame animations in JavaScript yourself. Nevertheless, we'll end this chapter on that latter subject, as there are some tips and tricks for doing it well within Windows Store apps.

Systemwide Enabling and Disabling of Animations

Before we go any further, take a moment to check PC Settings > Ease of Access > Other Options > Play Animations in Windows and make sure that it's turned on (see Figure 14-1). If it's off, you won't be seeing many animations in the system, including those that you trigger through WinJS APIs. This can be very confusing when writing an app that uses animations!



Play animations in Windows
On

Show Windows background
On

Show notifications for
5 seconds

Cursor thickness

Show visual feedback when I touch the screen
On

Use darker, larger visual feedback (ideal for presentations)
Off

FIGURE 14-1 A very important setting for animation in the desktop control panel.

The idea behind this check box is that for some users, animations are a real distraction that can make the entire machine more difficult to use. For medical reasons too, some users might elect to minimize on-screen movement just to keep the whole experience more calm. So when this option is checked, the WinJS animations don't actually do anything, and it's recommended that apps also disable many if not all of their own custom animations as well.

The setting value is obtained through the `Windows.UI.ViewManagement.UISettings` class in its `animationsEnabled` property:

```
var settings = new Windows.UI.ViewManagement.UISettings();
var enabled = settings.animationsEnabled;
```

You can also just call the `WinJS.UI.isAnimationEnabled` method that will return `true` or `false` depending on this property. WinJS uses this internally to manage its own animation behavior.

WinJS also adds an enablement count that you can use to temporarily enable or disable animations in conjunction with the `animationsEnabled` value. You change this count by calling `WinJS.UI.enableAnimations` and `WinJS.UI.disableAnimations`, the effects of which are cumulative, and the `animationsEnabled` property counts as 0 if the PC Settings option is off and 1 if it's on.

When implementing your own animations either with CSS or with mechanisms like `setInterval` or `requestAnimationFrame`, be sensitive to the `animationsEnabled` setting where appropriate. I add this condition because if an animation is essential to the actual content of an app, like a game, then it's not appropriate to apply this setting. The same goes for animating something like a clock within a clock app. It's really about animations that add a fast-and-fluid effect to the content but can be turned off without ill effect.

The WinJS Animations Library

When considering animations for your app, the first place you should turn is the Animations Library in WinJS, found in the `WinJS.UI.Animation` namespace. Each animation is a function within this namespace that you call when you want a certain kind of animation or transition to happen. The benefit of using these is that they directly embody the Windows look and feel and, in fact, are what WinJS itself uses to animate its own controls, flyouts, and so forth to match the user interface design guidelines. What's more, because they are built with CSS transitions and animations, they aren't dependent on WinRT and are fully functional within web context pages that have loaded WinJS (but they do again pay attention to whether animations are enabled as described in the previous section).

All of the animations, as listed in the table below, have guidance as to when and how they should be applied. These are again really design questions more than implementation questions, as stated earlier. By being aware of what's in the animations library, designers can more readily see where animations are appropriately applied and include them early on in their app design, which makes your life as a developer all the more predictable.

You can find full guidance in Animating Your UI and its subtopics in the documentation, which will also contain specific guidelines for the individual animations below. I will only summarize here.

Key point Built-in controls and other UI elements like those we've worked with in previous chapters already make use of the appropriate animations. For example, you don't need to animate a button tap in the `button` element nor animate the appearance or disappearance of controls like `WinJS.UI.Appbar`. You'll primarily use them when implementing UI directly with HTML layout or when building custom controls.

Animation Name	WinJS.UI.Animation methods	Description and Usage
Page Transition	`enterPage`, `exitPage`	Animates a whole page into or out of view, such as when bringing in the first page of an app after the splash screen or when switching between app pages. Avoid using `enterPage` when content is already on screen—that is, use it only when changing the entirety of the content.
Content Transition	`enterContent`, `exitContent`	Animates one or more elements into or out of view, specifically used for content that wasn't ready when a page was loaded or when a section of a page is changing within a container. If other content needs to move in response to the container change, such as if it is resizing, you can move those other elements by using expand/collapse or reposition animations.
Fade In/Out	`fadeIn`, `fadeOut`	Used to show or hide transient UI or controls, as is done with scrollbars or when a placeholder is replaced with a loaded item. These are also good default animations for situations where other specific animations don't apply.
Crossfade	`crossFade`	Softens the transition between different states of an item. This is also used in refresh scenarios, such as when a news app updates all of its content at once.
Pointer Up/Down	`pointerUp`, `pointerDown`	Provides immediate feedback for a successful tap or click on an item or tile-like elements. Note that built-in controls like the button and ListView already incorporate these animations.
Expand/Collapse	`createExpandAnimation`, `createCollapseAnimation`	Adds or removes extra space within content, such as making room for error messages or hiding an option that isn't needed.
Reposition	`createRepositionAnimation`	Used when moving an element to a new position.
Show/Hide Popup	`showPopup`, `hidePopup`	Used to show and hide popup UI like menus, flyouts, tooltips, and other contextual UI that appears above an app canvas (dialogs, however, use Fade In). Avoid using for elements that are part of that canvas directly—use Content Transition and Fade In/Out animations instead. You also don't need to use these directly when using built-in controls, as those controls already apply the animations.
Show/Hide Edge UI Show/Hide Panel UI	`showEdgeUI`, `hideEdgeUI`, `showPanel`, `hidePanel`	Used to show and hide edge-oriented UI like app bars and the soft keyboard. The Edge UI animations are for elements that only move a short distance onto the screen; the Panel animations are for those that move longer distances. These should not be used for UI that's not moving from or toward an edge; use the Reposition animation instead. Crossfade is also typically applied after showing and simultaneous with hiding. The built-in edge controls like the app bar and settings pane already apply these animations.
Peek (for tiles)	`createPeekAnimation`	Animates a tile update when alternating between image and text areas; see Chapter 16, "Alive with Activity." Can also be used to cycle through tile updates. This is the animation used for live tiles on the Windows Start screen.
Badge Update	`updateBadge`	Used to update the number on a tile badge.
Swipe Hint	`swipeReveal`	Used in response to a tap-and-hold event to indicate that an item can be selected with a swipe.
Swipe Select/Deselect	`swipeSelect`, `swipeDeselect`	Animates an item when swiped to select or deselect it.
Add/Delete from List	`createAddToListAnimation`,	Animates the insertion or deletion of items from a list, as

	`createDeleteFromListAnimation`	used by the ListView control. The add animation repositions existing items to make space for the new items and then brings them; the delete animation pulls items out and repositions those that remain. Avoid using these to display or remove a container or to add or remove the entire contents of the collection; use Content Transitions instead.
Add/Delete from Search List	`createAddToSearchListAnimation`, `createDeleteFromSearchListAnimation`	These animations are similar to those for adding and removing from a list, but they are designed for much more rapid changes as happens when populating a list of search results. Simply said, they have shorter durations.
Start/End Drag-Drop	`dragSourceStart`, `dragSourceEnd`, `dragBetweenEnter`, `dragBetweenLeave`	Provides visual feedback during drag-and-drop operations as seen on the Start screen when you move tiles around. The start and end animations are for the item being moved and should always be used together; the enter and leave animations are for rearranging the area around a potential drop point, which helps to show how the content will appear if the drop happens. For this purpose you'll need to define the size of potential target areas (rectangles) so that you can track pointer movement in and out of those areas.

If you want to see what these animations are actually doing, you can find all of that in the WinJS source code's ui.js file. Just search for the method, and you'll see how they're set up. The Crossfade animation, for example, animates the incoming element's opacity property from 0 to 1 over 167ms with a linear timing function, while animating the outgoing element's opacity from 1 to 0 in the same way. The Pointer Down animation changes the element's scale from 100% to 97.5% over 167ms according to a cubic-bezier curve, while Pointer Up does the opposite.

Knowing these characteristics—known as *animation metrics*—can be important when creating web-based content to integrate with your app. Because you cannot presently use WinJS on a remote web page, knowing how these animations work will help you emulate that behavior on such pages.

Within your app, though, avoid hard-coding the metrics. Instead, acquire them programmatically through the API in `Windows.Core.UI.AnimationMetrics`. You can find a demonstration of using this API in the Animation metrics sample.

As interesting as such details might be, of course, they are always subject to change (hence the API). And in the end, what's important is that you choose animations not for their visual effects but for their semantic meaning, using the right animations at the right times in the right places. So let's see how we do that.

Tip #1 All of the WinJS animations are implemented using the `WinJS.UI.executeAnimation` and `WinJS.UI.executeTransition` functions, which you can use for custom animations as well.

Tip #2 While an animation is running, always avoid changing an element's contents and its CSS styles that affect the same properties. The results are unpredictable and unreliable and can cause performance problems.

Animations in Action

To see all of the WinJS animations in action, run the HTML animation library sample. There are many different animations to illustrate, and this sample most certainly earns the award for the largest number of scenarios: twenty-two! In fact, the first thing you should do is go to scenario 22 and see whether animations are enabled, as that will most certainly affect your experience with the rest of the sample. The output of that scenario will show you whether the `UISettings.animationsEnabled` flag is set and allow you to increment or decrement the WinJS enablement count. So go check that now, because if you're like me (I dislike waiting for my task bar to animate up and down), you might have turned off system animations a long time ago for a snappier desktop experience. I didn't realize at first that it affected WinJS in this way!

Clearly, with 22 scenarios in the sample I won't be showing code for all of them here; indeed, doing so isn't necessary because many operate in the same way. The only real distinction is between those whose methods start with `create` and those that don't, as we'll see in a bit.

All the animation methods return a promise that you can use to take additional action when the animation is complete (at which point the completed handlers given to `then`/`done` will be called). If you already know something about CSS transitions and animations, you'll rightly guess that these promises encapsulate events like `transitionend` and `animationend`, so you won't need to listen for those events directly if you want to chain or synchronize animations. For chaining, you can just chain the promises; for synchronization, you can obtain the promises for multiple animations and wait for their completion using methods like `WinJS.Promise.join` or `WinJS.Promise.any`.

Animation promises also support the `cancel` method, which removes the animation from the element. This immediately sets the affected property values to their final states, causing an immediate visual jump to that end state. And whether you cancel an animation or it ends on its own, the promise is considered to have completed successfully; canceling an animation, in other words, will call the promise's completed handler and not its error handler.

Be aware that because all of the WinJS animations are implemented with CSS, they won't actually start until you give control back to the UI thread. This means that you can set up multiple animations knowing that they'll more or less start together once you return from the function. So even though the animation methods return promises, they are not like other asynchronous operations in WinRT that start running on another thread altogether.

Anyway, let's look at some code! In the simplest case, all you need to do is call one of the animation methods and the animation will execute when you yield. Scenario 6 of the sample, for instance, just adds these handlers to the `pointerdown` and `pointerup` events of three different elements (js/pointerFeedback.js):

```
function onPointerDown(evt) {
    WinJS.UI.Animation.pointerDown(evt.srcElement);
}

function onPointerUp(evt) {
    WinJS.UI.Animation.pointerUp(evt.srcElement);
}
```

We typically don't need to do anything when the animations complete, so there's no need for us to call done or provide a completed handler. Truly, using many of these animations is just this simple.

The crossFade animation, for its part (scenario 10), takes two elements: the incoming element and the outgoing element (all of which must be visible and part of the DOM throughout the animation, mind you!). Calling it then looks like this (js/crossfade.js):

```
WinJS.UI.Animation.crossFade(incoming, outgoing);
```

Yet this isn't the whole story. A common feature among the animations is that you can provide an *array* of elements on which to execute the same animation or, in the case of crossFade, two arrays of elements. While this isn't useful for animations like pointerDown and pointerUp (each pointer event should be handled independently), it's certainly handy for most others.

Consider the enterPage animation. In its singular form it accepts an element to animate and an optional initial offset where the element begins relative to its final position. (Generally speaking, you should omit this offset if you don't need it, because it will result in better performance—the sample passes null here, which I've omitted in the code below.) enterPage can also accept a collection of elements, such as the result of a querySelectorAll. Scenario 1 (html/enterPage.html and js/enterPage.js) provides a choice of how many elements are animated separately:

```
switch (pageSections) {
    case "1":
        // Animate the whole page together
        enterPage = WinJS.UI.Animation.enterPage(rootGrid);
        break;
    case "2":
        // Stagger the header and body
        enterPage = WinJS.UI.Animation.enterPage([[header, featureLabel], [contentHost,
           footer]]);
        break;
    case "3":
        // Stagger the header, input, and output areas
        enterPage = WinJS.UI.Animation.enterPage([[header, featureLabel],
        [inputLabel, input], [outputLabel, output, footer]]);
        break;
}
```

When the element argument is an array, the offset argument, if provided, can be either a single offset that is applied to all elements, or an array to indicate individual offsets for each element. Each offset is an object whose properties define the offset. See js/dragBetween.js for scenario 13 where this is used with the dragBetweenEnter animation:

```
WinJS.UI.Animation.dragBetweenEnter([box1, box2],
    [{ top: "-40px", left: "0px" }, { top: "40px", left: "0px" }]);
```

Here's a modification showing a single offset that's applied to both elements:

```
WinJS.UI.Animation.dragBetweenEnter([box1, box2], { top: "0px", left: "40px" });
```

Scenario 4 (js/transitioncontent.js) shows how you can chain a couple of promises together to transition between two different blocks of content:[102]

```
WinJS.UI.Animation.exitContent(outgoing, null).done( function () {
    outgoing.style.display = "none";
    incoming.style.display = "block";
    return WinJS.UI.Animation.enterContent(incoming, null);
});
```

Things get a little more interesting when we look at the `create*` animation methods, together referred to as the *layout animations*, which are for adding and removing items from lists, expanding and collapsing content, and so forth. Each of these has a three-step process where you (1) create the animation, (2) manipulate the DOM, and then (3) execute the animation, as shown in scenario 7 (js/addAndDeleteFromList.js):

```
// Create addToList animation.
var addToList = WinJS.UI.Animation.createAddToListAnimation(newItem, affectedItems);

// Insert new item into DOM tree. This causes the affected items to change position.
list.insertBefore(newItem, list.firstChild);

// Execute the animation.
addToList.execute();
```

The reason for the three-step process is that in order to carry out the animation on newly added items or items that are being removed, they all need to be in the DOM when the animation executes. The process here lets you create the animation with the initial state of everything, manipulate the DOM (or just set styles and so forth) to create the ending state, and then execute the animation to "let 'er rip." You can then use the `done` method on the promise returned from `execute` to perform any final cleanup. Scenario 5 (js/expandAndCollapse.js) makes this point clear:

```
// Create collapse animation.
var collapseAnimation = WinJS.UI.Animation.createCollapseAnimation(element, affected);

// Remove collapsing item from document flow so that affected items reflow to their new
// position. Do not remove collapsing item from DOM or display at this point, otherwise the
// animation on the collapsing item will not display.
element.style.position = "absolute";
element.style.opacity = "0";

// Execute collapse animation.
collapseAnimation.execute().done(
    // After animation is complete (or on error), remove from display.
    function () { element.style.display = "none"; },
    function () { element.style.display = "none"; }
```

[102] Note that the sample erroneously passes a variable `output` as the first parameter to `exitContent` and `enterContent`; the code should appear as shown here, with `outgoing` passed to `exitContent` and `incoming` passed to `enterContent`.

```
);
```

As a final example—because I know you're smart enough to look at most of the other cases on your own—scenario 21 (js/customAnimation.js) shows how to use the `WinJS.UI.executeAnimation` and `WinJS.UI.executeTransition` methods.

```
function runCustomShowAnimation() {
    var showAnimation = WinJS.UI.executeAnimation(
        target,
        {
            // Note: this keyframe refers to a keyframe defined in customAnimation.css.
            // If it's not defined in CSS, the animation won't work.
            keyframe: "custom-opacity-in",
            property: "opacity",
            delay: 0,
            duration: 500,
            timing: "linear",
            from: 0,
            to: 1
        }
    );
}

function runCustomShowTransition() {
    var showTransition = WinJS.UI.executeTransition(
        target,
        {
            property: "opacity",
            delay: 0,
            duration: 500,
            timing: "linear",
            to: 1
        }
    );
}
```

If you want to combine multiple animations (as many of the WinJS animations do), note that both of these functions return promises so that you can combine multiple results with `WinJS.Promise.join` and have a single completed handler in which to do post-processing. This is exactly what WinJS does internally.

And if you know anything about CSS animations and transitions already, you can probably tell that the objects you pass to `executeAnimation` and `executeTransition` are simply shorthand expressions of the CSS styles you would use otherwise. In short, these methods give you an easy way to set up your own custom animations and transitions through the capabilities of CSS. Let's now look at those capabilities directly.

Sidebar: Parallax/Panorama Animations

Developers often ask how to create a *parallax* or *panorama* background animation as seen on the Windows Start screen. If you're not familiar with this concept, go to the Start screen and pan around a little, noticing how the background pans as well but slower than the tiles. This creates a sense of the tiles floating above the background.

While it is possible to implement this effect in JavaScript (see the KidsBook example on the Internet Explorer TestDrive site), we don't recommend it or at least recommend providing a setting to turn the effect off. At issue is the fact that such animations run *dependently* (on the CPU) rather than *independently* (on the GPU), as described in the next section. As such, the threading and rendering model of JavaScript results in choppy movement except on high-power devices; the effect will be very pronounced on low-power and especially ARM devices. In addition, such animations can be costly in terms of CPU and battery utilization.

This is one case in which using C++ and DirectX (or even C#/VB and XAML) has a clear advantage over JavaScript, and would be a consideration if you absolutely must have this effect in your app.

CSS Animations and Transitions

As noted before, many animation needs can be achieved through CSS rather than with JavaScript code running on intervals or animation frames. The WinJS Animations Library, as we've just seen, is entirely built on CSS. Using CSS relieves us from writing a bunch of code that worries about how much to move every element in every frame based on elapsed time and synchronized to the refresh rate. Instead, we can simply declare what we want to happen (perhaps using the `WinJS.UI.executeAnimation` and `WinJS.UI.executeTransition` helpers as shown earlier in "Animations in Action") and let the app host take care of the details. Delegation at its best! In this section, then, let's take a closer look at the capabilities of CSS for Windows Store apps.

Another huge benefit of performing animations and transitions through CSS—specifically those that affect only transform and opacity properties—is that they can be used to create what are called *independent animations* that run on a GPU thread rather than the UI thread. This makes them smoother and more power-efficient than *dependent animations* that are using the UI thread. Dependent animations happen when you create animations in JavaScript using intervals, use CSS animations and transitions with properties other than transform and opacity, or run animations on elements that are partly or wholly obscured by other elements. (If you want to get into the details, refer to Independent Composition: Rendering and Compositing in Internet Explorer 10, which applies to Windows Store apps written in JavaScript.)

We'll come back to this subject in a bit when we look at sample code. Let's first review how CSS animations and transitions work (and you can find many more tutorials on the web). I say both animations *and* transitions because there are, in fact, two separate CSS specifications: CSS animations and CSS transitions. So what's the difference?

Normally when a CSS property changes, its value jumps immediately from the old value to the new value, resulting in a sudden visual change. *Transitions* instruct the app host how to change one or more property values gradually, according to specific delay, duration, and timing curve parameters. All of this is declared within a specific style rule for an element (as well as `:before` and `:after` pseudo-elements) using four individual styles:

- `transition-property` (`transitionProperty` in JavaScript) Identifies the CSS properties affected by the transition (the transitionable properties are listed in section 7 of the transitions spec).
- `transition-duration` (`transitionDuration` in JavaScript) Defines the duration of the transition in seconds (fractional seconds are supported, as in `.125s`; negative values are normalized to `0s`).
- `transition-delay` (`transitionDelay` in JavaScript) Defines the delayed start of the transitions relative to the moment the property is changed, in seconds. If a negative value is given, the transition will appear to have started earlier but the effect will not have been visible.
- `transition-timing-function` (`transitionTimingFunction` in JavaScript) Defines how the property values change over time. The functions are `ease`, `linear`, `ease-in`, `ease-out`, `ease-in-out`, `cubic-bezier`, `step-start`, and `step-end`. The W3C spec has some helpful diagrams that explain these, but the best way to see the difference is to try them out in running code.

For example, a transition for a single property appears like so:

```
#div1 {
    transition-property: left;
    transition-duration: 2s;
    transition-delay: .5s;
    transition-timing-function: linear;
}
```

When defining transitions for multiple properties, each value in each style is separated by a comma:

```
.class2 {
    transition-property: opacity, left;
    transition-duration: 1s, 0.25s;
}
```

Again, transitions don't specify any actual beginning or ending property values—they define how the change actually happens *whenever* a new property is set through another CSS rule or through JavaScript. So, in the first case above, if `left` is initially 100px and it's set to 300px through a `:hover` rule, it will transition after 0.5 seconds from 100px to 300px over a period of 2 seconds. Doing the math, the visual movement with a `linear` timing function will run at 100px/second. Other timing functions will show different rates of movement at different points along the 2-second duration.

If a bit of JavaScript then sets the value to -200px—ideally after the first transition completes and fires its `transitionend` event—the value will again transition over the same amount of time but now from 300px to -200px (a total of 500px). As a result, the element will move at a higher speed (250px/second, again with the `linear` timing function) because it has more ground to cover for the same transition duration.

What's also true for transitions is that if you assign a style (e.g., `class2` above) to an element, nothing will happen until an affected property changes value. Changing a style like this also has no effect if a transition is already in progress. The exception is if you change the `transition-property` value, in which case that transition will stop. With this, it's important to note that the default value of this property is `all`, so clearing it (setting it to `""`) doesn't stop all transitions—it enables them! You instead need to set the property to `none`.

> **Note** Elements with `display: none` do not run CSS animations and transitions at all, for obvious reasons. The same cannot be said about elements with `display: hidden`, `visibility: hidden`, `visibility: collapsed`, or `opacity: 0`, which means that hiding elements with some means other than `display: none` might end up running animations on nonvisible elements, which is a complete waste of resources. In short, use `display: none`.

Animations work in an opposite manner to transitions. Animations are defined separately from any CSS style rules but are then attached to rules. Assigning that style to an element then triggers the animation immediately applying any delay defined in the animation. (The animation can be initially paused as well if you want to start it at a later time.) Furthermore, groups of affected properties are defined together in *keyframes* and are thus animated together.

A CSS animation, in other words, is an instruction to progressively update one or more CSS property values over a period of time. The values change from an initial state to a final state through various intermediate states defined by a set of keyframes. Here's an example (from scenario 1 of the HTML independent animations sample we'll be referring to later in this chapter):

```
@keyframes move {
    from { transform: translateX(0px); }
    50% { transform: translateX(400px); }
    to { transform: translateX(800px); }
}
```

More generally:

- Start with `@keyframes <identifier>` where `<identifier>` is a name for the animation (like *move* above). You'll refer to this identifier elsewhere in style rules.
- Within this animation, create any number of individual keyframes (also called *rule sets*), each of which represents a different snapshot of the animated element at different stages in the overall animation, demarked by percentages. The `from` and `to` keywords, as shown above, are simply aliases for 0% and 100%, respectively.

- Within each keyframe, define the desired value of *any number of style properties* (just transform in the example above), with each separated by a semicolon as with CSS styles. If a value for a property is the same as in the previous rule set, no animation will occur for that property. If the value is different, the rendering engine will animate the change between the two values of that property across the amount of time equivalent to *<overall animation time>* * *(<toPercentage> - <fromPercentage>)/100*. A timing function can also be specified for each rule set using the animation-timing-function style. For example:

```
50% { transform: translateX(400px); animation-timing-function: ease-in;}
```

One thing you'll notice here is that while the keyframe can indicate a timing function, it doesn't say anything about actual *timings*. This is left for the specific style rules that *refer* to the animation. In scenario 1 of the sample, for instance:

```
.ball {
    animation-name: move;
    animation-duration: 2s;
    animation-timing-function: linear;
    animation-delay: 0s;
    animation-iteration-count: infinite;
    animation-play-state: running;
}
```

Here, the animation-name style (animationName in JavaScript) identifies the animation to apply, and we'll look at the other styles in a moment. Before that, it's essential to understand that this *ball* style class is what gets added to an element to start the *move* animation according to the other timing properties. Again, the animation begins when that class is assigned. In typical practice, you can define a separate class like this that contains just the animation styles and then add that class to any number of elements individually. Scenario 1 of the sample does exactly this with two different elements, one that runs independently and one that runs dependently (js/scenario1.js):[103]

```
IndependentAnimationelement.className = 'ball';
DependentAnimation.className = 'ball';
IndependentAnimationelement.style.animationPlayState = 'running';
DependentAnimation.style.animationPlayState = 'running';
```

Setting the animationPlayState properties here is not actually necessary because the *ball* class already sets that. If, however, the style class initially sets paused, setting the play state to running will then start the animation.

Note too that you can also have any number of classes with different animation* styles that all refer to the same animation-name, which allows you to reuse the same animation with any number of timing variations.

[103] In this code, any existing style classes will be removed from the elements by virtue of setting className. If you want to add and remove classes without affecting others, use the WinJS.Utilities.addClass and removeClass methods.

In any case, the other `animation-*` styles describe how the animation should execute:

- `animation-duration` (`animationDuration` in JavaScript) The duration of the animation in seconds (fractions allowed, as in `0.4s`) or milliseconds (as in `400ms`). Negatives are the same as `0s`.
- `animation-timing-function` (`animationTimingFunction` in JavaScript) Defines, as with transitions, how the property values are interpolated over time—`ease` (the default), `linear`, `ease-in`, `ease-out`, `ease-in-out`, `cubic-bezier`, `step-start`, and `step-end`.
- `animation-delay` (`animationDelay` in JavaScript) Defines the number of seconds (`s` suffix) or milliseconds (`ms` suffix) after which the animation will start when the style is applied. This can be negative, as with transitions, which will start the animation partway through its cycle.
- `animation-iteration-count` (`animationIterationCount` in JavaScript) Indicates how many times the animation will repeat (default is 1). This can be a number or `infinite`, as shown above. If you want an animation to run forwards and then backwards, such as creating a temporary swell effect, set this count to 2 and the `animation-direction` (below) to `alternate`, as each direction will count as an iteration.
- `animation-direction` (`animationDirection` in JavaScript) Indicates whether the animation should play `normal` (forward), `reverse`, `alternate` (back and forth), or `alternate-reverse` (back and forth starting with `reverse`). The default is `normal`. Note again that `alternate` and `alternate-reverse` require you to have at least two iterations because each direction counts as a separate iteration.
- `animation-play-state` (`animationPlayState` in JavaScript) Sets the initial run state of the animation: the default state of `running` plays the animation as soon as the style is set on the element, whereas `paused` will hold off starting the animation until you set the style to `running`. You can use this style from JavaScript to start and pause animations whenever needed until the animation has completed. After that time, setting the play state has no effect.
- `animation-fill-mode` (`animationFillMode` in JavaScript) Defines which property values of the named keyframe will be applied when the animation is not executing, such as during the initial delay or after it is completed. The default value of `none` applies the values of the `0%` or `from` rule set if the direction is `forward` and `alternate` directions; it applies those of the `100%` or `to` rule set if the direction is reverse or `alternate-reverse`. A fill mode of `backwards` flips this around. A fill mode of `forwards` always applies the `100%` or `to` values (unless the iteration count is zero, in which case it acts like `backwards`). The other option, `both`, is the same as indicating both `forwards` and `backwards`.
- `animation` (`animation` in JavaScript) The shorthand style for all of the above (except for `animation-play-state`) in the order of name, duration, timing function, delay, iteration count, direction, and fill mode.

Again, applying a style that contains animation-name will trigger the animation for that element if the play state is running. You can do this by setting an element's className directly, as shown in the code from the HTML independent animations sample shown earlier, but it's better to use WinJS.Utilities.addClass because this will not affect any of the element's existing classes.

Triggering an animation will also happen automatically if the animation is named in a style that's applied to an element by default in your CSS. You can set the animation property for an element, of course, and if the play state is initially set to paused in CSS, you can then trigger the animation on demand by changing the animationPlayState property to running.

The nature of a CSS animation is to run until it's complete, during which time you can pause and start it through animationPlayState, but once the animation is complete you have to remove the animation styles from an element and then add them again to restart that animation.

This is where the element's animationend event comes into play: it's the right place to remove the animation class from the element, ideally using WinJS.Utilities.removeClass, which lets you remove a specific class from an element without affecting any others. For example, here's how I'd set up a handler for the event to automatically reset the *ball* animation for an element:

```
element.addEventListener("animationend", resetBallAnimation);
WinJS.Utilities.addClass(element, "ball");

function resetBallAnimation(e) {
    e.target.removeEventListener("animationend", resetBallAnimation);
    WinJS.Utilities.removeClass(e.target, "ball");
}
```

This way, the next time I add the *ball* class to the element, the animation will run again.

Keyframes, although typically defined in CSS, can also be defined in JavaScript. The first step is to build up a string that matches what you'd write in CSS, and then you insert that string to the stylesheet. This is shown in scenario 7 of the HTML independent animations sample (js/scenario7.js, which also has a demonstration of the animationend event):

```
var styleSheet = document.styleSheets[1];
var element1 = document.getElementById("ballcontainer");
var animationString = '@keyframes bounce1 {'
    // ...
    + '}';

styleSheet.insertRule(animationString, 0);

window.setImmediate(function () {
    element1.style.animationName = 'bounce1';
});
```

Note how setImmediate is used to yield to the UI thread before setting the animationName property to trigger the animation. This ensures that whatever other code follows (not shown here) will execute first, as it does some other work the sample wants to complete before the animation begins.

More generally, it's good to again remember that CSS animations and transitions start only when you return from whatever function is setting them up. That is, nothing happens visually until you yield back to the UI thread and the rendering engine kicks in again, just like when you change nonanimated properties. This means you can set up however many animations and transitions as desired, and they'll all execute simultaneously. Using a callback with `setImmediate`, as shown above, is a simple way to say, "Run this code as soon as there is no pending work on the UI thread."[104] Such a pattern is typically for triggering one or more animations once everything else is set up.

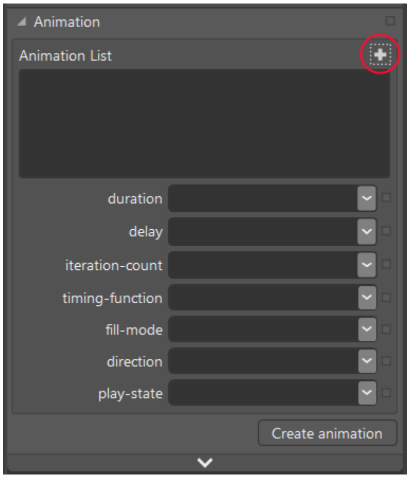
Of course, you should ideally prioritize your animation work in relation to other work happening on the UI thread, using the WinJS scheduler API as discussed in Chapter 3, "App Anatomy and Performance Fundamentals."

As a final note for this section, you might be interested in The Guide to CSS Animation: Principles and Examples (Smashing Magazine). This will tell you a lot about animation design beyond just how CSS animations are set up in your code.

Designing Animations in Blend for Visual Studio

Although you can define CSS animations directly in text, it helps to have a way to immediately visualize the results and make adjustments. Blend for Visual Studio provides just such a design experience, which I demonstrate in Video 14-1. (Another demonstration can be found in the //build 2013 session 2-311, What's New in Blend for HTML Developers, between time indices 42:00 and 46:00.)

You start in the Live DOM pane by selecting the element to which you want to apply an animation. Under the CSS properties pane, then, expand the Animations group that appears as follows:

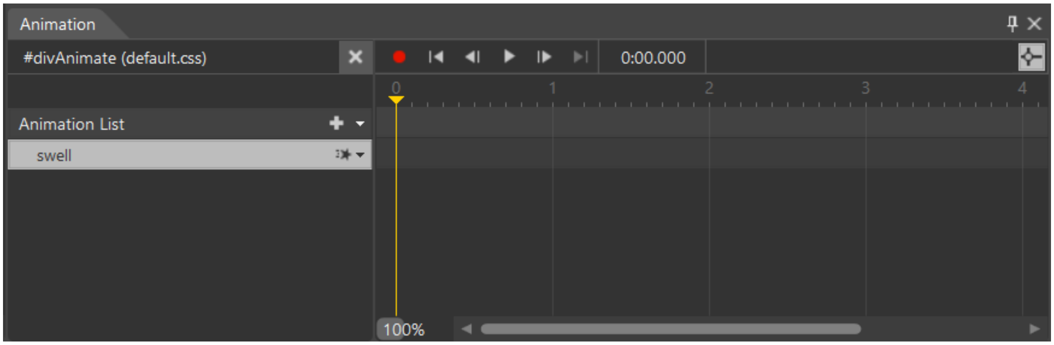


[104] For more on this topic, see http://dvcs.w3.org/hg/webperf/raw-file/tip/specs/setImmediate/Overview.html.

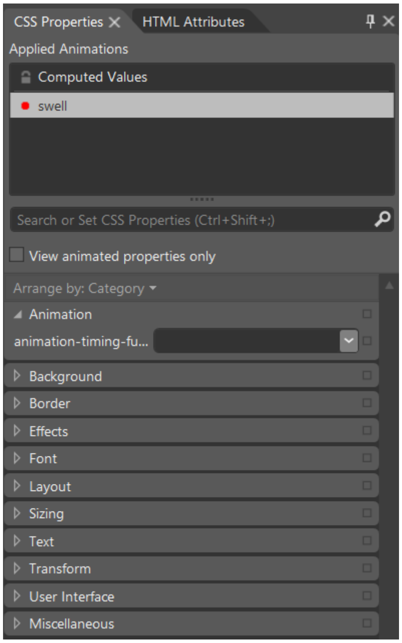
Click the + in the upper right (circled) to create an animation with whatever name you enter, and then set the animation's timing properties in the drop-downs below. This will create the appropriate animation reference for the element. For example, if I've selected an element with the id *divAnimate* and create a simple back-and-forth animation called *swell* (to briefly enlarge the element, which requires two iterations), the following will be added to the CSS:

```
#divAnimate {
    animation: swell 300ms ease-in 0ms 2 alternate forwards;
    animation-play-state: running;
}
```

Next we need to define the animation's keyframes. Click the Create Animation button at the lower right and Blend opens up a timeline view:



At this point you're in Recording Mode, which is indicated by the red dot at the left end of the playback controls and by a red dot and frame around the artboard. Here, set the playback cursor to a position where you want to define a keyframe. In the CSS Properties pane you'll see the name of the animation at the top; the list of properties that appear underneath are those that can be animated:



Recording Mode means that any properties you set in this CSS properties pane are recorded in the keyframe for the current timeline position. And as long as you're in this mode, you can move the timeline position around and edit more keyframes. For example, if I place the playback position to 0s and set a scaling transform to 1, and then I move the playback position to the end of the first iteration and set a scaling transform to 1.2 plus, I get the following in CSS:

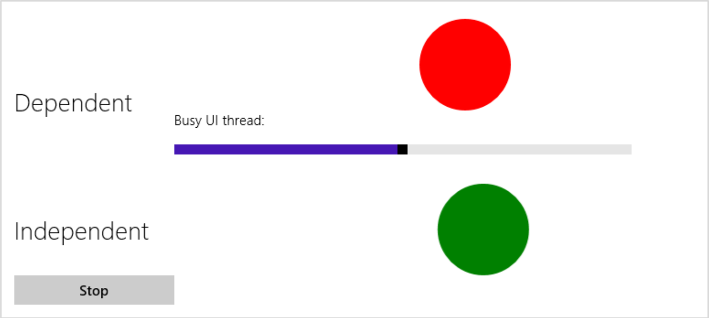
```
@keyframes swell {
    0% {
        transform: scale(1, 1);
    }

    100% {
        transform: scale(1.2, 1.2);
    }
}
```

In the animation timeline you can then use the other playback controls to see the result. You can also drag the timeline positioner back and forth. The animation will run in Interactive Mode as well.

When you're done editing, turn recording off by clicking the red dot; you'll see the CSS Properties pane revert to the normal view.

The HTML Independent Animations Sample

Let's now more fully examine the HTML independent animations sample. Scenario 1 gives a demonstration of an independent versus a dependent animation by eating some time on the UI thread (that is, blocking that thread) according to a slider. As a result, the top red ball (see image below) moves choppily, especially as you increase the work on the UI thread by moving the slider—the dynamic effect is shown in Video 14-2. The green ball on the bottom, on the other hand, continues to move smoothly the whole time.

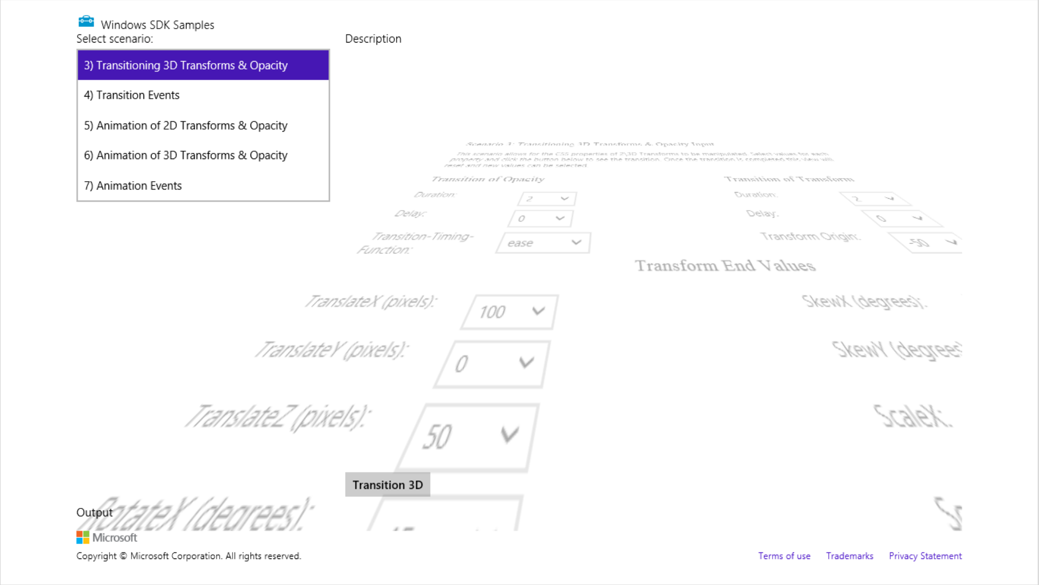


What's tricky to understand about this sample is that both balls use the same CSS style rule named *ball* that we saw earlier. In fact, just about everything about the two elements is exactly the same. So why does the movement of the red ball get choppy when additional work is happening on the UI thread?

The secret is in the `z-index: -1;` style on the red ball in css/scenario1.css (and a corresponding lack of `position: static` which negates `z-index`). For animations to run independently, *they must be free of obstruction*. This really gets into the subject of how layout is being composed within the HTML/CSS rendering engine of the app host—an animating element that's somewhere in the middle of the z-order might end up being independent or dependent. The short of it is that the `z-index` style is the only lever that's available for you to pull here.

As I noted before, independent animations are limited to those that affect only the `transform` and `opacity` properties for an element. If you animate any property that affects layout, like `width` or `left`, the animation will run as dependent (and similar results can be achieved with a scaling and translation transform anyway). Other factors also affect independent animations, as described on the Best Practices: Animating topic in the documentation. For example, if the system lacks a GPU, if you overload the GPU with too many independent animations, or if the elements are too large, some of the animations will revert to dependent. This is another good reason to be purposeful in your use of animations—overusing them will produce a terrible user experience on lower-end devices, thereby defeating the whole point of using animations to enhance the user experience!

The other scenarios of the HTML independent animations sample let you play with CSS transitions and animations by setting values within various controls and then running the animation. Scenarios 2 and 3 work with CSS transitions for 2D and 3D transforms, respectively, with an effect of the latter shown in Figure 14-2 and both shown more dynamically in Video 14-3. As you can see, the element that the sample animates is the container for all the input controls! Scenarios 5 and 6 then let you do similar things with CSS animations. In all these cases, the necessary styles are set directly in JavaScript rather than using declarative CSS, so look in the .js files and not the .css files for the details.



Scenario 3: Transitioning 3D Transforms & Opacity Input

This scenario allows for the CSS properties of 2\3D Transforms to be manipulated. Select values for each property and click the button below to see the transition. Once the transition is completed this view will reset and new values can be selected.

Transition of Opacity

Duration: 2

Delay: 0

Transition-Timing-Function: ease

Transition of Transform

Duration: 2

Delay: 0

Transform Origin: -50

Transform End Values

TranslateX (pixels): 100

TranslateY (pixels): 0

TranslateZ (pixels): 50

RotateX (degrees):

SkewX (degrees):

SkewY (degrees

ScaleX:

Copyright © Microsoft Corporation. All rights reserved.

FIGURE 14-2 Output of scenario 3 of the HTML independent animations sample.

Scenarios 4 and 7 show something we've only touched on so far, which are the few simple events that are raised for transitions and animations (and actually have nothing to do with independent versus dependent animations). In the former case, any element on which you execute a CSS transition will fire `transitionstart` and `transitionend` events. You can use these to chain transitions together.

With animations, there are three events: `animationstart` (which comes after any delay has passed), `animationend` (when the animation finished), and `animationiteration` (at the end of each iteration, unless `animationend` also fires at the same time). As with transitions, all of these can be used to chain animations or otherwise synchronize them. The `animationiteration` event is also helpful if you need to run a little code every time an animation finishes a cycle. In such a handler you might check conditions that would cause you to stop an animation, in which case you can set the `animationPlayState` to paused when needed.

Rolling Your Own: Tips and Tricks

If you're anything like me, I imagine that one of the first things you did when you started playing with JavaScript is to do some kind of animation: set up some initial conditions, create a timer with `setInterval`, do some calculations in the handler and update elements (or draw on a `canvas`[105]), and keep looping until you're done. After all, this sort of thing is at the heart of many of our favorite games!

[105] Or possibly multiple layered canvases, where you can isolate different animation groups on their own canvases. For more on this, see Optimize HTML5 canvas rendering with layering (IBM developerWorks).

(For an introductory discussion on this, just in case you haven't done this on your own yet, see How to animate canvas graphics.)

Considerable wisdom on this subject is available in the community if you decide to go this route. I put it this way because by now, having looked at the WinJS animations library and the capabilities of CSS, you should be in a good position to decide whether you actually *need* to go this route at all. Some people have estimated that a vast majority of animations needed by most apps can be handled entirely through CSS: just set a style and let the app host do the rest.[106] But if you decide that you still need to do low-level animation, the first thing you should do is ask yourself this question:

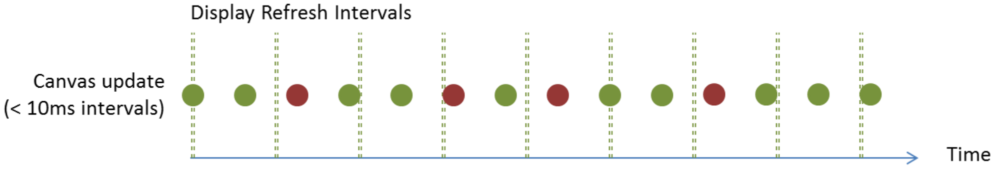
What is the appropriate animation interval?

This is a very important question because oftentimes developers have no idea what kind of interval to use for animation. It's not so much of an issue for long intervals, like 500ms or 1s, but developers often just use 10ms because it seems "fast enough."

To be honest, 10ms is overkill for a number of reasons. 60 frames per second (fps)—an animation interval of 16.7ms—is about the best that human beings can even discern and is also the best that most displays can even handle in the first place. In fact, the best results are obtained when your animation frames are synchronized with the screen refresh rate.

Let's explore this a little more. Have you ever looked at a screen while eating something really crunchy and noticed how the pixels seem to dance all over the place? This is because display devices aren't typically just passive viewports onto the graphics memory. Instead, displays typically cycle through graphics memory at a set refresh rate, which is most commonly 60Hz or 60fps (but can also be as low as 50Hz or as high as 100Hz).

This means that trying to draw animations at an interval faster than the refresh rate is a waste of time, is a waste of power (it has been shown to reduce battery life by as much as 25%!), and results in dropped frames. The latter point is illustrated below, where the red dots are frames that get drawn on something like a `canvas` but never make it to the screen because another frame is drawn before the screen refreshes:



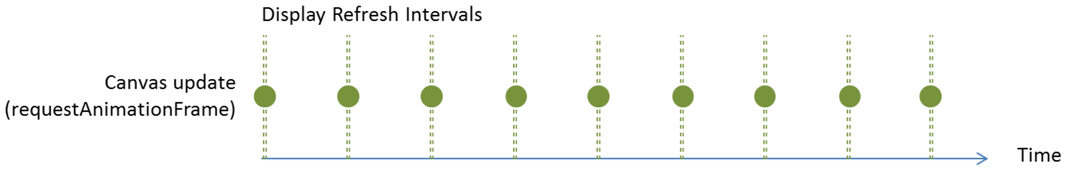
This is why it's common to animate on multiples of 16.7ms using `setInterval`. However, using 16.7 assumes a 60Hz display refresh, which isn't always the case. The right solution, then, for both Windows

[106] It's also worth noting that you could use the `MediaStreamSource` that we discussed at the end of Chapter 13, "Media," to dynamically generate a video, which is effectively an animation but would then provide playback controls, Play To capabilities, and so forth.

Store apps in JavaScript and web apps is to use requestAnimationFrame. This API simply takes a function to call for each frame:

```
requestAnimationFrame(renderLoop);
```

You'll notice that there's not an interval parameter; the function rather gives you a way to align your frame updates with display refreshes so that you draw only when the system is ready to display something:



What's more, requestAnimationFrame also takes page visibility into account, meaning that if you're not visible (and animations are thus wasteful), you won't be asked to render the frame at all. This means you don't need to handle page visibility events yourself to turn animations on and off: you can just rely on the behavior of requestAnimationFrame directly.

Tip If you really want an optimized display, consider doing all your app's drawing work (not just animations) within a requestAnimationFrame callback. That is, when processing a change, as in response to an input event, update your data and call requestAnimationFrame with your rendering function rather than doing the rendering immediately. And always be mindful to redraw only when you need to redraw, as we'll see in a moment, to make the best use of CPU and battery power.

It's also good to know that attempting to animate a canvas that's partly obscured by an element with display: inline-block has been found to result in very poor performance and large gaps between frames because of excessive region invalidation. Using a different display model such as table-cell avoids this issue.

Calling this method once will invoke your callback for a single frame. To keep up a continuous animation, your handler should call requestAnimationFrame again.

Tip Be mindful about JavaScript closures within the callbacks for requestAnimationFrame, setInterval, and setTimeout, especially when you renew the callback again. An accumulation of non-garbage-collected closures can accumulate large memory leaks, especially at 60fps!

Keeping up a continuous animation is shown in the Using requestAnimationFrame for power efficient animations sample (this wins fourth place for long sample names!), which draws and animates a clock with a second hand:

The first call to `requestAnimationFrame` happens in the page's `ready` method, and then the callback refreshes the request (js/scenario1.js):

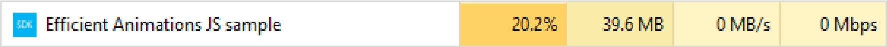
```
window.requestAnimationFrame(renderLoopRAF);

function renderLoopRAF() {
    drawClock();
    window.requestAnimationFrame(renderLoopRAF);
}
```

where the `drawClock` function gets the current time and calculates the angle at which to draw the clock hands:

```
function drawClock() {
    // ...

    // Note: this is modified from the sample to create a Date only once, not each time
    var date = new Date();
    var hour = date.getHours();
    var minute = date.getMinutes();
    var second = date.getSeconds();

    // ...

    var sDegree = second / 60 * 360 - 180;
    var mDegree = minute / 60 * 360 - 180;
    var hDegree = ((hour + (minute / 60)) / 12) * 360 - 180;

    // Code to use the context's translate, rotate, and drawImage methods
    // to render each clock hand
}
```

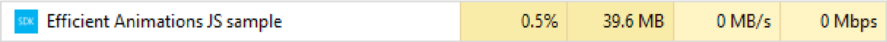
Here's a challenge for you: *What's wrong with this code?* Run the sample and look at the second hand. Then think about how `requestAnimationFrame` aligns to screen refresh cycles with an interval like 16.7ms. What's wrong with this picture?

What’s wrong is that even though the second hand is moving visibly only *once per second*, the drawClock code is actually executing nearly *50, 60, or 100 times more frequently* than that! Thus the “Efficient and Smooth Animations” title that the sample shows on screen is anything but! Indeed, if you run Task Manager, you can see that this simple “efficient” clock is ironically consuming a disproportionate amount of CPU. Yikes! (The percentage depends on your hardware, clearly—the 20% shown here is on an older laptop; my newer device shows more like 8%).



Efficient Animations JS sample 20.2% 39.6 MB 0 MB/s 0 Mbps

Remember that an interval aligned with ~16.7ms screen refreshes (on a 60Hz display) implies 60fps rendering. If you don’t need that much, you should skip frames yourself according to elapsed time, thereby saving power, and not blindly redraw as this sample is doing. In fact, if all we need is a once-per-second movement in a clock like this, repeated calls to requestAnimationFrame is sheer overkill. We could instead use setInterval(function () { requestAnimationFrame(drawClock) }, 1000) to coordinate 1s intervals with screen refreshes. If you make this change in the ready method, for example, the CPU usage will drop precipitously (even down to less than 0.1%):



Efficient Animations JS sample 0.5% 39.6 MB 0 MB/s 0 Mbps

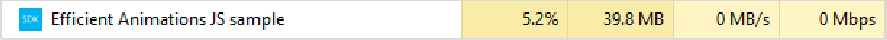
But let’s say we actually want to put 60fps animation and 20% of the CPU to good use. In that case, we should at least make the clock’s second hand move smoothly, which can be done by simply adding milliseconds into the angle calculation in the drawClock method (and reversing the previous setInterval change):

```
var date = new Date();
var second = date.getSeconds() + date.getMilliseconds() / 1000;
```

Still, 8% to 20% is a lot of CPU power to spend on something so simple and 60fps is still serious overkill. ~10fps is probably sufficient for good effect. In this case we can calculate elapsed time within renderLoopRAF to call drawClock only when 0.1 seconds have passed:

```
var lastTime = 0;

function renderLoopRAF() {
    var fps = 10;  // Target frames per second
    var interval = 1000 / fps;
    var curTime = Math.floor(Date.now() / interval);

    if (lastTime != curTime) {
        lastTime = curTime;
        drawClock();
    }

    requestAnimationFrame(renderLoopRAF);
}
```

That's not quite as smooth—10fps creates the sense of a slight mechanical movement—but it certainly has much less impact on the CPU (about 1/4th of the 60fps usage):



Efficient Animations JS sample	5.2%	39.8 MB	0 MB/s	0 Mbps

I encourage you to play around with variations on this theme to see what kind of interval you can actually discern with your eyes. 10fps and 15fps give a sense of mechanical movement; at 20fps I don't see much difference from 60fps at all, and the CPU usage is cut in half. You might also try something like 4fps (quarter-second intervals) to see the effect. In this chapter's companion content I've included a variation of the original sample where you can select from various target rendering rates.

The other thing you can do in the modified sample is draw the hour and minute hands at fractional angles. In the original code, the minute hand will move suddenly when the second hand comes around to the top. Analog clocks don't actually work this way: their complex gearing moves both the hour and the minute hand ever so slightly with every tick. To simulate that same behavior, we just need to include the seconds in the minutes calculation, and the resulting minutes in the hours, like so:

```
var second = date.getSeconds() + date.getMilliseconds() / 1000;
var minute = date.getMinutes() + second / 60;
var hour   = date.getHours() + minute / 60;
```

In real practice, you'd generally want to avoid just running a continuous animation loop like this: if there's nothing moving on the screen that needs animating (for which you might be using setInterval as a timer) and there are no input events to respond to, there's no reason to call requestAnimationFrame. Also, be sure when the app is paused that you stop calling requestAnimationFrame until the animation starts up again. (You can also use cancelAnimationFrame to stop one you've already requested.) The same is true for setTimeout and setInterval: don't generate unnecessary calls to your callback functions unless you really need to do the animation. For this, use the visibilitychange event to know if your app is visible on screen. While requestAnimationFrame takes visibility into account (the sample's CPU use will drop to 0% before it is suspended), you need to do this on your own with setTimeout and setInterval.

In the end, the whole point here is that really understanding the animation interval you need (that is, your frame rate) will help you make the best use of requestAnimationFrame, if that's needed, or setInterval/setTimeout. They all have their valid uses to deliver the right user experience with the least consumption of system resources.

Did you know? One change introduced with Windows 8 and Internet Explorer 10 (and thus supported in subsequent versions) is that setTimeout and setInterval, along with setImmediate, all support including parameters that you can pass to the callback functions.

What We've Just Learned

- In PC Settings (or the desktop control panel), users can elect to disable most (that is, nonessential) animations. Apps should honor this, as does WinJS, by checking the `Windows.UI.ViewManagement.UISettings.animationsEnabled` property.
- The WinJS animations library has many built-in animations that embody the Windows personality. These are highly recommended for apps to use for the scenarios they support, such as content and page transitions, selections, list manipulation, and others.
- All WinJS animations are built using CSS and thus benefit from hardware acceleration. When the right conditions are met, such animations run in the GPU and are thus not affected by activity on the UI thread.
- Apps can also use CSS animations and transitions directly, according to the W3C specifications.
- Apart from WinJS and CSS, apps can also use functions like `setInterval` and `requestAnimationFrame` to implement direct frame-by-frame animation. The `requestAnimationFrame` method aligns frames with the display refresh rate, leading to the best overall performance.

Chapter 15

Contracts

Some time ago I discovered a delightfully quirky comedy called *Interstate 60* that is full of delightfully quirky characters. One of them, played by Chris Cooper, is a former advertising executive who, having discovered he was terminally ill with lung cancer, decided to make up for a career built on lies by encouraging others to be more truthful. As such, he was very particular about agreements and contracts, especially those in writing.

We really get to see the character's quirkiness in a scene at a gas station. He's approached by a beggar with a sign, "Will work for food." Seeing this, he offers the man an apple in exchange for cleaning his car's windshield. But when the man refuses to honor the written contract on his sign, Cooper's character gets increasingly upset over the breach…to the point where he announces his terminal illness, rips open his shirt, and reveals the dynamite wrapped around his body and the 10-second timer that's already counting down!

In the end, he drives away with a clean windshield and the satisfaction of having helped someone—in his delightfully quirky way—to fulfill their part of a written contract. And he reappears later in the movie in a town that's 100% populated with lawyers; I'll leave it to you to imagine the result, or at least enjoy the film!

Setting the dynamite and impending threats of bodily harm aside—which have absolutely nothing to do with this chapter—agreements between two parties are exceptionally important in a well-running computer system just as they are in a civil society. Agreements are especially important where apps provide extensions to the system and where apps written by different people at different points in time cooperate to fulfill certain tasks.

Such is the nature of various contracts within Windows, which as a collection constitute perhaps one of the most powerful features of the entire system. The overarching purpose of contracts has been described as "launching apps for a purpose and with context." That is, instead of just starting apps in isolation, contracts make it possible to start them in relationship to other apps and in the context of those other apps. Information can then be shared seamlessly between those apps for a real purpose, rather than through the generic intermediary of the file system where such context is lost.

With any given contract, one party is the *consumer* or receiver of information involved in the contract. The other party is the *source* or *provider* of that information. The contract itself is generic: neither party needs any specific knowledge of the other, just knowledge of their side of the contract. It might not sound like much, but what this allows is a degree of extensibility that gets richer and richer as more apps that support contracts are added to the system. When users really start to experience what these contracts provide, they'll more and more look for and choose apps from the Windows Store that use contracts to enrich their system and create increasingly powerful user experiences.

Within the apps themselves, consuming contracts typically happens through an API call, such as the file pickers we've already see in Chapter 11, "The Story of State, Part 2," or is already built into the system through UI like the Charms bar. *Providing* information for a contract is often the more interesting part, because an app needs to respond to specific events (when running), or announce the capability through its manifest and then handle different contract activations.

The table below summarizes all the contracts and other extensions in Windows (in alphabetical order), some of which serve to allow apps to work together while others serve to allow apps to extend system functionality. Full descriptions for most of these can be found on App contracts and extensions. Those that are covered in this chapter are colored in green, namely share, search, URI scheme associations, contacts (people), and appointments—contracts that many apps will rely upon. We've also seen a number of contracts in previous chapters, and a few more will come along later. Also, the provider side of certain contracts, which are somewhat uncommon for apps to implement, can be found in Appendix D, "Provider-Side Contracts." The docs and samples picks up a couple of others that aren't covered in this book.

Tip For a comparison of the different options for exchanging data—the share contract, the clipboard, and the file save picker contract—refer to Sharing and exchanging data. This topic outlines different scenarios for each option and when you might implement more than one in the same app.

Also note that there are many WinRT events involved in these different contracts, so be mindful of the need to call `removeEventListener` as described in Chapter 3, "App Anatomy and Performance Fundamentals," in the section "WinRT Events and removeEventListener."

Contract/Extension	Provider	Consumer	Description, Documentation, and Samples
Account picture provider (Chapter 4)	Apps that can take a picture	Windows (PC Settings > Accounts > Your Account)	When user changes an account picture, they can either select an existing one or acquire a new one from a provider, see Account picture name sample.
Appointments	Windows (system calendar)	Apps that create and manage appointments	Provides the ability for apps to add, remove, and update appointments on the user's calendar, with popup UI that lets the user control each action. See Quickstart: Managing appointments and the Appointments API sample.
AutoPlay (Chapters 11, 17)	Apps that want to be listed as an AutoPlay option	Windows	See Auto-launching with AutoPlay and the Removable storage sample.
Background tasks (Chapters 16, 4)	Apps that have background tasks	Windows	Allows apps to run small tasks in the background (that is, when otherwise suspended or not running) without user interaction. See Introduction to Background Tasks (whitepaper) as well as Chapter 16. Background file transfers are a special case supported by specific APIs; see Transferring data in the background and Chapter 4.
Cached file updater (Appendix D)	Apps that provide access to their data through file pickers and want to synchronize updates	Apps using the file picker API and the file APIs to manage them	Provider apps allow the consumer to maintain a cached copy of a file, and the provider can manage updates between the local copy and the source copy. See Integrating with file picker contracts.
Camera settings (Chapter 13)	Apps with custom camera UI	Windows Camera Capture UI	See Developing Windows 8 device apps for cameras.
Contacts (via contact	Windows (through	Apps exposing	A system-provided UI through which the user can take

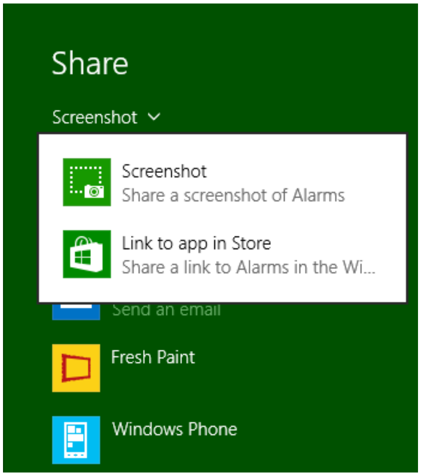
card)	the People app); other apps can handle contact card actions (Appendix D).	actions for contacts	specific actions on a contact, without the calling app needing to access contact information directly. See Managing contact cards and the Contact manager API sample.
Contact picker	Apps that manage contact data (like an address book)	Apps using the contact picker API (like email)	Launches an app to provide a list of possible contacts to select, providing that information to the app. See Managing user contacts.
File activation (file type association, Chapter 11)	Apps that can open files of a particular type	Windows Explorer and apps that use the launcher API	Launches an app to open/service a file when needed. See How to handle file activation and Auto-launching with file and URI associations.
File open/save picker, folder picker (Chapter 11 and Appendix D)	App with data that can appear as files to other apps for opening and/or saving (two separate contracts).	Apps using the file picker API (also certain Windows features)	Makes data that is otherwise hidden inside and managed by apps appear as if they were part of the file system. See Integrating with file picker contracts.
Game explorer (not covered)	Game apps with a Game Definition File (GDF)	Windows (parental controls)	Manages age ratings for games. See Creating a GDF file.
Play To (Chapter 13)	Apps that can play media to a DLNA device	Windows (Devices charm > Connect)	See Streaming media to devices using Play To.
Print task settings (Chapter 17)	Printer device apps	Windows (Device charm > Print)	See Developing Windows 8 device apps for printers.
Protocol activation (URI scheme association)	Apps that can open URIs that begin with a particular URI scheme	Windows Explorer and apps that use the launcher API	Launches an app to open/service a URI when needed. See How to handle protocol activation and Auto-launching with file and URI associations.
Search	Apps with searchable data	Windows (Search charm)	Provides the ubiquitous ability to search any app from anywhere. See Adding search to an app.
Settings (Chapter 10)	Apps with settings	Windows (Settings charm)	Provides a standard place for app settings. See Adding app settings.
Share	Apps with sharable data	Apps that can receive data to incorporate into itself or share to a service	Provides a linkage of data transfer between apps so that source apps don't need to be particularly aware of individual targets like Facebook, Twitter, etc. See Adding share.
SSL/certificates (not covered)	Apps that need to install a certificate	Apps needing to supply a certificate to another service	See Encrypting data and working with certificates.

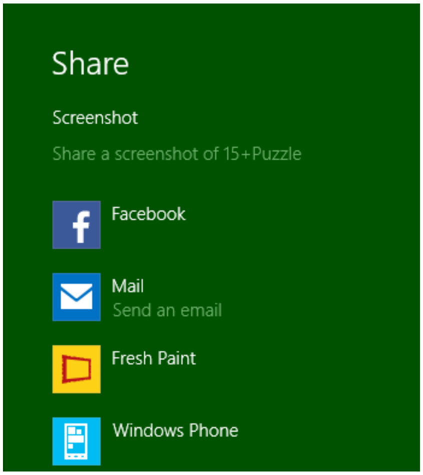
Share

Though Search appears at the top of the Charms bar, the first contract I want to look at in depth is Share—after all, it's one of the first things you learn as a child! In truth, I'm starting with Share because we've already seen the source side of the story starting back in Chapter 2, "Quickstart," with the Here My Am! app, and our coverage here will also include a brief look at the age-old clipboard at the end of this section.

Let's review the basic process of Share and its user experience. Note that all object classes referred to here come from the `Windows.ApplicationModel.DataTransfer` namespace unless noted.

- First, if the Share charm is invoked for an app that doesn't participate in the contract at all, the charm provides options to share a screenshot or a link to the app's page in the Store (if it's published), as shown below left. Screenshots can be disabled for protected content—see "Sidebar: Disabling Screen Capture" at the end of this section. If an app is not yet published and doesn't yet have Share features, such as my game called 15+Puzzle that I've been working on while writing this book, it appears as shown below right:

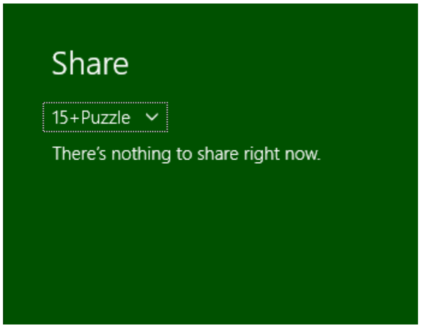




- To share content, an app listens for the [datarequested]() event from the object returned by `DataTransferManager.getForCurrentView()`. This WinRT event (for which you should be mindful of using `removeEventListener`) is fired whenever the user invokes the Share charm.

- In its `datarequested` handler, the app determines whether it has anything to share *at the moment*. If it doesn't—for example, the app shares selected content but currently has no selection—it just returns from the handler:

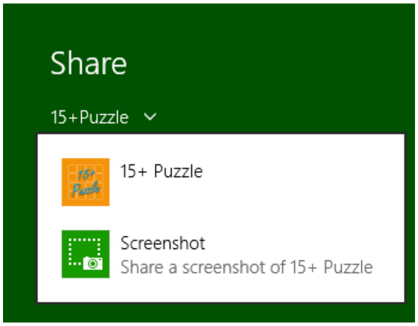
```
var dtm =
    Windows.ApplicationModel.DataTransfer.DataTransferManager.getForCurrentView();
dtm.addEventListener("datarequested", shareHandler); //Remove this later!

function shareHandler (e) {
    //Nothing to share right now
}
```



Share

15+Puzzle

There's nothing to share right now.



Share

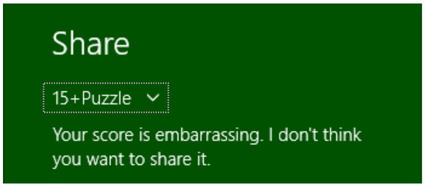
15+Puzzle

15+ Puzzle

Screenshot
Share a screenshot of 15+ Puzzle

You can customize the message by calling the `eventArgs.request.failWithDisplayText` method:

```
function shareHandler (e) {
    e.request.failWithDisplayText(
        "Your score is embarrassing. I don't think you want to share it.");
}
```



Share

15+Puzzle

Your score is embarrassing. I don't think you want to share it.

(And no, my real app in the Store doesn't do this, but it was worth a little laugh!)

- If the source app *does* have data to share, it populates the `DataPackage` object provided in the event args (a [`DataRequestedEventArgs`](#) object whose `request.data` property is the package). It specifically uses the package's `set*` methods, like `setHtmlFormat`, to provide whatever formats are applicable. You'll typically want to share as many formats as you can, to increase the number of potential targets. In the Here My Am! app, for example, we share text and images together (see pages/home/home.js at the end of the file):

```
function provideData(e) {
    var request = e.request;
    var data = request.data;

    if (!lastPosition || !lastCapture) {
        return;  //Nothing to share, so exit
    }

    data.properties.title = "Here My Am!";
    data.properties.description = "At ("
        + app.sessionState.lastPosition.latitude +
        ", " + app.sessionState.lastPosition.longitude + ")";
    //When sharing an image, include a thumbnail
    var streamReference =
        Windows.Storage.Streams.RandomAccessStreamReference.createFromFile(lastCapture);
    data.properties.thumbnail = streamReference;

    data.setStorageItems([lastCapture]);
    data.setBitmap(streamReference);
}
```

- Based on the data formats in the package, Windows—that is, the *share broker* that manages the contract—determines the share target apps to display to the user (as in the first image above). The user can also control which apps are shown via PC Settings > Search and Apps > Share.
- When the user picks a target, the associated app is activated in a partial overlay and receives the data package to process however it wants. Because the target is activated in an overlay, the source app remains on the screen so that the user always remains aware of that original context.

The more complete sequence of events between the source app, the share broker, and the target app is shown in Figure 15-1.

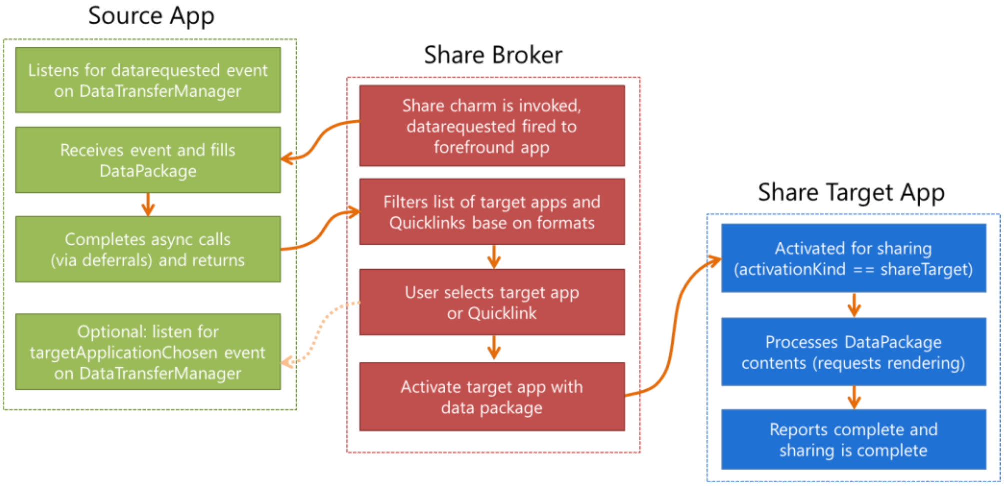

Source App
Listens for datarequested event on DataTransferManager
Receives event and fills DataPackage
Completes async calls (via deferrals) and returns
Optional: listen for targetApplicationChosen event on DataTransferManager
Share Broker
Share charm is invoked, datarequested fired to forefround app
Filters list of target apps and Quicklinks base on formats
User selects target app or Quicklink
Activate target app with data package
Share Target App
Activated for sharing (activationKind == shareTarget)
Processes DataPackage contents (requests rendering)
Reports complete and sharing is complete

FIGURE 15-1 Processing the Share contract as initiated by the user's selection of the Share charm.

This whole process provides a very convenient shortcut for users to take something they love in one app and get it into another app with a simple edge gesture and target app selection, without leaving the context of the source app. It's like a semantically rich clipboard in which you don't have to figure out how to get connected to other apps. What's very cool about the Share contract, in other words, is that the source doesn't have to care what happens to the data—its only role is to provide whatever data is appropriate for sharing at the moment the user invokes the Share charm (if, in fact, there is appropriate data—sometimes there isn't). This liberates source apps from the burden of having to predict, anticipate, or second-guess what users might want to do with the data (though there's nothing wrong with providing dedicated controls for specific scenarios). Perhaps they want to email it, share it via social networking, drop it into a content management app…who knows?

Well, only the user knows, so what the share broker does with that data is let the user decide! Given the data package from the source, the broker matches the formats in that package to target apps that have indicated support for those formats in their manifests. The broker then displays that list to the user. That list can contain apps, for one, but also something called a *quicklink* (a ShareTarget.Quicklink object, to be precise), which is serviced by some app but is much more specific. For instance, when an email app is shown as an option for sharing, the best it can do is create a new message with no particular recipients. A quicklink, however, can identify specific email addresses, say, for a person or persons you email frequently. The quicklink, then, is essentially an app plus specific configuration information.

Whatever the case, some app is launched when the user selects a target. With the Share contract, the app is launched with an activation kind of shareTarget. This tells it to *not* bring up its default UI but to rather show a specific share pane (with light-dismiss behavior) in which the user can refine exactly what is being shared and how. A share target for a social network, for instance, will often provide a place to add a comment on the shared data before posting it. An email app would provide a means to edit the message before sending it. A front-end app for a photo service could allow for adding a caption, specifying a location, identifying people, and so on. You get the idea. All of this combines together to provide a smooth flow from having something to share to an app that facilitates the sharing and allows the user to add customizations.

Overall, then, the Share contract gets apps connected to one another for this common purpose without any of them having to know anything about the others. This creates a very extensible and scalable experience: because all the potential target choices appear only in the Share charm pane, they never need to clutter a source app as we see happening on many web pages. This is the "content before chrome" design principle in action. (Though you might still implement specific sharing scenarios directly within the app for purposes other than general sharing, such as a share-to-Facebook function that rewards the user with some in-app currency.)

Source apps also don't need to update themselves when a new target becomes popular (e.g., a new kind of social network): all that's needed is a single target app. As for those target apps, they don't have to evangelize themselves to the world: through the contract, source apps are automatically able to use any target apps that come along in the future. And from the end user's point of view, their experience of the Share charm gets better and better as they acquire more Share-capable apps.

At the same time, it is possible for the source app to know something about how its shared data is being used. Alongside the datarequested event, the DataTransferManager also fires a targetApplicationChosen event to those sources who are listening. The eventArgs in this case contain only a single property: applicationName. This isn't really useful for any other WinRT APIs, mind you, but is something you can tally within your own telemetry. Such data can help you understand whether you'd provide a better user experience by sharing richer data formats, for example, or, if common target apps also support custom formats that you can support in future updates.

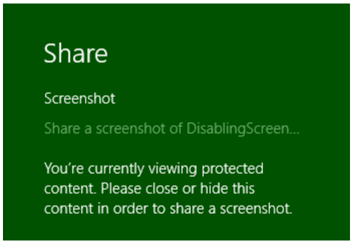
Sidebar: Disabling Screen Capture

Some apps, to protect sensitive or rights-managed information, for example, need to disable screen capture both through the default Share charm and through the Alt+Print Screen and Windows+Print Screen keyboard combinations.

Screen capture is controlled through the ApplicationView.isScreenCaptureEnabled property. Setting this to false will disable screen capture, as you'd do when a rights-protected email message is visible:

```
Windows.UI.ViewManagement.ApplicationView.getForCurrentView().isScreenCaptureEnabled
    = false;
```

If you invoke the Share charm with capture disabled, you'll see this message:



Share

Screenshot

Share a screenshot of DisablingScreen...

You're currently viewing protected content. Please close or hide this content in order to share a screenshot.

Setting isScreenCaptureEnabled to true will reenable capture, as you'd do when protected content is no longer visible. Both scenarios—which constitute only a single line of code each!—can be found in the Disabling screen capture sample.

Share Source Apps

Let's complete our understanding of source apps by looking at a number of details we haven't fully explored yet, primarily around how the source populates the data package and the options it has for handling the request. For this purpose, I suggest you obtain and run the Sharing content source app sample and the Sharing content target app sample. We'll be looking at both of these, and the latter provides a helpful way to see how a target app consumes the data package created in the source.

The source app sample provides a number of scenarios that demonstrate how to share different types of data. It also shows how to programmatically invoke the Share charm—this isn't typically recommended, but it is possible:

```
Windows.ApplicationModel.DataTransfer.DataTransferManager.showShareUI();
```

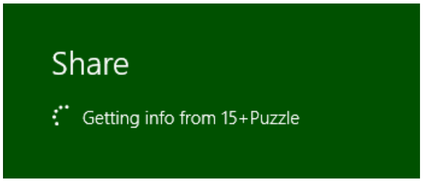
Calling this will, as when the user invokes the charm, trigger the datarequested event where eventArgs.request object is again a DataRequest object, which contains two properties and two methods:

- data is the DataPackage to populate. It contains methods to make various data formats available, though it's important to note that not all formats will be immediately rendered. Instead, they're rendered only when a share target asks for them. Equally important is the metadata that you configure for the package through data.properties.

- deadline is a Date property indicating the time in the future when the data you're making available will no longer be valid (that is, will not render). This recognizes that there might be an indeterminate amount of time between when the source app is asked for data and when the target actually tries to use it. With delayed rendering, as noted above for the data property, it's possible that some transient source data might disappear after some time, such as when it's just part of a cache that the source is managing. By indicating that time in deadline, rendering requests that occur past the deadline will be ignored.

- failWithDisplayText, as mentioned earlier, is a method to tell the share broker that sharing isn't possible right now, along with a string that will tell the user why (perhaps the lack of a usable selection). You call this when you don't have appropriate data formats or an appropriate selection to share, or if there's an error in populating the data package for whatever reason. The text you provide will then be displayed in the Share charm (and thus should be localized). Scenario 8 of the source app sample shows the use of this in the simple case when the app doesn't provide data in response to the datarequested event.

- getDeferral provides for async operations you might need to perform while populating the data package (just like other deferrals elsewhere in the WinRT API). That is, once you return from datarequested, the Share charm assumes you've populated the package; by retrieving the deferral object, the charm will wait until you call the deferral's complete method. During this time it will display a progress ring:



Share

Getting info from 15+Puzzle

Tip Be mindful to always complete the deferral if you use it and to thoroughly test your sharing code to ensure that your handler always returns. If your datarequested handler crashes, does not return, or fails to complete a deferral, this Share charm message turns into a spinning donut of death (that is, the graphic above will last forever!), which your users will likely report in your ratings.

The basic structure of a datarequested handler, then, attempts to populate data formats through request.data.set*, populates the necessary metadata fields of request.data.properties, and calls eventArgs.request.failWithDisplayText when errors occur. We see this structure in most of the scenarios in the sample, which I've generalized here:

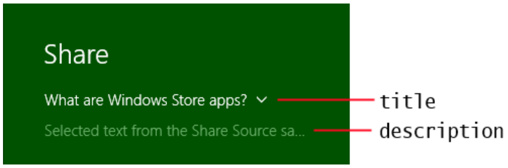
```
var dataTransferManager =
    Windows.ApplicationModel.DataTransfer.DataTransferManager.getForCurrentView();

// Remove this listener as required
dataTransferManager.addEventListener("datarequested", dataRequested);

function dataRequested(e) {
    var request = e.request;

    // Assume variables like shareTitle, shareDescription, etc., are defined elsewhere.

""    if ( /* Check if there is appropriate data to share */ ) {
        // Populate desired metadata. title is required.
        request.data.properties.title = shareTitle;
""        request.data.properties.description = shareDescription;

        // Highly recommended to set app links
            request.data.properties.contentSourceApplicationLink =
                new Windows.Foundation.Uri(scenario.applink);
        });

        // Call request.data.setText, setUri, setBitmap, setData, etc.
        // Request a deferral if async work is necessary.
        request.data.setText(shareText);
    } else {
        request.failWithDisplayText(/* Error message */ );
    }
}
```

The following sections explore the details of the different aspects of the handler: metadata, populating data formats, and deferrals.

Populating Metadata

Even though the real "goods" in a Share operation are stored in a data package through the `set*` methods, how you *describe* that information through metadata is very important for how the data is consumed, how it appears in the system UI, and how that data refers back to the source app. The latter is especially important because it can help you acquire new users when the shared data ultimately goes out beyond the local device via email, via sharing to social networks, and so forth.

All metadata for Share is populated through fields of the `request.data.properties` object, which is a [`DataPackagePropertySet`](). This object is technically a `PropertySet` and thus has methods like `first`, `lookup`, and `remove` to support custom properties (when using custom formats) and allows for future extensibility. We'll talk about custom formats a little later. What's important first are the named fields in `request.data.properties`. These fall into two groups—descriptive/UI properties and app reference properties—as detailed in the following table.

Descriptive/UI Properties	
`title`	(Required) The text that appears at the top of the Share flyout:  Share What are Windows Store apps? Selected text from the Share Source sa...
`description`	The text that appears below the title in the Share flyout (see image above). This should describe the contents of the package.
`thumbnail`	A stream containing a thumbnail image; obtaining this image is typically why you'd use the `request.getDeferral` method. The target app can choose to display this as a preview instead of using the full image data from the package. The Reading List app, for example, uses an image because the thumbnail is typically too small.
`square30x30Logo`	Sets the source app's logo that the target app can use to identify the source. (The Share charm works with the source's logo from its manifest.)
`logoBackgroundColor`	A `Color` object describing the color to use for the logo background; defaults to the source app's background color in the manifest (Visual Assets).
`fileTypes`	A string vector, where strings should come from the `StandardDataFormats` enumeration but can also be custom formats.
`size`	The number of items when the data in the package is a collection, e.g., files.
App Reference Properties	
`applicationName`	A string that helps target apps gather the same kind of telemetry information that the source app can obtain from the `targetApplicationChosen` event. You can set this property directly (if you want to attribute a different source than the app) or use the value from `Windows.ApplicationModel.Package.current.id.name`.
`packageFamilyName`	The exact identifier of the source app as used in the Windows Store, which is set automatically to the sane value as `Windows.ApplicationModel.Package.current.id.familyName`. It allows a target app wishing to activate the source app to provide this identifier as a fallback when attempting to launch the `contentSourceApplicationLink`, such that the user can go acquire the source app from the Store.
`applicationListingUri`	The URI of your app's page in the Windows Store, which should be set to the value from `Windows.ApplicationModel.Store.CurrentApp.linkUri`. This is a very simple way to increase your app's visibility, so don't skip this step!
`contentSourceApplicationLink`	A `Windows.Foundation.Uri` object that provides a deep link into the exact location of the source data in the source app.
`contentSourceWebLink`	A `Windows.Foundation.Uri` object that provides a web link to the content that's being shared, as when the content is hosted in a webview.

The relationships and uses of the reference properties deserve a little more explanation. In the past, sharing through mechanisms like the clipboard has been a one-way process: once data leaves the source app, it loses its relationship with that source (with the exception of sharing a URI with a custom scheme). The overall purpose of these reference properties is to maintain that connection between the data and its source, even though the data might go far afield even beyond the immediate target app. For example, a target that shares to a social network can post the shared data and provide one of the links back to the source app or the web content that the source app was hosting. This invites consumers of that data, wherever they encounter it, to navigate back to the source app, possibly acquiring the app from the Store along the way.

The `applicationListingUri`, then, clearly provides a link to the source app in the Store. Truly, you should always set this, because it's one line of code that can very much help you acquire new users. Of course, until you upload an app to the Store for the first time, you won't know this URI, which is why you should just assign the value from `Windows.ApplicationModel.Store.CurrentApp.linkUri`:

```
e.request.data.properties.applicationListingUri =
   Windows.ApplicationModel.Store.CurrentApp.linkUri;
```

Tip If you're using the `CurrentAppSimulator` during development, use that object in place of `CurrentApp` in this code snippet. It's best, actually, to have a single app variable that contains either `CurrentApp` or `CurrentAppSimulator` so that you can change it in one place prior to onboarding to the Store. In any case, when using the `CurrentAppSimulator` (see Chapter 20, "Apps for Everyone, Part 2,"), the `linkUri` property will return whatever is in the `<LinkUri>` element of your WindowsStoreProxy.xml file.

The `contentSourceApplicationLink` and `contentSourceWebLink` properties are more specific. Their purpose is to provide direct links to where the content originally came from, be it a section of an app or a website, respectively.

For example, if you share from an RSS reader app, set `contentSourceApplicationLink` to a URI that will activate the app to navigate directly back to that page at a later time. Navigating to an app, of course, means that the URI is something *other* than `http[s]`, typically a custom scheme. As a result, this requires that the app implements protocol activation, as we'll discuss later in this chapter under "Protocol Activation." The Sharing content source app sample uses the code below to create deep links to the specific scenario page from which data is shared (this example is from js/text.js):

```
SdkSample.scenarios.forEach(function (scenario) {
   if (scenario.url === "/html/text.html") {
      request.data.properties.contentSourceApplicationLink =
         new Windows.Foundation.Uri(scenario.applink);
   }
});
```

Whatever app ultimate consumes this link will likely want to activate it using `Windows.System.Launcher.launchUriAsync`. In that call, the `LauncherOptions` argument can contain a preferredApplicationPackageFamilyName, which would be set to the packageFamilyName property from the source app. As a result, if there's no other app handling the URI scheme already, the user will be invited to install that original source app.

If your app hosts and shares web content in a webview element, things are a little different. Here you set the contentSourceWebLink to the appropriate http[s] URI. If you remember from Chapter 4, "Web Content and Services," in the section "Capturing Webview Content," the webview element allows you to retrieve a DataPackage for whatever content is selected within it (using the captureSelected-ContentToDataPackageAsync method. You can assign this package directly to request.data within the datarequested event if you want to share straight through, or you can copy data from this package to the one provided in request.data. (And of course, if you use an async API like this within the event handler, you'll need to use the deferral object as well.)

I'd refer you to scenario 7 of the HTML webview control sample in the SDK to see a bit of this, but unfortunately it doesn't share the selection within a webview (just its URI and a bitmap), and it does two other things incorrectly. It fails to use the deferral when making an async call, and it uses the package's deprecated setUri method to set a data type instead of setting the metadata field contentSourceWebLink. To address these shortcomings, scenario 4 of the Webview extras example in Chapter 4's companion content provides a more complete and accurate demonstration, specifically sharing the selected content from the webview, or a bitmap if there is no selection (js/scenario4.js):

```
//Wrap the webview's capture methods promises.
function getWebviewSelectionAsync(webview) {
    return new WinJS.Promise(function (cd, ed) {
        var op = webview.captureSelectedContentToDataPackageAsync();
        op.oncomplete = function (args) { cd(args.target.result); };
        op.onerror = function (e) { ed(e); };
        op.start();
    });
}

function getWebviewBitmapAsync(webview) {
    return new WinJS.Promise(function (cd, ed) {
        var op = webview.capturePreviewToBlobAsync();

        op.oncomplete = function (args) {
            var ras = Windows.Storage.Streams.RandomAccessStreamReference;
            var bitmapStream = ras.createFromStream(args.target.result.msDetachStream());
            cd(bitmapStream);
        };

        op.onerror = function (e) { ed(e); };
        op.start();
    });
}


function dataRequested(e) {
    var webview = document.getElementById("webview");
    var dataPackage = e.request.data;
```

```
//Obtain a deferral
var deferral = e.request.getDeferral();

//Set the data package's properties.  These are displayed within the Share UI
//to give the user an indication of what is being shared.  They can also be
//used by target apps to determine the source of the data.
dataPackage.properties.title = webview.documentTitle;
dataPackage.properties.description = "Content shared from Webview";
dataPackage.properties.applicationName = "Webview Extras Example";

//Web link is the same as the webview's source URI.
dataPackage.properties.contentSourceWebLink = new Windows.Foundation.Uri(webview.src);

//To support app links in Share, the app typically uses protocol activation with
//a custom protocol.
var applink = "progwin-js-webviewextras:navigate?page=ShareWebview";
dataPackage.properties.contentSourceApplicationLink = new Windows.Foundation.Uri(applink);

// Set the data being shared from the webview's selection, or else use the whole webview.
getWebviewSelectionAsync(webview).then(function (selectionPackage) {
    if (selectionPackage != null) {
        //There's a selection, so use that as the data package. First copy the key
        //properties from the original package to the new one.
        var props = ["title", "description", "applicationName", "contentSourceWebLink",
            "contentSourceApplicationLink"];

        for (var i = 0; i < props.length; i++ ) {
            selectionPackage.properties[props[i]] = dataPackage.properties[props[i]];
        }

        //Now provide the webview's package as a whole for the data
        e.request.data = selectionPackage;

        //We return a promise to make a chain; in this case we just return a Boolean
        //indicating what was rendered (true for selection).
        return WinJS.Promise.as(true);
    } else {
        //With no selection, render the whole webview and provide its URI as text.
        dataPackage.setText(webview.src);
        dataPackage.setUri(new Windows.Foundation.Uri(webview.src));

        return getWebviewBitmapAsync(webview).then(function (bitmapStream) {
            dataPackage.setBitmap(bitmapStream);
            return false;
        });
    }
}).done(function (selectionRendered) {
    //Be sure to complete the deferral on success or error either way
    WinJS.log && WinJS.log("Selection rendered: " + selectionRendered, "app");
    deferral.complete();
}, function (e) {
    deferral.complete();
});
```

```
}
```

> **Note** Setting the contentSourceApplicationLink and contentSourceWebLink properties have different semantics than populating the data package with the *ApplicationLink* or *WebLink* data *formats*, which we'll meet in the next section. Specifically, when the only data you're sharing is a link itself, that's when you use the link formats. If you're sharing other content from an app page or a webview, that's when you use formats like text and HTML and set the contentSource*Link properties accordingly, as the code above demonstrates.

Populating Data Formats

Populating metadata to describe the shared content is all well and good, but then we cannot forget about placing content in the data package itself! This is done in your datarequested handler by calling the various set* methods in the DataPackage object for as many formats as you can provide. Supporting more formats will enable more potential targets and thus the likelihood that data from your app will be shared. This includes calling setData for custom formats and setDataProvider for deferred rendering, which are described in the two sections that follow this one.

For standard formats identified by values in the StandardDataFormats enumeration, there are discrete methods: setText, setHtmlFormat, setApplicationLink, setWebLink, setBitmap, and setStorageItems (for files and folders), and setRtf.[107] All of these except for setRtf are represented in the Sharing content source app sample as follows.

Sharing text—scenario 1 (js/text.js):

```
var dataPackageText = document.getElementById("textInputBox").value;
request.data.setText(dataPackageText);
```

Sharing links—scenarios 2 and 3 (js/weblink.js and js/applicationlink.js), used for remote content and deep linking into the app, respectively:

```
request.data.setWebLink(
    new Windows.Foundation.Uri(document.getElementById("weblinkInputBox").value));

var dataPackageApplicationLink = document.getElementById("selectionList").value;
request.data.setApplicationLink(new Windows.Foundation.Uri(dataPackageApplicationLink));
```

Be mindful that when using *WebLink* or *ApplicationLink* formats, you're saying that the link itself *is* the whole content. This is different from using the package's contentSourceApplicationLink and contentSourceWebLink properties to indicate the source (app page or web page, respectively) of the content being shared. In other words, the link *formats* are generally mutually exclusive with their associated contentSource*Link properties. It is, however, certainly possible to share a *WebLink* format

[107] RTF stands for rich text format, a comparably ancient and somewhat uncommon precursor to HTML. There is also the setUri method that is deprecated in favor of setApplicationLink and setWebLink.

and set a contentSourceApplicationLink to enable deep linking back to the page in the app (or share both *WebLink* and *ApplicationLink* formats).

Sharing an image and a storage item—scenario 4 (js/image.js):

```
var imageFile; // A StorageFile obtained through the file picker

// In the data requested event
var streamReference =
   Windows.Storage.Streams.RandomAccessStreamReference.createFromFile(imageFile);
request.data.properties.thumbnail = streamReference;

// It's recommended to always use both setBitmap and setStorageItems for sharing a
// single image since the Target app may only support one or the other

// Put the image file in an array and pass it to setStorageItems
request.data.setStorageItems([imageFile]);

// The setBitmap method requires a RandomAccessStreamReference
request.data.setBitmap(streamReference);
```

Sharing files—scenario 5 (js/file.js):

```
var selectedFiles; // A collection of StorageFile objects obtained through the file picker

// In the data requested event
request.data.setStorageItems(selectedFiles);
```

As for sharing HTML, this isn't quite as simple as calling setHtmlFormat with a bit of HTML. The string must be formatted to include a few extra headers that a target app (or the clipboard) requires. (Refer back to Figure 4-3 in Chapter 4 to see what those headers look like.)

For this purpose you might find the DataTransfer.HtmlFormatHelper object, well, helpful—it provides methods to build such properly formatted markup. Specifically, its createHtmlFormat method takes whatever bit of HTML you want to share and gives it the necessary headers:

```
var htmlString = "<p>A big hi hello to all <em>intelligent</em> life out there...</p>"
var shareString =
   Windows.ApplicationModel.DataTransfer.HtmlFormatHelper.createHtmlFormat(htmlString);
request.data.setHtmlFormat(shareString);
```

What's also true with HTML is that it often refers to other content like images that aren't directly contained in the markup. So how do you handle that? Fortunately, the designers of this API thought through this need: you employ the data package's resourceMap property to associate relative URIs in the HTML with an image stream. We see this in scenario 7 of the sample (js/html.js):

```
var path = document.getElementById("htmlFragmentImage").getAttribute("src");
var imageUri = new Windows.Foundation.Uri(path);
var streamReference =
    Windows.Storage.Streams.RandomAccessStreamReference.createFromUri(imageUri);
request.data.resourceMap[path] = streamReference;
```

The other interesting part of scenario 7 is that it replaces the data package in the eventArgs with a new one that it creates as follows:

```
var range = document.createRange();
range.selectNode(document.getElementById("htmlFragment"));
request.data = MSApp.createDataPackage(range);
```

As you can see, the MSApp.createDataPackage method takes a DOM range (in this case a portion of the current page) and creates a data package from it, where the package's setHtmlFormat method is called in the process (which is why you don't see that method called explicitly in scenario 7). For what it's worth, there is also MSApp.createDataPackageFromSelection, which does the same job with whatever is currently selected in the DOM. You would obviously use this if you have editable elements on your page from which you'd like to share.

Also, as noted in the previous section, the webview element's captureSelectedContentToDataPackageAsync method makes it simple to extract content from a webview for use with the Share contract. In this case, any HTML format that's contained in the package from the webview already has the necessary headers, which is what's shown in Figure 4-3 of Chapter 4.

Custom Data Formats: schema.org

Long ago, I imagine, API designers decided it was an exercise in futility to try to predict every data format that apps might want to exchange in the future. The WinRT API is no different, so alongside the format-specific set* methods of the DataPackage we find the generic setData method. This takes a format identifier (a string) and the data to share. This is illustrated in scenario 8 of the Sharing content source app sample using the format "http://schema.org/Book" and data in a JSON string (js/custom.js):

```
request.data.setData(dataFormat, JSON.stringify(book));
```

Because the custom format identifier is just a string, you can literally use anything you want here; a very specific format string might be useful, for example, in a sharing scenario where you want to target a very specific app, perhaps one that you authored yourself. However, unless you're very good at evangelizing your custom formats to the rest of the developer community (and have a budget for such!), chances are that other share targets won't have any clue what you're talking about.

Fortunately, there is a growing body of conventions for custom data formats maintained by http://schema.org. This site is the point of agreement where custom formats are concerned, so we highly recommend that you draw formats from it. See http://schema.org/docs/schemas.html for a complete list.

Here's the JSON book data used in the sample:

```
var book = {
    type: "http://schema.org/Book",
    properties: {
        image: "http://sourceuri.com/catcher-in-the-rye-book-cover.jpg",
        name: "The Catcher in the Rye",
        bookFormat: "http://schema.org/Paperback",
        author: "http://sourceuri.com/author/jd_salinger.html",
        numberOfPages: 224,
        publisher: "Little, Brown, and Company",
        datePublished: "1991-05-01",
        inLanguage: "English",
        isbn: "0316769487"
    }
};
```

You can easily express this same data as plain text, as HTML (or RTF), as a link (perhaps to a page with this information), and an image (of the book cover). This way you can populate the data package with all the standard formats alongside specific custom formats.

Deferrals and Delayed Rendering

Deferrals, as mentioned before, are a simple mechanism to delay completion of the `datarequested` event until the deferral's `complete` method is called within an async operation's completed handler. The documentation for [`DataRequest.getDeferral`]() shows an example of using this when loading an image file:

```
var deferral = request.getDeferral();

Windows.ApplicationModel.Package.current.installedLocation.getFileAsync(
    "images\\smalllogo.png")
    .then(function (thumbnailFile) {
        request.data.properties.thumbnail = Windows.Storage.Streams.
            RandomAccessStreamReference.createFromFile(thumbnailFile);
        return Windows.ApplicationModel.Package.current.installedLocation.getFileAsync(
            "images\\logo.png");
    })
    .done(function (imageFile) {
        request.data.setBitmap(
            Windows.Storage.Streams.RandomAccessStreamReference.createFromFile(imageFile));
        deferral.complete();
    });
```

Another example was shown with the webview example in the "Populating Metadata" section earlier.

Delayed *rendering* is a different matter, though the process typically employs the deferral. The purpose here is to avoid rendering the shared data until a target actually requires it, referred to as a *pull operation*. The `set*` methods that we've seen so far all copy the full data into the package. Delayed rendering means calling the data package's [`setDataProvider`]() method with a data format identifier and a function to call when and if the data is needed. Here's how it's done in scenario 6 of the [Sharing content source app sample]() where `imageFile` is selected with a file picker (js/delay-render.js):

```
// When sharing an image, don't forget to set the thumbnail for the DataPackage
var streamReference =
    Windows.Storage.Streams.RandomAccessStreamReference.createFromFile(imageFile);
```

```
request.data.properties.thumbnail = streamReference;
request.data.setDataProvider(
    Windows.ApplicationModel.DataTransfer.StandardDataFormats.bitmap,
    onDeferredImageRequested);
```

As indicated in the comments, it's a really good idea to provide a thumbnail with delayed rendering so that the target app has something to show the user. Then, when the target needs the full data, the data provider function gets called—in this case, onDeferredImageRequsted—where we see a good flashback to the encoding processes we learned about in Chapter 13, "Media":

```
function onDeferredImageRequested(request) {
    if (imageFile) {
        // Here we provide updated Bitmap data using delayed rendering
        var deferral = request.getDeferral();

        var imageDecoder, inMemoryStream;

        imageFile.openAsync(Windows.Storage.FileAccessMode.read).then(function (stream) {
            // Decode the image
            return Windows.Graphics.Imaging.BitmapDecoder.createAsync(stream);
        }).then(function (decoder) {
            // Re-encode the image at 50% width and height
            inMemoryStream = new Windows.Storage.Streams.InMemoryRandomAccessStream();
            imageDecoder = decoder;
            return Windows.Graphics.Imaging.BitmapEncoder.createForTranscodingAsync(
                inMemoryStream, decoder);
        }).then(function (encoder) {
            encoder.bitmapTransform.scaledWidth = imageDecoder.orientedPixelWidth * 0.5;
            encoder.bitmapTransform.scaledHeight = imageDecoder.orientedPixelHeight * 0.5;
            return encoder.flushAsync();
        }).done(function () {
            var streamReference = Windows.Storage.Streams.RandomAccessStreamReference
                .createFromStream(inMemoryStream);
            request.setData(streamReference);
            deferral.complete();
        }, function (e) {
            // didn't succeed, but we still need to release the deferral to avoid
            //a hang in the target app
            deferral.complete();
        });
    }
}
```

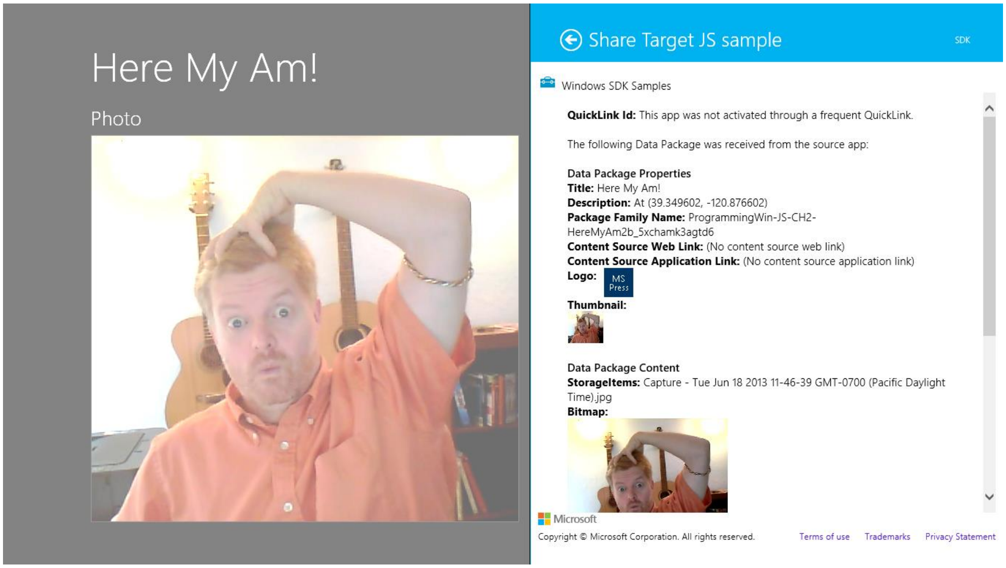
The *request* argument passed to this function is a simplified hybrid of the DataRequest and DataPackage objects called a DataProviderRequest. This contains a deadline property (with the same meaning as in DataRequest), a formatId property, a getDeferral method, and a setData method through which you provide the data that matches formatId.

Note that the sample here doesn't actually check request.formatId before proceeding because it calls setDataProvider only once for a single format. If you make multiple calls to setDataProvider with different formats and use the same handler, be sure to check formatId and render the proper data. Of course, you can use discrete handlers for each format.

Share Target Apps

Looking back to Figure 15-1, we can see that while the interaction between a source app and the share broker is driven by the single `datarequested` event, the interaction between the broker and a target app is a little more involved. For one, the broker needs to determine which apps can potentially handle a particular data package, for which purpose each target app includes appropriate details in its manifest. When an app is selected, it gets launched with an activation kind of `shareTarget`, in response to which it should show a specific share UI rather than the full app experience.

Let's see how all this works with the Sharing content target app sample, whose appearance is shown in Figure 15-2 (borrowing from Figure 2-20 we saw ages ago). Be sure to load this app in Visual Studio and run it once so that it's effectively installed and it will appear on the list of apps when we invoke the Share charm.



QuickLink Id: This app was not activated through a frequent QuickLink.

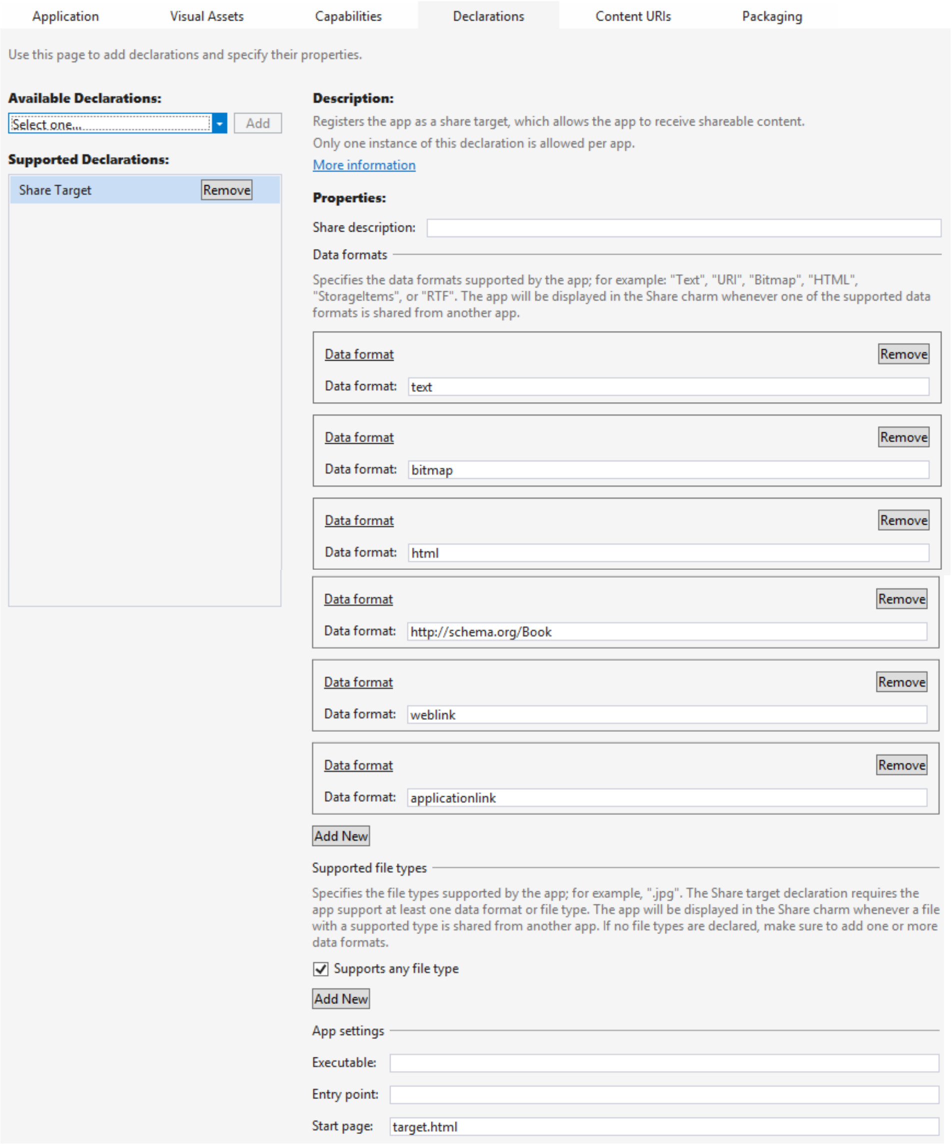
The following Data Package was received from the source app:

Data Package Properties
Title: Here My Am!
Description: At (39.349602, -120.876602)
Package Family Name: ProgrammingWin-JS-CH2-HereMyAm2b_5xchamk3agtd6
Content Source Web Link: (No content source web link)
Content Source Application Link: (No content source application link)
Logo: MS Press
Thumbnail:

Data Package Content
StorageItems: Capture - Tue Jun 18 2013 11-46-39 GMT-0700 (Pacific Daylight Time).jpg
Bitmap:

FIGURE 15-2 The appearance of the Sharing content target app sample (the right-hand nonfaded part).

The first step for a share target is to declare the data formats it can accept in the Declarations section of its manifest, along with the page that will be invoked when the app is selected as a target. As shown in Figure 15-3, the target app sample declares it can handle text, URI, bitmap, html, and the http://schema.org/Book formats, and it also declares it can handle whatever files might be in a data package (you can indicate specific file types here). Way down at the bottom it then points to target.html as its Share target page.



Use this page to add declarations and specify their properties.

Available Declarations:

Select one... Add

Supported Declarations:

Share Target Remove

Description:

Registers the app as a share target, which allows the app to receive shareable content.

Only one instance of this declaration is allowed per app.

More information

Properties:

Share description:

Data formats

Specifies the data formats supported by the app; for example: "Text", "URI", "Bitmap", "HTML", "StorageItems", or "RTF". The app will be displayed in the Share charm whenever one of the supported data formats is shared from another app.

Data format Remove
Data format: text

Data format Remove
Data format: bitmap

Data format Remove
Data format: html

Data format Remove
Data format: http://schema.org/Book

Data format Remove
Data format: weblink

Data format Remove
Data format: applicationlink

Add New

Supported file types

Specifies the file types supported by the app; for example, ".jpg". The Share target declaration requires the app support at least one data format or file type. The app will be displayed in the Share charm whenever a file with a supported type is shared from another app. If no file types are declared, make sure to add one or more data formats.

☑ Supports any file type

Add New

App settings

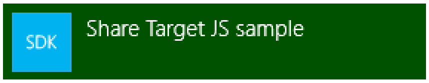
Executable:

Entry point:

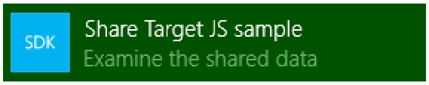
Start page: target.html

FIGURE 15-3 The Share content target app sample's manifest declarations.

Tip The Share Description field on the Share Target Declarations page in the manifest determines the subtext below the app name in the share charm. That is, if this field is left empty, only the target app's logo and name appears in the charm:



Here's how it looks with "Examine the shared data" in the Share Description field:

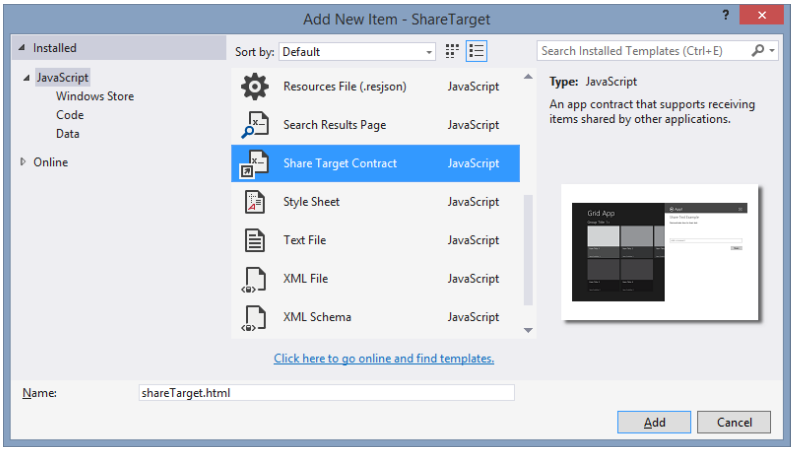


Share Target JS sample
Examine the shared data

The Share start page, target.html, is just a typical HTML page with whatever layout you require for performing the share task. This page typically operates independently of your main app: when your app is chosen through Share, this page is loaded and activated by itself and thus has an entirely separate script context. This page should not provide navigation to other parts of the app and should thus load only whatever code is necessary for the sharing task. (The Executable and Entry Point options are not used for apps written in HTML and JavaScript; those exist for apps written in other languages.)

Tip Because the Share start page is activated separately from the main app, you can use the `document.onbeforeunload` event to detect when the user dismisses it.

Much of this structure is built for you automatically through the Share Target Contract *item* template provided by Visual Studio and Blend, as shown in Figure 15-4; the dialog appears when you right-click your project and select Add > New Item or select the Project > Add New Item menu command.


Add New Item - ShareTarget
?
✕
Installed
JavaScript
Windows Store
Code
Data
Online
Sort by: Default
Search Installed Templates (Ctrl+E)
Resources File (.resjson) JavaScript
Search Results Page JavaScript
Share Target Contract JavaScript
Style Sheet JavaScript
Text File JavaScript
XML File JavaScript
XML Schema JavaScript
Click here to go online and find templates.
Add
Cancel

Type: JavaScript

An app contract that supports receiving items shared by other applications.

Name: shareTarget.html

FIGURE 15-4 The Share Target Contract item template in Visual Studio and Blend.

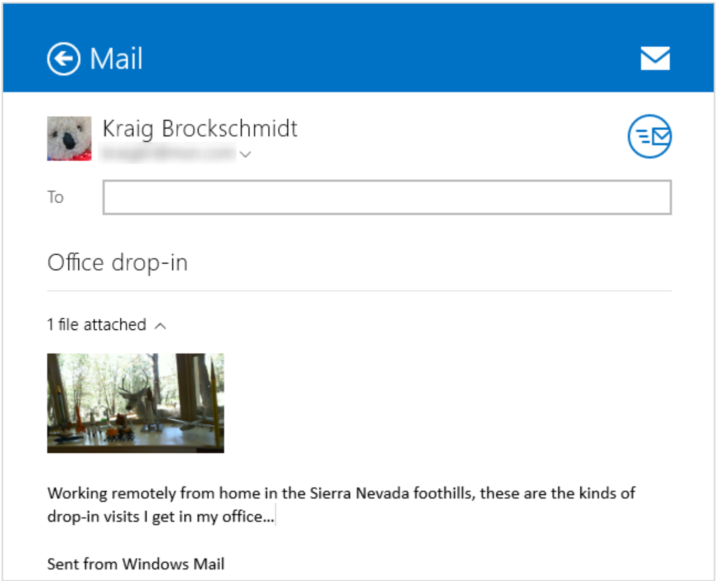
This item template will give you HTML, JS, and CSS files for the share target page and will add that page to your manifest declarations along with text and URI formats. So you'll want to update those declarations as appropriate.

Before we jump into the code, a few notes about the design of a share target page, summarized from Guidelines for sharing content:

- Maintain the app's identity and its look and feel, consistent with the primary app experience.
- Keep interactions simple to quickly complete the share flow: avoid text formatting, tagging, and setup tasks, but do consider providing editing capabilities especially if posting to social networks or sending a message. (See Figure 15-5 from the Mail app for an example.) A social networking target app would generally want to include the ability to add comment; a photo-sharing target would probably include the ability to add captions.
- Avoid navigation: sharing is a specific task flow, so use inline controls and inline errors instead of switching to other pages. Another reason to avoid this is that the share page of the target app runs in its own script context, so being able to navigate elsewhere in the app within a separate context could be very confusing to users.
- Keep sign-in and configuration interactions simple—that is, have one step to sign in instead of a multistep process. If more steps are necessary, encourage the user to open the full app to perform them.
- Avoid links that would distract from or take the user away from the sharing experience. Remember that sharing is a way to shortcut the oft-tedious process of getting data from one app to another, so keep the target app focused on that purpose.
- Avoid light-dismiss flyouts because the Share charm already works that way.
- Acknowledge user actions when you start sending the data off (to an online service, for example) so that users know something is actually happening.
- Put important buttons within reach of the thumbs on a touch device; refer to Windows 8 Touch Posture topic in the documentation for placement guidance.
- Make previews match the actual content—in other words, don't play tricks on the user!

With this page design, it's good to know that you do *not* need to worry about different views—this page really just has one view as a flyout. It *does* need to adapt itself well to varying dimensions, mind you, but not to random widths. Basing the layout on a CSS grid with fractional rows and columns is a good approach here.

Caution Because a target app can receive data from any source app, it should treat all such content as untrusted and potentially malicious, especially with HTML, URIs, and files. The target app should avoid adding such HTML or file contents to the DOM, executing code from URIs, navigating to the URI or some other page based on the URI, modifying database records, using `eval` with the data, and so on.


Mail

Kraig Brockschmidt
[redacted]

To

Office drop-in

1 file attached

Working remotely from home in the Sierra Nevada foothills, these are the kinds of drop-in visits I get in my office...

Sent from Windows Mail

FIGURE 15-5 The sharing UI of the Windows Mail app (the bottom blank portion has been cropped); this UI allows editing of the recipient, subject, and message body, and managing attachments.

Let's now look at the contents of the template's JavaScript file as a whole, because it shows us the basics of being a target. First, as you can see, we have the same structure as a typical default.js for the app, using the `WinJS.Application` object's methods and events.

```
(function () {
    "use strict";

    var app = WinJS.Application;
    var share;

    function onShareSubmit() {
        document.querySelector(".progressindicators").style.visibility = "visible";
        document.querySelector(".commentbox").disabled = true;
        document.querySelector(".submitbutton").disabled = true;

        // TODO: Do something with the shared data stored in the 'share' var.

        share.reportCompleted();
    }

    // This function responds to all application activations.
    app.onactivated = function (args) {
        var thumbnail;

        if (args.detail.kind ===
            Windows.ApplicationModel.Activation.ActivationKind.shareTarget) {
            document.querySelector(".submitbutton").onclick = onShareSubmit;
            share = args.detail.shareOperation;
            document.querySelector(".shared-title").textContent =
                share.data.properties.title;
            document.querySelector(".shared-description").textContent =
                share.data.properties.description;

            thumbnail = share.data.properties.thumbnail;
            if (thumbnail) {
                // If the share data includes a thumbnail, display it.
                args.setPromise(thumbnail.openReadAsync().then(
                    function displayThumbnail(stream) {
                        document.querySelector(".shared-thumbnail").src =
                            window.URL.createObjectURL(stream);
                    }));
            } else {
                // If no thumbnail is present, expand the description  and
                // title elements to fill the unused space.
                document.querySelector("section[role=main] header").style
                   .setProperty("-ms-grid-columns", "0px 0px 1fr");
                document.querySelector(".shared-thumbnail").style.visibility = "hidden";
            }
        }
    };

    app.start();
})();
```

When this page is loaded and activated, during which time the app's splash screen will appear in the flyout, its `WinJS.Application.onactivated` event will fire—again independently of your app's main `activated` handler that's typically in default.js. As a share target you just want to make sure that the activation kind is `shareTarget`, after which your primary responsibility is to provide a preview of the data you'll be sharing along with whatever UI you have to edit it, comment on it, and so forth. Typically, you'll also have a button to complete or submit the sharing, on which you tell the share broker that you've completed the process.

The key here is the `args.detail.shareOperation` object provided to the `activated` handler. This is a [`ShareTarget.ShareOperation`] object, whose `data` property contains a read-only package called a [`DataPackageView`] from which you obtain all the goods:

- To check whether the package has formats you can consume, use the `contains` method or the `availableFormats` collection.
- To obtain data from the package, use its `get*` methods such as `getTextAsync`, `getBitmapAsync`, and `getDataAsync` (for custom formats), among others (and note that `getUriAsync` is deprecated). When pasting HTML you can also use the `getResourceMapAsync` method to get relative resource URIs. The view's `properties` like the `thumbnail` are also useful to provide a preview of the data.

As you can see, the Share target item template code above doesn't do anything with shared data other than display the title, description, and thumbnail; clearly your app will do something more by requesting data from the package, like the examples we see in the share target sample. Its js/target.js file contains an `activated` handler for the target.html page (in the project root), and it also displays the thumbnail in the data package by default. It then looks for different data formats and displays those contents if they exist:

```
if (shareOperation.data.contains(
    Windows.ApplicationModel.DataTransfer.StandardDataFormats.text)) {
    shareOperation.data.getTextAsync().done(function (text) {
        displayContent("Text: ", text, false);
    });
}
```

The same kind of code appears for the simpler formats. Consuming a bitmap is a little more work but straightforward:

```
if (shareOperation.data.contains(
    Windows.ApplicationModel.DataTransfer.StandardDataFormats.bitmap)) {
    shareOperation.data.getBitmapAsync().done(function (bitmapStreamReference) {
        bitmapStreamReference.openReadAsync().done(function (bitmapStream) {
            if (bitmapStream) {
                var blob = MSApp.createBlobFromRandomAccessStream(bitmapStream.contentType,
                    bitmapStream);
                document.getElementById("imageHolder").src = URL.createObjectURL(blob,
                    { oneTimeOnly: true });
                document.getElementById("imageArea").className = "unhidden";
            }
        });
    });
}
```

For HTML, it looks through the markup for `img` elements and then sets up their `src` attributes from the resource map. The `iframe` used to display the content (which could also be a webview) already has the HTML content from the package by this time:

```
var images = iFrame.contentDocument.documentElement.getElementsByTagName("img");
if (images.length > 0) {
    shareOperation.data.getResourceMapAsync().done(function (resourceMap) {
        if (resourceMap.size > 0) {
            for (var i = 0, len = images.length; i < len; i++) {
                var streamReference = resourceMap[images[i].getAttribute("src")];
                if (streamReference) {
                    // Call a helper function to map the image element's src
                    // to a corresponding blob URL generated from the streamReference
                    setResourceMapURL(streamReference, images[i]);
                }
            }
        }
    });
}
```

The setResourceMapURL helper function does pretty much what the bitmap-specific code did, which is call openReadAsync on the stream, call MSApp.createBlobFromRandomAccessStream, pass that blob to URL.createObjectURL, set the img.src with the result, and close the stream.

After the target app has completed a sharing operation, it calls the ShareOperation.reportCompleted method, as shown earlier with the template code. This lets the system know that the data package has been consumed, the share flow is complete, and all related resources can be released. The share target sample does this when you explicitly click a button for this purpose, but normally you automatically call the method whenever you've completed the share. Do be aware that calling reportCompleted will close the target app's sharing UI, so avoid calling it as soon as the target activates: you want the user to feel confident that the operation was carried out.

Long-Running Operations

To provide a fast and fluid user experience, a target app can dismiss the Share pane when it wants by calling the ShareOperation.dismissUI method. After this, the target app has 10 seconds to complete its processes or else it will be terminated.

If you need more time, it's necessary to tell Windows that you have a long-running operation. When you run the Sharing content target app sample and invoke the Share charm from a suitable source app, there's a little expansion control near the bottom labeled "Long-running Share support." If you expand that, you'll see some additional controls and a bunch of descriptive text, as shown in Figure 15-6. The buttons shown here tie into a number of other methods on the ShareOperation object alongside reportCompleted. These help Windows understand exactly how the share operation is happening within the target: reportStarted, reportDataRetrieved, reportSubmittedBackgroundTask, and reportError. As you can see from the descriptions in Figure 15-6, these generally relate to telling Windows when the target app has finished cooking its meal, so to speak, and the system can clean the dishes and put away the utensils:

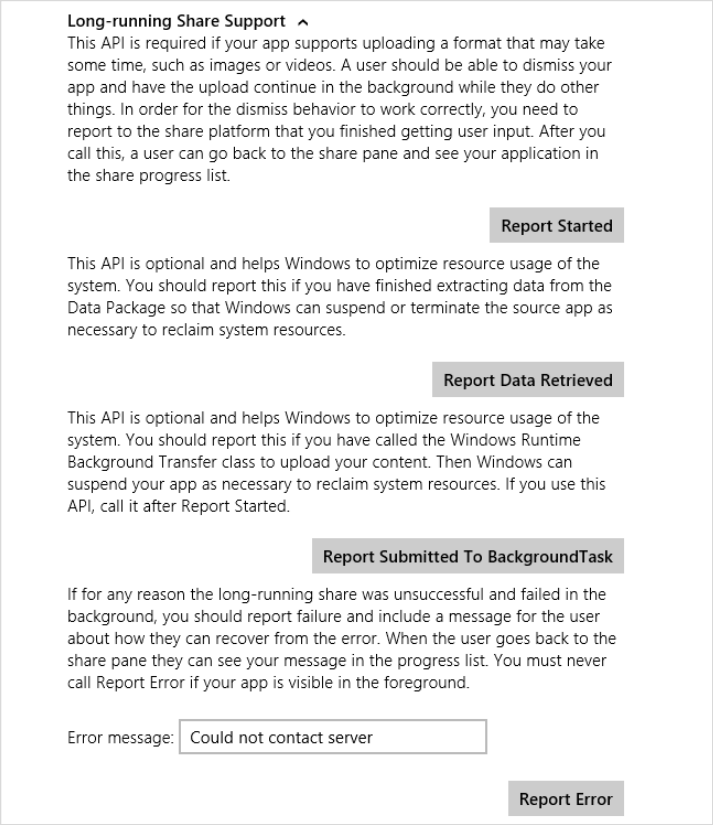
- reportStarted informs Windows that your sharing operation might take a while, as if you're uploading the data from the package to another place or sending an email attachment with what ends up being large images and such. This specific method indicates that you've obtained all necessary user input and that the share pane can be dismissed.

- reportDataRetrieved informs Windows that you've extracted what you need from the data package such that it can be released. If you've called MSApp.createBlobFromRandomAccessStream for an image stream, for example, the blob now contains a copy of the image that's local to the target app. If you're using images from the package's resourceMap, on the other hand, don't call reportDataRetrieved unless you explicitly make a copy of those references whose URIs refer to bits inside the data package. In any case, if you need to hold on to the package throughout the operation, you don't need to call this method because you'll later call reportCompleted to release the package.

- reportSubmittedBackgroundTask tells Windows that you've started a background transfer using the Windows.Networking.BackgroundTransfer.BackgroundUploader class (see

Chapter 4). As the sample description in Figure 15-6 indicates, this lets Windows know that it can suspend the target app and not disturb the sharing operation. If you call this method with a local copy of the data being uploaded, go ahead and call `reportCompleted` method so that Windows can clean up the package; otherwise wait until the transfer is complete.

- `reportError` lets Windows know if there's been an error during the sharing operation.



Long-running Share Support

This API is required if your app supports uploading a format that may take some time, such as images or videos. A user should be able to dismiss your app and have the upload continue in the background while they do other things. In order for the dismiss behavior to work correctly, you need to report to the share platform that you finished getting user input. After you call this, a user can go back to the share pane and see your application in the share progress list.

This API is optional and helps Windows to optimize resource usage of the system. You should report this if you have finished extracting data from the Data Package so that Windows can suspend or terminate the source app as necessary to reclaim system resources.

This API is optional and helps Windows to optimize resource usage of the system. You should report this if you have called the Windows Runtime Background Transfer class to upload your content. Then Windows can suspend your app as necessary to reclaim system resources. If you use this API, call it after Report Started.

If for any reason the long-running share was unsuccessful and failed in the background, you should report failure and include a message for the user about how they can recover from the error. When the user goes back to the share pane they can see your message in the progress list. You must never call Report Error if your app is visible in the foreground.

Error message: Could not contact server

FIGURE 15-6 Expanded controls in the Sharing content target app sample for Long-Running Share Support. The Report Completed button is always shown and isn't specific to long-running tasks despite its placement in the sample's UI. Don't let that confuse you!

Quicklinks

The last aspect of the Share contract for us to explore is something we mentioned early on in this section: quicklinks. These serve to streamline the Share process such that users don't need to re-enter information in a target app. For example, if a user commonly shares data with particular people through email, each contact can be a quicklink for the email app. If a user commonly shares with different people or groups through a social networking app, those people and/or groups can be represented with quicklinks. And as these targets are much more user-specific than target apps in general, the Share charm UI shows these at the top of its list (see Figure 15-7 below).

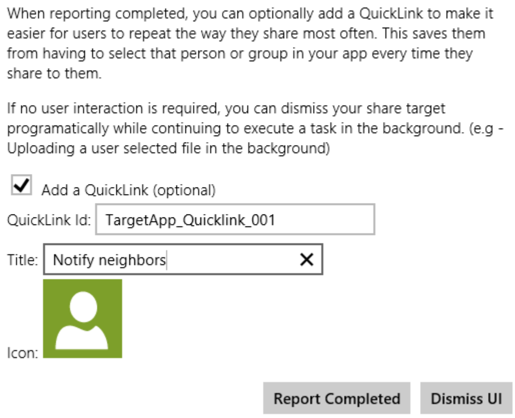
Each quicklink is associated with and serviced by a particular target app and simply provides an identifier to that target. When the target is invoked through a quicklink, it then uses that identifier to retrieve whatever data is associated with that quicklink and prepopulates or otherwise configures its UI. It's important to understand that quicklinks contain *only* an identifier, so the target app must store and retrieve the associated data from some other source, typically local app data where the identifier is a filename, the name of a settings container, etc. The target app could also use roaming app data or the cloud for this purpose, but quicklinks themselves do not roam to another device—they are strictly local. Thus, it makes the most sense to store the associated data locally.

A quicklink itself is just an instance of the `Quicklink` class. You create one with the `new` operator and then populate its `title`, `thumbnail`, `supportedDataFormats`, `supportedFileTypes`, and `id` properties. The data formats and file types are what Windows uses to determine if this quicklink should be shown in the list of targets for whatever data is being shared from a source app (independent of the app's manifest declarations). The `title` and `thumbnail` are used to display that choice in the Share charm, and the `id` is what gets passed to the target app when the quicklink is chosen.

> **Tip** For the thumbnail, use an image that's more specifically representative of the quicklink (such as a contact photo) rather than just the target app. This helps distinguish the quicklink from the general use of the target app.

An app then registers a quicklink with the system by passing it to the `ShareOperation.reportCompleted` method. As this is the *only* way in which a quicklink is registered, it tells us that creating a quicklink *always* happens as part of another sharing operation. It's a way to create a specific target that might save the user some time and encourage her to choose your target app again in the future.

Let's follow the process within the Sharing content target app sample to see how this all works. First, when you invoke the Share charm and choose the sample, you'll see that it provides a check box for creating a quicklink (Figure 15-7). When you check this, it provide fields in which you can enter an id and a title (the thumbnail just uses a default image). When you press the Report Completed button, it calls `reportCompleted` and the quicklink is registered. On subsequent invocations of the Share charm with the appropriate data formats from the source app, this quicklink will then appear in the list, as shown in Figure 15-8 where the app servicing the quicklink is always indicated under the provided title.



When reporting completed, you can optionally add a QuickLink to make it easier for users to repeat the way they share most often. This saves them from having to select that person or group in your app every time they share to them.

If no user interaction is required, you can dismiss your share target programatically while continuing to execute a task in the background. (e.g - Uploading a user selected file in the background)

☑ Add a QuickLink (optional)

QuickLink Id: TargetApp_Quicklink_001

Title: Notify neighbors

Icon:

FIGURE 15-7 Controls to create a quicklink in the Sharing content target app sample.

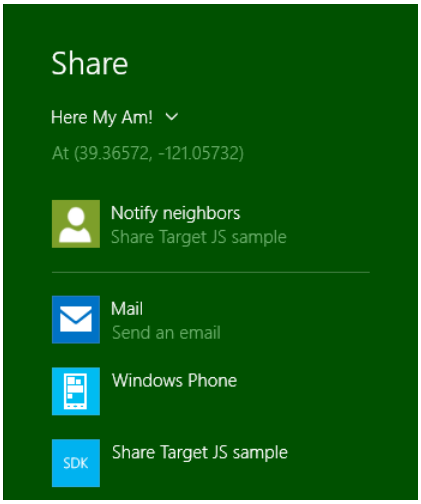


FIGURE 15-8 A quicklink from the Sharing content target app sample as it appears in the Share charm target list.

Here's how the share target sample creates the quicklink within the function `reportCompleted` (js/target.js) that's attached to the Report Completed button (some error checking omitted):

```
if (addQuickLink) {
    var quickLink = new Windows.ApplicationModel.DataTransfer.ShareTarget.QuickLink();

    var quickLinkId = document.getElementById("quickLinkId").value;
    quickLink.id = quickLinkId;

    var quickLinkTitle = document.getElementById("quickLinkTitle").value;
    quickLink.title = quickLinkTitle;

    // For quicklinks, the supported FileTypes and DataFormats are set independently
    // from the manifest
    var dataFormats = Windows.ApplicationModel.DataTransfer.StandardDataFormats;
    quickLink.supportedFileTypes.replaceAll(["*"]);
    quickLink.supportedDataFormats.replaceAll([dataFormats.text, dataFormats.uri,
        dataFormats.bitmap,
        dataFormats.storageItems, dataFormats.html, customFormatName]);

    // Prepare the icon for a QuickLink
    Windows.ApplicationModel.Package.current.installedLocation.getFileAsync(
    "images\\user.png").done(function (iconFile) {
        quickLink.thumbnail = Windows.Storage.Streams.RandomAccessStreamReference
            .createFromFile(iconFile);
        shareOperation.reportCompleted(quickLink);
    });
```

Again, the process just creates the `Quicklink` object, sets its properties (perhaps settings a more specific thumbnail such as a contact's picture), and passes it to `reportCompleted`. In the share target sample, you can see that it doesn't actually store any other local app data; for its purposes the properties in the quicklink are sufficient. Most target apps, however, will likely save some app data for the quicklink that's associated with the `quicklink.id` property and reload that data when activated later on through the quicklink.

When the app (that is, the target page) is activated in this way, the `eventArgs.detail.shareOperation` object within the `activated` event handler will contain the `quicklinkId`. The sample simply displays this id, but you would certainly use it to load app data and prepopulate your share UI:

```
// If this app was activated via a QuickLink, display the QuickLinkId
if (shareOperation.quickLinkId !== "") {
    document.getElementById("selectedQuickLinkId").innerText = shareOperation.quickLinkId;
    document.getElementById("quickLinkArea").className = "hidden";
}
```

Note that when the target app is invoked *through* a quicklink, it doesn't display the same UI to *create* a quicklink, because doing so would be redundant. However, if the user edited the information related to the quicklink, you might provide the ability to update the quicklink, which means to update the data you save related to the id, or to create a new quicklink with a new id.

You should also provide a means through which a user can delete a quicklink. This is done by calling the `ShareOperation.removeThisQuickLink` method, which deletes the one identified in the `quickLinkId` property.

The Clipboard

Before the Share contract was ever conceived, the mechanism we know as the Clipboard was once the poster child of app-to-app cooperation. And while it may not garner any media attention nowadays, it's still a tried-and-true means for apps to share and consume data, including data coming from any kind of source, including the browser, desktop apps, and other Windows Store apps. Any necessary translation work between these layers is handled automatically.

For Windows Store apps, clipboard interactions build on the same DataPackage mechanisms we've already seen for sharing, so everything we've learned about populating that package, using custom formats, and using delayed rendering still applies. Indeed, if you make data available on the clipboard, you should make sure the same data is available for the Share contract!

The question is how to wire up commands like copy, cut, and paste—from the app bar, a context menu, or keystrokes—if an app provides them for its own content (many controls handle the clipboard automatically). In the web context, you can use the window.clipboardData object like you would in a browser, but attempting to access it will throw an Access Denied exception in the local context. There we must instead turn to the Windows.ApplicationModel.DataTransfer.Clipboard class.

As shown in the Clipboard app sample, the processes here are straightforward. For copy and cut:

- Create a new DataPackage (or use MSApp.createDataPackage or MSApp.createDataPackageFromSelection), and populate it with the desired data.

  ```
  var dataPackage = new Windows.ApplicationModel.DataTransfer.DataPackage();
  dataPackage.setText(textValue);
  //...
  ```

- (Optional) Set the package's requestedOperation property to values from DataPackageOperation: copy, move, link, or none (the latter is used with delayed rendering). Note that these values can be combined using the bitwise OR operator, as in:

  ```
  var dpo = Windows.ApplicationModel.DataTransfer.DataPackageOperation;
  dataPackage.requestedOperation = dpo.copy | dpo.move | dpo.link;
  ```

- Pass the data package to Clipboard.setContent:

  ```
  Windows.ApplicationModel.DataTransfer.Clipboard.setContent(dataPackage);
  ```

To perform a paste:

- Call Clipboard.getContent to obtain a read-only data package called a DataPackageView:

  ```
  var dataView = Windows.ApplicationModel.DataTransfer.Clipboard.getContent();
  ```

- Check whether it contains formats you can consume with the contains method (alternately, you can check the contents of the availableFormats vector):

  ```
  if (dataView.contains(
      Windows.ApplicationModel.DataTransfer.StandardDataFormats.text)) {
      //...
  }
  ```

- Obtain data using the view's get* methods such as getTextAsync, getBitmapAsync, and getDataAsync (for custom formats), among others. When pasting HTML, you can also use the getResourceMapAsync method to get relative resource URIs. The view's properties like the thumbnail are also useful, along with the requestedOperation value or values.

```
    dataView.getTextAsync().done(function (text) {
        // Consume the data
    }
```

If at any time you want to clear the clipboard contents, call the Clipboard class's clear method. You can also make sure data is available to other apps even if yours is shut down by calling the flush method (which will trigger any deferred rendering you might have set up).

Apps that use the clipboard also need to know when to enable or disable a paste command depending on available formats. At any time you can get the data package view from the clipboard (Clipboard.getContent) and use its contains method or availableFormats property and decide accordingly. You should also then listen to the Clipboard object's contentChanged event (a WinRT event), which will be fired when you or some other app calls the clipboard's setContent method. At that time you'd again enable or disable the commands. You won't receive this event when your app is suspended, so you should refresh the state of those commands within your resuming handler.

Again, the Clipboard app sample provides examples of these various scenarios, including copy/paste of text and HTML (scenario 1); copy and paste of an image (scenario 2); copy and paste of files (scenario 3); and clearing the clipboard, enumerating formats, clearing the content (scenario 4), and handling contentChanged (also scenario 4).

Note, finally, that pasted data can come from anywhere. Apps that consume data from the clipboard should, like a share target, treat the content they receive as potentially malicious and take appropriate precautions.

Launching Apps with URI Scheme Associations

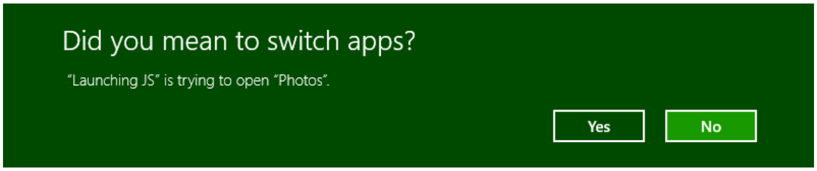
Back in Chapter 11, in the section "File Association and Launching," we learned about the Windows.System.Launcher API and the launchFileAsync method that allows one app to start another through a file association. It's time now to complete that story with the other launcher method: launchUriAsync. This launches another app through a protocol (URI scheme) association and supports variations through a LauncherOptions argument just like launchFileAsync.

Note With both launchFileAsync and launchUriAsync, Windows specifically blocks apps from launching any file or URI scheme that is handled by a system component and for which there is no legitimate scenario for a Windows Store app to insert itself into that process. The How to handle file activation and How to handle protocol activation topics generally list the specific file types and URI schemes in question, though these are specifically those you cannot implement as an association (the exact list of blocked associations is not published). Also, the file:// URI scheme is allowed in launchUriAsync but only for intranet URIs when you have declared the *Private Networks (Client & Server)* capability in the manifest. Furthermore, launchUriAsync does not recognize ms-appx, ms-appx-web, or ms-appdata URIs, because these already map to the current app and that app should just display those pages directly.

The result of the `launchUriAsync` call, as passed to your completed handler, is a Boolean: `true` if the launch succeeded, `false` if not. That is, barring a catastrophic failure, such as a low memory condition where the async operation will outright fail, `launchUriAsync` normally reports success to your completed handler with a Boolean indicating the outcome. You'll get a `false` result, for example, if you try to launch a URI that's blocked for security reasons.

However, you cannot know ahead of time what the result will be. This is the reason for the [`LauncherOptions`]() parameter, through which you can provide fallback mechanisms:

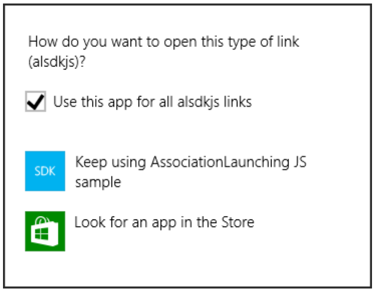
- `treatAsUntrusted` (a Boolean, default is `false`) displays a warning to the user that they'll be switching apps if they proceed (see image below). This is good to use when you're unsure about the source of the association, such as launching a URI found inside a PDF or other document, and want to prevent the user from experiencing a classic bait-and-switch!



Did you mean to switch apps?

"Launching JS" is trying to open "Photos".

- `displayApplicationPicker` (a Boolean, default is `false`) lets the use choose which app to launch as part of the process (see image below, which is using the protocol from the SDK sample). Note that the UI allows the user to change the default app for subsequent invocations. Also, the `LauncherOptions.ui` property (a [`LauncherUIOptions`]() object) can be used to control the placement of the app picker through its `invocationPoint`, `selectionRect`, and `preferredPlacement` properties. Beyond this, however, the picker cannot be customized.



How do you want to open this type of link (alsdkjs)?

☑ Use this app for all alsdkjs links

SDK
Keep using AssociationLaunching JS sample
Look for an app in the Store

- `desiredRemainingView` A value from [`Windows.UI.ViewManagement.ViewSizePreference`]() (see Chapter 8, "Layout and Views"), indicating how the calling app should appear after the launch: `default`, `useLess`, `useHalf`, `useMore`, `useMinimum`, or `useNone`. This helps when implementing cross-app scenarios, although the choice is not guaranteed.

- preferredApplicationDisplayName and preferredApplicationPackageFamilyName provide a suggestion to the user to acquire a specific app from the Windows Store if no other app is available to service the request (similar to what a Share source app uses, as described earlier under "Populating Metadata"). This is very useful with a particular URI scheme for which you provide an app yourself.
- Similarly, fallbackUri specifies a URI to which the user will be taken if no app can be found to handle the request and you don't have a specific suggestion in the Windows Store.
- Finally, the contentType option identifies the content type associated with a URI that controls which app is launched. This is primarily useful when the URI doesn't contain a specific scheme but simply refers to a file on a network using a scheme such as http or file that would normally launch a browser for file download. With contentType, the default app that's registered for that type, rather than the scheme, will be launched. That app, of course, must be able to them use the URI to access the file. In other words, this option is a way to pass a URI, rather than a whole file, to a handler app that you know can work with that URI.

Scenarios 2 of the Association launching sample provides a basic demonstration of using launchUriAsync (js/launch-uri.js):

```
var uri = new Windows.Foundation.Uri(document.getElementById("uriToLaunch").value);

// Launch the URI.
Windows.System.Launcher.launchUriAsync(uri).done(function (success) {
    // success indicates whether the URI was launched.
});
```

Scenario 2 also shows how to use options to launch with a warning, display the Open With dialog, and to control the view, through the following functions (js/launch-uri.js):

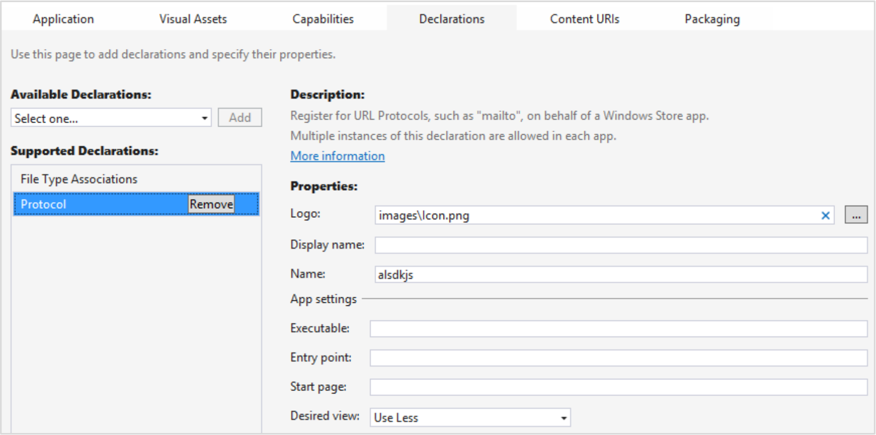
```
function launchUriWithWarning() {
    var uri = new Windows.Foundation.Uri(document.getElementById("uriToLaunch").value);

    var options = new Windows.System.LauncherOptions();  // Set the show warning option.
    options.treatAsUntrusted = true;

    Windows.System.Launcher.launchUriAsync(uri, options).done(function (success) {
        // success indicates whether the URI was launched.
    });
}

function launchUriOpenWith() {
    var uri = new Windows.Foundation.Uri(document.getElementById("uriToLaunch").value);

    var options = new Windows.System.LauncherOptions();  // Set the show picker option.
    options.displayApplicationPicker = true;

    // Position the Open With dialog so that it aligns with the button.
    // An alternative to using the rect is to set a point indicating where
    // the dialog is supposed to be shown.
    options.ui.selectionRect = getSelectionRect(
        document.getElementById("launchUriOpenWithButton"));
    options.ui.preferredPlacement = Windows.UI.Popups.Placement.below;

    Windows.System.Launcher.launchUriAsync(uri, options).done(function (success) {
        // success indicates whether the URI was launched.
    });
}

function launchUriSplitScreen() {
    var uri = new Windows.Foundation.Uri(document.getElementById("uriToLaunch").value);

    // Request to share the screen.
    var options = new Windows.System.LauncherOptions();

    // code omitted: Set options.desiredRemainingView from a drop-down list selection, e.g.:
    options.desiredRemainingView = Windows.UI.ViewManagement.ViewSizePreference.useHalf;

    Windows.System.Launcher.launchUriAsync(uri, options).done(function (success) {
        // success indicates whether the URI was launched.
    });
}
```

Scenario 4 then demonstrates handling protocol activation—that is, implementing the Protocol Activation contract (scenarios 2 and 4 deal with file type associations, which are covered in Chapter 11).

In comparison to file type associations, custom URI schemes are somewhat likely to launch your particular app. That is, a custom URI scheme offers the best route to have one app specifically launch another, as when you want to delegate certain tasks. The Maps app in Windows, for example, supports a *bingmaps* scheme for accomplishing mapping tasks. You can imagine the same for a stocks app, an email app (beyond *mailto*), line-of-business apps, and so forth. If you create such a scheme and want other apps to use it, you'll certainly need to provide documentation for its usage details, which means that another app can implement the same scheme and thus offer itself as another choice in the Windows Store. So there's no guarantee even with a very specific scheme that you can know for certain that you'll be launching another known app, but this is about as close as you can get to that capability.[108]

As with file types, the target app declares the URI schemes it wishes to service in the Declarations section of manifest. Here you add a Protocol declaration as shown in Figure 15-9.

[108] In any case, it's a good idea to register your URI scheme with the Internet Assigned Numbers Authority (IANA). RFC 4395 is the particular specification for defining new URI schemes.



Use this page to add declarations and specify their properties.

Available Declarations: Select one...

Supported Declarations:

File Type Associations

Protocol

Description:

Register for URL Protocols, such as "mailto", on behalf of a Windows Store app.

Multiple instances of this declaration are allowed in each app.

More information

Properties:

Logo: images\Icon.png

Display name:

Name: alsdkjs

App settings

Executable:

Entry point:

Start page:

Desired view: Use Less

FIGURE 15-9 The Declarations > Protocol UI in the Visual Studio manifest designer.

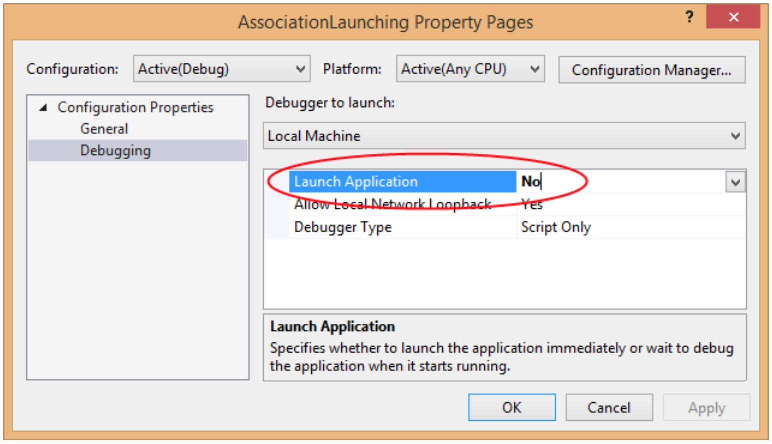
Under Properties, the Logo, Display Name, and Name all have the same meaning as with file type associations (see Chapter 11). Similarly, while you can specify a discrete start page, you'll typically handle activation in your main activation handler, as demonstrated in js/default.js of the Association launching sample (where the code path leads into js/scenario4.js). The last field is then the view in which you'd like the app to be launched (`useLess` in this case).

In the main activation handler you'll receive the activation kind of `protocol`, in which case `eventArgs.detail` is a [`WebUIProtocolActivatedEventArgs`]: its `uri` property contains the URI that the launching app passed to `launchUriAsync`. The sample doesn't make use of this itself, but just opens the page for scenario 4 passing the launching URI as an argument:

```
var url = null;
var arg = null;

// [Other activation kind checks omitted]
if (e.detail.kind === Windows.ApplicationModel.Activation.ActivationKind.protocol) {
    url = scenarios[3].url;  // Maps to /html/receive-uri.html

    arg = e.detail.uri;
}

if (url !== null) {
    e.setPromise(WinJS.UI.processAll().then(function () {
        return WinJS.Navigation.navigate(url, arg);
    }));
}
```

The receiving page—/html/receive-uri.html in this case—just outputs the URI to the console, but a real app will, of course, take other actions. Be warned, though, that URIs with some unique scheme can come from anywhere, including potentially malicious sources. Be wary of any data or queries in the URI, and avoid taking permanent actions with it. For instance, you can perhaps navigate to a new page, but don't modify database records or try to `eval` anything in the URI.

The last thing you'll need to know is how to debug protocol activation when the app isn't running, which means getting the app into the debugger when it's activated through a protocol. To do this, open the project's properties (Project > Properties menu command in Visual Studio) and then, under Configuration Properties > Debugging, set Launch Application to No:



Configuration: Active(Debug)
Platform: Active(Any CPU)

Debugger to launch: Local Machine

Launch Application	No
Allow Local Network Loopback	Yes
Debugger Type	Script Only

Launch Application
Specifies whether to launch the application immediately or wait to debug the application when it starts running.

Then set a breakpoint within your app's `activated` event handler; in the Association launching sample, you can do this within the function called *activated*. Then start the debugger in Visual Studio, which then waits until the app is launched some other way. To launch a protocol, then, you can just enter an appropriate URI like *alsdkjs://HelloWorld* into the Windows Run dialog (Win+R), and you'll see the app start up and hit your breakpoint in the debugger.

You can also just use scenario 2 of the Association launching sample, which allows you to try out the different launcher options. Do note that if you run this sample in the debugger with a breakpoint in the `activated` handler, you can observe how protocol activation will trigger that event when the app is already running.

Search

It hardly needs saying that Search is one of the most, if not the most, important capability in a world where the amount of data available to us continues to grows significantly faster than our ability to consume it. It's almost quaint, in fact, that we still refer to web *browsers* when most of the time they're really acting as viewports onto *search*. Sure, we still do some browsing here and there, but search is clearly where most of us begin because we typically know at the outset what it is we're seeking.

Recognizing this, user experience designers at Microsoft started, with Windows 8, to build a search paradigm directly into the fabric of the operating system that could quickly and efficiently get you to whatever it is you're looking for. In Windows 8 we first saw the idea that universal search was *always* available through the Search charm, shown in Figure 15-10, which is easily accessible alongside Share, Devices, and Settings.

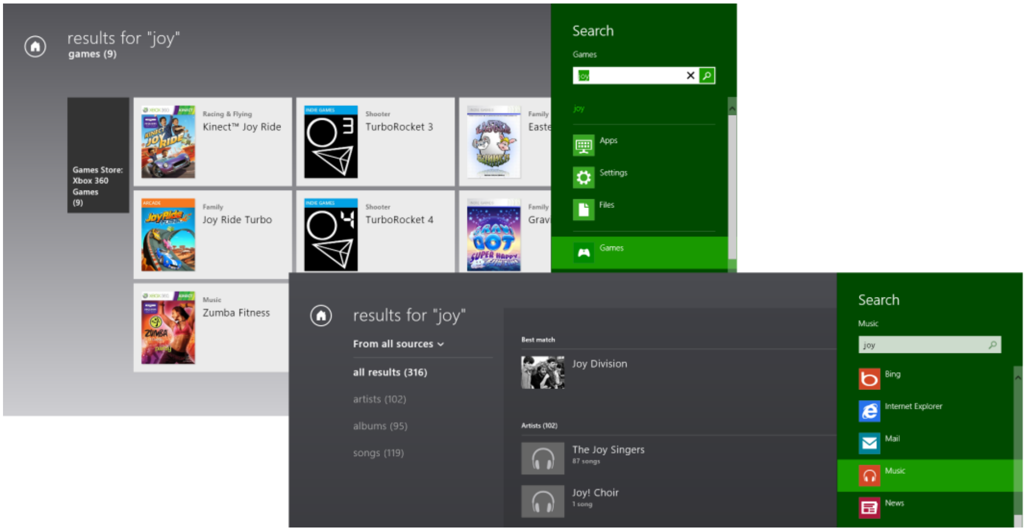


FIGURE 15-10 The Search charm experience of Windows 8, with results shown in the Games and Photos apps.

> **Tip** In both Windows 8 and Windows 8.1, you can determine which side of the screen the Search pane appears on through the `Windows.UI.ApplicationSettings.SettingsPane.edge` property. This can be used to lay out your results page if you don't want certain results obscured.

This design created a single go-to place to search for (and launch) apps on the Start screen with the keyboard, search settings (which are often hard to find), search files (whether local or cloud-based), and search within an app. And because you could use a browser like Internet Explorer as a target app, you also had the ability to search the web as well.

The Search charm meant that apps whose primary function wasn't oriented around search didn't need to provide their own UI controls but could simply implement whatever extent of the search contract was appropriate. And if a user didn't find what they were looking for in the app, the Search charm made it very quick and easy to search elsewhere by simply selecting a different search target. Users did not need to switch to a different app first before searching on the same term there—the current search is automatically applied as soon as a new target is selected. This again made search both quick and efficient.

When work began on Windows 8.1, the designers had lots of user feedback, telemetry, and experience to draw from to make search even better. The key, they realized, was to continue to improve efficiency by removing extra steps.

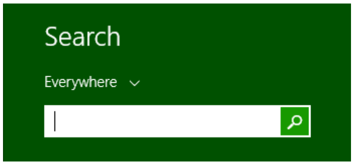
User telemetry also revealed the most common search targets: the foreground app, Internet Explorer, the Windows Store, and Settings. Very rarely did users actually select a different *app* target—I suspect this is because doing a generic search in different apps doesn't typically produce meaningful results, so users learned to not even bother. Thus search was revamped for Windows 8.1 with several key characteristics:

- Searching everywhere—files, apps, settings, and the web—is now the default when the Search charm is invoked, which saves the step of switching targets.
- For in-app search, the recommendation is to implement a search control directly on the app canvas. The `WinJS.UI.SearchBox` control is included in WinJS for this purpose. Be sure to refer to Guidelines for search for the full UI recommendations for in-app search.
- For apps that want to provide a search capability but aren't yet prepared to provide a full in-app experience, you can also have a simple button that invokes the Search charm directly and work with the Search contract from there. In this case the Search charm will be scoped to the app by default instead of "everywhere."

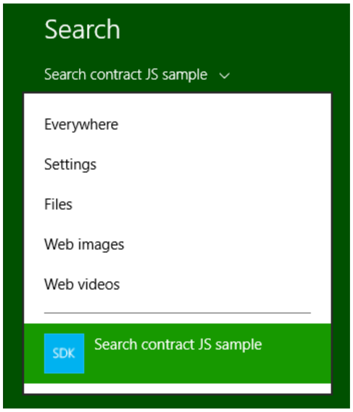
We'll look at all of these options in this section, but first let's do what we've done for other aspects of Windows by exploring the user experience of the Search charm as well as in-app search.

The Search Charm UI

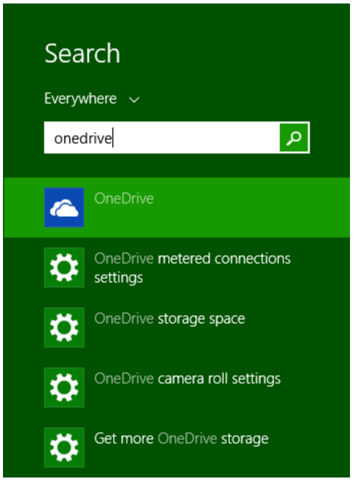
When you invoke the search charm in Windows 8.1 (which is done quickly by pressing Win+Q or Win+S), under most circumstances you see a Search Everywhere UI:

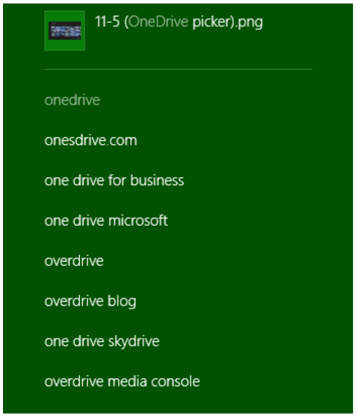


The exception is when the foreground app implements the Search contract and invokes the search charm programmatically, in which case the search is scoped to that app by default. In either case, you can tap the header with the down arrow to quickly change the scope, as shown here with the Search contract sample that we'll be seeing later:

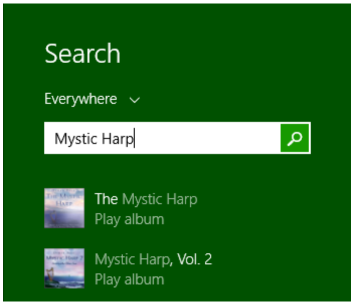


For our present purposes, though, let's assume a search scoped to Everywhere. As soon as you start typing a search phrase, results begin to immediately appear from all targets thanks to the file system indexer and the Bing search engine—something called "Smart Search." The idea with Smart Search is that it surfaces the most likely *result suggestions* at the top, results that include apps, settings, files, and high-confidence web results as they are available. This is what appears above the separator (see below left). Below the separator (below right) are then *query suggestions* based on what you've entered so far:

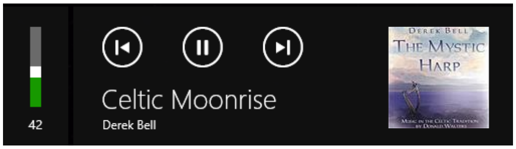

Search
Everywhere ⌄
onedrive
OneDrive
OneDrive metered connections settings
OneDrive storage space
OneDrive camera roll settings
Get more OneDrive storage


11-5 (OneDrive picker).png
onedrive
onesdrive.com
one drive for business
one drive microsoft
overdrive
overdrive blog
one drive skydrive
overdrive media console

Results suggestions save you the trouble of going to a separate results page and selecting a result there: these suggestions take you straight to that result. Each one might activate an app for that result (if the app fits the query), launch an app through file association, open PC Settings to a specific page, and so on. What's also very cool with certain results like music tracks or albums is that the system adds specific one-click actions in the Search charm, such as playing that track or album. These actions invoke a target app appropriately, such as starting playback (as the system media controls indicate) through the default media player without having to switch to that app at all, as shown below. That is, the app is launched directly to the background. At present, however, such actions are not extensible by apps.


Search
Everywhere ⌄
Mystic Harp
The Mystic Harp
Play album
Mystic Harp, Vol. 2
Play album

⇨



Selecting a query suggestion below the separator, pressing Enter in the text box, or tapping the search icon next to the text box takes you to a larger view of results, as shown in Figure 15-11 and in Figure 15-12, which includes local and web content together (this view is part of the Windows shell; it's not a separate app even though it appears as such). Depending on the search in question, you might see a large *hero* result (Figure 15-11) after local files and before specific web results. Also, if any web result can be serviced by an app—such as the Amazon result highlighted in Figure 15-12—invoking that result will launch the app. (You can swipe-select or right-click that item, as shown in the figure, for other options, such as opening a browser or copying the appropriate link.) These *app-mapped web results* (also known as Smart Search), provide an appcentric experience for web content. In Chapter 20 we'll see how to create this linkage between your website and your app.

The web results as shown in Figure 15-12 are curated over time based on real usage data so that the most common results that people use are shown first. Other results, if you pan to the right, are shown in a more condensed view, and more results will keep appearing as you continue to pan.

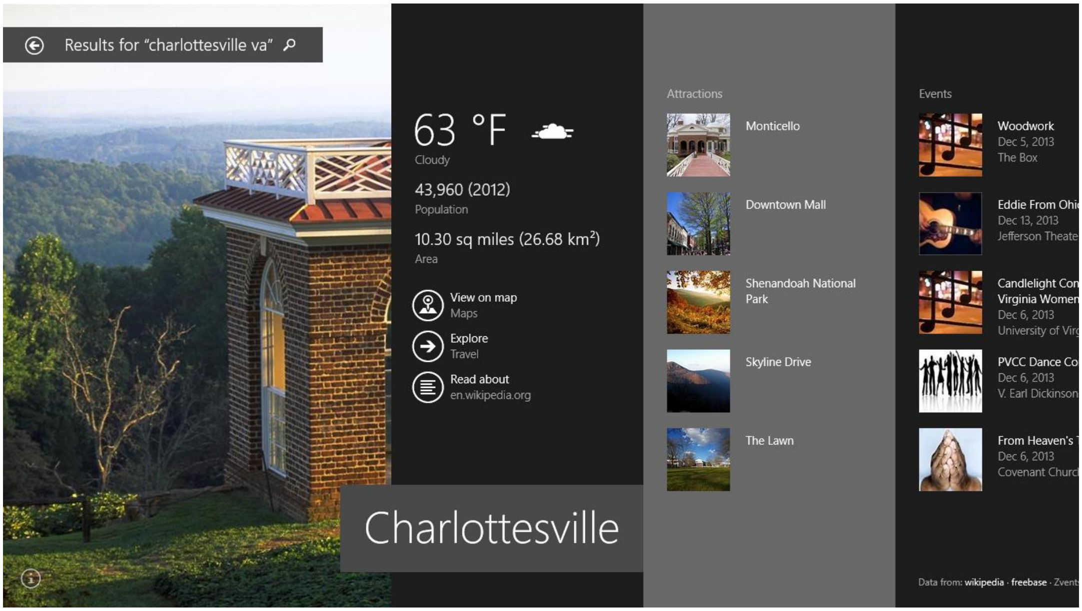
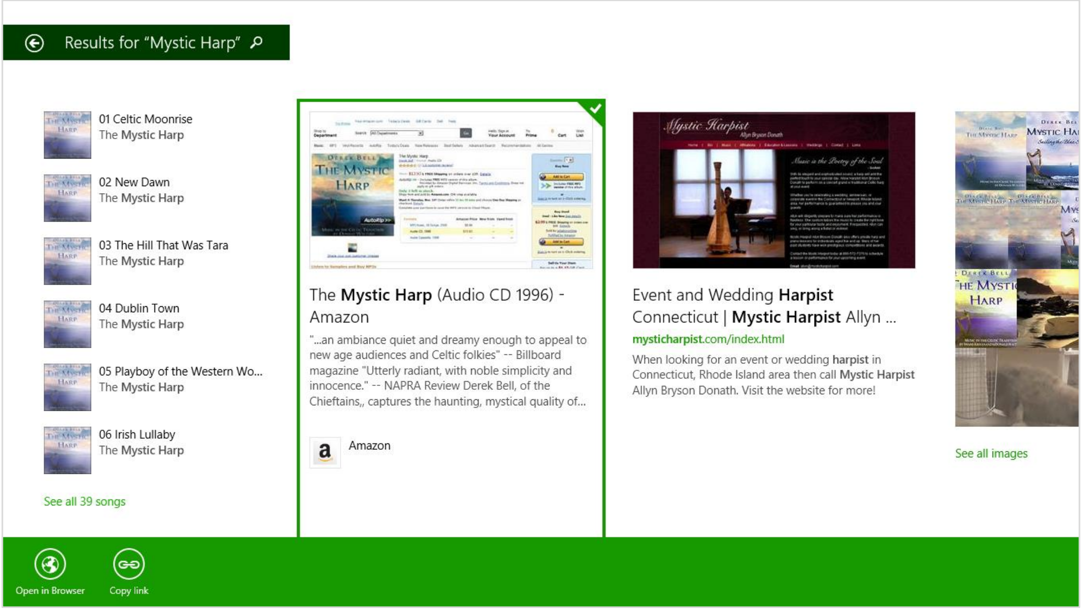


FIGURE 15-11 A large hero result in the search results view, showing in beautiful fashion the most common results that people typically look for. These are followed by additional web results if you pan to the right.


Results for "Mystic Harp"

01 Celtic Moonrise
The Mystic Harp

02 New Dawn
The Mystic Harp

03 The Hill That Was Tara
The Mystic Harp

04 Dublin Town
The Mystic Harp

05 Playboy of the Western Wo...
The Mystic Harp

06 Irish Lullaby
The Mystic Harp

See all 39 songs

The **Mystic Harp** (Audio CD 1996) - Amazon

"...an ambiance quiet and dreamy enough to appeal to new age audiences and Celtic folkies" -- Billboard magazine "Utterly radiant, with noble simplicity and innocence." -- NAPRA Review Derek Bell, of the Chieftains,, captures the haunting, mystical quality of...

Amazon

Event and Wedding **Harpist** Connecticut | **Mystic Harpist** Allyn ...

mysticharpist.com/index.html

When looking for an event or wedding **harpist** in Connecticut, Rhode Island area then call **Mystic Harpist** Allyn Bryson Donath. Visit the website for more!

See all images

FIGURE 15-12 Search results without a large hero, also showing files on left. The first Amazon result will open the Amazon app by default, but you can also open in the browser or copy a link by selecting the item and using the app bar commands, as shown here.

You can see that the Search charm itself is much richer and more powerful than it was in Windows 8. And because it defaults to a universal search, the UI guidance for in-app search changed as well. Whereas Windows 8 recommended that an app did not have its own search control in favor of the charm (which you can still do in Windows 8.1 as shown in the image above), it's now recommended that apps *do* have their own search controls for in-app content.

Having a search control on the app canvas—especially where search is a key feature for the app—keeps the user immersed in that app's content all the time, without going to the Search charm at all. That is, the app can show results (often filtered content) for the search term immediately right alongside the search control, rather than having the user go to the charm, enter a search term there, and have the app navigate to a separate results page. In other words, the recommended approach for Windows 8.1 keeps everything about search directly in the app.

For this purpose, WinJS provides the powerful `WinJS.UI.SearchBox` control with support for search history, type to search, suggested results, query suggestions, and filtering suggestions as a search term is being typed. As this is what most search-capable apps are expected to use, we'll be looking at it first in the sections that follow.[109]

[109] The `SearchBox` makes use of the [`Windows.ApplicationModel.Search.Core.SearchSuggestionManager`] class, which you'd use if you needed to implement a custom search control.

Then we'll take a look at a few matters of content indexing, which is applicable in this context, and we'll explore the Search contract in finer detail as you'd use when implementing your own search controls or participating in the contract directly.

> **Note** Because the `SearchBox` API and the search contract employ the same events and concepts, be sure to read the `SearchBox` sections even if you're planning to implement the contract, as the contact section at the end is fairly short and relies on the `SearchBox` material.

The WinJS.UI.SearchBox Control

The `WinJS.UI.SearchBox` control is the last control in all of WinJS that we haven't discussed so far in this book, so let's remedy that situation!

You typically place the `SearchBox` on the upper right of your app pages (or left with right to left languages), unless it should be nearer to the content that you can search. In the Mail app, for example, it's placed directly above the messages, thus indicating that messages are what's specifically being searched. Also, make sure the `SearchBox` is available in all views of a page; you can reduce it down to a button that then expands to show the full control, but don't make it disappear altogether.

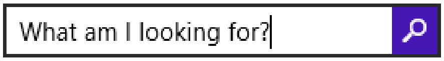
In its most basic usage, the `SearchBox` is an edit field with a search button that fires a `querysubmitted` event (as does pressing Enter). That is, with this minimal markup:

```
<div id="searchBoxId" data-win-control="WinJS.UI.SearchBox"></div>
```

and a little event code:

```
var searchBox = document.getElementById("searchBoxId").winControl;
searchBox.addEventListener("querysubmitted", querySubmittedHandler);

function querySubmittedHandler(eventObject) {
    var queryText = eventObject.detail.queryText;
}
```

you'll get the entered text (up to 2048 characters) in the `queryText` variable[110] at which point you can do anything you want with it. Whatever the case, here's how the control will appear with default styling (which includes its size):



What am I looking for?

The magnifying glass icon is specifically set in the WinJS stylesheets as follows, so you can override it with the same selector:

[110] The query is always available through the control's `queryText` property, and `eventArgs.detail.queryText` is also included with the `queryChanged` events. The `eventArgs.detail` properties also include `language` (the current locale), `linguisticDetails` (a `SearchQueryLinguisticDetails` object that provides information related to input method editors), and, for `querysubmitted`, `keyModifiers` to indicate the state of the Ctrl, Shift, Menu, and Windows keys.

```
.win-searchbox-button.win-searchbox-button:before {
    content: "\E094";
}
```

For all the other `win-searchbox*` styles, see the "SearchBox Styling" section later in this chapter.

Sidebar: The Search Results Page Item Template

Obtaining a query from a `SearchBox` is one thing, but displaying the results of a query is another. As this typically involves showing results in a ListView control, Visual Studio and Blend provide a Search Results Page item template. Right-click your project and select Add > New Item, or use the Project > Add New Item… menu command and choose Search Results Page from the list of templates. This adds page control files (.html, .js. and .css) for a search results page. There's not much exciting to show here visually because the template code very much relies on there being some real data to work with. Nevertheless, the template gives you a great structure to start from, including a filtering mechanism, and it's wholly expected that you then rework the layout and use styling that integrates with the rest of your app's experience. Some further details can be found on Adding a Search results item template.

Beyond a text box with a button, the `SearchBox` has a number of other features:

- As the user types, the `SearchBox` raises `querychanged` events, indicating that the search term in its `queryText` property has changed. Handling this is appropriate if you have logic you need to run outside of providing suggestions, such as previewing results (that is, immediate filtering) directly in the app itself. Typically, if you do active filtering or previewing, submitting the query (and the `querysubmitted` event, see below) is a moot point because it has already been processed.
- When the control receives focus and as the user types, the control also raises `suggestionsrequested` events. In response to this you populate a collection of suggested search terms (query suggestions), including default suggestions before the user has entered anything. You can use existing in-app content for this or make an HTTP request, if you like, because the `SearchBox` supports providing suggestions asynchronously.
- You can also tell the `SearchBox` to produce query suggestions from an Advanced Query Syntax (AQS) query on the file system through its `setLocalContentSuggestionSettings` method.
- When the user presses Enter in the `SearchBox`, taps the magnifying glass button, or selects a query suggestion, the `SearchBox` fires a `querysubmitted` event—this tells you to take the user to a results page. Also, if you set the `chooseSuggestionOnEnter` property to `true`, pressing Enter in the `SearchBox` automatically chooses the first query suggestion and sends that string with the `querysubmitted` event.

- If, on the other hand, the user selects a *result* suggestion, the control fires a resultsuggestionschosen event (note the "suggestions" in the middle is plural), which is intended to take the user directly to that content rather than a results page.
- By setting its focusOnKeyboardInput flag to true, you instruct the SearchBox to automatically take the focus on keyboard input. This makes it easy to implement type-to-search behavior as you have on the Windows Start screen. When this happens, the control raises a receiving-focusonkeyboardinput event in response to which you can do things like make other UI visible. Be mindful to enable this on a page-by-page basis: disable the feature on pages where you have any other text input controls. Typically, type-to-search is enabled on an app's main page, a search results page, and gallery pages. See Guidelines for search in the section "Enabling type-to-search."
- The SearchBox automatically tracks automatic search history, which can be turned off by setting the searchHistoryDisabled property to true. You can have a single SearchBox manage different histories by changing the searchHistoryContext property. Note, however, that the history itself isn't accessible, and if an app wants to clear the history (a capability you should ideally offer through Settings), call the Windows.ApplicationModel.Search.Core.-SearchSuggestionManager.clearHistory method.
- The placeholderText property lets you assign text that appears in the SearchBox when there is no user-entered text. Always set placeholder text to something that describes *what kind of data can be searched*—this immediately tells the user the applicable scope for searches. For example, as described in Guidelines for search, a music app might use "Album, artist, or song name" here. The Sports app uses "Enter a team name or player name." Having good placeholder text goes a long way to helping your users understand and have appropriate expectations of your search feature (saving them time and frustration).
- The SearchBox fully supports IME input for east-Asian languages.

Let's now take a look at the details of working with query suggestions and result suggestions in the next two sections. Then we'll check out the APIs for indexing your own content, which can very much help improve the performance of your searches.

Providing Query Suggestions

Handling the SearchBox control's suggestionsrequested event is how you provide the kind of auto-complete feature that many users have come to expect (beyond matching search terms in the control's history). It's not required, of course, because sometimes there just isn't space in your app's UI to show those suggestions, or your content is so varied that you really can't make suggestions in the first place. That is, if you're essentially doing a full-text search, like the Mail app does over email messages, then making guesses at what the user is looking for doesn't make sense. But for a general topic search, making topic suggestions can shortcut the user’s path between searching and enjoying your content.

The SearchBox control is set up to handle two types of suggestions: those coming directly from the app, and those coming from the local file system.

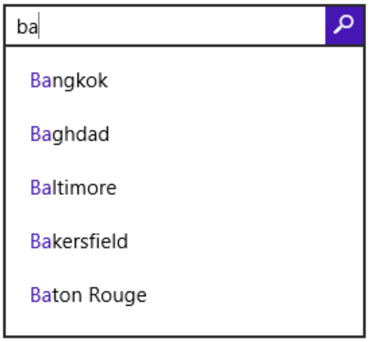
To supply the first type, in your handler for the suggestionsrequested event you populate the eventArgs.detail.searchSuggestionCollection with your suggested strings using the collection's methods named appendQuerySuggestion (for a single string) or appendQuerySuggestions (plural, which takes an array). Each string can be up to 512 characters.

Supporting multiple or east-Asian languages? Here you want to pay attention to the BCP-47 tag in the eventArgs.detail.language property if you support multiple languages and also to the linguisticDetails property that provides more information about text entered through an Input Method Editor (IME), specifically linguistic alternatives. If you expect to have Japanese or Chinese users, it's highly recommended to also search for these alternatives.

Here's how it's done in scenario 1 of the SearchBox control sample, where suggestionList is just a hard-coded list of city names and this handler is wired up to suggestionsrequested (js/S1-SearchBoxWithSuggestions.js):

```
function suggestionsRequestedHandler(eventObject) {
    var queryText = eventObject.detail.queryText,
        query = queryText.toLowerCase(),
        suggestionCollection = eventObject.detail.searchSuggestionCollection;

    if (queryText.length > 0) {
        for (var i = 0, len = suggestionList.length; i < len; i++) {
            if (suggestionList[i].substr(0, query.length).toLowerCase() === query) {
                suggestionCollection.appendQuerySuggestion(suggestionList[i]);
            }
        }
    }
}
```

So if query contains "ba", the first 5 names in suggestionList will be Bangkok, Baghdad, Baltimore, Bakersfield, and Baton Rouge:


ba

Bangkok

Baghdad

Baltimore

Bakersfield

Baton Rouge

Ideally, your event handler returns in half a second or less, and it's important to know that all the suggestions must be in the collection when your handler returns. Of course, a real app will be often drawing suggestions from its own database or from a service. In the process you might need to do some asynchronous work—to plug into the control's deferral mechanism, obtain a promise for your async work and pass it to `eventArgs.detail.setPromise`. Scenario 4 of the sample, for instance, calls an OpenSearch service via `WinJS.xhr` to provide suggestions (js/S4-SuggestionsOpenSearch.js):

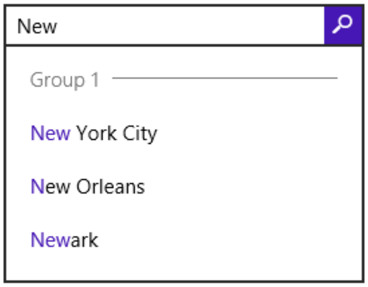
```
xhrRequest = WinJS.xhr({ url: suggestionUri });
eventObject.detail.setPromise(xhrRequest.then(
    function (request) {
        if (request.responseText) {
            var parsedResponse = JSON.parse(request.responseText);
            if (parsedResponse && parsedResponse instanceof Array
                && parsedResponse.length >= 2) {
                var suggestions = parsedResponse[1];
                if (suggestions) {
                    suggestionCollection.appendQuerySuggestions(suggestions);
                }
            }
        }
    }));
```

OpenSearch, in case you're unfamiliar with it, is a standard JSON format for search suggestions. See the OpenSearch Suggestions specification. The advantage of this, as we see in the code above, is that the JSON response can be directly parsed into an array and passed in one call to `appendQuerySuggestions`. There is also an XML format for search suggestions that's documented in the XML Search Suggestions Format Specification and demonstrated in scenario 5 of the sample. In this case, a function named `generateSuggestions` provides a generic parser routine for such a response, sending them onto `appendQuerySuggestion[s]` as well as `appendResultSuggestion`, which we'll see shortly.

In some cases you might want to group query suggestions together and divide those groups by separators (or divide result suggestions from query suggestions). For this, call the collection's `appendSearchSeparator` method with a label for the group, which is done in scenario 5 of the sample. Or, as a simpler demonstration, if you insert the following line before the `for` loop in the sample's scenario 1 (in js/S1-SearchBoxWithSuggestions.js):

```
suggestionCollection.appendSearchSeparator("Group 1");
```

you'll see the following in the `SearchBox` suggestions:

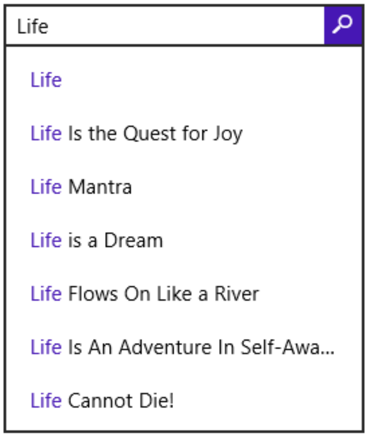


Now the other source for query suggestions is the local file system, as defined by one or more `StorageFolder` objects. Here you can have the `SearchBox` automatically provide suggestions by calling its `setLocalContentSuggestionsSettings` method. You need only do this once for any particular configuration, so you typically make the call when initializing the page or changing the page's data context, rather than from inside an event handler.

Here, you first create a new `LocalContentSuggestionSettings` object (in the `Windows.ApplicationModel.Search` namespace), set its `enabled` flag to `true`, populate its `locations` vector with the `StorageFile` objects you want to search, and then provide an Advanced Query Syntax filter string and/or an array of Windows Properties (like *System.Title*) that defines the scope of the search (in the `aqsFilter` and `propertiesToMatch` properties).

For example, scenario 3 of the SDK sample looks for music files in the Music library (js/S3-SuggestionsWindows.js, in the page control's `ready` method):

```
var localSuggestionSettings = new
    Windows.ApplicationModel.Search.LocalContentSuggestionSettings();
localSuggestionSettings.enabled = true;
localSuggestionSettings.locations.append(Windows.Storage.KnownFolders.musicLibrary);
localSuggestionSettings.aqsFilter = "kind:=music";

searchBox.winControl.setLocalContentSuggestionSettings(localSuggestionSettings);
```


Life

Life

Life Is the Quest for Joy

Life Mantra

Life is a Dream

Life Flows On Like a River

Life Is An Adventure In Self-Awa...

Life Cannot Die!

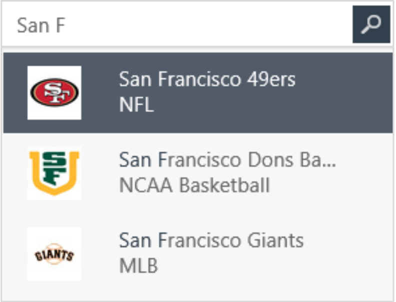
Because enumerating files in the `locations` folders requires programmatic access to those folders, you need to make sure your app has the appropriate capabilities set in its manifest, retrieves the folder from the `Windows.Storage.AccessCache`, or has obtained programmatic access through the folder picker. In the latter case, the app would provide UI elsewhere to configure the search locations (through the Settings pane, for instance).

The `aqsFilter` property determines which files in the folders vector will be searched. If you leave this blank, all files will be searched. The `propertiesToMatch` property (a string vector) then determines which file *properties* are involved. If you leave this blank, all properties will be searched; by setting this you can limit the scope however specifically you want. For details on AQS and Windows properties, refer to Chapter 11 in the section "Custom Queries."

Providing Result Suggestions

Providing query suggestions, as described in the previous section, is just a matter of providing possible search strings that will ultimately be passed to the app through to the SearchBox control's querysubmitted event. Typically this is used to give the user a list of possible search results.

Result suggestions are different in that they pinpoint a specific result right away, such that selecting one of those results goes immediately to that item, bypassing any search results page and streamlining the whole process. For example, when searching for a team name or player name in the Sports app, specific results can be identified from the underlying data set:



To provide result suggestions, you still handle the SearchBox.onsuggestionsrequested event as before but now take the extra step of calling the appendResultSuggestion method of the eventArgs.detail.searchSuggestionCollection. In fact, you'll typically want to populate result suggestions first, add a separator, and then populate query suggestions.

Caveat If you're using the search box's setLocalContentSuggestionSetting method to provide automatic query suggestions, any result suggestions you add will end up at the bottom of the list and won't be visible unless there are only a couple of query suggestions.

The appendResultSuggestion method takes five arguments in the following order, which fully describe the result. Be mindful of any necessary localization here:

- *text* The first line of text for the suggestion, such as "San Francisco 49ers" in the earlier graphic.
- *detailText* The second line of text for the suggestion, as with "NFL" in the graphic.
- *tag* The string you want to receive in the resultSuggestionChosen event (see below).
- *image* A RandomAccessStreamReference for the image to display (see below).
- *imageAlternateText* The alt attribute for the image.

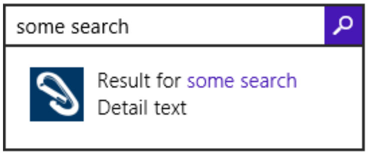
For the *image* argument, you can easily obtain the necessary object through the static createFromFile, createFromUri, and createFromStream methods that you'll find on the RandomAccessStreamReference class, depending on your image source (a StorageFile, a Windows.Foundation.Uri, or RandomAccessStream, respectively). The base size of this image is 40x40 for 100% scale, 56x56 for 140%, and 72x72 for 180%. Take these sizes into account if you dynamically generate images for the result suggestions or request them from a service, but the SearchBox will scale whatever image you provide.

Tip createFromUri accepts ms-appx:/// and ms-appdata:/// URIs along with http[s]://. Be sure to have a default image in your package in case you can't obtain a more specific one; there is no built-in default.

Here's a simple example to add a single suggestion that just reflects the query text and uses the app's store logo as a hack, but it shows the code you need:

```
var searchBox = document.getElementById("mainSearchBox").winControl;
searchBox.addEventListener("suggestionsrequested", suggestResults);

function suggestResults(e) {
    var uri = new Windows.Foundation.Uri("ms-appx:///images/storelogo.png")
    var stream = Windows.Storage.Streams.RandomAccessStreamReference.createFromUri(uri);

    e.detail.searchSuggestionCollection.appendResultSuggestion(
        "Result for " + e.detail.queryText, "Detail text", "result1", stream, "A suggestion");
}
```

This will produce the following result:



As noted in the previous section, the generateSuggestions function found in scenario 5 the SearchBox control sample provides a generic parser that turns XML search suggestions into the appropriate appendResultSuggestion calls. (You just have to follow through all the code.)

Providing a result suggestion, of course, is valuable only if the user can tap on it and get to that result immediately. For this you must also handle the resultsuggestionchosen event. For example, if you have a page called resultItem.html in which you'll show the result, you might handle the event as follows, where the eventArgs.detail.tag property is the same as the *tag* argument passed to appendResultSuggestion:

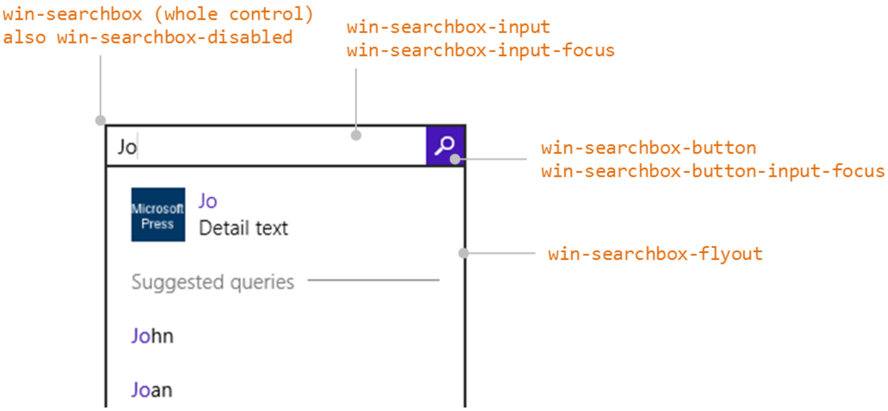
```
searchBox.addEventListener("resultsuggestionchosen", navToResult);

function navToResult(e) {
    WinJS.Navigation.navigate("/pages/resultItem.html", { item: e.detail.tag });
}
```

To summarize, handling the querysubmitted event means that you're taking the query text and populating a list of results in your own page. Each of those items will then navigate to an appropriate detail page. The resultSuggestionChosen event tells you that the same thing has happened directly within the SearchBox with a result that you placed there. For the most part, the detail pages to which you'll navigate are likely the same from either part of your UI.

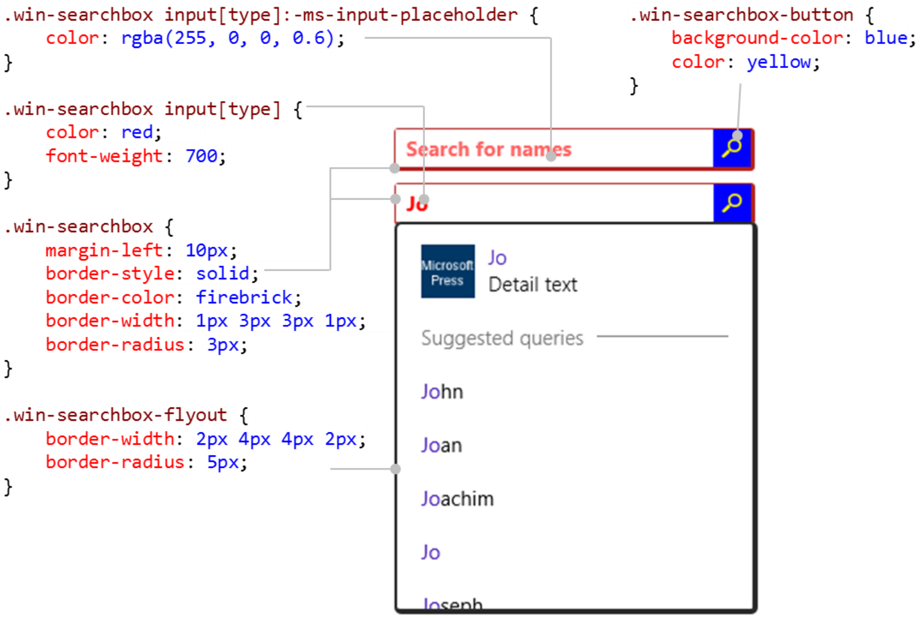
SearchBox Styling

As we've done with other controls in this book, the following annotated images identify which win-searchbox-* style classes are attached to which parts of the control. First of all, it's helpful to understand that the control as a whole (which is given the win-searchbox style as well as win-searchbox-disabled), is made up of three child elements:

- An <input> element where you type the query, which has the win-searchbox-input style class and win-searchbox-input-focus if it has the focus.
- A <button>, with the magnifying glass, which has the win-searchbox-button style class and win-searchbox-button-focus when it has the focus.
- A flyout containing the suggestions, tagged with win-searchbox-flyout.



With a few styles of our own, then, we can change the overall appearance like so:



```
.win-searchbox input[type]:-ms-input-placeholder {
    color: rgba(255, 0, 0, 0.6);
}

.win-searchbox input[type] {
    color: red;
    font-weight: 700;
}

.win-searchbox {
    margin-left: 10px;
    border-style: solid;
    border-color: firebrick;
    border-width: 1px 3px 3px 1px;
    border-radius: 3px;
}

.win-searchbox-flyout {
    border-width: 2px 4px 4px 2px;
    border-radius: 5px;
}
```

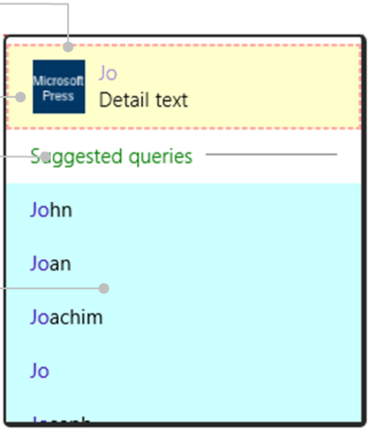
```
.win-searchbox-button {
    background-color: blue;
    color: yellow;
}
```

Note that it's necessary to use the `.win-searchbox input[type]` selector to style the `<input>` element rather than `.win-searchbox-input` because of CSS specificity. That is, although the latter class does identify this particular element, the WinJS stylesheet rules that affect it have a higher specificity and thus take precedence over any `win-searchbox-input` styles you assign yourself. So you have to use `.win-searchbox input[type]` to match that specificity. Note also the use of the `-ms-input-placeholder` pseudo-class for the placeholder text.

Tip You can see all this in Visual Studio's DOM Explorer. If you add styles for `win-searchbox-input`, the DOM Explorer will show that they're generally being overruled by styles from WinJS. Fortunately, that tool also shows the exact selectors that are involved, allowing you to write rules with the necessary specificity as I'm showing here.

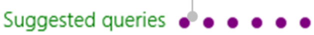
Now let's look at the content in the flyout, namely the sections that contain the result suggestions, separator, and query suggestions. I've used atrocious styles here so that we can clearly see what's going on—I hope you and your designers don't follow such a terrible example!

```
.win-searchbox-suggestion-result {
    border: dashed 2px #faa;
}

.win-searchbox-suggestion-selected {
    background-color: #ffc;
}

.win-searchbox-suggestion-separator {
    color: green;
}

.win-searchbox-suggestion-query {
    background-color: #cff;
}
```



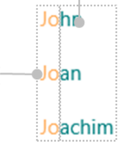
Each suggestion, as well as the separator, have their own component elements. For example, you can style the `<hr>` element of the separator specifically:

```
.win-searchbox-suggestion-separator hr {
    border-style: dotted;
    border-color: purple;
    border-width: 8px 0px 0px 0px;
}
```

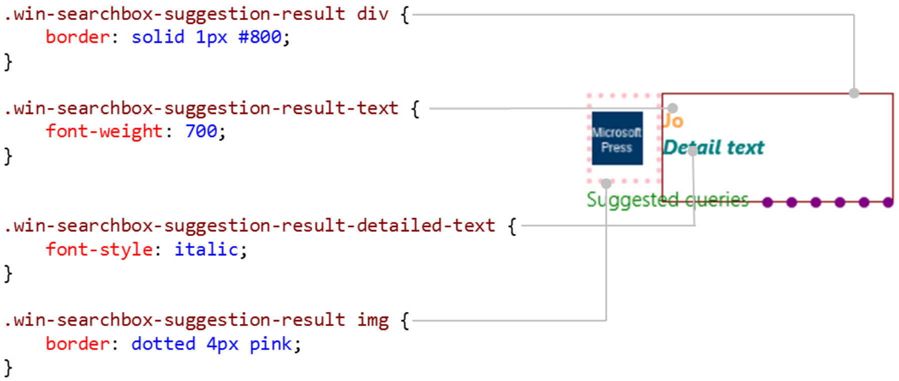


The query suggestions are made up of a `<span>` for the whole string, with a child `<span>` for the highlighted letters:

```
.win-searchbox-hithighlight-span {
    color: teal;
}

.win-searchbox-flyout-highlighttext {
    color: #fa5;
}
```



Finally, a result suggestion consists of a `div`, an `img`, the main result text, and the detail text. Here I'm still showing a styled separator so that you can see how the result `div` overlaps a bit, but it's where you'd adjust the margins between the image and the result text or otherwise set styles that affect the whole text area:



```
.win-searchbox-suggestion-result div {
    border: solid 1px #800;
}

.win-searchbox-suggestion-result-text {
    font-weight: 700;
}

.win-searchbox-suggestion-result-detailed-text {
    font-style: italic;
}

.win-searchbox-suggestion-result img {
    border: dotted 4px pink;
}
```

Indexing and Searching Content

A big part of implementing search capabilities in your app is not so much integrating the `SearchBox` UI as it is actually *performing* a search over your data for the `querysubmitted` event, as well as suggesting queries and results for that data. You can, of course, use the local content suggestions API of the `SearchBox` to easily search over the local file system, but how do you approach doing the same for your own local content, such as what exists in your app data? (For online data on a back-end service, you'll want to have that service implement search capabilities so that you can just submit a URI query string and get back the appropriate results.)

If you're using some kind of query-capable database for your storage, you'll simply perform your searches that way and translate the query results into the appropriate forms for your `SearchBox` suggestions and your results page. We talked about some of the options back in Chapter 10, "The Story of State, Part 1," in the section "IndexedDB, SQLite, and Other Database Options." (IndexedDB is not an inherently queryable database, by the way—to make it work like that, you have to build keyword indexes manually.)

Proper databases aside, there are two other options in the WinRT APIs: using indexed app data files and using the Content Indexer API for nonfile data, as the following sections describe.

Sidebar: Semantic Text Queries, Text Segmentation, and Unicode

For searching in-memory text, you can of course use any facilities that JavaScript provides, such as regular expressions. WinRT also provides an API in `Windows.Data.Text` that helps you identify segments in text that match a specific Advanced Query Syntax string, such that you can highlight those segments and so forth. This is very handy if you want to take content that the indexer APIs in this section identify as part of a result set and show the exact locations of search terms within those results. For a simple demonstration, refer to the Semantic text query sample.

Similarly, `Windows.Data.Text` has APIs to delineate individual words (the `WordsSegmenter` class) or selection ranges (`SelectableWordsSegmenter`) within a piece of text, with sensitivity to the user's language. These can be useful if you want to do your own segment highlighting and are demonstrated in the Text segmentation API sample.

A third sample of interest here is the Unicode string processing API sample, which shows using the `Windows.Data.Text.UnicodeCharacters` class "to tokenize lexical identifiers within a string…[detecting] surrogate pairs and [delimiting] the identifiers…." This class also works with different languages.

Using Indexed AppData Folders

If your content is file-based, store those files in a folder called *Indexed* (not case-sensitive) within your local or roaming app data (also mentioned in Chapter 10). By doing so, the system indexer will automatically process those files such that queries using `StorageFolder.createFileQuery` and `createFileQueryWithOptions` will return very quickly, as is essential when providing suggestions.

For more about these APIs, refer to Chapter 11 in the sections "Simple Enumeration and Common File Queries" and "Custom Queries." The short of it is that you use Advanced Query Syntax strings to write the query, as we'll see in a moment.

Any term in the AQS string that doesn't identify a particular Windows property is used in a full-text search, which is again very fast because the file contents are fully indexed (to whatever extent they can be indexed, of course, because some files are purely binary and/or do not support metadata). Those terms that *do* identify a Windows property (such as *System.Author: Jo*) will perform a search against file metadata. In the latter case you want to make sure to populate that metadata. If you're generating files in the *Indexed* folder programmatically, perhaps saving responses from HTTP requests into a cache, you can set each `StorageFile` object's properties directly by using its `properties` object and its `getBasicPropertiesAsync` method. Refer back to Chapter 11 in the section "File Properties" for all the details.

Alternately, you can generate what are called *appcontent-ms* files (with that extension, as in datafile.appcontent-ms) and place them in the *Indexed* folder. The appcontent-ms format is an XML format in which you can specify both properties and content together in text, which is typically a simpler way to go about it than writing file metadata through the `StorageFile` API. It also allows for indexing metadata with files whose formats don't support open metadata (like a .txt file). The Windows Reading List app takes advantage of this, for instance, generating an appcontent-ms file every time you add something to the list. To create some example files, then, just share content to the Reading List app through the Share charm, and then go to *%localappdata%\packages\Microsoft.WindowsReadingList_<hash>\RoamingState\indexed* and you'll see a file with content like the following, which I've annotated with comments to explain the schema:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- Root element can have any name -->
<ReadingListData>
  <Version>1.2</Version>
  <!-- Properties can contain Name, Keywords, Comment, and AdditionalProperties elements. -->
  <Properties xmlns="http://schemas.microsoft.com/Search/2013/ApplicationContent">
    <!-- Name corresponds to System.ItemName and System.ItemNameDisplay -->
    <Name>{00000000-0000-0000-7E31- 86D9A26AE4F}.appcontent-ms</Name>

    <!-- Keywords corresponds to System.Keywords and contains one or more Keyword children -->
    <Keywords>
      <Keyword>{69851795-6131-4b39-9079-3f592b6bd585}</Keyword>
    </Keywords>

    <!-- Comment corresponds to System.Comment -->
    <Comment>
      <!-- Any string -->
    </Comment>

    <!-- Name, Keywords, and Comment are the only strongly-typed properties; any others
        can be specified under AdditionalProperties with specific property names in
        the Key attribute. -->
    <AdditionalProperties>
      <Property xml:lang="en-US" Key="System.Title">
        Specs for built-in animations (animation metrics)
      </Property>
      <Property xml:lang="en-US" Key="System.Author">Internet Explorer</Property>
      <Property xml:lang="en-US" Key="System.Comment">The question occasionally comes up about
        the exact specifications for the built-in Windows animations that express the Windows
        look and feel. Often this is because someone wants to replicate the ef...</Property>

      <!-- Dates are ISO 8601 format -->
      <Property Key="System.DateAcquired">2013-12-12T15:32:18.6350000-08:00</Property>
      <Property Key="System.IsDeleted">false</Property>
      <Property Key="System.IsFlagged">false</Property>
    </AdditionalProperties>
  </Properties>

  <!-- Adjacent to the Properties elements, apps can include any other children for its
      own use. The Reading List app include an ItemData element, but this is not part
      of the appcontent-ms format) -->
  <ItemData>
    <!-- Other data -->
  </ItemData>
</ReadingListData>
```

In some cases a property might allow for multiple values, such as *System.Contact.EmailAddresses*, in which case each value is specified with a `<Value>` child element:

```
<Property xml:lang="en-US" Key="System.Contact.EmailAddresses">
  <Value>bryan@contoso.com</Value>
  <Value>vincent@contoso.com</Value>
</Property>
```

You can find additional examples in the Indexer sample, where scenarios 5, 6, and 7 demonstrate use of the *Indexed* folder and appcontent-ms files. The files here show that you can include any other custom elements in the XML (outside `<Properties>`) with the `IndexableContent="true"` attribute to include it in the indexing process (appcontent-ms/sample1.appcontent-ms):

```
<IndexerSampleSpecificElement sc:IndexableContent="true"
  xmlns:sc="http://schemas.microsoft.com/Search/2013/ApplicationContent">
  The text included here will be indexed, enabling full-text search.
</IndexerSampleSpecificElement>
```

To query the contents of an *Indexed* folder, you again use the `StorageFolder.createQuery*` methods and an AQS string. The query produces an array of `StorageFile` objects. You then use the metadata and contents of each individual file to generate your result suggestions and/or results page. If you want to display any kind of preview of each result, you could use this data if you like parsing XML, but it's generally easier to use the `Windows.Data.Text` APIs, which also have locale-sensitive support for word breaking, as noted earlier in "Sidebar: Semantic Text Queries, Text Segmentation, and Unicode."

Scenario 6 of the Indexer sample demonstrates querying, though nothing is different from what we saw in Chapter 11: the query will just complete quickly because the content is indexed. And this kind of query will work whether or not you're using appcontent-ms files (js/retrieveWithAppContent.js):

```
function retrieveAllItems() {
    var applicationData = Windows.Storage.ApplicationData.current,
        localFolder = applicationData.localFolder;
    var output;

    localFolder.createFolderAsync("Indexed",
    Windows.Storage.CreationCollisionOption.openIfExists).then(function (indexedFolder) {
        // Queries for all files in the "LocalState\Indexed" folder and sorts the
        // results by name
        var queryAll =
           indexedFolder.createFileQuery(Windows.Storage.Search.CommonFileQuery.orderByName);
        return queryAll.getFilesAsync();
    }).then(function (indexedItems) {
        var promiseArray = [];
        output = "";

        for (var i = 0, len = indexedItems.length; i < len; i++) {
           promiseArray[i] = indexedItems[i].properties.retrievePropertiesAsync(
               [Windows.Storage.SystemProperties.itemNameDisplay,
                Windows.Storage.SystemProperties.comment,
                Windows.Storage.SystemProperties.keywords,
                Windows.Storage.SystemProperties.title])
           .then(function (map) {
                // Retrieves the ItemNameDisplay, Comment, Keywords, and Title
                // properties for the item
               output += "Name: " + map[Windows.Storage.SystemProperties.itemNameDisplay];
               output += "\nKeywords: " + IndexerHelpers.createKeywordString(
                   map[Windows.Storage.SystemProperties.keywords]);
               output += "\nComment: " + map[Windows.Storage.SystemProperties.comment];
               output += "\nTitle: " + map[Windows.Storage.SystemProperties.title] + "\n\n";
           });
        }
        return WinJS.Promise.join(promiseArray);
    }).done(function () {
    });
}
```

Scenario 7 then shows how to apply a custom search term in a query—as you'd get from a `SearchBox`—where we employ a `QueryOptions` object and its `applicationSearchFilter` or `userSearchFilter` AQS strings (simplified from js/retrieveWithAppContent.js):

```
function retrieveMatchedItems() {
    var applicationData = Windows.Storage.ApplicationData.current,
        localFolder = applicationData.localFolder,
        queryOptions = new Windows.Storage.Search.QueryOptions();

    queryOptions.indexerOption = Windows.Storage.Search.IndexerOption.onlyUseIndexer;

    // Create an AQS (Advanced Query Syntax) query which will look for ItemNameDisplay
    properties which contain "Sample 1"
    queryOptions.applicationSearchFilter =
        Windows.Storage.SystemProperties.itemNameDisplay + ":\"Sample 1\"";

    var output;
    localFolder.createFolderAsync("Indexed",
        Windows.Storage.CreationCollisionOption.openIfExists).then(function (indexedFolder) {
        var query = indexedFolder.createFileQueryWithOptions(queryOptions);
        return query.getFilesAsync();
    }).done(function (indexedItems) {
        // Process the results
    });
}
```

Again, you don't have to query against any specific properties. If you set `queryOptions.userSearchFilter` to something without any property names, such as `"CSS styling"`, you'll use that term in a full text search against the file contents.

To remove files from the index, simply remove them from the *Indexed* folder by using methods like `StorageFile.deleteAsync` or `moveAsync`.

Using the ContentIndexer API

The second option for using the system indexer applies to nonfile content or content that cannot live in an *Indexed* folder. Here you employ the `Windows.Storage.Search.ContentIndexer` API, which allows you to do per-app indexing of just about any content you want (including language-specific content) using Windows Properties and full text. Once indexed, you can then query that content using AQS. This is convenient for many scenarios, and the index persists across app sessions so it's not necessary to re-create it every time the app is launched. Note, however, that because the index can be reset at any time, and because you cannot retrieve the indexed content itself, treat the index as just a durable cache and not as storage mechanism in itself.

You start with a call to the static method `ContentIndexer.getIndexer`, which takes an optional *name* argument that lets you maintain multiple independent indexes within the same app. To just use the default index, use this code:

```
var indexer = Windows.Storage.Search.ContentIndexer.getIndexer();
```

The object you get back is a ContentIndexer instance, whose sole property, revision, is automatically incremented with each add, update, or delete operation. We'll look at how to use this in a moment, because it won't mean anything until we add some content to the indexer through its addAsync method. To this you pass an instance of the Windows.Storage.Search.IndexableContent class that has the following properties:

Property Name	Description
Id	A unique app-defined string identifier for the content item.
properties	A Map (an object with the Windows.Foundation.Collections.IMap interface) of the content's properties (key-value pairs). You populate the map through its insert method using standard Windows property names (a string or a value from Windows.Storage.SystemProperties) and a value. What you set here determines what you can search for in the index, and can include the full textual content if needed.
stream	A RandomAccessStream object through which you can provide the full text of the content if it exists in a file or other data stream.
streamContentType	A string identifying the MIME content type of the stream property, if stream is used.

Examples of this are found in the SDK's Indexer sample, specifically scenarios 1, 2, 3, and 4. First, here's a bit of code from js/helperFunctions.js (in a routine called *_addItemsToIndex*, slightly simplified for clarity). This just adds three basic items to the default index and maintains a local appdata setting against which to match the revision property:

```
// Initialize the value used to track the expected index revision number.
var localSettings = Windows.Storage.ApplicationData.current.localSettings;

if (!localSettings.values["expectedIndexRevision"]) {
    localSettings.values["expectedIndexRevision"] = 0;
}

var indexer = Windows.Storage.Search.ContentIndexer.getIndexer();
var content = new Windows.Storage.Search.IndexableContent();
var numberOfItemsToIndex = 3;
var promiseArray = [];

for (var i = 0; i < numberOfItemsToIndex; i++) {
    var itemKeyValue = "SampleKey" + i.toString(),
        itemNameValue = "Sample Item Name " + i.toString(),
        itemKeywordsValue = "Sample keyword " + i.toString(),
        itemCommentValue = "Sample comment " + i.toString();
    content.properties.insert(Windows.Storage.SystemProperties.itemNameDisplay, itemNameValue);
    content.properties.insert(Windows.Storage.SystemProperties.keywords, itemKeywordsValue);
    content.properties.insert(Windows.Storage.SystemProperties.comment, itemCommentValue);
    content.id = itemKeyValue;
    promiseArray[i] = indexer.addAsync(content);
}
WinJS.Promise.join(promiseArray).then(function (resultArray) {
    // We can now query the index
    if (localSettings.values["expectedIndexRevision"] !== indexer.revision) {
        // There is a mismatch between the expected and reported index revision numbers
        if (indexer.revision === 0) {
            // The index has been reset, so code would be added here to re-push all data
        } else {
            // The index hasn't been reset, but it doesn't contain all expected updates, so
            // add code to get the index back into the expected state.
        }

        // After doing the necessary work to get back to a synchronized state, set the expected
        // index revision number to match the reported revision number
        localSettings.values["expectedIndexRevision"] = indexer.revision;
    }
});
```

Each item in the index here has three properties—a display name, keywords, and a comment—but you can add properties with any other Windows property key.

The addAsync method returns a promise, so all the promises are joined together so that we can take further action once we know all the items have been processed. Notice that the IndexableContent object can be reused after each addAsync call.

Important If you use the IndexableContent.stream property to index content, rather than a specific property, the objects you get back from queries will not include that content, only the properties. That is, whatever stream content exists in the IndexableContent object when you add it will be indexed, yes, and will be included in searches but will *not* be accessible through the search results. If you want to display that content in your results UI, make sure that the IndexableContent.id property is sufficient to retrieve the full content from wherever it's stored.

To remove content from the ContentIndexer, use its deleteAsync (passing an id), deleteMultipleAsync (passing an array of id's), and deleteAllAsync methods (see scenario 2 in the sample). To update an item (also in scenario 2), use the updateAsync method passing the updated IndexableContent object. Note that updateAsync will throw an exception if the index doesn't contain an item with the given id.

Tip To remove a property from an indexed item, use the properties.insert method for the same property name with a value of null, then call updateAsync.

Now that we understand how we manage the indexed content, we can talk about using the ContentIndexer.revision property. This specifically tells you about the state of the index itself:

- The revision value always starts at zero when you first obtain the ContextIndexer.
- Each successful add, remove, and update operation increments revision.
- If the index is cleared out—which should be a rare occurrence but can happen if the index becomes corrupted or the user manually rebuilds the index—then revision is reset to zero.

In most cases, you want to check revision anytime your app is activated or resumed—that is, whenever the app becomes active again after an indeterminate amount of time since it was last running. If the index was reset during that time, revision will be zero and you should repopulate all of your items. If revision is not zero and doesn't match your expectations, you have the very rare situation where some but not all of your items are in the index, perhaps due to an app crash or loss of power while the indexing was going on. At this point, if your number of items is small, you could just repopulate them all again. If you have a larger number of items, it makes sense to track (in your app data) which batches of items fall within which range of the revision count so that you can then just repopulate the affected batch.

As shown in the earlier code, you can take similar steps after a given batch if updates are complete. It's again possible that the index was reset during that time or that for some reason one or more items didn't make it in. Either way, you should you should attempt to repopulate the necessary items to get the index into the expected state. And then, of course, be sure that your expected count local setting matches the value of revision once you're all done so that the two don't get out of sync with later changes to the index.

Once you have content in the indexer, the next step is to use it for searches. In scenario 2 of the sample you'll also see calls to ContentIndexer.retrievePropertiesAsync. This is how you do a lookup of item properties in the indexer, because there isn't a method to retrieve the original IndexableContent object. That is, this method, given an item id and an array of property names, results in a map of property names and values:

```
indexer.retrievePropertiesAsync(itemKeyValue,
    [Windows.Storage.SystemProperties.keywords]).then(function (map) {
        var originalRetrievedKeywords = map[Windows.Storage.SystemProperties.keywords];
        // ...
    }
```

The last step with the indexer is querying its content (scenario 3 in the sample) through the ContentIndexer.createQuery method. The first two required arguments are *searchFilter*, the AQS string to apply, and *propertiesToRetrieve*, an array of Windows property names that you want returned. Optional arguments are *sortOrder*, an array of SortEntry objects (each of which indicates a property to sort by and the order, and multiple SortEntry objects are applied in the order they appear in the array), and *searchFilterLanguage*, a BCP-47 language tag (e.g., en-US) that indicates what language should be used to parse *searchFilter*. Here's a simple example that looks for any item with "0" in any of its properties (js/retrieveWithAPI.js):

```
var query = indexer.createQuery("0", [Windows.Storage.SystemProperties.itemNameDisplay,
    Windows.Storage.SystemProperties.keywords, Windows.Storage.SystemProperties.comment]);
```

The result of createQuery is a ContentIndexerQuery object through which you run the query and access the results:

Member	Description
getCountAsync	(Method) Provides the number of items in the query results.
getAsync	(Method) Runs the query, resulting in a vector of IndexableContent objects. Two optional integer arguments control how many items are returned from which position in the result set.
getPropertiesAsync	(Method) Retrieves a vector (array) of property-value maps. Two optional integer arguments control how many items are returned from which position in the result set. The vector will contain a map for each item in the result set, where the map contains each matching property and that property's value.
queryFolder	(Property, read-only) The StorageFolder that contains the indexed items. This allows use of the ContextIndexer API with all the StorageFolder query APIs that we saw in Chapter 11, which is convenient if you're using a ListView and a StorageDataSource, for example.

Using the query created in the previous bit of code (in scenario 3 again), here's the code to retrieve and process the results (js/retrieveWithAPI.js, slightly modified):[111]

```
var output;
query.getAsync().done(function (indexedItems) {
    output = createItemString(indexedItems);
    WinJS.log && WinJS.log(output, "sample", "status");
});

function createItemString(indexedItemArray) {
    var output;
    var wsp = Windows.Storage.SystemProperties;

    if (indexedItemArray) {
        output = "";
        for (var i = 0, len = indexedItemArray.length; i < len; i++) {
            var retrievedItemName = indexedItemArray[i].properties[wsp.itemNameDisplay],
                retrievedItemComment = indexedItemArray[i].properties[wsp.comment],
                retrievedItemKey = indexedItemArray[i].id,
                retrievedItemKeywords = indexedItemArray[i].properties[wsp.keywords];
            output += "Key: " + retrievedItemKey;
            output += "\nName: " + retrievedItemName;
            output += "\nKeywords: " +
                IndexerHelpers.createKeywordString(retrievedItemKeywords);
            output += "\nComment: " + retrievedItemComment;
            if (i < len - 1) {
                output += "\n\n";
            }
        }
    }
    return output;
}
```

[111] A comment in the sample erroneously states that getAsync retrieves an array of StorageItem objects. These are, as indicated in the table, IndexableContent objects.

The Search Contract

Although the SearchBox is the preferred way for an app to provide search capabilities, you can also implement the search contract directly to interact with the Search charm. In addition, if you register your app with Bing for Smart Search, this contract also comes into play (see Chapter 20).

In many ways, the interaction between the app and the Search pane is very similar to the SearchBox, with the same event names and other classes. I won't be going into details about those; I'll be referring to the previous sections instead.

Implementing the contract is demonstrated in the Search contract sample and begins with the Search target declaration in the app manifest. All you need here in Windows 8.1 is the declaration itself; you can leave the Start Page field blank because you'll search the app from the charm only when the app is in the foreground.

Next, instead of working with a SearchBox control for search-related events, you instead obtain the Windows.ApplicationModel.Search.SearchPane object. For example:

```
var searchPane = Windows.ApplicationModel.Search.SearchPane.getForCurrentView();
searchPane.onquerysubmitted = function (eventArgs) {
   WinJS.Navigation.navigate(searchPageURI, eventArgs);
};
```

> **Note** The SearchBox control will interact with the Search charm and contract on your behalf, so you should not attempt to implement the contract directly. Doing so will throw an Access Denied exception when you try to obtain the SearchPane object.

The SearchPane object has many of the same features as the SearchBox, including placeholderText, setLocalContentSuggestionsSettings, searchHistoryEnabled, and searchHistoryContext. For type-to-search, the property is called showOnKeyboardInput because this involves showing the Search pane as a whole and not just changing focus.

The SearchPane has a few unique features. Its show method allows you to show the pane programmatically, its visible property and visibilitychanged event will tell you its status, and the trySetQueryText is what you use to update the search term instead of just settings its queryText property directly.

Like the SearchBox, the SearchPane also has querychanged, querySubmitted, suggestionsrequested, and resultsuggestionchosen events that are identical to those of the SearchBox except for the fact that these are WinRT events, so you must call removeEventListener appropriately to avoid memory leaks.

The other difference with events is that the eventArgs.request.getDeferral for suggestionsrequested provides a deferral mechanism like other WinRT events, rather than the WinJS-style setPromise mechanism that we have with the SearchBox.

The piece you need to be aware of with the Search contract in a Windows 8.1 app is that it will again be used only for in-app searches and to invoke apps that have registered with Bing to have their content appear as web-powered search results. In the latter case, the app can be activated with `ActivationKind.search` through the search contract from the Smart Search results pane, and this is the only place where such activation will occur. The Search contract sample implements code for this activation path, but it won't ever be called through using the Search charm.

Contacts

When you first played around with a Windows 8 or Windows 8.1 device, you probably fired up the People app and added a few accounts such as Outlook (Live), Facebook, Twitter, Google+, LinkedIn, Skype, and so on. In doing so you could see that the People app nicely aggregates all those contacts into a single collection, combining whatever phone numbers, email addresses, and IM user names it can find from those sources.

What you probably didn't know is that the People app also serves as the systemwide, indexed repository of all those contacts, and that it provides contact-related features to all other apps. It's a glorified address book, in other words! But this means that as you implement people-related capabilities into your own app, you can save yourself a lot of time and trouble by working with those features instead of trying to manage contact information directly.

There are two ways to do this. The first is through a Windows 8.1 feature called *contact cards*, which have the significant benefit of keeping contact information isolated from the app (for security and privacy) and still provides many different actions for that contact. What's more, it's simple to use, and it's expected that most apps that simply want to display contact information and actions, and otherwise have no need to process contact information directly, will use contact cards.

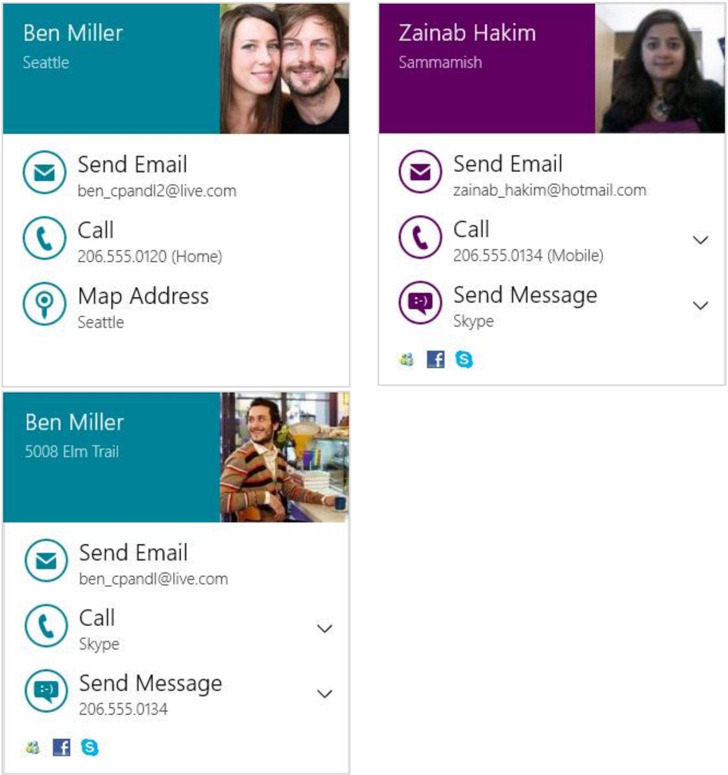
The second, which is available in both Windows 8 and Windows 8.1, is to invoke the Contact Picker. This brings up a contact provider app (which is the People app by default) through which the user can select one or more contacts, the information for which is then returned to the app directly. The app is responsible for protecting that information, of course, but can otherwise do whatever it needs with it.

We'll dive into each in turn in the following sections. For the provider/target side of each story—apps that handle contact card actions and/or serve as contact picker providers—see Appendix D.

Contact Cards

A *contact card* is a bit of system-provided UI that you invoke with whatever bits of contact information you have, which at minimum requires an email address or a phone number. Windows then does its best to find a matching contact and displays its contact card, drawing information from the system database and ignoring whatever you passed in. If no match is found, however, the contact card provides the option to add the contact to the database using the information you provide. Generally speaking, you want to give as much information as you can when invoking a contact card, which helps Windows find an existing contact and makes it easy for the user to add that contact if needed.

A few examples of contact cards are shown below:



Ben Miller
Seattle
Send Email
ben_cpandl2@live.com
Call
206.555.0120 (Home)
Map Address
Seattle

Zainab Hakim
Sammamish
Send Email
zainab_hakim@hotmail.com
Call
206.555.0134 (Mobile)
Send Message
Skype

Ben Miller
5008 Elm Trail
Send Email
ben_cpandl@live.com
Call
Skype
Send Message
206.555.0134

Tip The contact card's color scheme uses the app's tile background color setting in the manifest. However, if Windows determines that there's not enough contrast between this color and a black or white font color, it will revert to a black font on a white background to ensure legibility. Also note that if you change colors in the manifest and run the app from Visual Studio, you might not immediately see the proper change. Try running the app again, or, if needs be, uninstall first and then run again.

In addition to displaying contact information along with icons indicating the source of the contact's information, the user can take a number of actions directly from the card depending on what information is available (these have the same UI as the People app). The down arrows that appear on the right side of an action let you choose from different emails, phone numbers, messaging transports if available.

In general, contact card actions are handled by provider apps that implement that side of the contract, as discussed in Appendix D. The user can control the associations here through PC Settings > Search and Apps > Defaults. The Email action is an exception; it's handled by launching a `mailto:` URI. Phone numbers can also be handled through the `tel:` URI, but it's better for provider apps to implement the contract for that purpose.

What actions appear also depend on the contact information. In the above left image, for example, the contact isn't IM-capable, so a mapping option appears instead. More actions might be possible, of course, but the card displays up to the three most common ones. The rest are accessed through a More Details command that can appear on the lower right of the card and that opens the contact in the People app.

Again, there are two ways to invoke a contact card, which uses the API in the Windows.ApplicationModel.Contacts namespace. The first way is to provide minimal information and let Windows fill in the rest:

- Create an instance of the Contact class.
- Populate whatever properties of that object you can, at minimum either an emailAddress (a ContactEmail object) or phoneNumber (a ContactPhone object). This guarantees that something will appear in the contact card if no other information is found.
- Call ContactManager.showContactCard with the Contact object and placement information (to position the flyout in your UI).

These steps are shown in scenario 1 of the Contact manager API sample in the SDK. It starts by defining a few constants (js/ScenarioShowContactCard.js):

```
var ContactsNS = Windows.ApplicationModel.Contacts;

// Length limits allowed by the API
var MAX_EMAIL_ADDRESS_LENGTH = 321;
var MAX_PHONE_NUMBER_LENGTH = 50;
```

Then we create the Contact:

```
var contact = new ContactsNS.Contact();
```

The sample provides an input field for an email address and another for a phone number. Whichever ones aren't blank are added to the contact (code slightly simplified):

```
var emailAddress = document.getElementById("inputEmailAddress").value;
var phoneNumber = document.getElementById("inputPhoneNumber").value;

if (emailAddress.length > 0 && emailAddress.length <= MAX_EMAIL_ADDRESS_LENGTH) {
    var email = new ContactsNS.ContactEmail();
    email.address = emailAddress;
    contact.emails.append(email);
}

if (phoneNumber.length > 0 && phoneNumber.length <= MAX_PHONE_NUMBER_LENGTH) {
    var phone = new ContactsNS.ContactPhone();
    phone.number = phoneNumber;
    contact.phones.append(phone);
}
```

Tip As noted earlier, you can supply as much other information as you want when invoking the card, such as names (first and/or last), an address, job information, alternate phone numbers, a website URI, a thumbnail, and so on. However, all of this is ignored if Windows finds a match in the People app's aggregated data, which it will use for the contact card. If no data exists for the contact, however, the information you supply is what's shown in the card and the user will have an option to add the contact to the People app's database, as we'll see shortly.

Contact pictures To provide a picture for the contact, set the `Contact.thumbnail` property to a `RandomAccessStreamReference`, as obtained from that class's static methods `createFromFile`, `createFromUri`, and `createFromStream`. We've seen this for thumbnails elsewhere in this chapter.

Next we create an object describing where the information exists in our UI that was used to invoke the contact card. This could be an email address in a message header, a phone number in an IM, an address on a map, and so forth. In this case, `evt.srcElement` is a button, so we use its rectangle:

```
var boundingRect = evt.srcElement.getBoundingClientRect();
var selectionRect = { x: boundingRect.left, y: boundingRect.top,
    width: boundingRect.width, height: boundingRect.height };
```

Lastly, we invoke the contact card with the Contact and selection rectangle, along with an optional value from `Windows.UI.Popups.Placement` to indicate where the contact card should appear relative to that rectangle:

```
ContactsNS.ContactManager.showContactCard(contact, selectionRect,
    Windows.UI.Popups.Placement.default);
```

The placement values are `default`, `above`, `below`, `left`, or `right`. Unless you have a reason to do otherwise, though, use `default` so that Windows can place the flyout where it's fully visible.

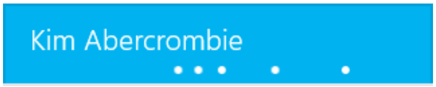
At the same time, there are cases when you do want to specify a placement directly, especially with the second means to invoke a contact card: `ContactManager.showDelayLoadedContactCard`. This method allows the app to go out and get its data asynchronously, perhaps through a database lookup or HTTP request.

To call this method, as demonstrated in scenario 2 of the Contact manager API sample, we create a Contact instance as before with whatever information we'd like to initially appear in the contact card (js/ScenarioShowContactCardDelayLoad.js):

```
var contact = new ContactsNS.Contact();
contact.firstName = "Kim";
contact.lastName = "Abercrombie";

var email = new ContactsNS.ContactEmail();
email.address = "kim@contoso.com";
contact.emails.append(email);

var boundingRect = evt.srcElement.getBoundingClientRect();
var selectionRect = { x: boundingRect.left, y: boundingRect.top,
    width: boundingRect.width, height: boundingRect.height };
```

```
var delayedDataLoader = ContactsNS.ContactManager.showDelayLoadedContactCard(
    contact, selectionRect, Windows.UI.Popups.Placement.below);
```

This displays a small contact card with just a name and a progress control (if you don't set the `firstName`/`lastName` properties, the email address will appear instead):



Kim Abercrombie

Note the use of `Placement.below` in this example rather than default. By default, a small contact card like this will likely be placed *above* the given rectangle, but once the rest of the data is populated, the contact card will likely become larger. As a result, the automatic placement could move the contact card to appear below the rectangle. To prevent that visual switch, the sample here is just forcing placement below the rectangle to begin with.

As before, you must provide either an email address or a phone number initially so that there's at least one action that can be taken from the contact card, especially if the app provides no other additional information.

The return value of `showDelayLoadedContact` card is an object of type `ContactCardDelayedDataLoader`. It has two methods of concern for apps written in JavaScript:

- `setData` Supplies a fully populated `Contact` object to the card and instruction Windows to show it.
- `close` Tells Windows that you don't have any more data and thus won't be calling `setData`. Windows will then just display the card with what you provided to `showDelayLoadedContact`.

Tip You have four seconds to call `setData`, otherwise the contact card will time out and call `close`.

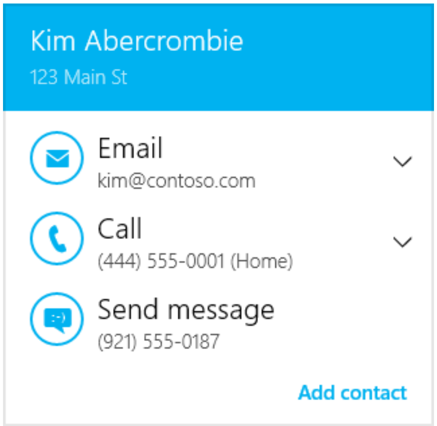
To demonstrate, scenario 2 of the sample uses `setTimeout` inside a promise to simulate making an async call to retrieve the contact data, transferring it into the `contact` object created earlier (js/ScenarioShowContentCardDelayLoad.js):

```
function downLoadContactDataAsync(contact) {
    return new WinJS.Promise(function (comp) {
        // Simulate the download latency by delaying the execution by 2 seconds.
        setTimeout(function () {
            // Add more data to the contact object.
            var workEmail = new ContactsNS.ContactEmail();
            workEmail.address = "kim@adatum.com";
            workEmail.kind = ContactsNS.ContactEmailKind.work;
            contact.emails.append(workEmail);

            var homePhone = new ContactsNS.ContactPhone();
            homePhone.number = "(444) 555-0001";
            homePhone.kind = ContactsNS.ContactPhoneKind.home;
            contact.phones.append(homePhone);

            var workPhone = new ContactsNS.ContactPhone();
            workPhone.number = "(245) 555-0123";
            workPhone.kind = ContactsNS.ContactPhoneKind.work;
            contact.phones.append(workPhone);

            var mobilePhone = new ContactsNS.ContactPhone();
            mobilePhone.number = "(921) 555-0187";
            mobilePhone.kind = ContactsNS.ContactPhoneKind.mobile;
            contact.phones.append(mobilePhone);

            var address = new ContactsNS.ContactAddress();
            address.streetAddress = "123 Main St";
            address.locality = "Redmond";
            address.region = "WA";
            address.country = "USA";
            address.postalCode = "23456";
            address.kind = ContactsNS.ContactAddressKind.home;
            contact.addresses.append(address);

            comp({ fullContact: contact, hasMoreData: true });
        },
        2000);
    });
}
```

Here's how it's used with the `ContactCardDelayedDataLoader` (code simplified):

```
downLoadContactDataAsync(contact).then(
    function complete(result) {
        if (result.hasMoreData) {
            delayedDataLoader.setData(result.fullContact);
        }
        else {
            delayedDataLoader.close();
        }
    }
}
```

Assuming all goes well and *result* gets the object from the promise that contains the updated Contact, you'll see a fuller contact card:



Kim Abercrombie
123 Main St

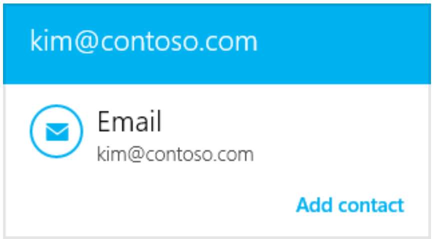
Email
kim@contoso.com

Call
(444) 555-0001 (Home)

Send message
(921) 555-0187

Here, the "Add contact" command at the bottom right—which appears also with `showContactCard` if the contact isn't found in the People app's data—will take the user to a page in the People app where all the information is prepopulated in a new contact form. The user just has to save that contact, and it'll be added to the system database.

If the app doesn't call `setData` within the four-second timeout or if it calls `close` instead of `setData`, you'll see a card with only the initial data:



kim@contoso.com

Email
kim@contoso.com

You can simulate this either by setting the interval in `setTimeout` to something greater than 4000 (simulating a timeout) or by changing the app's `hasMoreData` property in the promise's result (simulating a case where the app didn't retrieve any further data).

Using the Contact Picker

Contact cards, as we saw in the previous section, provide a convenient way to display information about a contact that appears somewhere in the app's UI. But how does such a contact get into the app's UI in the first place?

A user can, of course, just enter contact information into an app form of some kind, but this is terribly inconvenient for contacts that already exist in the system database. For this reason Windows provides the Contact Picker UI much along the same lines as the File Pickers we've already seen. Here the user can just select one or more contacts and have that information returned to the app.

An obvious place you'd need a contact is when composing an email, as shown in Figure 15-13 with the Mail app. Here, tapping the To or Cc controls will open the contact picker, which defaults to the Windows People app, as shown in Figure 15-14 (its splash screen) and Figure 15-15 (its multiselect picker view, where I have blurred my friends' identities so that they don't start blaming me for unwanted attention!). As we saw with the File Picker UI, the provider app supplies the UI for the middle portion of the screen while Windows supplies the top and bottom bars, the header, and the down-arrow menu control using information from the provider app's manifest. Figure 15-16 shows the appearance of the Contact Picker app sample in its provider mode (which we'll talk about more in Appendix D), as well as the menu that allows you to select a different provider (those who have declared themselves as a contact provider).

When I select one or more contacts in any provider app and press the Select button along the bottom of the screen, those contacts are then brought directly back to the first app—Mail in this case. Just as the file picker contract allows the user to navigate into data surfaced as files by any other app, the contact contract (say that ten times fast!) lets the user easily navigate to people you might select from any other source.

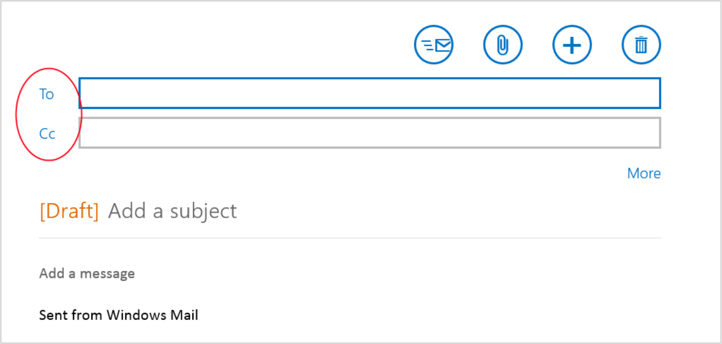


FIGURE 15-13 The Mail app uses the contact picker to choose a recipient.

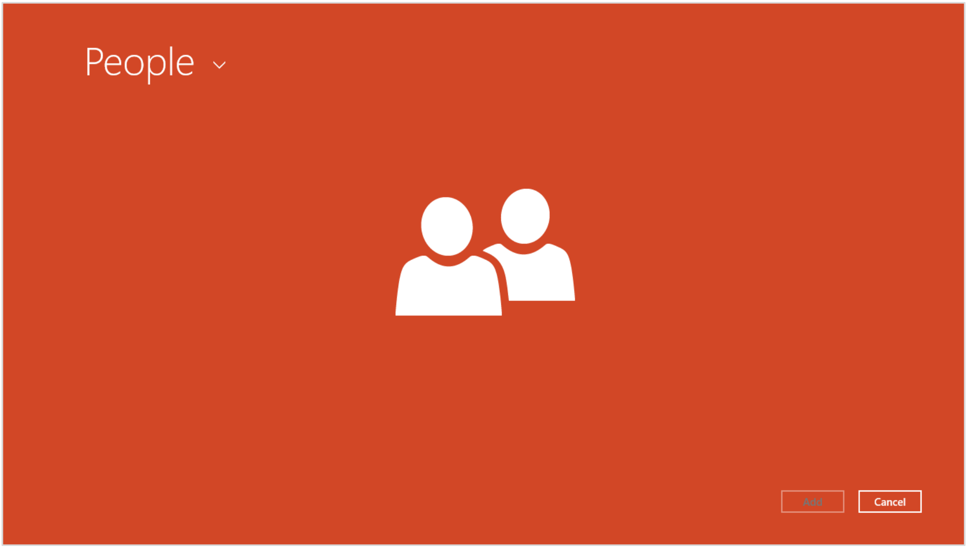


FIGURE 15-14 The People app on startup when launched as a contact provider.

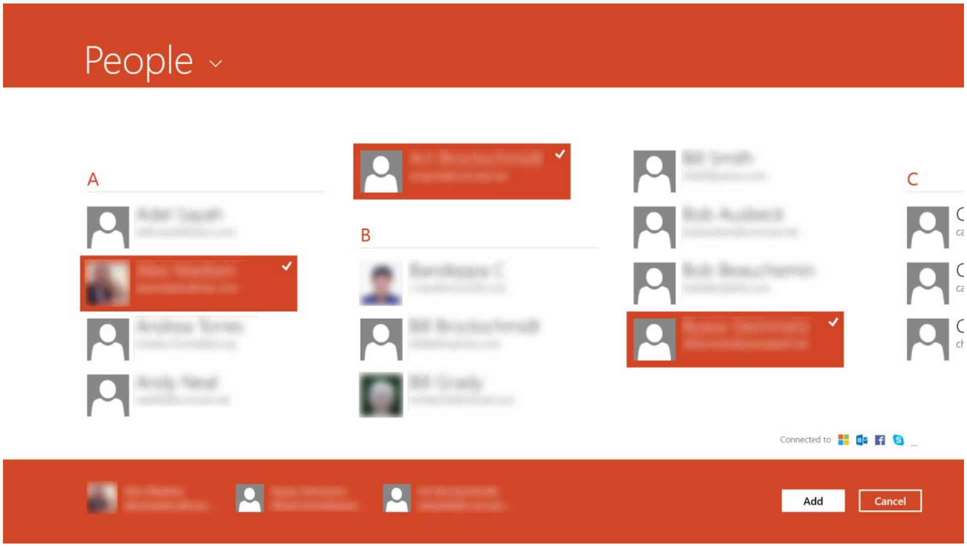
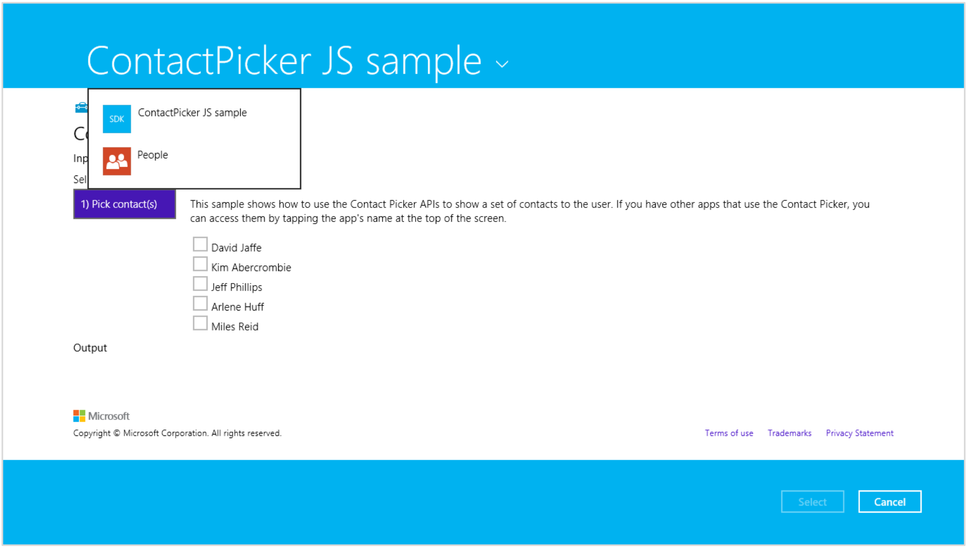


FIGURE 15-15 The picker UI within the People app, shown for multiple selection (with my friends blurred because they're generally not looking for fame amongst developers). The selections are gathered along the bottom in the basket.



This sample shows how to use the Contact Picker APIs to show a set of contacts to the user. If you have other apps that use the Contact Picker, you can access them by tapping the app's name at the top of the screen.

□ David Jaffe
□ Kim Abercrombie
□ Jeff Phillips
□ Arlene Huff
□ Miles Reid

Output

Microsoft
Copyright © Microsoft Corporation. All rights reserved.
Terms of use Trademarks Privacy Statement

FIGURE 15-16 The Contact Picker sample's UI when used as a provider, along with the header flyout menu allowing selection of a picker provider.

Invoking the contact picker happens through the ContactPicker object (in Windows.ApplicationModel.Contacts). After creating an instance of this object, you can set the commitButtonText for the first (left) button in the picker UI (as with "Select" in the earlier figures). You can also set the selectionMode property to a value from the ContactSelectionMode enumeration: either contact (the default) or fields. In the former case, the whole contact information is returned; in the latter, the picker works against the contents of the picker's desiredFields. Refer to the documentation on that property for details.

When you're ready to show the UI, call the picker's pickSingleContactAsync or pickMultipleContactsAsync methods. These provide your completed handler with a single Contact object or a vector of them, respectively, at which point you can work with whatever information in that object or objects you need.

Picking a single contact and displaying its information is demonstrated in scenario 1 of the Contact Picker app sample (js/scenarioSingle.js):

```
var picker = new Windows.ApplicationModel.Contacts.ContactPicker();
picker.commitButtonText = "Select";

// Open the picker for the user to select a contact
picker.pickSingleContactAsync().done(function (contact) {
    if (contact !== null) {
        // Consume the contact information...
    }
});
```

Choosing multiple contacts (scenario 2, js/scenarioMultiple.js) works the same way, just using pickMultipleContactsAsync. In either case, the calling app then applies the Contact data however it sees fit, such as populating a To or Cc field like the Mail app. Be mindful that a number of properties, such as addresses, emails, and phones (among others), are themselves vectors of other object types, such as ContactAddress, ContactEmail, and ContactPhone, with their own set of properties. I don't want to bore you by going into the details, so look through the documentation for the Contact class as appropriate. Scenario 1 of the sample has some code that shows how to consume these vector properties by iterating them with their forEach methods.

Appointments

Working with the user's calendar is very similar to working with their address book through contact cards. Every user has some kind of calendar (typically associated with their Microsoft account), for which the built-in Calendar app is the default provider. Most apps, however, don't need to own or manage the entire calendar—they just need to create and manage individual appointments on that calendar, as with booking travel, making restaurant reservations, or arranging meetings with friends over other communication channels.

That said, it'd be sheer lunacy to just let any arbitrary app create and remove appointments programmatically without some kind of user consent—some creative soul, if we're generous enough to refer to them that way, would surely delight themselves by subjecting your calendar to outrageous spamming experiments!

The Windows.ApplicationModel.Appointments API serves as a broker between apps and the user's calendar (managed by whatever app is serving as the provider), giving users control over the process. As with contact cards, an app populates the suitable object instance, in this case an Appointment, and then asks the AppointmentManager to add, remove, or update that entry. In response, the API launches the appointments provider and gives it a space in which to display appropriate UI for the operation. For details on writing an appointment provider app, refer to Appendix D.

An Appointment is a rather detailed object whose properties are described in the following table:

Property	**Type (max length)**	**Description**
startTime	Date	The date and time for the beginning of the appointment.
duration	Number	The length of the appointment in milliseconds. (Note that if you see references to 100ns TimeSpan units, that's how it appears in other languages; it's projected into JavaScript as miliseconds.)
allDay	Boolean	If true, indicates that the appointment occupies the entire date in startTime and that duration should be ignored.
subject	string (255 char max)	The appointment's title.
location	string (32K char max)	The location of the appointment.
details	string (1M char max)	A description body.
organizer	AppointmentOrganizer	An object that describes the person making the appointment.
invitees	Vector of AppointmentInvitee objects	The list of invited attendees.

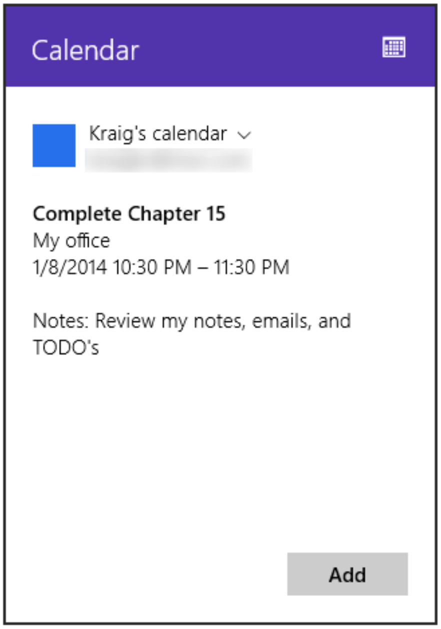
recurrence	AppointmentRecurrence	An object describing the appointment's recurrence, if needed.
reminder	Number	The time in milliseconds before the appointment to issue a reminder.
busyStatus	AppointmentBusyStatus	Indicates how an attendee's status will show during the appointment, one of busy (default), tentative, free, outOfOffice, and workingElsewhere.
sensitivity	AppointmentSensitivity	Indicates public (default) or private.
uri	Windows.Foundation.Uri	A URI to associate with the appointment that can bring the user back to the source app from the calendar.

For a demonstration of creating and populating an Appointment object, refer to scenario 1 of the Appointments API sample (about 200 lines of code in js/AppointmentProperties.js), as there are additional details for populating objects like AppointmentInvitee. Scenario 6 is there you'll find code to set up a recurring appointment.

Note that scenario 1 here, although it's described as "Create an Appointment," only creates an Appointment object; it does not add it to the user's calendar. To take that step, look at scenario 2, or better still, copy the following bit of code from scenario 2 into scenario 1 at the end of the create-Appointment function. Note that I've added the *e* argument to the function so that we get the bounding rectangle of the invoking button):

```
function createAppointment(e) {
    var boundingRect = e.srcElement.getBoundingClientRect();
    var selectionRect = { x: boundingRect.left, y: boundingRect.top,
        width: boundingRect.width, height: boundingRect.height };

    Windows.ApplicationModel.Appointments.AppointmentManager.showAddAppointmentAsync(
        appointment, selectionRect, Windows.UI.Popups.Placement.default)
       .done(function (appointmentId) {
           // appointmentId is non-null if the appointment was added
    });
}
```

As you can see here, the AppointmentManager.showAddAppointmentAsync method is what you call to add the appointment, where the Placement value is optional. This displays a piece of UI from the appointments provider app, shown below, in which the user confirms the addition by tapping Add or rejects it by tapping outside the popup:



Kraig's calendar

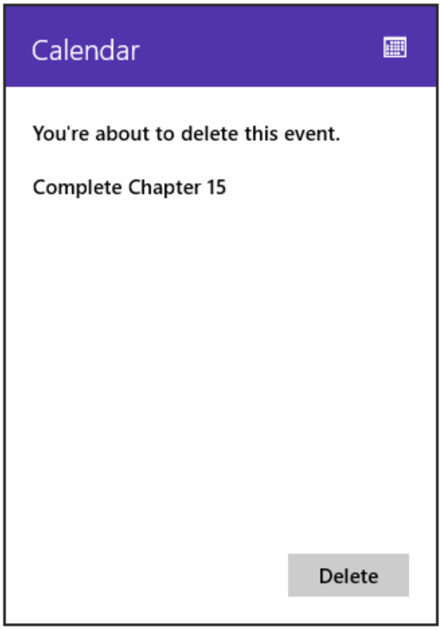
Complete Chapter 15
My office
1/8/2014 10:30 PM – 11:30 PM

Notes: Review my notes, emails, and TODO's

Again, this bit of UI comes from the appointments provider app, not Windows specifically—in fact, you'll see the app's splash screen appear in the popup at first. The title bar of the popup also reflects the provider app's branding of course.

If the appointment is added successfully, the completed handler you give to `showAddAppointmentAsync` will receive a unique appointment id that you can save in your local or roaming app data for later reference on any of the user's devices (as they will all be tied to the same user's calendar).

If you will later want to manage the appointments you've created, be sure to save their ids. To remove an appointment, pass its id to the `showRemoveAppointmentAsync` method, in which case another bit of provider-supplied UI will appear, asking the user to confirm:



Calendar

You're about to delete this event.

Complete Chapter 15

Similarly, you can update an appointment by passing its id and the new Appointment object to the showReplaceAppointmentAsync method, which will again show a little provider UI through which the user confirms the action.

The one other method of the AppointmentManager object, showTimeFrame, lets you conveniently invoke the appointments provider app for a particular date and time range. Scenario 5 of the sample uses the current date with a range of one hour (js/ShowTimeFrame.js):

```
var dateToShow = new Date();
Windows.ApplicationModel.Appointments.AppointmentManager.showTimeFrameAsync(
    dateToShow, (60 * 60 * 1000)).done(function () {
        // ...
    });
```

In this case, the API launches the provider app separately in a side-by-side view and not in a flyout. How the provider then displays that particular time frame is up to it—the built-in Calendar app, for its part, shows the current day and the next day together. Other implementations could show just the current day. What's important, though, is that you'd use this API from an app to let the user see what other appointments are near to one they're trying to create, so the exact details aren't important.

What We’ve Just Learned

- Contracts provide the ability for any number of apps to extend system functionality as well as extend the functionality of other apps. Through contracts, installing more apps that support them creates a richer overall environment for users.
- The Share contract provides a shortcut means through which data from one app can be sent to another, eliminating many intermediate steps and keeping the user in the context of the same app. A source app packages data it can share when the Share charm is invoked; target apps consume that data, often copying it elsewhere (in an email message, text message, social networking service, and so forth).
- The Share target provides for delayed rendering of items (such as graphics), for long-running operations (such as when it’s necessary to upload large data files to a service), and for providing quicklinks to specific targets within the same app (such as frequent email recipients).
- The Search contract provides integration between an app and the Search charm. From the charm users can search the current app as well perform broad searches on local content and the web. The search contract allows apps to also provide query suggestions and result suggestions.
- In-app search is more commonly implemented by using the `WinJS.UI.SearchBox` control, whose interface is very similar to the search contact and provides the same capabilities.
- To simplify searches, apps can store files in the Indexed folder in local app data, using appcontent-ms XML files to include metadata for searches. Alternately, apps can use the `ContentIndexer` API to add arbitrary content and metadata to the system index.
- File type and URI scheme associations are how apps can launch other apps. An app’s associations are declared in its manifest allowing it to be launched to service those associations. URI scheme associations are an excellent means for an app to provide workflow services to others.
- Working with contacts can happen through contact cards, which provide the users with convenient actions for a contact, or through the Contact Picker wherein the user can choose one or more contacts for the app to manipulate.
- The user’s calendar is made available to apps through the Appointments API, where the provider app supplies bits of UI to confirm additions, updates, and deletions. Apps can also ask the provider to display appointments for a particular time frame.

Chapter 16

Alive with Activity: Tiles, Notifications, the Lock Screen, and Background Tasks

At the risk of seriously dating myself once again, I still remember how a friend and I marveled when we first acquired modems that allowed us to do an online chat. At that time the modems ran at a whopping 300 baud (not Kb, not Mb—just b) and we connected by one of us calling the other's phone number directly. It would have been far more efficient for us to just talk over the phone lines we were tying up with our bitstreams! Such were the early days of the kind of connectivity we enjoy today, where millions of services are ready to provide us with just about any kind of information we seek with transfer speeds that once challenged the limits of believability.

Even so, almost from the genesis of online services it's been necessary to enter some kind of app, be it a client app or a web app, to view that information and get updates. When computers could run *only* a single app at a time (like the one I was using with that 300 baud modem[112]), this could become quite cumbersome, and it made it difficult, if not impossible, to move data from one program to another. With the advent of multitasking operating systems like Windows, you could run apps side-by-side and transfer information between them, a model that has stuck with us for several decades now. Even many web apps, for the most part, still operate this way. There have been innovations in this space, certainly, such as mashups that bring disparate information together into a more convenient place, but such an experience is still often hidden within an app.

Windows 8 changed that. As one columnist put it, "Using Windows 8 is like living in a house made out of Internet…The Start screen is a brilliant innovation, [a] huge improvement on the folder-littered desktops on every other OS, which serve exactly no purpose except to show a background photo. The Start screen makes it possible to check a dozen things"—if not more, I might add!—"in five seconds—from any app, just tap the Windows key and you can check to see if you have a new email, an upcoming appointment, inclement weather, or any breaking news. Tap the Windows key again and you're back to your original app."[113] He goes on to suggest how long this would take if you had to go into individual apps to check the same information, even with high-speed broadband!

What makes the Start screen come alive in this way are what we call *live tiles*, Microsoft's answer to

[112] If you want the actual make and model, you'll have to look for it in the footnotes of Chapter 1 of my book *Mystic Microsoft*, found on mysticmicrosoft.com or through my website, kraigbrockschmidt.com.

[113] From This is my next: Windows 8, by David Pierce.

the need to bring information from many sources together at the core of the user experience, an experience that "is constantly changing and updating," as the same writer puts it, "because its every fiber is connected to the Internet." With live tiles, each one is a small window onto whatever wealth of information an app is built around; the app is essentially extracting the most important pieces of that information according to each user's particular interests. And as the user adds more apps to the system—which adds more tiles to the Start screen that the user can rearrange and group however he or she likes—the whole information experience becomes richer.

Even so, live tiles and the Start screen are just the beginning of the story. It's ironic that this chapter has one of the longest titles in this book, listing off four things that do not, at first glance, appear to be related: tiles, notifications, the lock screen, and background tasks. Maybe you're just thinking that I couldn't figure out where else to put them all! In truth, they together form what is essentially a single topic: how apps work with Windows to create an environment that is constantly *alive with activity* while those apps often are not actually running or are allowed to run just a little bit.

Let's begin by exploring those relationships and the general means through which apps wire their tiles and other notifications to their information sources.

Before going further Refer back to "Systemwide Enabling and Disabling of Animations" early in Chapter 14, "Purposeful Animations," and check the PC Settings > Ease of Access > Other Options > Play Animations in Windows option. If animations are turned off, live tiles won't be animated and you won't see the complete experience they can provide.

Second, because all the topics of this chapter are related to one another, various aspects that I'll discuss in one part of this chapter—in the context of tiles, for example—also apply to other parts—such as toast notifications. For this reason it is best to read this chapter from start to finish.

Third, many aspects of what we cover in this chapter are not enabled within the Windows Simulator, such as live tiles, toast notifications, and the lock screen. When running some of the samples within Visual Studio, be sure to use the Local Machine or Remote Machine debugging options.

Finally, the tile and notifications API is generally found within `Windows.UI.Notifications`, which is a lot to spell out every time. Unless noted, assume that the WinRT APIs we're talking about come from that namespace.

Alive with Activity: A Visual Tour

After an app is acquired from the Windows Store and installed on a device, a user can pin its primary *app tile* is to the Start screen. As shown in Figure 16-1, these tiles can appear in four sizes: small (70x70 at 100% scaling), medium (150x150), wide (310x150), and large (310x310). The available options depend on what graphics the app provides in its package (for that full list, refer back to Chapter 3, "App Anatomy and Performance Fundamentals"). To customize a tile, the user can tap-and-hold (using touch) or right-click (using the mouse) a tile to bring up tile commands, as shown in Figure 16-2. Here she can unpin the tile, uninstall the app, change the tile size, and disable updates.

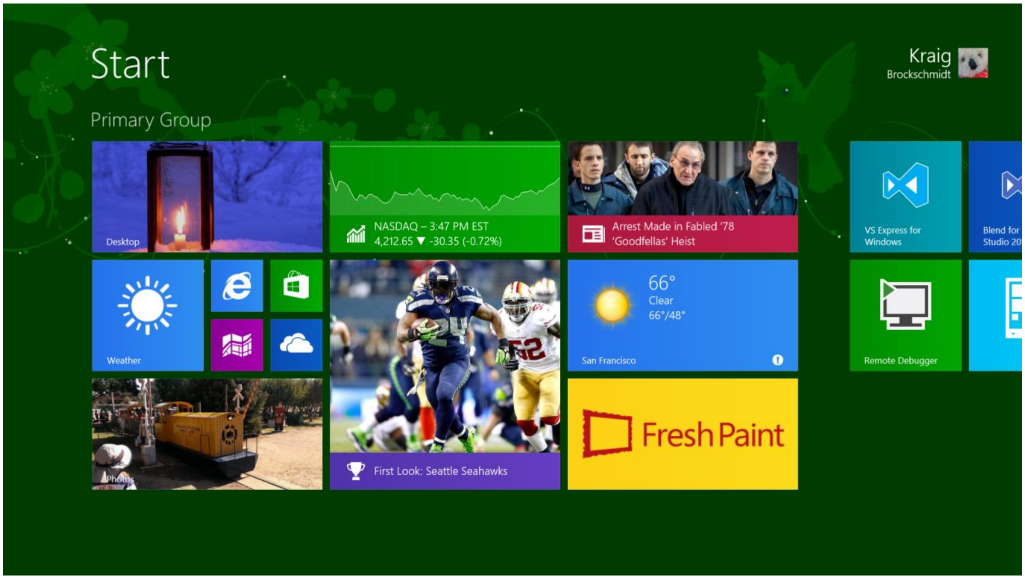


FIGURE 16-1 A typical Start screen with a variety of tile sizes. This happens to be my current configuration, where you can see a few development tools on the right.

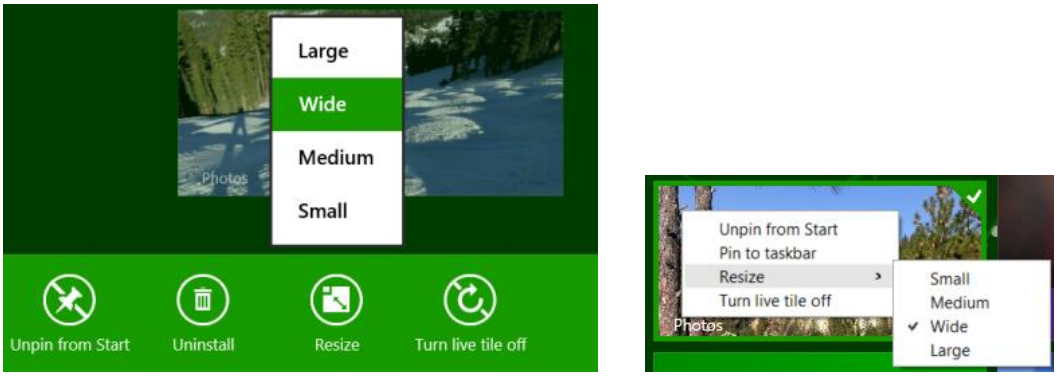


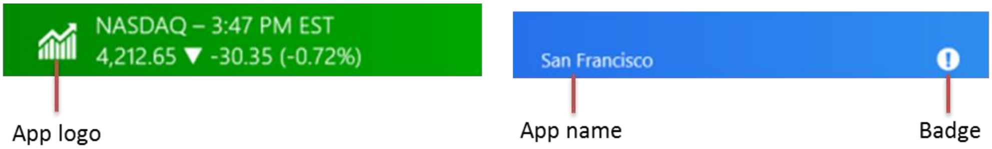
FIGURE 16-2 The app bar commands (left, for touch) and the menu (right, for mouse right-click and the keyboard menu key or Shift+F10) for a tile, including the options for changing a tile's size. The Turn Live Tile Off command will disable updates for a given tile, so be careful not to annoy your customers with too much noise! Note also that the mouse right-click menu is available starting with Windows 8.1 Update 1.

When you first install Windows on a device, you'll be greeted with a Start screen that lights up quite quickly, even before you've run any apps.[114] Tiles for new apps that you install from the Windows Store but haven't yet run can also become active right away (there's a way to do this in the app manifest). In

[114] This is assuming you have Internet connectivity, which I mention with great irony because when I first wrote this chapter there was a fiber optic breakdown between Sacramento and Oakland, California, that had myself and many thousands of others completely offline!

short, whatever Start screen you see when you first install Windows or get a new device won't stay that way for long! Many tiles will change every few seconds as you can easily see by returning to your own Start screen.

What can appear on any given tile is quite extensive and varied. As you can see in Figure 16-1, everything but the smallest tiles can display text, images, and an app name or logo (at the lower left). All tiles, including the small size, can also show small glyphs or numbers called *badges* at the lower right. Here are some examples of all of these:



A key feature of "alive with activity" is that tiles can receive updates when the app is suspended or not otherwise running, as we'll see next in "The Four Sources of Updates and Notifications." Tiles can also cycle through up to five updates, an important feature that reduces the overall number of updates that actually need to be retrieved from the Internet (thus using less power). That is, by cycling through different updates, a tile can appear very alive without constantly pinging a service.

Copious inspiration! Hidden in the documentation at the end of the lengthy Tile template catalog is a section on Using live tiles that gives many examples of different design possibilities. It's a great place to spend some time during the design phase of your project.

Tip Even though live tiles can be updated frequently through push notifications, be careful not to abuse that right. Think of live tiles as views into app content rather than gadgets: avoid trying to make a live tile an app experience unto itself (like a clock) because you cannot rely on high-frequency updates or update order. Furthermore, a tile update consists only of XML that defines the tile content—updates cannot trigger the execution of any code. In the end, think about the real experience you want to deliver through your live tile and use the longest update period that will still achieve that goal.

In the introduction I mentioned how acquiring more apps from the Windows Store is a way that the Start screen becomes increasingly richer. But new apps are not the only way that more tiles might appear. Apps can also create *secondary tiles* with all the capabilities of the app tile. Secondary tiles are essentially bookmarks into an app. A secondary tile is created from within an app, typically through a Pin command on its app bar. Upon the app's request to create the tile, Windows automatically prompts the user for confirmation, as shown for the Weather app in Figure 16-3, thus always keeping the user in control of the Start screen (that is, apps cannot become litterbugs on that real estate!).

In this case the Weather app lets you pin secondary tiles for each location you've configured; the secondary tile always includes specific information that is given back to the app when it's launched, allowing it to navigate to the appropriate page. The secondary tile flyout also lets the user choose the size of the tile. Apps must support at least small and medium tile sizes; wide and large sizes are optional (supporting large requires that you also support wide). In addition, apps can provide a selection of images for each tile size, all of which are available through the flyout's FlipView.

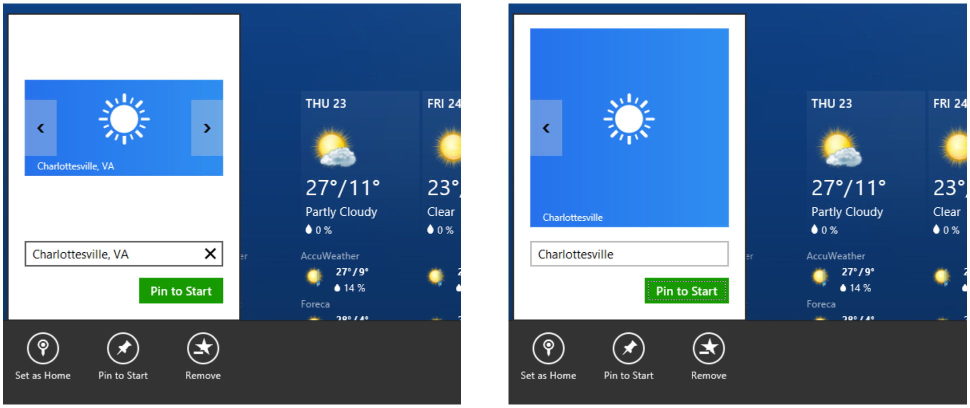


FIGURE 16-3 Pinning a secondary tile in the Weather app by using a Pin To Start app bar command, shown here with the automatic confirmation flyout in which you can select a tile size and image.

In the People app, similarly, you can pin—that is, create secondary tiles for—specific individuals. In the Mail app you can pin different accounts and folders. In Internet Explorer you can pin your favorite websites. You get the idea: secondary tiles let you populate the Start screen with very personalized views into different apps. The user can also unpin any app tile at any time, including the primary app tile (as can happen when one has created a number of secondary tiles for more specific views). An app can itself ask to unpin a secondary tile programmatically, in response to which Windows will again prompt the user for confirmation.

> **User tip** The default customization view of the Start screen (which appears when you tap-and-hold a tile, or invoke the Customize command on the app bar) lets you rearrange individual tiles and name groups. To rearrange the groups as a whole, do a semantic zoom out on the Start screen (a pinch gesture, Ctrl+mouse wheel down, or the Ctrl+minus key), then drag-and-drop groups as you'd like.

In many ways, live tiles might reduce the need for a user to ever launch the app that's associated with a tile. Yet this isn't really the case. Because tiles are limited in size and must adhere to predefined configurations (templates), they provide essential details while simultaneously serving as teasers. They give you enough useful information for an at-a-glance view but not so much that your appetite for details is fully satisfied. Instead of being a deterrent to starting apps, they're actually an invitation: they both inform and engage. For this reason, live tiles should be considered an essential app feature where they are appropriate.

I encourage you to get creative with what kinds of interesting information you might surface on a tile, even if your app doesn't draw from web-based content. Games, for example, can cycle through tile updates that show progress on various levels, high scores, new challenges, and so forth—all of which invite the user to re-engage with that app. Do remember, though, that the user can always disable updates for any given tile, so don't give them a reason to defeat your purpose altogether!

As additional background on live tiles, check out the Updating live tiles without draining your battery post on the Building Windows blog. It's good background on the system's view of efficiently managing tiles.

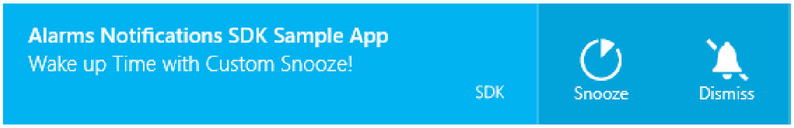
Now, for all the excellence of live tiles, the Start screen isn't actually where users will be spending the majority of their time—we expect them, of course, to mostly be engaged in apps themselves. Even so, users may want to be notified when important events occur, such as the arrival of an email, the triggering of an alarm, or perhaps a change in traffic conditions that indicates a good time to head home for the day (or a change in weather conditions that indicates a great time to go out skiing!).

For this purpose—surfacing typically time-sensitive information from apps that aren't in the foreground—Windows provides *toast notifications*. These transient messages pop up (like real toast but without the bread crumbs) in the upper right corner of the screen (or the upper left in right-to-left languages). They appear on top of the foreground app, as shown in Figure 16-4, as well as on the Start screen, the desktop, and in certain cases the lock screen. Up to three toasts can appear at any one time, and each can be accompanied by a predefined sound, if desired. Apps can issue toasts to appear immediately or can schedule them to appear sometime in the future.

Toasts are, like tile updates, created using predefined templates and can be composed of images, text, and logos; they always use the originating app's color scheme, as defined in that app's manifest (the Foreground Text and Background Color settings in the Visual Assets section).

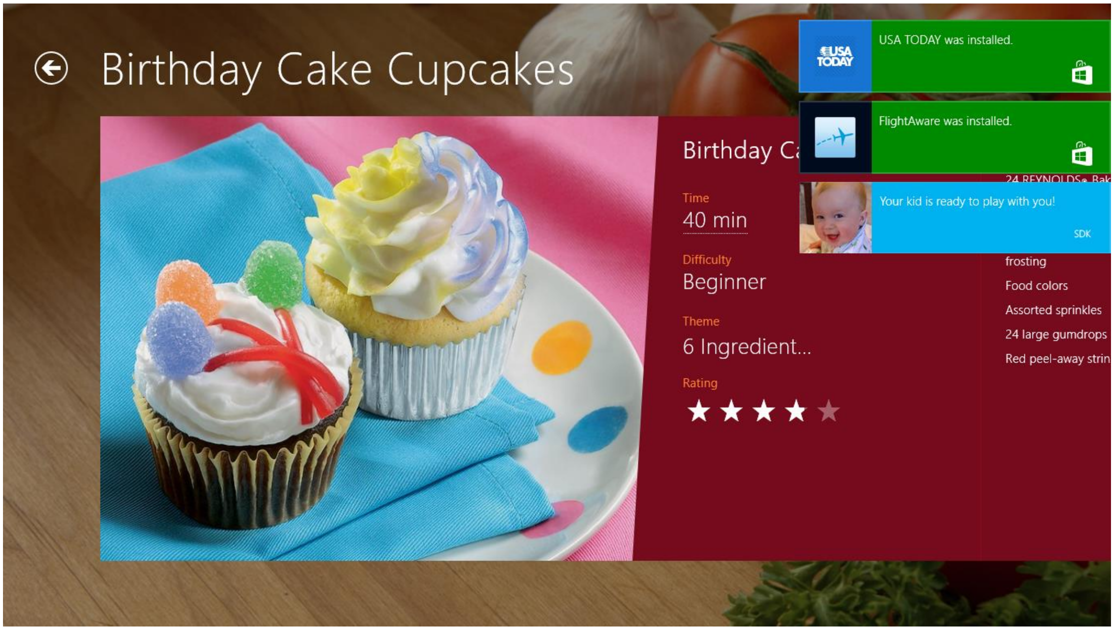
The purpose of toasts is, again, to give the user alerts and other time-sensitive information, but by default they appear only for a short time before disappearing. The default toast duration is five seconds, but this can be set to as long as five minutes in PC Settings > Ease of Access > Other Options, as shown in Figure 16-5. Apps can create long-duration toasts that remain visible for 25 seconds or the Ease of Access setting, whichever is longer. Furthermore, apps can create a toast with looping audio for events like a phone call or other situation where another human being might be waiting on the other end and it's appropriate to keep the notification active for some time.

There is also a special case for *alarm* toast notifications, as employed by the Windows Alarms app. Alarm toasts can include Snooze and Dismiss commands:



Alarms Notifications SDK Sample App
Wake up Time with Custom Snooze!

A few similar commands are also available for VoIP apps, as we'll see later.


Birthday Cake Cupcakes

Birthday Ca

Time
40 min

Difficulty
Beginner

Theme
6 Ingredient...

Rating
★★★★☆

24 REYNOLDS® Bak

frosting

Food colors

Assorted sprinkles

24 large gumdrops

Red peel-away strin

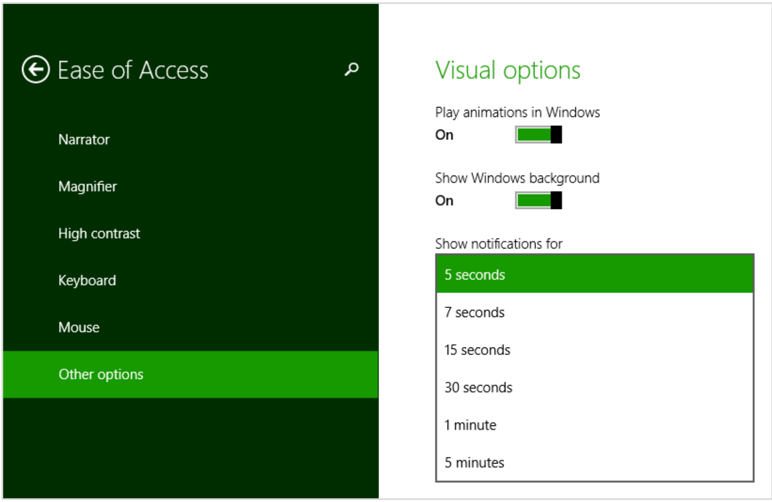
USA TODAY was installed.

FlightAware was installed.

Your kid is ready to play with you!

SDK

FIGURE 16-4 Up to three toast notifications can appear on top of the foreground app (including the desktop and the Start screen). Each notification can also play one of a small number of predefined sounds.


Ease of Access

- Narrator
- Magnifier
- High contrast
- Keyboard
- Mouse
- Other options

Visual options

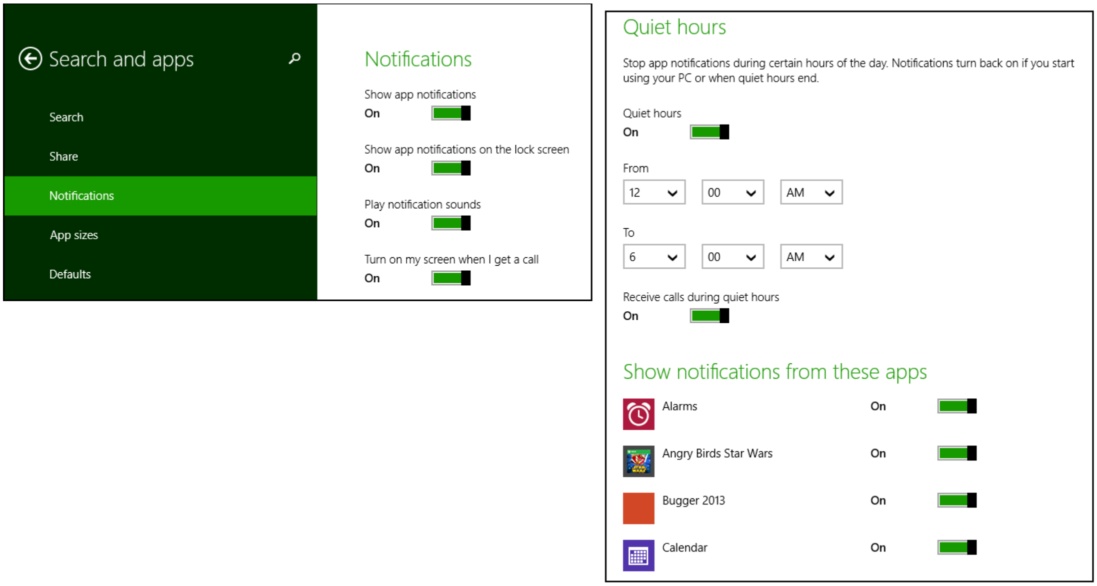
Play animations in Windows
On

Show Windows background
On

Show notifications for
- 5 seconds
- 7 seconds
- 15 seconds
- 30 seconds
- 1 minute
- 5 minutes

FIGURE 16-5 Toast duration settings (a drop-down list) in PC Settings > Ease of Access > Other Options.

As with tile updates, the user has complete control over toast notifications: for the entire system, for the lock screen, and for individual apps. Users do this through PC Settings > Search & Apps > Notifications, as shown in Figure 16-6. Here the user can turn notifications on and off completely, turn them on and off for the lock screen specifically, and enable or disable sounds. This area of PC Settings also controls *quiet hours,* a part of the day where the user probably wants to stay asleep! (Most background activity is disabled during quiet hours as well; see Updates to background task management for specifics.) And further down the user can control notifications for any individual app. All of this ultimately means that you want to make your notifications valuable to the user. If you toss up lots of superfluous toast, chances are that the user will turn them off for your app or for the whole system (and give you bad reviews in the Windows Store).



Search and apps

Search

Share

Notifications

App sizes

Defaults

Notifications

Show app notifications
On

Show app notifications on the lock screen
On

Play notification sounds
On

Turn on my screen when I get a call
On

Quiet hours

Stop app notifications during certain hours of the day. Notifications turn back on if you start using your PC or when quiet hours end.

Quiet hours
On

From
12 00 AM

To
6 00 AM

Receive calls during quiet hours
On

Show notifications from these apps

Alarms	On
Angry Birds Star Wars	On
Bugger 2013	On
Calendar	On

FIGURE 16-6 The user can exercise fine control over notifications in PC Settings > Search & Apps > Notifications, including the configuration of "Quiet Hours," during which notifications are turned off temporarily.

As with secondary tiles, each toast notification contains specific data that is given to its associated app if the user taps the toast to activate the app. If the app is suspended, of course, Windows switches to that app and fires its `activated` event with the notification data. If the app isn't running, Windows will launch it. (The Win+V key, by the way, will cycle the keyboard focus through active toasts, and pressing Enter will activate the one with the focus.)

This brings up the point that toast notifications, like tile updates, can originate from sources other than a running app—which should be obvious because nonforeground apps will typically be suspended! Again, we'll talk about those sources soon. At the same time, you might be wondering if the last item in this chapter's subtitle—*background tasks*—comes into play here.

Indeed it does. Background tasks are an essential part of the whole "alive with activity" story. As we've already seen with background transfers in Chapter 4, "Web Content and Services," geofencing in Chapter 12, "Input and Sensors," and background audio in Chapter 13, "Media," it's not a hard-and-fast rule that apps are always suspended in the background. It's just that Windows, on its quest to optimize battery life, doesn't allow arbitrary apps to keep themselves running for arbitrary reasons. Instead, Windows allows apps to run focused background tasks for specific purposes—called *triggers*—subject to specific quotas on CPU time and network I/O. As you might expect, an app declares such background tasks in its manifest.

Triggers include a change in network connectivity, a time zone change, an update of an app, the expiration of a timer (with a 15-minute resolution), or the arrival of a push notification from an online source (that is, a notification sent in response to a condition that's completely external to the device itself). Each trigger can also be configured with conditions such as whether there's Internet connectivity. Whatever the case, the whole purpose of background tasks is not to launch an app—in fact, background tasks cannot display arbitrary UI. It is rather to allow them to update their internal state and, when needed, issue tile updates or toast notifications through which the user can make the choice to activate the app for further action.

One additional aspect of background tasks is that Windows also places a limit on the total number of apps that can handle certain kinds of triggers: timers, receipt of push notifications, and receipt of network traffic on a *control channel* as used by real-time communications apps. The limit is imposed by the fact that such apps must be added to the *lock screen* for their tasks to run at all.

The lock screen, as you certainly know by now and as shown in Figure 16-7, is what's displayed anytime the user must log into the device. A device will be locked directly by the user or after a period of inactivity. An exception is made when an app has disabled auto-locking through the `Windows.System.Display.DisplayRequest` API, as discussed in Chapter 13 in "Disabling Screen Savers and the Lock Screen During Playback."

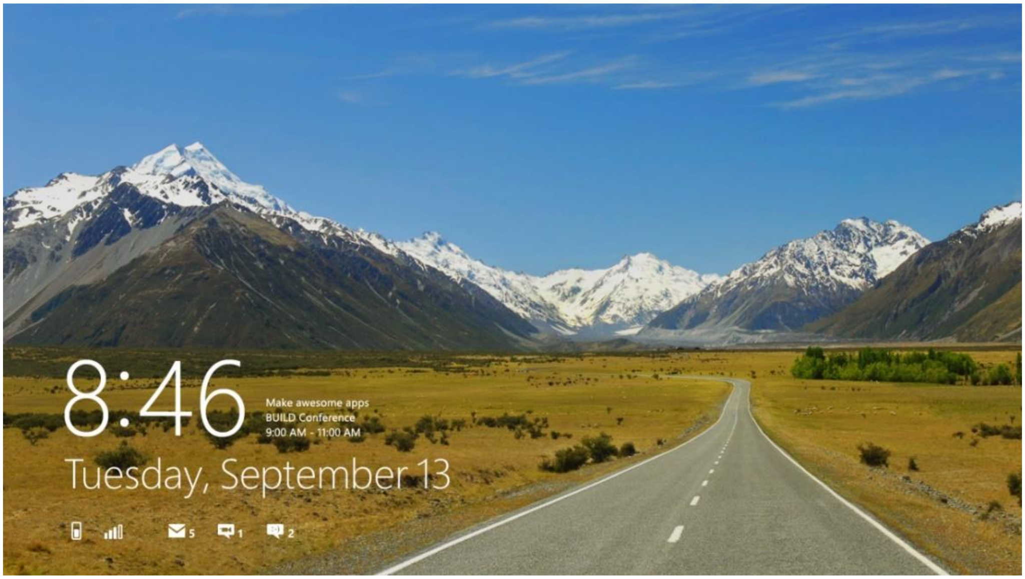
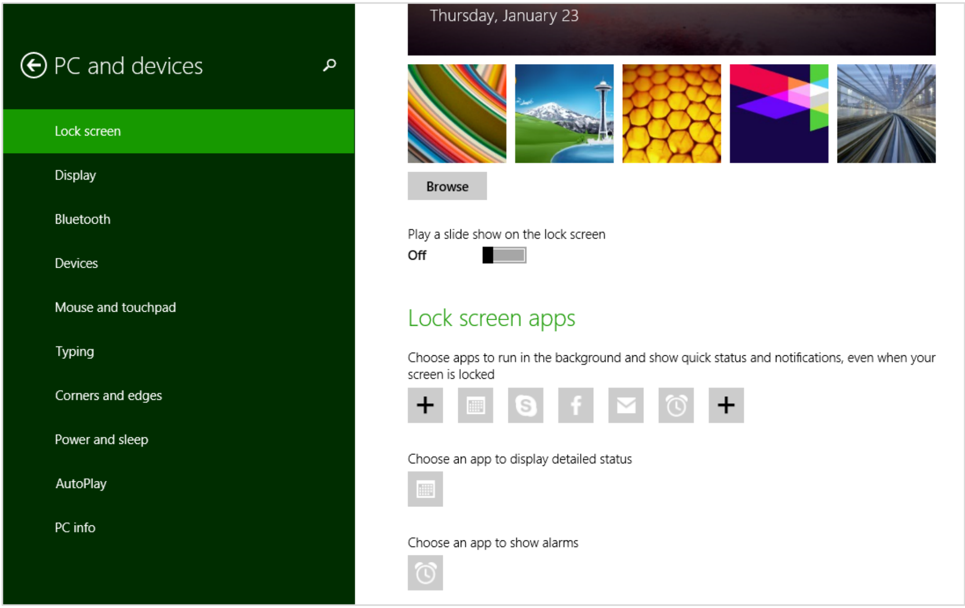


FIGURE 16-7 A typical lock screen. Up to seven apps can display badges along the bottom of the screen; one app can display text next to the clock.

Yet Windows doesn't want to force the user to log in just to see the most important information from their most important apps. Through PC Settings, as shown in Figure 16-8 (where I've panned up a bit), the user can add up to seven apps to the lock screen (provided those apps have requested access, which is subject to user consent). These apps must be registered for lock screen–related background tasks during which they can issue badge updates to the lock screen—these are what you see above along the bottom of Figure 16-7, where each badge glyph (the numbers) is also accompanied by a monochrome graphic, referred to as the Badge Logo in the app manifest. This graphic is 24x24 at 100%, 33x33 at 140%, and 43x43 at 180%, and it must contain only white or transparent pixels.

In addition, the user can indicate a single app that can display a piece of text (but not an image) next to the clock, and a single app to show alarms. Note that toast notifications raised by these apps will surface on the lock screen; if tapped, the lock screen will bounce and the app will be activated once the user signs in.



Play a slide show on the lock screen
Off

Lock screen apps

Choose apps to run in the background and show quick status and notifications, even when your screen is locked

Choose an app to display detailed status

Choose an app to show alarms

FIGURE 16-8 Configuring the lock screen and lock screen apps in PC Settings > PC and Devices > Lock Screen (I've panned the view on the right up a bit to show the Lock Screen Apps section).

Thus we complete the story of how Windows works with apps to be alive with activity—on the Start screen, on the lock screen, and while the user is engaged in other apps—while yet conserving battery power by intelligently managing how and when apps can issue their various updates. Let's now see exactly how that's accomplished, ideally without needing apps to run at all.

The Four Sources of Updates and Notifications

When an app is active in the foreground, it can clearly issue whatever notifications it wants: updates to any of its tiles, badge updates, and toast notifications. A background task running on the app's behalf can do the same. Together these updates are simply referred to as *local* updates because they originate from running app code and are applied immediately, as shown in Figure 16-9. A running news app, as an example, might issue up to five updates to its tiles so that recent headlines continue to cycle when the user switches to another app. Such updates can also be set to expire at some date and time in the future so that they'll disappear automatically (perhaps fulfilling the adage, "No news is good news"!). With toasts, note that a foreground app should use inline messages, flyouts, and message dialogs for errors that pertain to the currently visible content; toasts are only appropriate for alerts about content in some other part of the app.

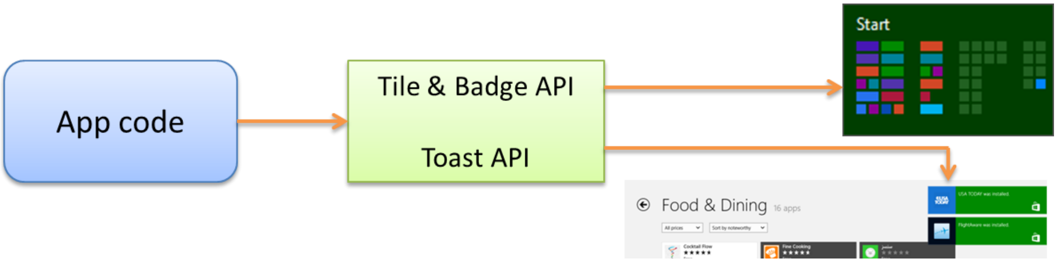


FIGURE 16-9 Local updates from a running app are applied immediately.

The second source of updates are *scheduled notifications* that apply to tile updates and toasts. A running app or background task issues these to the system with the date and time when the update or notification should appear, regardless of whether the app will be running, suspended, or not running at that future time. This is illustrated in Figure 16-10. A calendar app, for example, would typically use scheduled notifications for appointment reminders. An alarms app uses scheduled toasts as well but also requests alarm access such that its scheduled notifications appear on the lock screen and can be accompanied by Snooze and Dismiss commands, as shown earlier in "Alive with Activity: A Visual Tour."

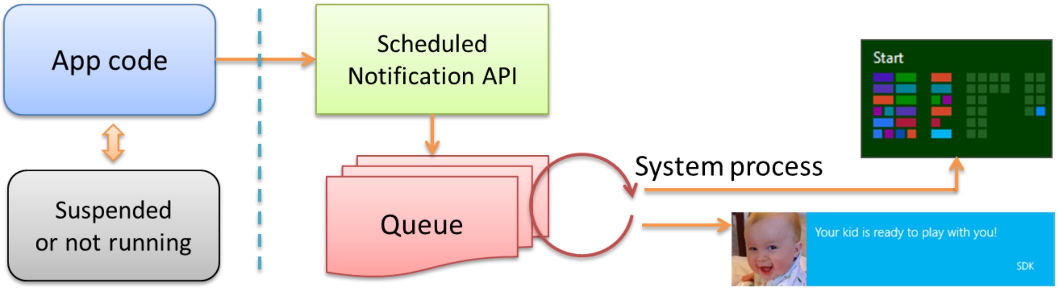

App code
Suspended or not running
Scheduled Notification API
Queue
System process
Start
Your kid is ready to play with you!
SDK

FIGURE 16-10 Scheduled notifications are managed by the system and will appear at the requested time irrespective of the state of the originating app.

The third way an app can issue updates—in this case for tiles and badges only—is through a *periodic update*. As illustrated in Figure 16-11, a running app configures the system's tile and badge updaters to request an update from a specific URI at low-frequency intervals (the minimum is 30 minutes) beginning at a specified time, if desired. You can also specify the URI directly in the app manifest (Application tab > Tile Update) to get a live tile going as soon as the app is installed (but not badges). That is, when your app is installed, Windows checks the manifest for your URI and recurrence settings. It then makes the periodic update request on your behalf so that updates begin immediately. After this, any periodic updates configured from the running app will take precedence.

The REST endpoint or web service at the specified URI (which I'll typically just refer to as a *service*) responds to these HTTP requests with an XML payload that's equivalent to what a running app would provide in a local update, and updates can be set with an expiration date/time so that they're automatically removed from the update cycle when appropriate. With all these capabilities, periodic updates are wholly sufficient for many apps to create very dynamic live tiles with relatively little effort.

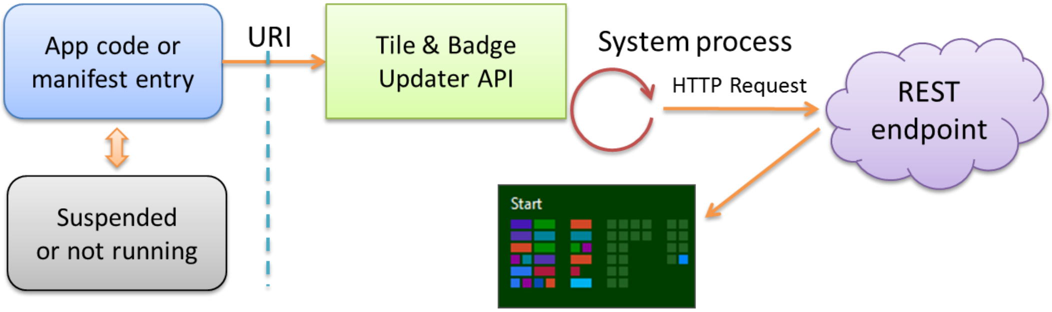


FIGURE 16-11 Periodic updates for tiles and badges are registered with the system's tile updater, which will request an update from a given REST endpoint or web service at regular intervals.

Of course, a 30-minute minimum interval is simply not fast enough when an app wants to notify a user as soon as possible. Thus we have the fourth means for updates—*push notifications*—and this method applies across tiles and toasts, as well as non-UI (raw) notifications.

Push notifications are, as the name implies, sent directly to a device not at the request of an app but at the request of some associated server process that is typically monitoring information or other conditions around the clock. As illustrated in Figure 16-12, that server process employs the free Windows Push Notification Service (WNS for short) to send notifications to those apps that have created a channel for this purpose. Each channel is specific to a user and the device. This requires the app to be run at least once, because it's during that first launch that the app establishes a WNS channel for its host device. On subsequent launches the app typically refreshes the channel, as it will otherwise expire after 30 days. A background task can also periodically refresh the channel.

Note Although push notifications can happen at any time with much greater frequency than other update methods, be aware that Windows will throttle the amount of push notification traffic on a device if it's on battery power, if it's in connected standby mode, or if notification traffic becomes excessive. This means there is no guarantee that all notifications will be delivered (especially if the device is turned off). So don't try to use push notifications to implement a clock tile or other tile gadgets with a similar kind of frequency or time resolution. Instead, think about how you can use push notifications to connect tiles and notifications to a backend service that has interesting and meaningful information to convey to the user, which invites them to re-engage with your app.

A push notification can contain an XML payload as with other tile updates and toast notifications, or it can contain a *raw notification* with arbitrary data (see [Guidelines for raw notifications]() for appropriate uses). A raw notification must be received by a running app or a by lock screen app with a background task for with the push notification trigger—otherwise the system clearly won't know what to do with it! A standard update or toast can be handled either by the system or app code.

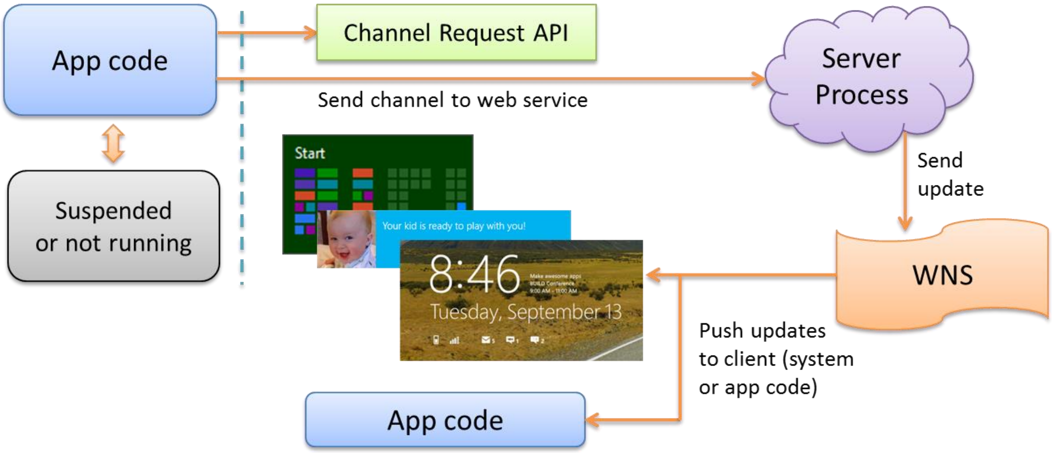


FIGURE 16-12 Push notifications originate with an always-running server process and are then sent to the Windows Push Notification Service for delivery to specific clients (a specific app on a specific user device) through their registered WNS channels.

A helpful summary of these different update mechanisms can be found on [Choosing a notification delivery method]() in the documentation, a topic that includes various examples of when you might use each method. I've also summarized the available options for different types of notifications in the table below. Whatever the case, we're now ready to see the details of how we employ all of them in an app to help keep a system alive with activity.

Notification type	Cycling	Scheduled	Expiring	Recurring	Audio	Periodic	Push
Tile	✓	✓	✓			✓	✓
Badge			✓			✓	✓
Toast		✓	✓	✓	✓		✓
Raw							✓

Sidebar: Connectivity and Remote Images in Live Tiles and Toasts

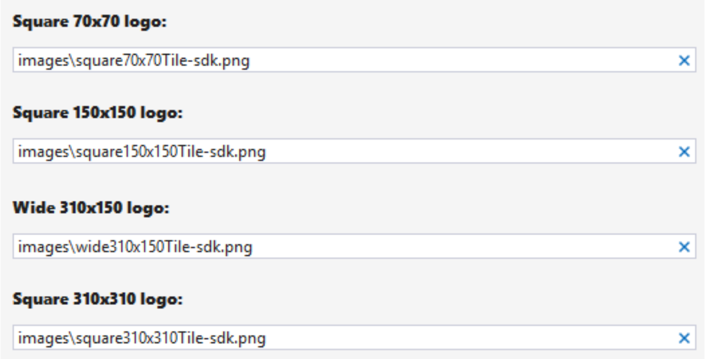
Periodic updates and push notifications are completely dependent on connectivity and will not operate without it. A running app, on the other hand, can still issue updates when the device is offline and can schedule updates and notifications for some time in the future when the device could then be offline. Under such circumstances, references to remote images will not be resolved without connectivity because the tile and toast systems do not presently support the use of local fallback images. When using remote images, then, consider whether to cache those images and use local image references instead (`ms-appx:///` or `ms-appdata:///` URIs), or opt for text-only tile and toast templates.

Sidebar: Windows Azure Mobile Services

Windows Azure Mobile Services is a set of cloud tools and a client library that together help you create a scalable backend for an app, including structured cloud data, identity, service endpoints (as for a periodic update service), scheduled jobs, and push notifications. In short, a mobile service (as I'll refer to it), provides a prebuilt solution via REST endpoints that is very handy for a number of things we'll need in this chapter! Various introductions and documentation can be found at the link above, and we'll cover some parts in appropriate contexts.

Tiles, Secondary Tiles, and Badges

The first thing you should know about your app tile is that if you want to enable live wide or large tiles (including secondary ones), you must include wide and large logo images in the Visual Assets section of your manifest, as shown below (condensing the UI). Without these, you can still have live medium tiles (which is the only required size), but wide and large tile updates will be ignored. Small tiles (70x70) are always static but can display badges like all the other sizes.


Square 70x70 logo:

images\square70x70Tile-sdk.png ×

Square 150x150 logo:

images\square150x150Tile-sdk.png ×

Wide 310x150 logo:

images\wide310x150Tile-sdk.png ×

Square 310x310 logo:

images\square310x310Tile-sdk.png ×

At this point I encourage you to go back to Chapter 3 and review "Branding Your App 101," where I discuss how different bits in the manifest affect your tiles, such as the Short Name and Show Name settings. As also covered there, remember to provide different scaled versions of each logo image for best quality. Even though you might issue tile updates as soon as your app is run, your static tiles will be essential to the user's first impression of your app after it's acquired from the Windows Store. The static tiles are also what the user will see if he or she turns your live tiles off or if all your updates expire. So, even if you plan for live tiles, be sure to still invest in great static tile designs as well.

> **Note** If you don't provide a small 70x70 image, Windows will automatically scale down the 150x150 asset. You can also provide different sizes for secondary tiles independent of the images provided in the manifest.

Again, providing static tiles for the 150x150, 310x150, and 310x310 sizes enables you to issue live tile updates for all of them, including secondary tiles of those sizes. In all cases, try to think through what the user would most want to see. When users select a wide or large tile—that is, electing to have your tile occupy more prime real estate on the Start screen—it's likely that they're looking for content that adds value to the Start screen. If users choose a medium tile, on the other hand, they're probably more interested in only the most essential information: the number of new email messages (as expressed through a badge), for example, rather than the first line of those messages, or the current temperature in a location rather than a more extended forecast. And don't worry if the user selects a small tile for your app, where only badges will appear: it typically means that they spend so much time in the app itself that they don't need any info on the Start screen!

The Guidelines for tiles and badges topic in the documentation provides rather extensive design guidance where all this is concerned, along with appropriate use of logos, names, badges, and updates. Also see Using live tiles at the end of the Tile Template Catalog for inspiration. Here we'll concern ourselves with how such updates and badges are sent to a tile, a process that involves *tile templates*, which are predefined configurations that you populate with text, images, and other properties via XML updates. These templates (which are also defined as XML, but don't confuse them with the update schema) apply to all forms of tiles and update methods, which we'll examine in a moment. First, however, let's see how secondary tiles are managed, because everything we talk about thereafter applies equally to all tiles for the app.

Secondary Tiles

A secondary tile is a kind of bookmark into an app, to achieve what's also called *deep linking*: a way to launch an app into a particular state or to a particular page. Secondary tiles allow the user to personalize the Start screen with more specific views of an app. As suggested on Guidelines for secondary tiles (a topic I highly recommend you read), offering the ability to create a secondary tile is a good idea whenever you have app state that could be a useful target or destination unto itself. But don't abuse secondary tiles, such as using them for virtual command buttons—that would only educate customers that they shouldn't bother to pin tiles from your app!

An app creates a secondary tile in response to a Pin command that it typically includes on its app bar (using the `WinJS.UI.AppBarIcon.pin` icon). Offer this command when the app is displaying pinnable content or the user has made an appropriately pinnable selection; hide or disable the command if the content or selection is not pinnable. In addition, change it to an Unpin command if the content is already pinned. For details on managing commands in the app bar, refer to Chapter 9, "Commanding UI."

When the Pin command is invoked, the app makes the request to create the tile. Windows then prompts the user for their consent, as shown earlier in Figure 16-3.

Once created, a secondary tile has all the same capabilities as your app tile, including the ability to receive updates from any source. They key difference between the app tile and secondary tiles is that the former launches the app into its default (or current) state, whereas the latter launches the app with specific arguments that your activation handler uses to launch (or activate) the app into a specific state. Let's see how it all works.

> **Note** Although the tile and notifications API is found within `Windows.UI.Notifications`, the API for creating secondary tiles come from `Windows.UI.StartScreen`. Assume that is our context in the next four sections unless otherwise noted.

Creating Secondary Tiles

The process for creating a secondary tile in response to a pin command is quite simple: first create an instance of `Windows.UI.StartScreen.SecondaryTile` with the desired options, and then call either its `requestCreateAsync` or `requestCreateForSelectionAsync` method. If the user confirms the creation of the tile, it will be added to the Start screen and your completed handler will receive a result argument of `true`. If the user dismisses the flyout (by tapping outside it), the completed handler will be called with a result argument of `false`. The error handler for these methods will be called if there is an exception, such as if you fail to provide required properties in the `SecondaryTile`.

When creating a `SecondaryTile` object, you can use one of three different constructors (there are two others, but they are deprecated in Windows 8.1):

- `SecondaryTile()` Creates a `SecondaryTile` with default properties; you must then set required properties directly before attempting to pin, update, or delete the tile.

- SecondaryTile(tileId) Initializes the SecondaryTile with a specific ID, typically used when creating an object before an update or when unpinning the file.
- SecondaryTile(tileId, displayName, arguments, logo, tileSize) Creates a SecondaryTile with the properties for a default medium tile (whose image is in logo).

The constructor arguments clearly correspond to the following SecondaryTile properties, all of which are required to be set when you call a requestCreate* method:

- tileId A unique string (a maximum of 64 alphanumeric characters including . and _) that identifies the tile within the package. You need this when you want to update or delete a tile, and it should always be set. This value is typically derived from the content related to the file. If you create secondary tiles with a tileId that already exists, the new one will takes its place.
- displayName The tile's display name that will be shown in the tile's tooltip, next to the app in the Start screen's All Apps view, and in a few other areas within Windows. This can be whatever length you want and can contain any characters.
- arguments A string that's passed to the app's activation handler when the tile is invoked. logo A URI for the square (medium) 150x150 tile image. This can use either the ms-appx:/// or ms-appdata:///local schema. Be sure to avoid storing a dynamically generated image in temporary storage, and avoid deleting it unless all secondary tiles that reference it are deleted.

In addition to these you can set any of the other properties described in the following table before attempting to pin a secondary tile:

Property	Description
roamingEnabled	Indicates that the secondary tile is roamed to the cloud and replicated on other devices where the same user installs the same app.
lockScreenBadgeLogo lockScreenDsiplayBadgeAndTileText	Indicate the secondary tile's relationship to the lock screen; see "Lock Screen Dependent Tasks and Triggers" later in this chapter.
phoneticName	For setting UI sort ordering in some languages.
visualElements	A SecondaryTileVisualElements object (see the properties below)
visualElements.backgroundColor	A Windows.UI.Color value that overrides the app's manifest color.
visualElements.foregroundText	A ForegroundText value, either dark or light.
visualElements.square30x30Logo visualElements.square70x70Logo visualElements.square150x150Logo visualElements.wide310x150Logo visualElements.square310x310Logo	ms-appx:/// or ms-appdata:///local URIs for logo images, overriding the defaults in the manifest.
visualElements.showNameOnSquare150x150Logo visualElements.showNameOnWide310x150Logo visualElements.showNameOnSquare310x310Logo	Booleans that indicate whether the app name is shown on specific tiles, overriding settings in the manifest.

Note The following SecondaryTile properties are deprecated in Windows 8.1 because they are handled through the visualElements property instead: backgroundColor, foregroundText, shortName, logo, smallLogo, wideLogo, and most values of tileOptions. The tileOptions.copyOnDeployment value is also replaced by of roamingEnabled.

At run time, you can also retrieve any of these properties to check the state of the secondary tile if needed. If you modify any properties for a SecondaryTile that has already been pinned, be sure to call its updateAsync method to propagate those changes.

The requestCreate* methods also have variations to control the placement of the user consent flyout (the flyout that's shown in Figure 16-3). Calling requestCreateAsync by itself results in a default placement in a lower corner of the display. It's usually better, however, for that flyout to appear close to the command that invoked it. For this purpose requestCreateAsync accepts an optional Windows.Foundation.Point, specifying where to place the lower right corner of the flyout.

requestCreateForSelectionAsync has two variations itself. The first takes a Windows.Foundation.Rect describing the selection. The flyout will appear above that rectangle if possible. If you expect that this default placement will obscure the secondary tile's content, you can also pass an optional value from Windows.Popup.Placement to indicate where the flyout should appear relative to that rectangle: above, below, left, and right.

You can play around with all of these options in the Secondary tiles sample. Scenarios 1 and 2 pin and unpin a secondary tile using on-canvas buttons, respectively, with scenario 7 doing the same through the app bar. We'll see some of the other scenarios in the sections that follow. For the moment, the pinning function in scenario 1 shows the creation process using requestCreateForSelectionAsync (js/pintile.js, copious comments removed):

```
function pinSecondaryTile() {
    var Scenario1TileId = "SecondaryTile.Logo";

    var square70x70Logo = new Windows.Foundation.Uri(
        "ms-appx:///Images/square70x70Tile-sdk.png");
    var square150x150Logo = new Windows.Foundation.Uri(
        "ms-appx:///Images/square150x150Tile-sdk.png");
    var wide310x150Logo = new Windows.Foundation.Uri(
        "ms-appx:///Images/wide310x150Tile-sdk.png");
    var square310x310Logo = new Windows.Foundation.Uri(
        "ms-appx:///Images/square310x310Tile-sdk.png");
    var square30x30Logo = new Windows.Foundation.Uri(
        "ms-appx:///Images/square30x30Tile-sdk.png");

    // Create activation arguments...
    var currentTime = new Date();
    var newTileActivationArguments = Scenario1TileId  + " WasPinnedAt=" + currentTime;

    var tile = new Windows.UI.StartScreen.SecondaryTile(Scenario1TileId,
        "Title text shown on the tile",
        newTileActivationArguments,
        square150x150Logo, Windows.UI.StartScreen.TileSize.Square150x150);

    // Setting other options
    tile.visualElements.wide310x150Logo = wide310x150Logo;
    tile.visualElements.square310x310Logo = square310x310Logo;
    tile.visualElements.square70x70Logo = square70x70Logo;
    tile.visualElements.square30x30Logo = square30x30Logo;
    tile.visualElements.showNameOnSquare150x150Logo = false;
    tile.visualElements.showNameOnWide310x150Logo = true;
    tile.visualElements.showNameOnSquare310x310Logo = true;

    tile.visualElements.foregroundText = Windows.UI.StartScreen.ForegroundText.dark;
    tile.roamingEnabled = false;

    var selectionRect = document.getElementById("pinButton").getBoundingClientRect();

    tile.requestCreateForSelectionAsync(
       { x: selectionRect.left, y: selectionRect.top, width: selectionRect.width,
         height: selectionRect.height },
       Windows.UI.Popups.Placement.below)
    .done(function (isCreated) {
       if (isCreated) {
          // The tile was successfully created
       } else {
          // The tile was not created
       }
    });
}
```

Note As mentioned in Chapter 9, the system flyout displayed when creating a secondary tile (and when removing it, see "Managing Secondary Tiles" below) will cause the app to lose focus and will dismiss a nonsticky app bar as a result. For this reason, scenario 7 of the Secondary tiles sample keeps the app bar visible by setting its `sticky` property to `true` before calling the secondary tile API.

Supplying Multiple Tile Graphics

In Figure 16-3 we saw that the secondary tile flyout lets the user flip through different tile sizes depending on what you're provided in the `SecondaryTile.visualElements` property, but this is only the beginning of the story. To help apps create a truly engaging Start screen presence with their secondary tiles, it's possible to provide up to three sets of additional tile graphics for each size, all of which are then shown in the flyout's FlipView. This is done by responding to the `SecondaryTile` object's `visualelementsrequested` event (a WinRT event, so remember to remove it).

This event is fired when you call a `requestCreate*` method, before the flyout appears. Its `eventObj.request` is a `VisualElementsRequest` object, and you specifically populate its `alternateVisualElements` vector with one or more additional `SecondaryTileVisualElements` objects. For reference. `eventObj.request.visualElements` is a copy of the original `visualElements` property from the `SecondaryTile`; that way you can avoid adding logos that are identical to the defaults.

Scenario 9 of the Secondary tiles sample provides a demonstration. First it sets an event handler before calling `requestCreateForSelectionAsync` (js/PinTileAlternateVisualElements.js):

```
// Most other details omitted
var tile = new Windows.UI.StartScreen.SecondaryTile(/* Includes 150x150 tile... */);
```

```
tile.visualElements.square70x70Logo = square70x70Logo; // Must set this to provide alternates
tile.visualElements.wide310x150Logo = wide310x150Logo;
tile.visualElements.square310x310Logo = square310x310Logo;

tile.onvisualelementsrequested = visualElementsRequestedHandler;
tile.requestCreateForSelectionAsync(/* ... */).done(function (isCreated) {
    // ...
});
```

Its handler then adds three sets of (admittedly uninspiring) 70x70, 150x150, 310x150, and 310x310 logos to the `alternateVisualElements` vector. To prevent you from nodding off while reading highly redundant code, let me just show the first of these (omitting some temporary variables as well; js/PinTileAlternateVisualElements.js):

```
function visualElementsRequestedHandler(args) {
    args.request.alternateVisualElements[0].square70x70Logo = new Windows.Foundation.Uri(
        "ms-appx:///Images/alternate1Square70x70Tile-sdk.png");
    args.request.alternateVisualElements[0].square150x150Logo = new Windows.Foundation.Uri(
        "ms-appx:///Images/alternate1Square150x150Tile-sdk.png");
    args.request.alternateVisualElements[0].wide310x150Logo = new Windows.Foundation.Uri(
        "ms-appx:///Images/alternate1Wide310x150Tile-sdk.png");
    args.request.alternateVisualElements[0].square310x310Logo = new Windows.Foundation.Uri(
        "ms-appx:///Images/alternate1Square310x310Tile-sdk.png");

    args.request.alternateVisualElements[0].backgroundColor = Windows.UI.Colors.green;
    args.request.alternateVisualElements[0].foregroundText =
        Windows.UI.StartScreen.ForegroundText.dark;
    args.request.alternateVisualElements[0].showNameOnSquare310x310Logo = true;

    // Setting up alternativeVisualElements[1] and [2] omitted
}
```

Note that if you provide alternate `square70x70Logo` URIs this way, you must have set the default `square70x70Logo` property in the original `visualElements`. In any case, the result of all this is shown in Figure 16-13, which is easier seen in Video 16-1 where you can see all the alternates. The sample, of course, is simply providing tiles with different colors, which isn't that much of a differentiation and also bad for branding. I fully expect that your apps will be much more creative! For example, if you're pinning a tile for a person, you can provide alternate portraits, and if you're pinning a location, you can supply different images of key landmarks. If you're pinning a news article or web page, you can provide different images from that page as options.

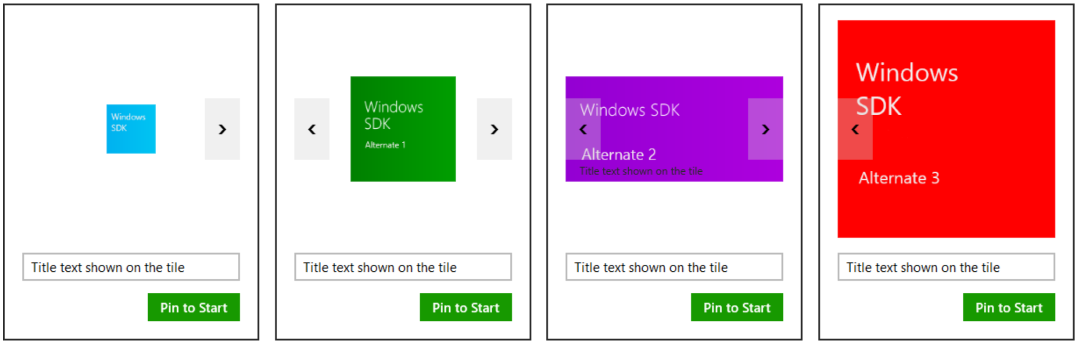

Windows SDK
›
Title text shown on the tile
Pin to Start
‹
Windows SDK
Alternate 1
›
Title text shown on the tile
Pin to Start
Windows SDK
‹
Alternate 2
Title text shown on the tile
›
Title text shown on the tile
Pin to Start
Windows SDK
‹
Alternate 3
Title text shown on the tile
Pin to Start

FIGURE 16-13 Flipping through alternate tile images in the Secondary tiles sample. The blue tiles are the defaults set in the `SecondaryTile` object. The green, purple, and red images, available in all sizes, are set through the `visualelementsrequested` event handler.

There are times, of course, when you want to use an image that you don't necessarily have on hand already. For example, if you're working with images of people in a contact database, you might need to run a query to retrieve them, or perhaps you simply need to open a file to get that information. These might be async operations, so the `VisualElementsRequest` object includes a `getDeferral` method and a `deadline` property (about five seconds from when the event is raised), These work like all other deferrals we've seen: once you call `getDeferral` for the deferral object, you can return from your handler and Windows will just display a progress ring in the flyout until you call the deferral's `complete` method or the deadline has passed. If the deadline had passed by the time you return from your handler, all of your alternates will be ignored.

Using the deferral is demonstrated in scenario 10 of the sample, which just uses a 3-second delay to simulate async work (js/PinTileAlternateVisualElementsAsync.js):

```
function visualElementsRequestedAsyncHandler(args) {
    var deferral = args.request.getDeferral();

    WinJS.Promise.timeout(3000).then(function () {
        assignVisualElements(args);
        deferral.complete();
    });
}
```

The code in `assignVisualElements` is the same as in scenario 8 except that it checks whether we're still within the deadline before populating the `alternateVisualElements` vector:

```
if (args.request.deadline > new Date()) {
    args.request.alternateVisualElements[0].square70x70Logo = /* ... */;
    // And so on...
}
```

App Activation from a Secondary Tile

Secondary tiles provide a way to activate an app to something other than its default state, according to the contents of the secondary tile's arguments property. When a secondary tile is tapped or clicked, the app's activated event is fired with an activation kind of launch and the tile's arguments value in eventArgs.detail.arguments. The app then takes whatever action is appropriate for that data, such as navigating to a particular page of content, retrieving a piece of content from an online source, and so on. In the Secondary tiles sample, the activation code in js/default.js navigates to its scenario 5 page, where we pass arguments as the options parameter of WinJS.Navigation.navigate:

```
function activated(eventObject) {
    if (eventObject.detail.kind ===
            Windows.ApplicationModel.Activation.ActivationKind.launch) {
        if (eventObject.detail.arguments !== "") {
            // Activation args are present (declared when the secondary tile was pinned)
            eventObject.setPromise(WinJS.UI.processAll().done(function () {
               // Navigate to scenario 5, where the user will be shown the activation args
               return WinJS.Navigation.navigate(scenarios[4].url,eventObject.detail.arguments);
            }));
        } else {
            // Activate in default state
        }
    }
}
```

The page control (js/LaunchedFromSecondaryTile.js) receives the arguments string in the options parameter of both the processed and ready methods. In the case of the sample it just copies that string to the display:

```
var page = WinJS.UI.Pages.define("/html/LaunchedFromSecondaryTile.html", {
    processed: function (element, options) {
        if (options) {
            document.getElementById("launchedFromSecondaryTileOutput").innerHTML += "<p>" +
                "App was activated from a secondary tile with the following activation" +
                "arguments : " + options + "</p>";
        }
    },

    ready: function (element, options) {
    }
});
```

That's the whole mechanism. Your own app, of course, will do something much more interesting with arguments!

Managing Secondary Tiles

In addition to the methods and properties to create secondary tiles, the SecondaryTile class has two static methods to manage your app's secondary tiles:

- exists Returns a Boolean indicating whether a secondary tile, identified with its tileId, is present on the Start screen. This tells you whether calling a requestCreate* method for a tile with that same tileId will replace an existing one. This is demonstrated in scenario 4 of the Secondary tiles sample.

- findAllAsync Retrieves a vector of SecondaryTile objects that have been created by the app. This will include any tiles roamed from another device (those created with the roamingEnabled option).[115] This is demonstrated in scenario 3 of the sample.

In addition, here are a few other methods for working with a specific SecondaryTile instance:

- requestDeleteAsync and requestDeleteForSelectionAsync Direct analogs, with the same placement variations, to the requestCreate* methods, because unpinning a secondary tile is also subject to user consent. This is demonstrated again in scenarios 2 and 7.
- updateAsync Propagates any changes made to the SecondaryTile properties since it was added to the Start screen. This is demonstrated in scenario 8.

If you've been keeping score, you might have noticed that I've yet to mention scenario 6 of the sample. That's because it shows how to make a secondary tile into a live tile with updates. To understand that, we need to look at updates more generally because the mechanisms involved apply to all tiles alike. This just so happens to be the next topic in this chapter—and yes, I planned it that way!

Basic Tile Updates

A local update for a tile, as described earlier in this chapter, is one that an app issues directly while it's running or issues from a background task. This is clearly efficient because running app code generally has the information it needs for making updates to any of its tiles. In a number of cases—especially when an app is *not* related to a server—the information needed for the app's live tiles is available only to the app's code. A game, for example, can send updates showing best scores, new challenges, progress toward achievements, and other kinds of compelling invitations to re-engage with the app. (I must personally admit that this works quite well with the Fruit Ninja game.)

The process of sending a local tile update is very straightforward using the APIs in the Windows.UI.Notifications namespace:

- Create the XML *payload*, as it's called, that describes the update within an XmlDocument object (this is in Windows.Data.Xml.Dom). The XML must always match one of the predefined tile templates. You can start with a system-provided XmlDocument, create it from scratch, or use the *Notifications Extensions Library*, which provides an object model and IntelliSense for this.
- Create a TileNotification object with that XML. The XML becomes the TileNotification object's content property and can be set separately.
- Optionally set the expirationTime and tag properties of the TileNotification. By default, a locally issued update does not expire and is removed only if it's evicted by a newer update or explicitly cleared. Setting expirationDate will automatically remove it at that particular time.

[115] The SecondaryTile class also has a variant of findAllAsync that takes a different app name along with findAllForPackageAsync that's described as enumerating secondary tiles for all apps in the same package. These were meant for packages that contain multiple apps, a feature that is not currently supported through the Windows Store.

(Cloud-issued notifications automatically expire after three days by default.) The `tag` property is a string of 16 or fewer characters that is used to manage the stack of updates that are cycled on the tile. More on this a little later.

- Call `TileUpdateManager.createTileUpdaterForApplication` to obtain a `TileUpdater` object that's linked to your app tile; call `TileUpdateManager.createTileUpdaterForSecondaryTile` (chew on that name!) to obtain a `TileUpdater` object for a secondary tile with a given `tileId`.
- Call `TileUpdater.update` with your `TileNotification` object. (The animation used to bring the update into view is similar to `WinJS.UI.Animation.createPeekAnimation`, as described in Chapter 14.)

Tip If you issue tile updates or other notifications when your app is running, think about whether it's also appropriate to issue updates within a `resuming` event handler if you aren't going to use other means like periodic updates or push notifications to refresh the tile. It may have been a while since you were suspended, so being resumed is a good opportunity to send updates.

Let's turn now to the App tiles and badges sample for how updates appear in code. Because the Visual Studio simulator doesn't enable live tiles and toasts, remember to run the sample with the Local Machine or Remote Machine option, and be sure that animations are enabled in PC Settings.

Assuming that we have our update `XMLDocument` in a variable named `tileXml`, sending the update takes only two lines of code (derived from the end of js/sendTextTile.js):

```
var tileNotification = new Windows.UI.Notifications.TileNotification(tileXml);
Windows.UI.Notifications.TileUpdateManager.createTileUpdaterForApplication()
    .update(tileNotification);
```

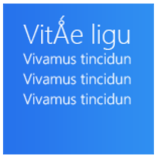
and similarly for secondary tiles in scenario 6 of the Secondary tiles sample (js/SecondaryTileNotification.js):

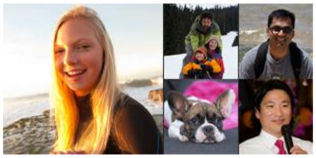
```
var tileNotification = new Windows.UI.Notifications.TileNotification(tileXml);
Windows.UI.Notifications.TileUpdateManager.createTileUpdaterForSecondaryTile(
    "SecondaryTile.LiveTile").update(tileNotification);
```

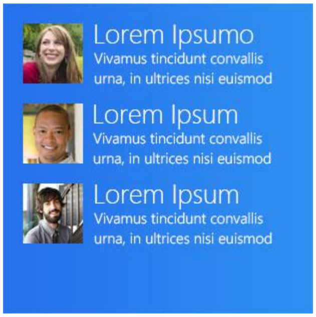
The more interesting question is how we create that `tileXml` payload in the first place. This involves choosing one of the predefined visual tile templates and understanding the options we can exercise within them, which are covered in the next two sections. Then it's a matter of choosing one of three methods to create the `XMLDocument`, which we'll also explore individually. Finally, we'll see how to properly reference images with the updates. Localization and accessibility are additional concerns for tile updates, but we'll return to that subject later in Chapter 19, "Apps for Everyone, Part 1."

Choosing a Tile Template

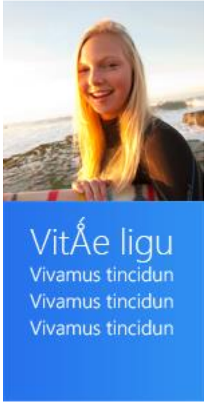
The first step in creating a tile update is to select an appropriate template from the Tile template catalog. Here you will find descriptions, images, and the exact XML for the 10 available medium templates, the 36 available wide templates, and the 29 available large templates—yes, 75 different templates in all, so I hope you understand why I'm not showing them all here! Some are text-only, some are image-only, and some are text and image both. Here are a few examples:

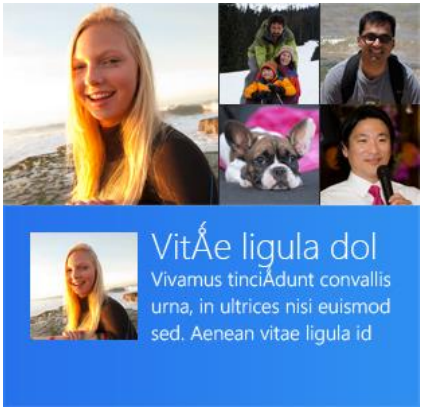





Lorem Ipsumo
Vivamus tincidunt convallis
urna, in ultrices nisi euismod
Lorem Ipsum
Vivamus tincidunt convallis
urna, in ultrices nisi euismod
Lorem Ipsum
Vivamus tincidunt convallis
urna, in ultrices nisi euismod

In addition, some of the options for medium and wide tiles are referred to as *peek* templates. These, if you look at them in the topic linked to above,[116] are really composed of two stacked sections that are each the size of the whole tile, as shown below for medium tiles (left) and wide tiles (right):




VitÅe ligula dol
Vivamus tinciÅdunt convallis
urna, in ultrices nisi euismod
sed. Aenean vitae ligula id

With peek templates you effectively get to show twice the content as the other templates. When a peek update is shown on a live tile, the upper portion will appear first and then the tile will flip or give you a "peek" at the lower portion, and then it will switch back to the upper portion, after which the live tile will switch to the next update in the cycle, if one exists. (The Travel app uses peek templates on its medium and wide tiles, if you want an example; and the animation that's employed here is again similar to `WinJS.UI.Animation.createPeekAnimation`.) Of course, both sections should contain related content because they are part of the same singular update.

[116] A more succinct list of templates is also found on the reference page for `TileTemplateType`. This includes the name of the template and a representative image but doesn't include the XML.

Hidden gems Among all the wide peek templates, the *TileWide310x150PeekImageAndText01* and *02* templates often go unnoticed because they don't look particularly distinctive in the template catalog. They're unique, though, in that on both sides of the peek they always show part of the text and part of the image. Check them out in scenario 6 of the App tiles and badges sample that we'll see shortly.

Why templates? Although it might seem limiting that you must choose from a predefined template for your tile design, this helps ensure a consistency amongst all the tiles on the Start screen. Templates also describe the relationships between different parts of the tile content, which can then be rendered uniquely (as on Windows and Windows Phone) without changing those relationships. Also, be sure to check out Using live tiles at the end of the catalog page, which is easy to miss in what is a long page in the documentation!

There are a number of important details with the template layouts. First, when you look at the template catalog, you'll see both version 1 (Windows 8) names like *TileSquareText01* and version 2 (Windows 8.1) names like *TileSquare150x150Text01*. Within an app, always use the latest version. When implementing a service that supplies updates to both version 1 and version 2 clients, you can include both names in the same XML payload. We'll come back to this in the next section.

Second, in many of the templates at present, the last line of text will not display if you're also showing a logo or a short name on the tile (to avoid overlaps). It's OK to include the extra text in the update, though—there's no penalty for doing so.

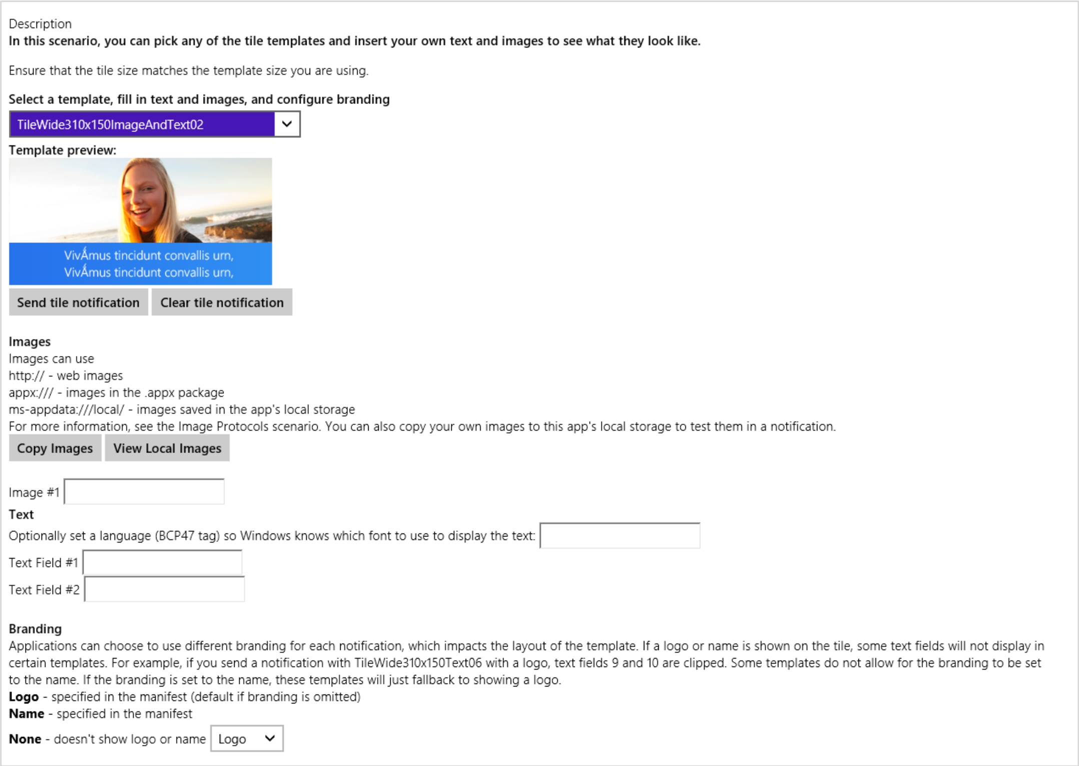
Third, images are limited to 1024x1024 and 200KB maximum; if an image exceeds these limits, the entire update is silently ignored. Clearly, it's better to avoid large images if you can help it because they'll just increase memory consumption and possibly network usage (if the image is being downloaded). It's also good to take the 80%, 100%, 140%, and 180% scale factors into account for tile images. However, if you don't want to deal with individual scaling factors, size your tile images for 180% and let the system scale them down (which uses a high-quality algorithm so that images will look as good as if you scaled them down with photo-editing software). Also, for photographs, consider using JPEG instead of PNG as the former has better compression for such images.

Fourth, if you supply an image that doesn't match the final aspect ratio, the image will be scaled and cropped to fit according to heuristics that generally produce good results with a variety of test images. Note too that a wide tile is not exactly a 2:1 aspect ratio; at 100% the wide tile is 310x150 pixels, meaning that an image occupying half of it will be something like 160x150 pixels depending on the template. Images in a collection view (the upper right portion of the rightmost image above) will also be special dimensions of their own. The tile template catalog documents all the sizes of these secondary images, and you can also find them in scenario 11 of the App tiles and badges sample.

Fifth, with some of the templates such as *TileWide310x150ImageCollection*, *TileWide310x150PeekImageCollection01*, and *TileSquare310x310SmallImagesAndTextList02*, which are the ones shown earlier with multiple images in the tile, there are specific dimensions for the small images that you should find in the template catalog. This can be helpful if you're creating a backend to serve up those images and want to size them appropriately.

And finally, as a last resort you can always use an image-only template (*TileSquare150x150Image*, *TileWide310x150Image*, and *TileSquare310x310Image*) with a graphic you generate at run time. Be mindful that images with text won't scale well, and don't mislead users by making it look like the tile has buttons or other separately clickable areas: the tile always acts as a single unit to invoke the app.

Scenario 6 of the App tiles and badges sample provides a very helpful experimentation and design tool for tiles, the core of which is shown in Figure 16-14. This part of the sample is intended as a tool rather than code would use directly in an app. It lets you easily play around with all the templates and their contents, including images referenced from local and remote sources, without having to write specific code every time. It also lets you exercise the various options for branding the app and sending the result as an update to the sample's tile on the Start screen. Scenario 11 of the sample, similarly, is a tool that lets you resize images for different parts of any selected template.



Description

In this scenario, you can pick any of the tile templates and insert your own text and images to see what they look like.

Ensure that the tile size matches the template size you are using.

Select a template, fill in text and images, and configure branding

TileWide310x150ImageAndText02

Template preview:

VivÅmus tincidunt convallis urn,
VivÅmus tincidunt convallis urn,

Images

Images can use
http:// - web images
appx:/// - images in the .appx package
ms-appdata:///local/ - images saved in the app's local storage
For more information, see the Image Protocols scenario. You can also copy your own images to this app's local storage to test them in a notification.

Image #1

Text

Optionally set a language (BCP47 tag) so Windows knows which font to use to display the text:

Text Field #1

Text Field #2

Branding

Applications can choose to use different branding for each notification, which impacts the layout of the template. If a logo or name is shown on the tile, some text fields will not display in certain templates. For example, if you send a notification with TileWide310x150Text06 with a logo, text fields 9 and 10 are clipped. Some templates do not allow for the branding to be set to the name. If the branding is set to the name, these templates will just fallback to showing a logo.
Logo - specified in the manifest (default if branding is omitted)
Name - specified in the manifest
None - doesn't show logo or name Logo

FIGURE 16-14 Scenario 6 of the App tiles and badges sample is a tool for testing out all the different tile templates and understanding how to populate them.

Tile Template Schema: Versioning, Branding, and Other Options

Before we look at the different methods for creating a tile payload, it's helpful to look at the overall XML schema for tile updates, which reveals otherwise hidden features:

```
<?xml version="1.0" encoding="utf-8" ?>
```

```
<tile>
  <visual version? = "integer" lang? = "string" baseUri? = "anyURI"
         branding? = "string" addImageQuery? = "boolean" >

    <!--One or more binding elements: place large tiles last for compatibility -->
    <binding template  = "TileSquare150x150Image" | "TileSquare150x150Block" | ...""
            fallback? = "TileSquareImage" | "TileSquareBlock" | …=""
            contentId = "string" lang? = "string" baseUri? = "anyURI"
            branding? = "string" addImageQuery? = "boolean" >
      <!--Some combination of image and text. Id’s used to de-dupe updates -->
      <image id = "integer" src = "string"
            alt? = "string" addImageQuery? = "boolean" />
      <text id = "integer" lang? = "string" />
    </binding>
  </visual>
</tile>
```

Note I know that this isn’t an exact schema, because that’s hard to capture in a concise view. For the precise details, refer to the Tile schema reference pages.

To begin with, the visual.version attribute must be 2 to support Windows 8.1 templates, including large tile sizes; version 1 supports only medium and wide Windows 8.0 templates. This matters when you have a service that supports both Windows 8 and Windows 8.1 apps through the same URI. In this case you want the service to provide a version 2 payload with the Windows 8.1 template name in the binding.template attribute and the Windows 8 template name in the binding.fallback attribute. This is what makes a single template compatible with both systems.

Next, the lang attribute that you see on all elements is where you can specify the current app language. This helps Windows determine the right font to use when rendering the tile and determine which localized images to pull from your resources. It’s also possible to localize text strings in the tile payload, but we’ll come back to that in Chapter 19. Note that any lang attribute on binding overrides that on visual, and any lang attribute on a text element overrides that of visual and binding.

The baseUri, alt, and addImageQuery attributes apply to images, which we’ll cover later in “Using Local and Web Images.”

Next, I noted in a comment that the single visual element can contain one or more binding elements. This is how you provide updates for medium, wide, and large tile sizes in a single payload, which allows the user to change tile sizes without losing information. The question here, though, is how you can maintain compatibility with Windows 8 apps when providing a version 2 payload that includes a large tile binding. The trick here is simple: always place the large binding last. A Windows 8 system will then process medium and wide bindings and ignore the large. Otherwise Windows 8 will stop processing the XML at the large binding and ignore anything that comes after it.

The ability to mix medium, wide, and large updates in single payload raises another issue, however, which is that some large updates can contain as much information as a two or three medium or wide updates. If you issue only a single update for your tile, this isn’t a problem. If, however, you use cycling to rotate through as many as five updates (discussed later in “Cycling, Scheduled, and Expiring Updates”), it will sometimes be necessary to send the same large tile content in payloads that contain different medium and wide tile content. However, Windows will naturally think that each large tile update is unique and cycle through them as it would the medium and wide updates. As a result, the user will see a large tile animation that doesn’t change the tile’s content at all!

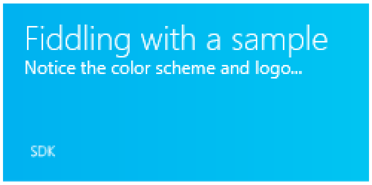
This is where the `binding.contentId` attribute comes in for the large tile. Set this to any string you want so long as it uniquely identifies the particular content for that binding. That is, if you send three payloads that contain different medium and wide bindings but the same large binding, make sure that each large binding has an identical `contentId`. Windows will then know that there’s really only one large binding for the tile and won’t do any cycling. For a demonstration, refer to scenario 12 of the App tiles and badges sample.

The final bit in the schema is the `branding` attribute of the `visual` and `binding` elements. This can be set to `none`, `logo` (the default), or `name` to indicate whether to include the app’s 30x30 logo or short name on the tile, both of which are provided in the app’s manifest. Scenario 6 of the aforementioned sample lets you play with these variations.

> **Tip** Large tile templates were designed with the assumption that you’d include branding elements at the bottom (hence the extra open space). Consider always including branding with large tile updates.

With branding, the Foreground Text and Background Color settings in the app manifest’s Visual Assets section define how tile text appears for all updates (and toasts for that matter), and they cannot be altered in the tile payload. This keeps the branding of the app consistent between updates; users would certainly find it confusing if multiple tiles from the same app showed up in different colors.

As a quick example, the App tiles and badges sample uses Light foreground text and a background color of #00b2f0. If you go to scenario 5, choose the TileWide310x150Text09 template, add some text, and select Logo for the branding, the result is shown below. (The small logo here contains a block with “SDK” in it, which is not a design to emulate. Logos should be *iconic* and not textual.)



Fiddling with a sample

Notice the color scheme and logo...

Now that we’ve seen everything we can do with a tile template, let’s see how to create the XML payloads themselves.

Creating the Payload, Method 1: Populating Template Content

The first way to create the XML payload for a given template is to use the `TileUpdateManager.getTemplateContent` method, to which you pass the name of a template (a value from `TileTemplateType`). This is shown in scenario 1 of the sample (js/sendTextTile.js; note that I’ve added the wn, tum, and tt variables for brevity in all the code snippets):

```
var wn  = Windows.UI.Notifications;
var tum = Windows.UI.Notifications.TileUpdateManager;
var tt  = Windows.UI.Notifications.TileTemplateType;

function sendTileTextNotificationWithXmlManipulation() {
       var square150x150Xml = tum.getTemplateContent(tt.tileSquare150x150Text04);
```

This method returns an XmlDocument object that contains the structure of the XML for the template but not any specific content. If you run the sample and examine tileXml just after the call above, it will contain only the following—elements but no real data values:

```
<tile>
  <visual version="2">
    <binding template="TileSquare150x150Text04" fallback="TileSquareText04">
      <text id="1"></text>
    </binding>
  </visual>
</tile>
```

The next step is to fill in the blanks (primarily attributes) by using the XmlDocument methods you probably already know (and may or may not love):

```
    var squareTileTextAttributes = square150x150Xml.getElementsByTagName("text");
        squareTileTextAttributes[0].appendChild(square150x150Xml.createTextNode(
        "Hello World! My very own tile notification"));
```

In general, if you supports wide and large tiles, include XML for medium, wide, and large formats in the same payload, because the user can change the tile size any time. The sample does it this way:

```
    var wide310x150Xml = tum.getTemplateContent(tt.tileWide310x150Text03);
    var tileTextAttributes = wide310x150Xml.getElementsByTagName("text");
    tileTextAttributes[0].appendChild(wide310x150Xml.createTextNode(
        "Hello World! My very own tile notification"));

    var square310x310Xml = tum.getTemplateContent(tt.tileSquare310x310Text09);
    square310x310Xml.getElementsByTagName("text")[0].setAttribute("id", "1");
    square310x310Xml.getElementsByTagName("text")[0].appendChild(
        square310x310Xml.createTextNode("Hello World! My very own tile notification"));

    node = square150x150Xml.importNode(wide310x150Xml.getElementsByTagName("binding")
        .item(0), true);
    square150x150Xml.getElementsByTagName("visual").item(0).appendChild(node);
    var node = square150x150Xml.importNode(square310x310Xml.getElementsByTagName("binding")
        .item(0), true);
    square150x150Xml.getElementsByTagName("visual").item(0).appendChild(node);
```

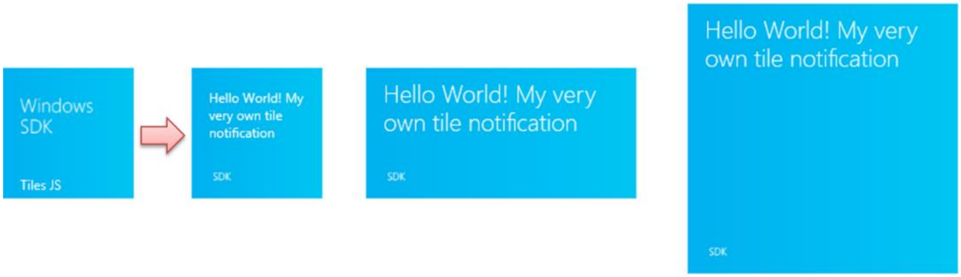
Notice how we're adding the wide and large templates directly into the original XmlDocument for the medium template by appending additional binding within the visual element, taking care to add the large binding last for compatibility with version 1 XML. The resulting structure (without the data) looks like this, where you'll also notice that there's no fallback for the large tile because version 1 templates don't support that size:

```
<tile>
  <visual version="2">
    <binding template="TileSquare150x150Text04" fallback="TileSquareText04">
      <text id="1"></text>
    </binding>
    <binding template="TileWide310x150Text03" fallback="TileWideText03">
      <text id="1"></text>
    </binding>
    <binding template="TileSquare310x310Text09">
      <text id="1"></text>
    </binding>
  </visual>
</tile>
```

With our XML payload build, we're now ready to create the `TileNotification` object and send the update to the tile:

```
    var tileNotification = new wn.TileNotification(square150x150Xml);

    // send the notification to the app's tile
    tum.createTileUpdaterForApplication().update(tileNotification);
}
```

Once this payload is sent, the default tile (below left) will show the updates on the right for the different tile sizes:


Windows SDK
Tiles JS
Hello World! My very own tile notification
SDK
Hello World! My very own tile notification
SDK
Hello World! My very own tile notification
SDK

Creating the Payload, Method 2: XML Strings

Instead of calling `TileUpdateManager.getTemplateContent` to obtain an `XmlDocument` with the tile template contents, you can just create that `XmlDocument` directly from a string. This is just like creating elements in the DOM by using `innerHTML` instead of the DOM API—it takes fewer overall function calls to create the payload you need and lends itself well to predefining a bunch of mostly populated tile updates ahead of time.

This method is simple: define an XML string with the update contents, create a new `XmlDocument`, and use its `loadXml` method to turn the string into the payload. In scenario 1 of the sample we see how this is done to create the exact same payload as in the previous section (js/sendTextTile.js):

```
function sendTileTextNotificationWithStringManipulation() {
    // create a string with the tile template xml
    var tileXmlString = "<tile>"
                 + "<visual version='2'>"
                 + "<binding template='TileSquare150x150Text04' fallback='TileSquareText04'>"
                 + "<text id='1'>Hello World! My very own tile notification</text>"
                 + "</binding>"
                 + "<binding template='TileWide310x150Text03' fallback='TileWideText03'>"
                 + "<text id='1'>Hello World! My very own tile notification</text>"
                 + "</binding>"
                 + "<binding template='TileSquare310x310Text09'>"
                 + "<text id='1'>Hello World! My very own tile notification</text>"
                 + "</binding>"
                 + "</visual>"
                 + "</tile>";

    var tileDOM = new Windows.Data.Xml.Dom.XmlDocument();
    tileDOM.loadXml(tileXmlString);  // Good idea to put this in a try/catch block

    var tile = new Windows.UI.Notifications.TileNotification(tileDOM);
    Windows.UI.Notifications.TileUpdateManager.createTileUpdaterForApplication()
        .update(tile);
}
```

Clearly, this method is very simple but has the drawback of requiring you to do manual escaping when necessary. It is also more difficult to debug. (Looking for tiny errors in strings is not my favorite pastime!) Fortunately, there is a third available method: the Notifications Extensions Library, which offers the simplicity of using strings with a high degree of reliability.

Creating the Payload, Method 3: The Notifications Extensions Library

The third means of creating the necessary XmlDocument for a tile update is to use what's called the Notifications Extensions Library. (Yes, it's a double plural.) This is a WinRT component written in C# that's included with a number of the SDK samples, including the App tiles and badges sample we're looking at here. (Notice that it's included in the project's References.) We'll be looking at the structure of such components in Chapter 18, "WinRT Components." It's possible that this library will become part of the Windows API in the future, so we do encourage developers to leverage it.

The library makes it easier to populate a template through object properties rather than XmlDocument methods, and because it's been very well-tested within Microsoft it's a more robust approach than creating an XmlDocument directly from strings. Here's how it's used in scenario 1 to create, once again, the same payload we've already seen (js/sendTileText.js; I've added the factory variable for brevity):

```
function sendTileTextNotification() {
    var factory = NotificationsExtensions.TileContent.TileContentFactory

    var tileContent = factory.createTileSquare310x310Text09();
    tileContent.textHeadingWrap.text = "Hello World! My very own tile notification";

    var wide310x150Content = factory.createTileWide310x150Text03();
    wide310x150Content.textHeadingWrap.text = "Hello World! My very own tile notification";
    var square150x150Content = factory.createTileSquare150x150Text04();
    square150x150Content.textBodyWrap.text = "Hello World! My very own tile notification";

    wide310x150Content.square150x150Content = square150x150Content;
    tileContent.wide310x150Content = wide310x150Content;

    Windows.UI.Notifications.TileUpdateManager.createTileUpdaterForApplication()
        .update(tileContent.createNotification());
}
```

Simply said, the library's `TileContentFactory` object provides methods to create objects equivalent to the XML documents provided by `TileUpdateManager.getTemplateContent`. As shown in the code above, those objects have properties equivalent to each field in the template, and when you're ready to pass it to `TileUpdater.update`, you just call its `createNotification` method.

> **Note** The medium tile object is stored in the `square150x150Content` property of the *wide* tile object, and the wide tile object is stored in the `wide310x150Content` property of the *large* tile object. This hierarchy implies that you should support progressively larger sizes rather than using an arbitrary mix.

The other reason this library exists is to simplify the process of creating an ASP.NET service for periodic updates and push notifications (where the latter can send tile updates, badge updates, and toast notifications). Instead of creating the XML payloads manually—a fragile and highly error-prone practice at best—the service can use the Notifications Extensions Library to easily and consistently create the XML for all these notifications.

Because the object model in the library clearly describes the XML, it's fairly easy to use. There is also a topic in the documentation for it called Quickstart: Using the NotificationsExtensions library in your code. The samples we look at in this chapter also show most use cases.

Using Local and Web Images

Scenario 1 of the sample, as we've seen, shows tile updates using text, but the more interesting ones include graphics as well (provided that the images stay under the 200KB and 1024 pixel limits). For tile updates, these can come either from the app package, local app data, or the web, using `ms-appx:///`, `ms-appdata:///local`, and `http[s]://` URIs. These URIs are simply assigned to the `src` attributes of `image` elements within the XML tile templates. (These are `image`, not `img` as in HTML.) Note again that the first two URIs typically have three slashes at the beginning to denote "the current app"; `http://` URIs also require that the *Internet (Client)* capability be declared in the app's manifest.

> **Note** If you plan on using remote images, review "Sidebar: Connectivity and Remote Images in Live Tiles and Toasts" at the end of "The Four Sources of Updates and Notifications" earlier in this chapter, because tiles do not support fallback images when a device is offline.

Scenario 2 of the sample (js/sendLocalImage.js) shows the use of `ms-appx:///` for images within the app package, with variants for all three methods we've just seen to create the payload. When using

XmlDocument methods, setting an image source looks like this:

```
var tileImageAttributes = wide310x150Xml.getElementsByTagName("image");
tileImageAttributes[0].setAttribute("src", "ms-appx:///images/redWideWide31x150.png");
```

The Notifications Extensions Library gives us properties to which we can assign a URI:

```
var tileContent = NotificationsExtensions.TileContent.TileContentFactory
    .createTileWide310x150ImageAndText01();
tileContent.textCaptionWrap.text = "This tile notification uses ms-appx images";
tileContent.image.src = "ms-appx:///images/redWide310x150.png";
```

And when using XML strings, you can just include the URI directly in the image element. Note also that you can add alt attributes to each image element that are aggregated into accessibility text for the tile (as used by screen readers).

Scenario 3 (js/sendWebImageTile.js) shows the same things except you can enter an http:// URI of your choice. This is a good way to see the effects of pointing to images that have varying aspect ratios as well as those that exceed the allowable 1024px dimensions and 200KB file size. As you'll see, the updates simply aren't shown in those cases.

As for ms-appdata:///local URIs (roaming and temp are not allowed), their use is demonstrated in scenario 9 where you choose an image with the file picker and the sample copies it to the local app data folder. It then references that file with an ms-appdata:///local URI in the update payload (js/imageprotocols.js):

```
tileContent = NotificationsExtensions.TileContent.TileContentFactory
    .createTileWide310x150Image();
tileContent.image.src = “ms-appdata:///local/” + imageRelativePath;
```

Because a number of tile templates let you specify multiple images, it's convenient to provide a common base URI for the entire update (or for each binding element) so that you can just use relative paths in each image.src. This is the purpose of the visual.baseUri and binding.baseUri attributes. For example, if you were always referencing images from ms-appdata:///local/, you could store that string just once in visual.baseUri. The default is ms-appx:///.

Scenario 9 in the sample lets you play with in-package and remote URIs as well, as does the tile update designer in scenario 6. I also updated the Here My Am! app for this chapter (in the companion content) with a medium and wide peek tile updates containing the most recent image and the location, along with a large tile that combines image and text; see “Sidebar: PNG vs. JPEG Image Sizes” below.

Speaking of tools and images, check out scenario 11 again where you can crop and adjust images according to varying pixel densities so that they'll work well with a selected tile template. You can save the images you adjust for inclusion in your app package. The code for the scenario can also be used to adjust images at run time. Like I said, very helpful!

A final capability with images is an option to have Windows automatically append a query string to http[s]:// URIs that describes the current scaling factor, contrast setting (for accessibility), and language. This enables your backend service to adjust images accordingly, avoiding the need to handle such concerns in the app itself. As described in the Tile schema reference, specifically for the image element, you indicate this option by setting the addImageQuery attribute of image to true (also supported on the visual and binding elements):

```
// XmlDocument form
var tileImageAttributes = tileXml.getElementsByTagName("image");
tileImageAttributes[0].setAttribute("addImageQuery", "true");

// XML string form (other lines omitted)
var tileXmlString = /* ... */  "<image id='1' addImageQuery="true"
    src='ms-appx:///images/redWide310x150.png'/>" /* ... */

// If using Notifications Extensions Library (see scenario 9 in the sample)
var tileContent = NotificationsExtensions.TileContent.TileContentFactory
    .createTileWideImageAndText01();
tileContent.image.src = "ms-appx:///images/redWide310x150.png";
tileContent.image.addImageQuery = true;
```

In all of these cases, the appended string will be of the form

```
?ms-scale=<scale>&ms-contrast=<contrast>&ms-lang=<language>
```

where <scale> is 80, 100, 140, or 180, <contrast> is standard, black, or white, and <language> is a BCP-47 language tag such as en-US, jp-JP, de-DE, and so forth. All of these are described on the Globalization and accessibility for tile and toast notifications in the documentation, including how to localize update text.

Sidebar: PNG vs. JPEG Image Sizes

When considering tile images for the larger 140% and 180% scales, the encoding you use for your images can make a big difference and keep them below the 200K size limit. As we saw in "Branding Your App 101" in Chapter 3, a wide tile at 180% is 558x270 pixels, a medium is 270x270 pixels, and a large is 558x558 pixels. With the wide and large tile sizes, a typical photographic PNG at this size will easily exceed 200K. I encountered this when first adding tile support to Here My Am! for this chapter, where it makes a smaller version of the current photo in the local appdata folder and uses ms-appdata:///local URIs in the tile XML payload. Originally the app was capturing PNG files that were too large, so I borrowed code from scenario 11 of the App tiles and badges sample, as we've been working with here, to create a smaller PNG from the img element using a temporary canvas and the blob APIs. This worked fine for a 270x270 medium tile image (a 180% scale that can be downsized), but for 558x270 and 558x558 the file was too large. So I borrowed code from scenario 3 of the Simple Imaging sample to directly transcode the StorageFile for the current image into a JPEG, where the compression is much better and we don't need to use the canvas. For this second edition's example I just capture JPEGs to begin with, which are already small enough, but I've preserved the code anyway for reference. You can find it in the transcodeImageFile function in pages/home/home.js, a routine that we'll also rewrite in Chapter 18 using C# in a WinRT component.

Such considerations are certainly important for services that handle the `addImageQuery` parameters for scale. For larger image sizes, it's probably wise to stick with the JPEG format to avoid going over the 200K limit.

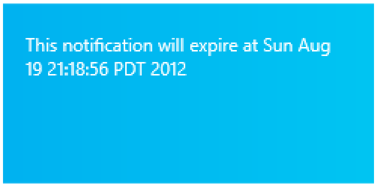
Cycling, Scheduled, and Expiring Updates

Although you might read the heading above and think we'll now be covering a grab bag of randomness, all it really means is that we're looking at additional methods of the `TileUpdater` object and revisiting the two properties of the `TileNotification` object that we already mentioned. Simply said, now that we've seen how to do all the basic tile updates, we're ready to start exploring the additional capabilities. Again, everything here applies to all tiles in the app, including secondary tiles.

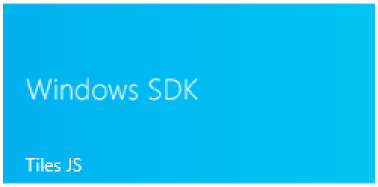
First is the ability to programmatically clear all updates and reset the tile to its default state as defined in the manifest. This happens with a simple call to `TileUpdater.clear` (shown in scenario 1 of the App tiles and badges sample, js/sendTileText.js):

```
Windows.UI.Notifications.TileUpdateManager.createTileUpdaterForApplication().clear();
```

The next capability, as already mentioned, is to set the `TileNotification.expirationTime` property before sending that notification to `TileUpdater.update`. This ensures that a locally issued notification will be automatically removed, and it lets you override the default three-day expiration period for cloud-issued updates. The update will appear immediately (at the next tile refresh, that is) and will then be removed from the tile after it expires. This is demonstrated in scenario 8 of the sample—sending an update with an expiration date will display an update as on the left below. When it expires, it's removed, which in this case causes the tile to revert to its default state, shown on the right:



This notification will expire at Sun Aug 19 21:18:56 PDT 2012



Windows SDK

Tiles JS

To *delay* the appearance of an update until a specified time, with or without an expiration, you do something a little different at the beginning: instead of creating a `TileNotification` object to send to the updater, create a `ScheduledTileNotification` and send it to `TileUpdater.addToSchedule`.

A `ScheduledTileNotification` is exactly the same as a `TileNotification` (including the expiration time) except that it contains an extra property called `deliveryTime` that indicates when—in UTC time, not local time!—the tile should first appear. For an example of this we have to take a brief detour to scenario 1 of the Scheduled notifications sample. But all that's really different is that we take whatever `XmlDocument` we've created with the payload—through any of the methods covered earlier—and create the notification with the delivery time. Here's a condensation of the code in the sample's js/scenario1.js file:

```
// Namespace variable
var Notifications = Windows.UI.Notifications;

// The delay in delivery time from the sample's control
var dueTimeInSeconds = parseInt(document.getElementById("futureTimeBox").value);

// The actual delivery date and time
var currentTime = new Date();
var dueTime = new Date(currentTime.getTime() + dueTimeInSeconds * 1000);

// In here we create the XmlDocument in the variable tileDOM

// Now create the update with the delivery date
var futureTile = new Notifications.ScheduledTileNotification(tileDOM, dueTime);
Notifications.TileUpdateManager.createTileUpdaterForApplication().addToSchedule(futureTile);
```

Other than these small changes, everything else about the tile update is the same as before.

It's certainly possible, as you can guess, to queue up many scheduled updates. At any time you can call `TileUpdater.getScheduledTileNotifications` to obtain a vector of active `ScheduledTileNotification` objects. You can also remove any of those updates with `TileUpdater.removeFromSchedule`.

What happens if you schedule a series of updates that will end up being active at the same time? In the Scheduled notifications sample, for instance, issue a series of tile updates for 10 seconds from now, and then quickly switch to the Start screen to see the results. Those updates will appear in sequence, and one of them might actually be dropped if another update is scheduled right on its heels. And once you reach the last update, it just stays there until it expires, the tile is cleared, or some new update comes along.

In such cases it would be better to have the tile cycle through a series of tiles, thereby keeping the tile active with more than just the last update.

Live tiles support cycling through up to five updates, where the duration of each is controlled by the system so that the whole Start screen has a consistent look and feel. To enable this, you must first call `TileUpdater.enableNotificationQueue(true)` or one of its `enableNotifcationQueueFor<size>` siblings (such as `enableNotifcationQueueForSquare150x150`), and you can call these with `false` to disable cycling for all tiles or a specific size. The queue itself is first in, first out (you cannot control the order otherwise), so the oldest notification is removed if a new one is added when the queue is already at maximum. In other words, if you enable the queue and issue updates as we've been doing, the five most recent updates will cycle.

Tip If you're sending duplicate large tile updates in payloads with distinct medium and wide updates, remember to set the *contentId* property of the large size binding element to prevent duplicates in the queue. Refer back to "Tile Template Schema" earlier in this chapter.

You might want to selectively replace existing updates already in the queue rather than rely on the first-in first-out behavior (or calling TileUpdater.clear and reissuing the update you want to retain). This is the purpose of the tag property in TileNotification and ScheduledTileNotification. The tag is again just a maximum 16-character string that identifies a particular update. If the queue is enabled, a new update with any given tag will replace any update already in the queue with the same tag. If that tag doesn't exist, the update will replace the oldest in the queue. So, for example, a news app might have five tags for different categories of headlines such as world, local, politics, business, and health; a stock app would obviously use tags for different ticker symbols. Similarly, a weather app might tag updates with a zip code or other location identifier.

You can play with all this in scenario 6 of the App tile and badges sample. In the process you might note that when activating an app from a cycling live tile, it does not receive an indicatio n as to which update is currently shown. For this reason, it's currently recommended that activating a cycling live tile opens the app on a hub page that displays relevant content for all the updates together.

Badge Updates

The last bit you can use to update a live tile is a *badge*. I've kept this topic separate because it works through a separate API and is not part of the tile update XML payloads we've been using.

To review, badges are simply small glyphs—a one- or two-digit number, or one of a small number of symbols—that appear on a tile regardless of any other update activity. They are meant to indicate the status of the app rather than provide a piece of content, and they're animated independently (using WinJS.UI.Animations.updateBadge, as briefly noted in Chapter 14).

How you send badges to your tile is structurally similar to a tile update. Start by creating an XmlDocument payload for the badge update by using a template from the Badge overview or using the Notifications Extensions Library, which contains full support for badges. In the case of badges, it's overkill to speak of an XML "document" because it contains only one element, badge, with one attribute, value, for which you can indicate a number from 1–99 (anything over that will display 99+) or one of 12 specific glyphs, as shown below. Note that although the glyphs are shown here against a blue background, the actual color will be the Background Color in your manifest:

Value	Glyph	XML
(none)	n/a	`<badge value="none"/>`
(number 1-99)	1	`<badge value="1"/>`
(number over 99)	99+	`<badge value="123456"/>`
activity	🔄	`<badge value="activity"/>`
alarm	🔔	`<badge value="alarm"/>`
alert	✱	`<badge value="alert"/>`
available	●	`<badge value="available"/>`

away	🟡	`<badge value="away"/>`
busy	🔴	`<badge value="busy"/>`
newMessage	✉	`<badge value="newMessage"/>`
paused	⏸	`<badge value="paused"/>`
playing	▶	`<badge value="playing"/>`
unavailable	⚪	`<badge value="unavailable"/>`
error	⊗	`<badge value="error"/>`
attention	❗	`<badge value="attention"/>`

If you like, you can use the BadgeUpdateManager.getTemplateContent function to obtain an XmlDocument with such contents; there's a bit of sample code on this method's reference page that shows how. But because the XML is so simple, it's just as easy to create the object from a string by using new Windows.Data.Xml.Dom.XmlDocument followed by a call to its loadXml method, as we've seen with tiles. The Notifications Extensions Library also has methods for this and doing updates. Both of these approaches are demonstrated in scenario 4 of the App tiles and badges sample.

However you create it, the next step is to instantiate a BadgeNotification with that XmlDocument:

```
var badge = new Windows.UI.Notifications.BadgeNotification(badgeDOM);
```

This notification object also supports an expirationTime property just like tiles do. That aside, the last step is to call BadgeUpdateManager.createBadgeUpdaterForApplication to obtain a BadgeUpdater for your app tile or—you can predict this one—BadgeUpdateManager.createBadgeUpdaterForSecondaryTile to obtain a BadgeUpdater for a secondary tile with a given tileId. After you obtain your BadgeNotification (and again, the Notifications Extensions Library can help here), you then call BadgeUpdate.update:

```
Windows.UI.Notifications.BadgeUpdateManager.createBadgeUpdaterForApplication().update(badge);
```

or:

```
Windows.UI.Notifications.BadgeUpdateManager.createBadgeUpdaterForSecondaryTile(
    "SecondaryTile.LiveTile").update(badge);
```

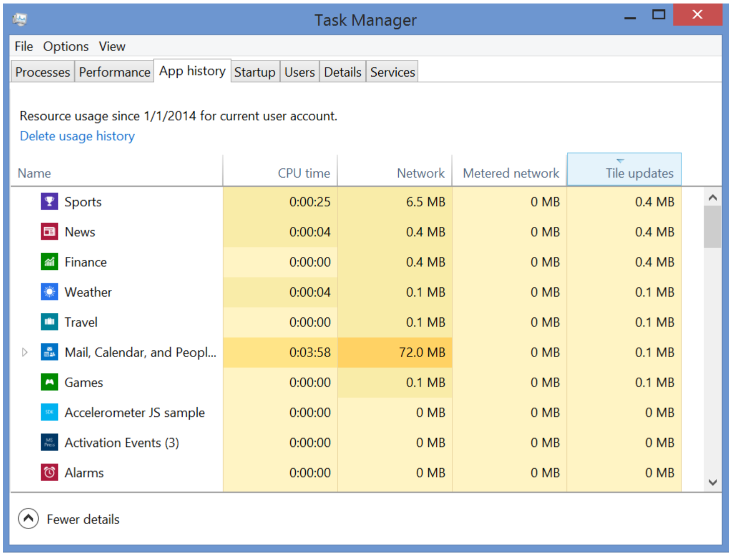
And as with tiles, BadgeUpdater.clear removes the badge entirely, which is equivalent to sending an update with the value of none.

Beyond that, the BadgeUpdater class has just two additional methods, startPeriodicUpdate and stopPeriodicUpdate, which are also found on the TileUpdater class. And wouldn't you know it?

Periodic updates just so happen to be our next topic—yes, I planned it this way too!

Sidebar: How Much Network Traffic for Tiles?

Included with the many improvements to Task Manager is the ability to track your app's network traffic for tile updates. Run Task Manager, make sure you click More Details in the lower left, and then click the App History tab. Network traffic for tile updates is shown on the rightmost column. This number will typically be small, but it's a metric you can monitor to see whether updates become excessive.


Task Manager
File Options View
Processes Performance App history Startup Users Details Services
Fewer details

Resource usage since 1/1/2014 for current user account.
Delete usage history

Name	CPU time	Network	Metered network	Tile updates
Sports	0:00:25	6.5 MB	0 MB	0.4 MB
News	0:00:04	0.4 MB	0 MB	0.4 MB
Finance	0:00:00	0.4 MB	0 MB	0.4 MB
Weather	0:00:04	0.1 MB	0 MB	0.1 MB
Travel	0:00:00	0.1 MB	0 MB	0.1 MB
Mail, Calendar, and Peopl...	0:03:58	72.0 MB	0 MB	0.1 MB
Games	0:00:00	0.1 MB	0 MB	0.1 MB
Accelerometer JS sample	0:00:00	0 MB	0 MB	0 MB
Activation Events (3)	0:00:00	0 MB	0 MB	0 MB
Alarms	0:00:00	0 MB	0 MB	0 MB

Periodic Updates

As described earlier in this chapter, periodic updates configure the system to automatically request tile and/or badge updates from a service on behalf of an app. Provided that the app has declared the *Internet (Client)* capability, this enables the app to continually keep its live tiles fresh without needing to run at all. Periodic updates for tiles (but not badges) can also be configured in your app manifest so that tile updates begin upon install, without the need to run the app.

Periodic updates are great proof that Microsoft listens to developer feedback. When the first Developer Preview of Windows 8 was released in September 2011, many developers were interested in implementing live tiles but it could only be done with push notifications and the Windows Push Notification Service, even if the tiles needed only low-frequency updates. In other words, push notifications were total overkill for apps that needed to get only a bit of data from a service every once in a while to create their updates. So developers asked, "Is it possible to have my app just run in the background, periodically request data from my service, and then issue tile or badge updates?"

It was a completely legitimate request, but, as described earlier, background tasks are carefully controlled and allowed only for very specific scenarios. Hearing this feedback, the tiles and notifications team at Microsoft studied the problem and found that creating a new class of background task would also be overkill for low-frequency tile updates. Furthermore, they found that even if apps could use a background task for this purpose, they'd all pretty much do the same thing: poll data from a service and populate a tile or badge template. So, instead of adding a background task, they added an API for system-managed periodic updates for Windows 8 and then added the ability to configure periodic updates from the manifest in Windows 8.1.

This API for setting up periodic updates from the app consists of the following methods of the TileUpdater and BadgeUpdater classes:

- TileUpdater.startPeriodicUpdate and BadgeUpdater.startPeriodicUpdate Configure Windows to request an update from a given URI with a specified period (see below). These calls remove any previous URIs registered for the tile. An app can specify an optional date and time around which regular polling should begin, and in all cases the first request will happen immediately (very helpful for debugging!). A good time to call these is when the app is launched, when it's resumed, and when a configuration changes that might alter the URI.

- TileUpdater.stopPeriodicUpdate and BadgeUpdater.stopPeriodicUpdate Cancels the current periodic update process but does not clear the tile of existing updates. To do that, call the updater's clear method.

- TileUpdater.startPeriodicUpdateBatch For tile updates only, is identical to startPeriodicUpdate but accepts an array of up to five URIs, automatically creating an update queue with the results (replacing any previous URIs). Note that TileUpdater.enableNotificationQueue must be set to true prior to using this method, as described earlier in "Cycling, Scheduled, and Expiring Updates."

In all these cases, each URI is represented by a Windows.Foundation.Uri object and the polling period is set with a value from the PeriodicUpdateRecurrence enumeration. Values are halfHour, hour, sixHours, twelveHours, and daily giving a clear indication that periodic updates are meant for content that changes relatively infrequently, like the weather, daily offers from local retailers, or the phases of the moon. For anything that requires more timely delivery, like appointment reminders, traffic conditions, current sports scores, or online auction status, you'll need to use push notifications.

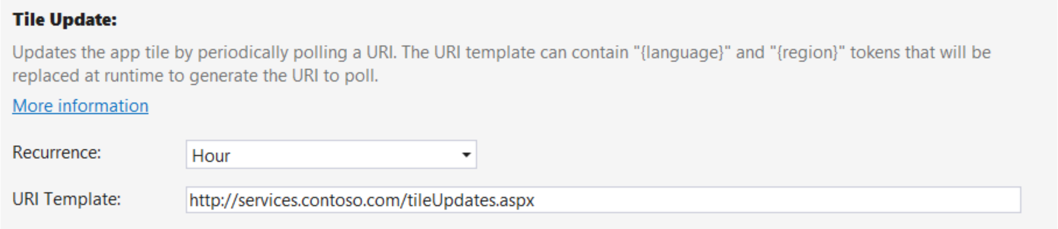
In addition, all these updates can employ tags and expiration times like local updates. One detail is that for any given polling request, a service can respond with only a single update payload—hence the need for startPeriodicUpdateBatch. That said, if the notification queue is enabled and the updates from the same service URI contain different tags, the live tile will behave the same as if those updates were issued locally: the most recent update for each tag (up to five) will be cycled through the queue.

Hint The periodic update API does not directly provide a means to authenticate with the service. This typically isn't necessary because periodic updates are not designed to be user-specific. However, you can certainly include encrypted credentials in the URI with a query string. You might also be able to use the *Enterprise Authentication* capability if the app is running on a domain-joined system.

The app side of the periodic update scene is demonstrated in the [Push and periodic notifications client-side sample]() (which I'll often refer to as just the Push notification sample). Specifically, see scenarios 4 and 5, which are somewhat general tools to employ the `TileUpdater` and `BadgeUpdater` methods with a given service URI as I've just described. A few lines of that code (condensed from js/scenario4.js) appear as follows:

```
var notifications = Windows.UI.Notifications;
var updater = notifications.TileUpdateManager.createTileUpdaterForApplication();

updater.enableNotificationQueue(true);
updater.startPeriodicUpdate(urisToPoll[0], recurrence);
updater.startPeriodicUpdateBatch(urisToPoll, recurrence);
```

To configure a periodic update through the app manifest, you specify the recurrence and the URI in the Application > Tile Update section:



Tile Update:

Updates the app tile by periodically polling a URI. The URI template can contain "{language}" and "{region}" tokens that will be replaced at runtime to generate the URI to poll.

More information

Recurrence: Hour

URI Template: http://services.contoso.com/tileUpdates.aspx

When the app is installed, Windows will take these two settings and call `startPeriodicUpdate` on your behalf, with the result that tile updates will begin before the app is even run the first time. Updating the tile from any other app code (including background tasks) will then override the settings in the manifest unless the app is uninstalled and reinstalled. This means that if you clear all updates from the tile and don't reestablish other updates, the tile will revert to your static logo images.

Tip The *{language}* and *{region}* tokens that you can place in the URI (as indicated in the manifest editor) are used to localize tile updates from the server. We'll cover these in Chapter 19.

As you can see, setting up periodic updates from the app is simple: the real work lies in the service itself, whose responsibility it is to return the appropriate XML with which Windows can create a local update. Indeed, being able to even run the client-side sample requires a service to which you can make requests, and unfortunately the Windows SDK does not provide one. So let's remedy that situation with a service of our own.

Because you'll likely use the client-side sample to play around with your own update service, though, you should make two changes, specifically to clear existing updates in the functions that stop polling. In js/scenario4.js, change the `stopTilePolling` function to read as follows:

```
function stopTilePolling() {
    var updater = notifications.TileUpdateManager.createTileUpdaterForApplication();
    updater.clear();
    updater.stopPeriodicUpdate();
    WinJS.log && WinJS.log("Stopped polling.", "sample", "status");
}
```

Similarly, change `stopBadgePolling` in js/scenario5.js to read as follows:

```
function stopBadgePolling() {
    var updater = notifications.BadgeUpdateManager.createBadgeUpdaterForApplication();
    updater.clear();
    updater.stopPeriodicUpdate();
    WinJS.log && WinJS.log("Stopped polling.", "sample", "status");
}
```

Without these changes, old updates will persist on the tile even after you stop the updates. If you then change your service but it has an error in the XML, you won't see any change on the tile and might think that the update worked when it really didn't. Trust me: making these small changes will simplify your life!

Creating an Update Service

Creating a service for periodic updates means creating a web page at some given URI whose sole purpose is to respond to an HTTP request with XML content for a `TileNotification` or `BadgeNotification` object. Ideally, such a page also handles the scaling, accessibility, and localization parameters provided in the query string described earlier in "Using Local and Web Images."

The page can be implemented using whatever language, tools, and host you want. In fact, unless you enjoy programming in Notepad, you'll want to utilize a good web development tool! One option covered later is to create an API through Windows Azure Mobile Services, but let's talk first about the Visual Studio tools. Visual Studio Express for Windows is not itself equipped for the task of creating websites or services: you'll need either the full version of Visual Studio or the free Visual Studio Express *for Web*; more on this in a moment. And if you use ASP.NET for your service, you can employ the Notifications Extensions Library for creating the tile XML.

As a minimal example, here is a functional one-line PHP page that generates XML for a badge update with the current day of the month:

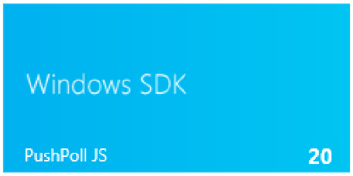
```
<?php echo "<badge value='".date("j")."'/>"; ?>
```

For proper XML we should also include a header element (the tile works with or without it):

```
<?php echo '<?xml version="1.0" encoding="utf-8"?>';
     echo "<badge value='".date("j")."'/>"; ?>
```

Drop this code into a .php file (see HelloTiles/dayofmonthservice.php in the companion content) on whatever web server you might have access to and—voila!—there's a basic service that delivers badge updates. In fact, I've deployed this to a test service I created for this book on Windows Azure: http://programmingwin8-js-ch13-hellotiles.azurewebsites.net/dayofmonthservice.php. I'll walk through this process in "Windows Azure and Azure Mobile Services." Note that I created this for the first edition of the book when the present chapter was Chapter 13 instead; I've kept that original URI to demonstrate that the service side of things can be independent of the client, as we'll also see with tile updates.

Anyway, you can use all this in scenario 5 of the Push and periodic notifications client-side sample—enter your service URI in the box, press the button to start polling, and then check the sample's tile on the Start screen. In a few seconds you should see the day of the month appear as a badge. (Of course, with this ultrasimplistic example the date will reflect the local time on the server rather than the device, which could be completely mismatched. A real service would be sensitive to time zone and other locale-specific considerations.)



Tip If your service generates an error when handling the update request, you might see exceptions in your app code that otherwise don't make any sense. With such mystery exceptions, carefully debug your service locally as described in "Debugging a Service Using the Localhost" later on.

Tip The tile and badge updaters are *very* sensitive to properly formed XML. In the PHP code above, leaving off the closing / for the badge element will cause the update to fail (that is, not appear). Avoiding such trivial errors is again why the Notifications Extensions Library was created, at least for ASP.NET. I'm hoping that some enterprising reader might consider a similar project for PHP and other server-side languages!

Another good option besides PHP is ASP.NET Razor syntax (typically in a .cshtml file), introduced with Microsoft WebMatrix. Razor/WebMatrix, along with tools such as Visual Studio Express for Web and a whole lot else, can be installed through the Web platform installer. To familiarize yourself with Razor, which works much in the same way as PHP, start with Walkthrough: Creating a Web Site using Razor Syntax in Visual Studio.

To make that long story short, here are the steps in Visual Studio Express for Web to create a simple tile update service with Razor:

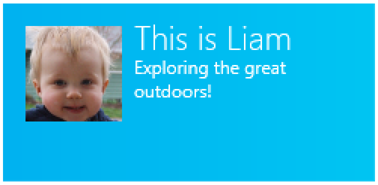
- Select File > New Web Site from the menu.
- In the New Web Site dialog, select ASP.NET Web Site, give it a project name and folder, and press OK. Call this site *HelloTiles*.
- Once the project is created, you should see the Default.cshtml file opened.

- Copy and paste the following Razor code (that begins with `@{`) into that file, replacing the default contents. The little piece of C# code at the top for the `weekDay` variable is something we'll use later to demonstrate debugging; it's not used in generating the XML. Here, note that I tested and generated the XML contents by using the tile designer in scenario 6 of the App tiles and badges sample (refer back to Figure 16-14); this saved me lots of time wondering whether my XML was correct.

```
@{
  //
  // This is where any other code would be placed to acquire the dynamic content
  // needed for the tile update. In this case we'll just return static XML to show
  // the structure of the service itself.
  //
  var weekDay = DateTime.Now.DayOfWeek;
}
<?xml version="1.0" encoding="utf-8" ?>
<tile>
@* version="2" enables Windows 8.1 templates but retains compatibility *@
    <visual version ="2" lang="en-US">
@* The fallback attribute specifies the Windows 8 template name *@
       <binding template="TileSquare150x150PeekImageAndText02" branding="none"
          fallback="TileSquarePeekImageAndText02" contentId="1">
          <image id="1" src="http://www.kraigbrockschmidt.com/images/Liam07.png"/>
          <text id="1">Liam--</text>
          <text id="2">Giddy on the day he learned to sit up!</text>
       </binding>
       <binding template="TileWide310x150SmallImageAndText04" branding="none"
          fallback="TileWideSmallImageAndText04" contentId="1">
          <image id="1" src="http://www.kraigbrockschmidt.com/images/Liam08.png"/>
          <text id="1">This is Liam</text>
          <text id="2">Exploring the great outdoors!</text>
       </binding>
@* Always put the large tile at the end for compatibility with Windows 8 systems *@
         <binding template="TileSquare310x310SmallImagesAndTextList03" branding="none"
            contentId="1">
          <image id="1" src="http://www.kraigbrockschmidt.com/images/Liam07.png"/>
          <text id="1">Liam--</text>
          <text id="2">Giddy on the day he learned to sit up!</text>
          <image id="2" src="http://www.kraigbrockschmidt.com/images/Liam08.png"/>
          <text id="3">This is Liam</text>
          <text id="4">Exploring the great outdoors!</text>
          <image id="3" src="http://www.kraigbrockschmidt.com/images/Liam13a.jpg"/>
          <text id="5">Still Liam</text>
          <text id="6">He's older now, almost 7</text>
       </binding>
    </visual>
</tile>
```

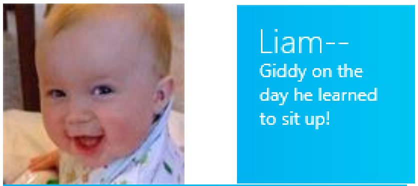
- Run the website in Internet Explorer using the Debug > Start Debugging command or the Internet Explorer toolbar button (where you'd find the Local Machine or Simulator options in Visual Studio Express for Windows). This will launch Internet Explorer with a URI like *http://localhost:52568/HelloTiles/Default.cshtml* where the port number is what routes the URI to the site running in the debugger. If you need to set up your localhost server, see “Debugging a Service Using the Localhost.”

- Run the Push notifications sample, switch to scenario 4, paste the URI into the URL 1 field, and press the Start periodic updates.
- If all is well, you should see the wide tile update as follows. (OK, so it’s another shameless picture of my kid…what can I say? You have to give us fathers a break!) If you don’t see the update right away, try right-clicking and changing the tile size.

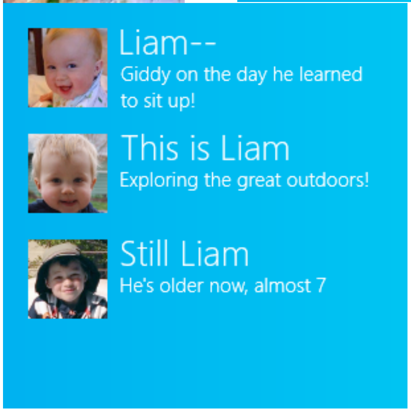


This is Liam
Exploring the great outdoors!

- If you make the tile smaller (to the medium size), you’ll see another gratuitous picture of my kid and peek text (below left), and the large tile size gives you yet more (below right):



Liam--
Giddy on the day he learned to sit up!



Liam--
Giddy on the day he learned to sit up!

This is Liam
Exploring the great outdoors!

Still Liam
He's older now, almost 7

The complete website code for all this can be found in the HelloTiles example in this chapter’s companion content (and the page itself is deployed to my Windows Azure host at http://programmingwin8-js-ch13-hellotiles.azurewebsites.net/Default.cshtml). As simple as it is, it provides the basic framework in which you can add code to generate more dynamic results. In the top part of the file, within the `@{ }`, you can write whatever C# code you need.

Let me point out that although the tile update from Default.cshtml is a version 2 XML payload, I'm hosting it at the same URI on Azure that I used for the first (Windows 8) edition of this book that supported only version 1 payloads. Everything in the Windows 8 samples will still work because I've made sure to place the large tile update last, thereby keeping the rest of the payload v1 compatible.

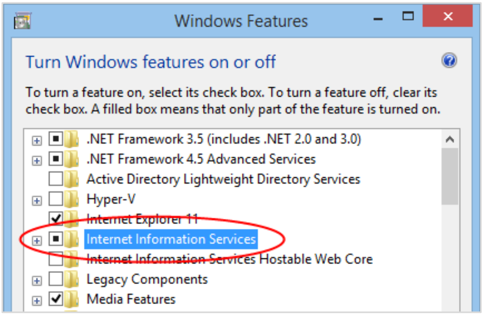
Furthermore, there is a second page named Default2.cshtml in HelloTiles (and on my Azure host) that has different medium and wide tile payloads but the same large tile payload. In this XML you can see that I've used the same `contentId` for the large tile so that it appears only once while the medium and wide tiles cycle through two. You can again use scenario 4 of the Push notifications sample to put in either localhost (or Azure) URIs to see the effect.

Finally, the HelloTiles example includes another page called *DefaultNE.aspx* that shows how to use the Notifications Extensions Library on a web server. This requires that the library be compiled for use on an ASP.NET host rather than as a WinRT component, which I explain in my blog post Alive with Activity, part 2: Writing and debugging services for live tiles, under "Writing Services." The companion content has a modified version of the library in the Notifications Extensions for Web folder.

Debugging a Service Using the Localhost

Debugging a tile and badge update service with periodic notifications can be a difficult proposition. You can just enter your service's URI in a browser and use its View Source command to examine the XML, but how do you step through your server-side code to isolate problems?

The solution is to run your service on your local machine, as we just did in the previous section, where the URI references your localhost server, however you want to set it up. You can install a server like Apache or you can use the Internet Information Services (IIS) that's built into Windows and integrated with the Visual Studio tools. To turn on IIS, go to Control Panel > Programs and Features > Turn Windows Features On Or Off as we saw also in Chapter 4. At the top level, check the Internet Information Services box at the top level, and when you click OK the core features will be installed:



Windows Features

Turn Windows features on or off

To turn a feature on, select its check box. To turn a feature off, clear its check box. A filled box means that only part of the feature is turned on.

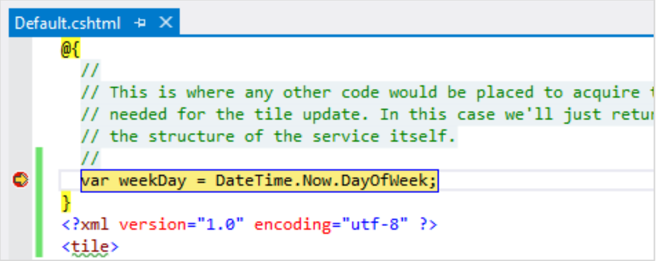
- ■ .NET Framework 3.5 (includes .NET 2.0 and 3.0)
- ■ .NET Framework 4.5 Advanced Services
- □ Active Directory Lightweight Directory Services
- □ Hyper-V
- ☑ Internet Explorer 11
- ■ Internet Information Services
- □ Internet Information Services Hostable Web Core
- □ Legacy Components
- ☑ Media Features

Once installation is complete, the `http://localhost/` URI is mapped to *c:\inetpub\wwwroot*. That's where you drop something like the PHP page described in the last section so that you can use a URI like *http://localhost/dayofmonthservice.php* in the Push notifications sample (scenario 5, in this case, for badge updates). In doing so, you can now use your favorite server-side tools for debugging.

To use PHP with IIS, you might need to install it through Microsoft's Web platform installer or the server-side code won't execute properly. After PHP installation, try entering the URI for the PHP page in your browser. If you get an error message that says "Handler PHP53_via_FastCGI has a bad module" (yeah, that's really helpful!), return to the Turn Windows Features On Or Off dialog, navigate to Internet Information Services > World Wide Web Services > Application Development Features, check the box for CGI, and press OK. Once the CGI engine is installed, your PHP page should work.

If you plan to work in ASP.NET or Razor, I highly recommend you also install Visual Studio Express for Web through the Web platform installer. When you run a website in its debugger, it assigns a port on localhost such as *http://localhost:53528* and launches Internet Explorer with that URI. The port links the browser to the debugger, so if you set a breakpoint in the page code, the debugger will stop at that point whenever there's a page request, allowing you to step through the code with the same features we've been enjoying with writing Windows Store apps.

For example, load up the HelloTiles example site with this chapter in Visual Studio Express for Web, set a breakpoint on the `var weekDay` line at the top (which is there only to give you something on which to set a breakpoint), and start debugging. Once Internet Explorer has loaded default.cshtml, copy and paste its URI into scenario 4 of the Push notifications sample. Press the Stop Periodic Updates button followed by the Start Periodic Update button to force a new request to the URI and—magic!—you should hit the breakpoint in the service:



```
@{
  //
  // This is where any other code would be placed to acquire t
  // needed for the tile update. In this case we'll just retur
  // the structure of the service itself.
  //
  var weekDay = DateTime.Now.DayOfWeek;
}
<?xml version="1.0" encoding="utf-8" ?>
<tile>
```

Windows Azure and Azure Mobile Services

Generally speaking, a real service that provides tile and badge updates would typically be connected on the server side to some other useful source of information, perhaps to an always-running process that can monitor other sites, extract the desired data, and generate updates. Those updates can be returned from page requests, as we've seen here, or fed directly to the Windows Push Notification Service (WNS) so that they can be pushed directly to specific clients (as we'll see later).

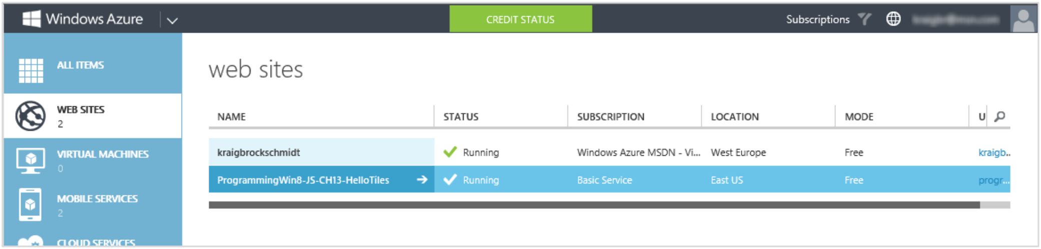
This is where the various capabilities of Windows Azure come in very handy, especially those of Windows Azure Mobile Services. But let's take a step back for a moment and consider the bigger picture of Windows Azure.

When you're ready to upload an app to the Windows Store and have real customers using your service, you'll need to consider where, exactly, you'll host that service so that it can scale to what hopefully becomes a very large customer base! During your development and testing process, of course, you can host the service anywhere you want because only a few instances of your app will ever call upon it. But if your app is acquired by many customers and each instance of that app starts banging on the host server for tile and badge updates, that host might soon become overloaded!

Windows Azure is thus a system that allows you to add more server power when it's necessary and scale back when it's not (and you pay only for what is actually used). To get started, visit http://www.windowsazure.com where you can set up a free 90-day trial for a hosted site and continue working within a free pricing tier indefinitely. The Windows Azure site also provides direct support and SDKs for .NET, node.js, PHP, Java, and Python, along with Visual Studio Express with Web for Windows Azure SDK—all available again through the Web platform installer. More importantly, many of the features you will likely need for apps are again provided through Windows Azure Mobile Services.

As a brief walkthrough to get you going, I started my Windows Azure trial, installed the Windows Azure SDK for .NET, and deployed the HelloTiles example service with these steps:

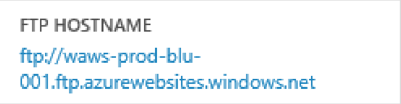
- Go to the Windows Azure Management Portal, and sign in with your account.
- Create a new website with some URI. I used *ProgrammingWin8-JS-CH13-HelloTiles* (originally for the first edition of this book), making the full site URI http://programmingwin8-js-ch13-hellotiles.azurewebsites.net/. This shows up in my portal under Web Sites:



web sites

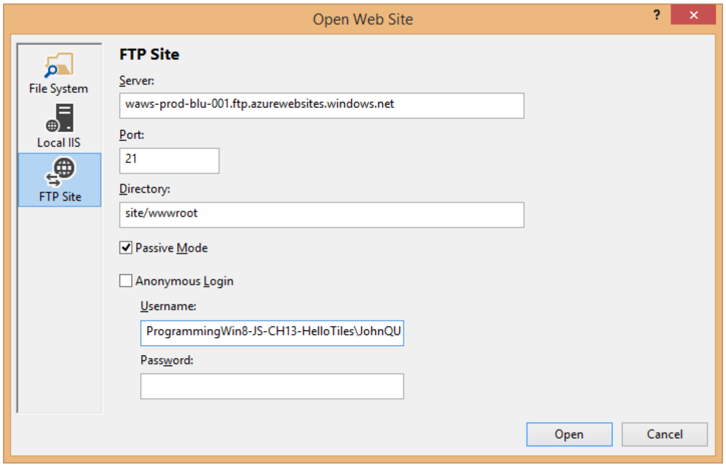
NAME	STATUS	SUBSCRIPTION	LOCATION	MODE	U
kraigbrockschmidt	✓ Running	Windows Azure MSDN - Vi...	West Europe	Free	kraigb..
ProgrammingWin8-JS-CH13-HelloTiles	✓ Running	Basic Service	East US	Free	progr...

- To upload the site, you must set up FTP credentials. Click the site in the list, which takes you to its specific dashboard. Under Publish Your App, click Set Up Deployment Credentials and enter a username and password.
- Click Dashboard along the top of the window, and scroll down the right side to find the FTP hostname:



FTP HOSTNAME
ftp://waws-prod-blu-001.ftp.azurewebsites.windows.net

- In Visual Studio Express for Web, right-click the project and select Copy Web Site. In the dialog that appears, click Connect along the top to open the Open Web Site dialog (below). Select FTP site on the left, enter the FTP hostname on the top, enter *site/wwwroot* in Directory, and then enter your credentials at the bottom. Make sure to prefix your username with the site name and a backslash (for example, ProgrammingWin8-JS-CH13-HelloTiles\JohnQUser) or you'll be confused by Azure's refusal to let you through the door! Finish by clicking Open.



FTP Site

Server: waws-prod-blu-001.ftp.azurewebsites.windows.net

Port: 21

Directory: site/wwwroot

☑ Passive Mode

☐ Anonymous Login

Username: ProgrammingWin8-JS-CH13-HelloTiles\JohnQU

Password:

- Once Visual Studio Express for Web is connected to the Azure site, you can upload your files through the Copy Web Site pane.

- When that's complete, you should be able to use the site URI plus a page name as the URI for the badge and tile update services. Again try *http://programmingwin8-js-ch13-hellotiles.azurewebsites.net/dayofmonthservice.php* with the Push and periodic notifications client-side sample, scenario 5, and both *http://ProgrammingWin8-JS-ch13-HelloTiles.azurewebsites.net/Default.cshtml* and Default2.cshtml with scenario 4, and you'll see the same results as when using the localhost. You can also just visit these URIs in a browser to see the XML they're producing.

With this, you now have the ability to scale up your Windows Azure hosting, with your app supplying the hosted URI to the periodic update API. This also puts you in a good position to use push notifications, as we'll see later, which specifically employs Windows Azure Mobile Services (which I refer to as AMS rather than its slang name "Zumo").

Put simply, a *mobile service* on Azure is a collection of REST endpoints that provide powerful capabilities that apps need:

- Service endpoints (APIs), such as those to provide periodic tile and badge updates.

- Scheduled background jobs that can make requests to other services and store the results in your database. An example of this, to obtain tweets from Twitter, can be found on the Schedule recurring jobs in Mobile Services topic.
- Cloud storage in a SQL Server database, which is accessible like any other web-hosted SQL Server database and as tables through the client-side JavaScript library. The database can serve as a central data-gathering point from any number of web sites, services, scheduled jobs, and so forth, as shown in Figure 16-15 (coming up).
- Support for scalable push notifications (tiles, badges, toasts, and raw notifications alike).
- Authentication through different identity providers.

You can learn much more about mobiles services and the Mobile Services SDK on the How to use an HTML/JavaScript client For Windows Azure Mobile Services.

The utility of a central database becomes apparent when you think about creating a periodic notification service that draws data from other services. Although your service's code could send off other HTTP requests to get data, those requests are asynchronous and are prone to a variety of failure conditions. This makes it difficult to just respond with an XML payload for a tile or badge update.

To simplify matters, we can take advantage of the fact that Windows will make requests to your service only at 30-minute or longer intervals, as enforced by the client API. This means you have lots of time during which other server-side processes—scheduled jobs, for example—can make external requests to monitor weather alerts, leaderboards, RSS feeds, and just about anything else for which there's a web API. Those processes store results in your database, which are then ready for your periodic notification service to query (synchronously) when it receives its next request.

Indeed, any number of agents can update the same database. For example, users might be entering data through your website. They might be using mobile phone apps to track their activities, with results being automatically uploaded to a database. The database might also be getting updated by friends who are using the same app on their devices.

Such an arrangement is again illustrated in Figure 16-15, showing how the database serves as the central store for your backend state, with the periodic notification service acting only as a simple consumer of that state.

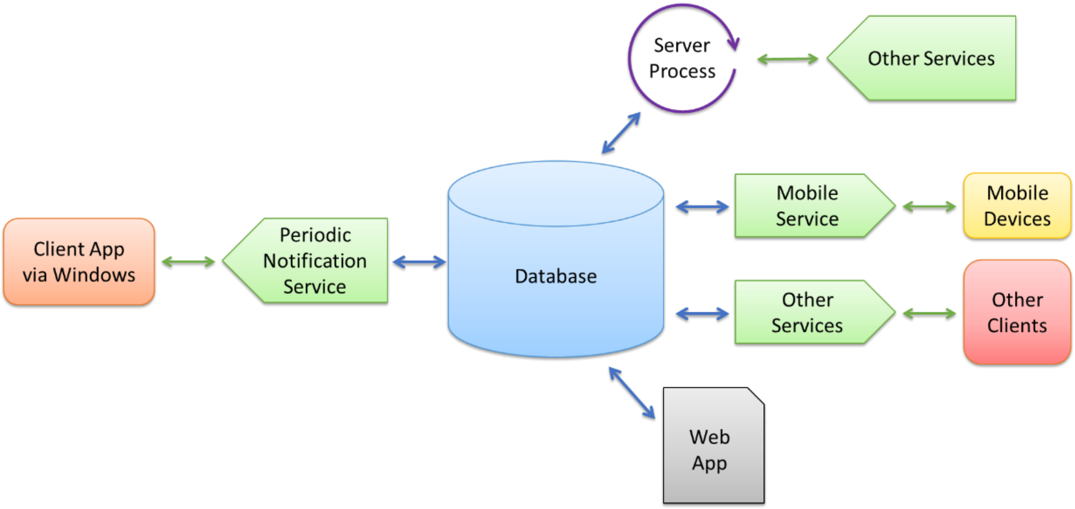
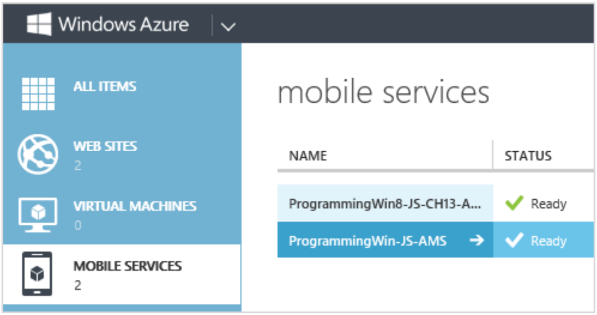


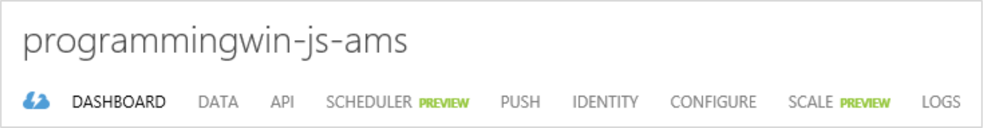
FIGURE 16-15 Using a central database to gather information that can be easily and synchronously consumed from a periodic notification service.

In any case, let's return for the moment to the question of creating a periodic update endpoint, which takes only a bit of node.js code within AMS. To create the mobile service, click Mobile Services on the left side of the Azure portal, click + New on the bottom, and enter your details. As you can see, I have two mobile services in my Azure account:



NAME	STATUS
ProgrammingWin8-JS-CH13-A...	✓ Ready
ProgrammingWin-JS-AMS	✓ Ready

Clicking the mobile service's name takes you to the dashboard where all the capabilities are arrayed along the top:



The most interesting parts here are the *Data* section, where you can create cloud storage tables that your app and services can easily access through the client library; *API*, where you can create endpoints to serve up badge and tile updates; *Scheduler*, where you create server jobs to accumulate data for your tiles; *Push*, where you configure push notifications; and *Identity*, where you can set up authentication services.

Because we're been talking about periodic updates, let's use the API feature to create the same badge update service that we implemented earlier with PHP. Click API, click Create a Custom API, and give a name such as *dayOfWeekBadge*. Also, make sure the permissions for GET are set to "Everyone" or else a nonauthenticated user—namely the Windows tile manager—will not get the response!

A few moments after you create the API, it will appear on the API page. Click that name and you'll open a script editor where you implement the service using node.js. Here's a bit of script that responds with the same badge XML payload as the PHP page:

```
exports.get = function (request, response) {
    var date = new Date();
    var day = date.getDate();

    var xml = "<?xml version='1.0' encoding='UTF-8'?><badge value='" + day + "'/>";
    response.set('content-type', 'application/xml');
    response.send(200, xml);
};
```

Once you save the script, your service endpoint is ready to go! The URI of the endpoint will be *http[s]://<service_name>.azure-mobile.net/api/<api_name>*. With my service shown here, it's *http://programmingwin-js-ams.azure-mobile.net/api/dayOfWeekBadge*. Put this URI again into scenario 4 of the Push notifications sample, and you'll see the badge on the tile. Put the URI into a browser, and you'll see the raw XML.

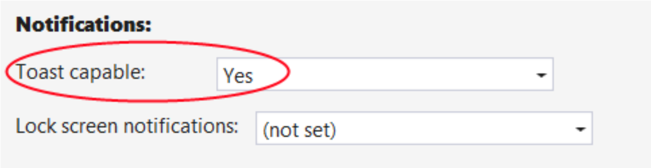
For more on node.js, see the Server script reference and Script debugging in the Azure documentation along with //build 2013 session 2-509 Node.js on Windows Azure.

Toast Notifications

So far we've exhausted the subject of tiles and tile updates, which is a great prelude to our next topic, toast notifications. This is because the process of creating and issuing toasts is quite similar to that for tiles and is simplified by the fact that toasts do not get periodic updates: they either come from the running app, from background tasks, or through push notifications, as we'll see in "Push Notifications and the Windows Push Notification Service" below. Fortunately, the topic of toasts is considerably shorter than that of tiles. Here are the salient aspects:

- Toasts always use the app's Background Color and Foreground Text settings in the manifest for branding, along with the small logo. There are no means to override this; the `branding` attribute in the XML is ignored for toasts.

- An app must set the Toast Capable setting in its manifest for any toasts to appear on its behalf. This is found in the Application > Notifications area show below. Alarms, VoIP, and certain other apps also set the Lock Screen Notifications option here as well.



Notifications:

Toast capable: Yes

Lock screen notifications: (not set)

- As shown long ago in Figure 16-6, the user can disable toasts for a particular app or disable them globally (which I find helpful when recording a screencast!). System administrators can also disable toasts by policy. To check this status programmatically, look at the `ToastNotifier.setting` property, a value from the `NotificationSetting` enumeration that will be `enabled`, `disabledForApplication`, `disabledForUser`, `disabledByGroupPolicy`, or `disabledByManifest` (meaning that you didn't set Toast Capable to Yes).

- When enabled, toasts always appear in the upper right corner of the screen (left-to-right languages) or the upper left corner (right-to-left languages). This is not configurable.

- Toasts are managed through instances of the `ToastNotification` or `ScheduledToastNotification` classes (for in `Window.UI.Notificatons`). The first supports an `expirationTime` property; the scheduled toast supports `deliveryTime`, `snoozeInterval`, and `maximumSnoozeCount` properties.

- As with tiles, the content of toasts are created with an XML payload from one of four text-only and four image-plus-text templates, as shown on the Toast template catalog. These are the same for Windows 8 and Windows 8.1 (that is, there is only one version). Toast templates are acquired from the `ToastNotificationManager` object's `getTemplateContent` method, can be created from strings, or can be created through the Notifications Extensions Library. Various options can be set in the XML:

    - Toasts can include text and an image, where the image can come from the app package, app data, or a remote source (given *Internet Client* capability). Images have the same limits as with tiles: 1024x1024 maximum resolution and 200KB maximum file size. Unlike tiles, however, if the image exceeds the limits, the notification will still show but with a gray placeholder image instead.

    - Toasts can specify a predefined sound to play when the toast appears, with a looping option. Custom sounds are not supported.

    - Toasts for alarms and incoming calls can display additional commands.

    - By default, toasts appear for seven seconds (five seconds opaque plus a two-second fade) or until activated or dismissed. The user can control this time through PC Settings > Ease of

Access > Other Options, and the opaque duration is available through the `Windows.UI.ViewManagement.UISettings.messageDuration` property. You can issue long-duration toasts, looping toasts, and recurring toasts that appear a given number of times with some interval in between.

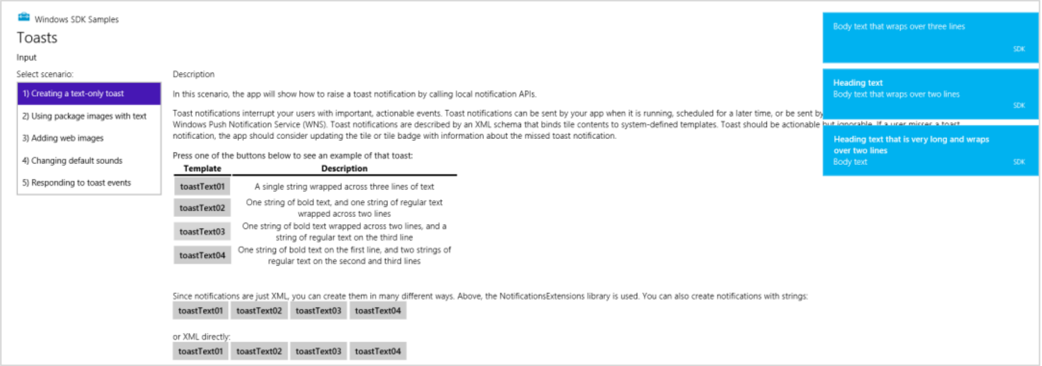
- A toast is issued through the `ToastNotifier` class, namely the `show` and `addToSchedule` methods for immediate and scheduled toasts, respectively. The `ToastNotifier` also provides methods to manage previously scheduled toasts.
- Like secondary tiles, toast notifications can (and generally should) be created with specific arguments in the XML payload that will be passed to the app's `activated` event handler with the activation kind of `launch`. Without such arguments, no `activated` event is raised, but otherwise an app handles toast notifications exactly as it would a secondary tile. Alternately, an app can listen to specific events that the toast itself will raise when it's activated or dismissed.

Now we'll examine a number of these steps, using the Toast notifications sample and Scheduled notifications sample for reference. I also recommend you review the Guidelines for toast notifications. The Lock screen call SDK sample involves toasts too, but we'll come back to that when we talk about the lock screen and background tasks.

Tip As noted before, toasts are not enabled within the Visual Studio simulator; you must run these samples on the Local Machine or a Remote Machine to see the toasts.

Creating Basic Toasts

Let's start with scenarios 1, 2 and 3 of the Toast notifications sample, which shows how to issue toasts from a running app using the text and text+image templates. As shown in Figure 16-16 and Figure 16-17 (for text-only and text+image toasts, respectively), up to three toasts can be visible at one time. Remember that you must have Toast Capable set to Yes in the app manifest for any of this to work.


Windows SDK Samples
Toasts
Input
Select scenario:
1) Creating a text-only toast
2) Using package images with text
3) Adding web images
4) Changing default sounds
5) Responding to toast events
Description
SDK
SDK
SDK

In this scenario, the app will show how to raise a toast notification by calling local notification APIs.

Toast notifications interrupt your users with important, actionable events. Toast notifications can be sent by your app when it is running, scheduled for a later time, or be sent by Windows Push Notification Service (WNS). Toast notifications are described by an XML schema that binds tile contents to system-defined templates. Toast should be actionable but ignorable. If a user misses a toast notification, the app should consider updating the tile or tile badge with information about the missed toast notification.

Press one of the buttons below to see an example of that toast:

Template	Description
toastText01	A single string wrapped across three lines of text
toastText02	One string of bold text, and one string of regular text wrapped across two lines
toastText03	One string of bold text wrapped across two lines, and a string of regular text on the third line
toastText04	One string of bold text on the first line, and two strings of regular text on the second and third lines

Since notifications are just XML, you can create them in many different ways. Above, the NotificationsExtensions library is used. You can also create notifications with strings:

toastText01 toastText02 toastText03 toastText04

or XML directly:

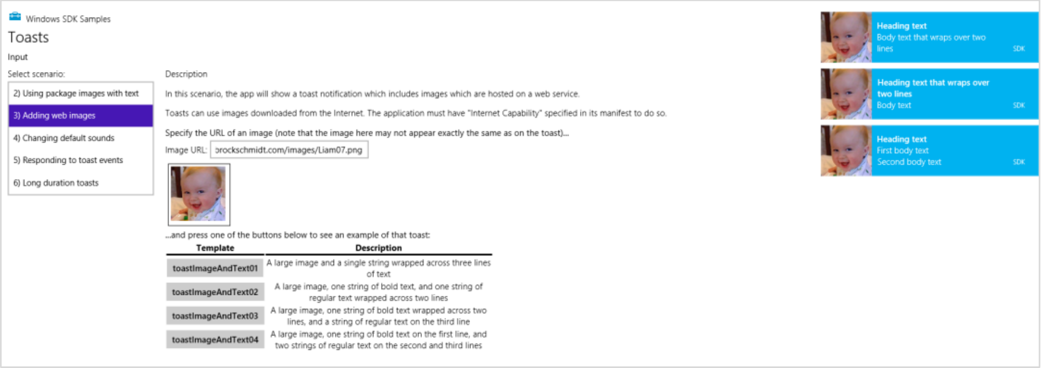
toastText01 toastText02 toastText03 toastText04

Body text that wraps over three lines

Heading text
Body text that wraps over two lines

Heading text that is very long and wraps over two lines
Body text

FIGURE 16-16 Issuing text toasts through scenario 1 of the Toast notifications sample. (The bottom of the app is cropped.)


Windows SDK Samples
Toasts
Input
Select scenario:
2) Using package images with text
3) Adding web images
4) Changing default sounds
5) Responding to toast events
6) Long duration toasts
Description
Image URL: brockschmidt.com/images/Liam07.png
Heading text
Body text that wraps over two lines
SDK
Heading text that wraps over two lines
Body text
SDK
Heading text
First body text
Second body text
SDK

In this scenario, the app will show a toast notification which includes images which are hosted on a web service.

Toasts can use images downloaded from the Internet. The application must have "Internet Capability" specified in its manifest to do so.

Specify the URL of an image (note that the image here may not appear exactly the same as on the toast)...

...and press one of the buttons below to see an example of that toast:

Template	Description
toastImageAndText01	A large image and a single string wrapped across three lines of text
toastImageAndText02	A large image, one string of bold text, and one string of regular text wrapped across two lines
toastImageAndText03	A large image, one string of bold text wrapped across two lines, and a string of regular text on the third line
toastImageAndText04	A large image, one string of bold text on the first line, and two strings of regular text on the second and third lines

FIGURE 16-17 Issuing text+image toasts through scenario 3 of the sample (the bottom of the app is again cropped). Scenario 2 does the same thing with in-package images that aren't nearly as interesting, in my paternal opinion, as my cute kid!

Just as we saw earlier with tiles, the sample shows how to create the XML payloads for toasts by using template content from ToastNotificationManager.getTemplateContent, the Notifications Extensions Library, or XML strings. The resulting XmlDocument is then used to create a ToastNotification object that is then passed to the ToastNotifier.show method.

For example, here's how scenario 1 (js/scenario1.js) issues a toast using the toastText01 template (a value from the ToastTemplateType enumeration) through getTemplateContent:

```
var Notifications = Windows.UI.Notifications;

function displayToastUsingXmlManipulation(e) {
    // toastTemplateName is set according to the button you click

    var notificationManager = Notifications.ToastNotificationManager;
    var toastXml = notificationManager.getTemplateContent(
        Notifications.ToastTemplateType[toastTemplateName]);

    // Populate the XmlDocument in toastXml (code omitted)

    var toast = new Notifications.ToastNotification(toastXml);
    notificationManager.createToastNotifier().show(toast);
}
```

The following code from scenario 3 (js/scenario3.js) demonstrates creating a toast with XML strings (toastImageAndText01). As with tiles, you can use ms-appx:///, ms-appdata:///local, or http:// URIs to refer to images (in-package, app data, and remote images, respectively), and beware of connectivity when using remote images, as explained earlier in the chapter in "Sidebar: Connectivity and Remote Images in Live Tiles and Toasts."

```
function displayWebImageToastWithStringManipulation(e) {
    // toastTemplateName is set according to the button you click

    var notificationManager = Notifications.ToastNotificationManager;
    var toastXmlString;

    if (templateName === "toastImageAndText01") {
        toastXmlString = "<toast>"
                    + "<visual version='1'>"
                    + "<binding template='toastImageAndText01'>"
                    + "<text id='1'>Body text that wraps over three lines</text>"
                    + "<image id='1' src='" + urlBox.value + "' alt='" + altText + "'/>"
                    + "</binding>"
                    + "</visual>"
                    + "</toast>";
    } else {
        // Other cases omitted
    }

    var toastDOM = new Windows.Data.Xml.Dom.XmlDocument();
    toastDOM.loadXml(toastXmlString);
    var toast = new Notifications.ToastNotification(toastDOM);
    notificationManager.createToastNotifier().show(toast);
}
```

Butter and Jam: Options for Your Toast

Beyond the properties you can assign when creating a ToastNotification object (or a ScheduledToastNotification), there are additional bits you can include within the XML, as described in the Toast schema:

- The root toast element in the XML has optional launch and duration attributes. The launch attribute can be assigned a string that will be passed to the app's activated handler as eventArgs.detail.arguments, exactly as happens with a secondary tile. (See "App Activation from a Secondary Tile" earlier in this chapter.) The duration attribute can have values of short (five seconds or the value from PC Settings > Ease of Access > Other Options) or long (25 seconds or the value from PC Settings, whichever is longer; refer back to Figure 16-6).

- The visual and binding elements in the XML can have addImageQuery attributes that act exactly like they do with tiles; refer back to "Using Local and Web Images" under "Tiles, Secondary Tiles, and Badges." The image element also supports addImageQuery for scale, language, and contrast settings.

- The branding attribute in the visual and binding elements is ignored.

- The visual, binding, and text elements support a lang attribute to identify the current app language, which helps Windows find the right font for your text.

- Toasts that are used for alarms or incoming calls can include a commands element with a scenario attribute that can be alarm or incomingCall. It then contains one or more command children with id attributes. For alarms, the command ids are snooze and dismiss; for calls we have video, voice, and decline. We'll see alarms soon; see "Lock Screen-Dependent Tasks and Triggers" for a bit more on calls.

The toast element can also have a child audio element through which you can add a sound to a toast notification provided that the user has not disabled notification sounds altogether in PC Settings > Search and Apps > Notifications. (Refer to Figure 16-6.) The particular sound is set with the src attribute whose value must be one of the following string values as described in the Toast audio options catalog:[117]

- ms-winsoundevent:Notification.Default
- ms-winsoundevent:Notification.IM
- ms-winsoundevent:Notification.Mail
- ms-winsoundevent:Notification.Reminder
- ms-winsoundevent:Notification.SMS
- ms-winsoundevent:Notification.Looping.Alarm[*n*] where *n* is an optional number from 2 to 10. These sounds are clearly used with alarms.
- ms-winsoundevent:Notification.Looping.Call[*n*] where *n* is an optional number from 2 to 10. These sounds are clearly used with incoming calls.

Separately, the audio.silent attribute controls whether audio plays at all (false, the default) or is muted (true). If the toast.duration attribute is set and you set audio.src to one of the "Looping" sounds above, you can also set audio.loop to true (to repeat the sound) or false (to play the sound only once, the default).

Scenario 4 in the Toast notifications sample lets you play with the different notification sounds—different buttons choose different sounds. The text of each button (in a variable named toastSoundSource) is appended to the ms-winsoundevent:Notification., as in this XML used to create the notification:

```
"<audio src='ms-winsoundevent:Notification." + toastSoundSource + "'/>"
```

Scenario 6 shows the use of the loop attribute in the XML as well (but not all the sounds):

```
"<audio loop='true' src='ms-winsoundevent:Notification.Looping.Alarm'/>"
```

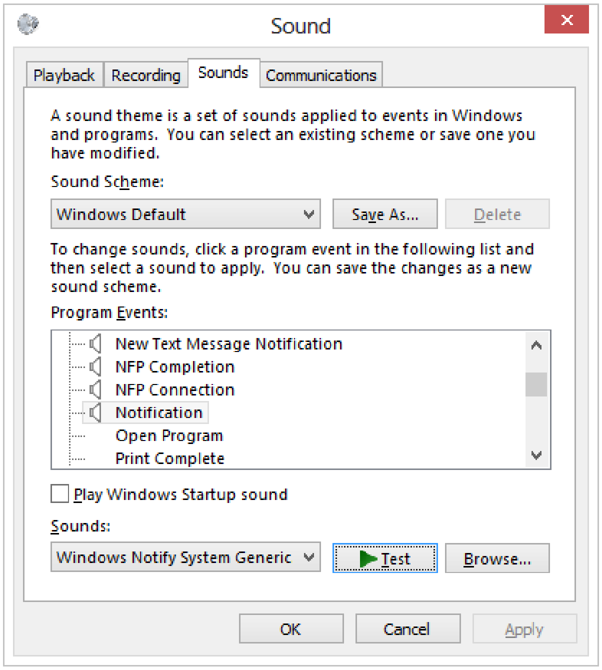
Did you actually hear any sounds? When I first ran these samples, I sure didn't! It took me a while to figure out why, so let me save you the trouble.

The values in the audio.src attribute simply map to various system sounds that are assigned in Control Panel > Hardware and Sounds > Change System Sounds, which displays the dialog box below. Having tired of all the beeps, boings, and dingalings that were once all the rage on personal computers, I routinely select the "No Sounds" option under Sound Scheme. As a result, there were no sounds assigned to anything in the Program Events list, so there were no sounds whatsoever for toast

[117] With all the catalogs we've seen in this chapter, it feels like we've been shopping! More seriously, the fact that you must use audio from this list means that custom audio is not supported.

notifications. When I selected the Windows Default scheme, I then heard sounds with the toasts.

In short, the user does have ultimate control over the sounds, both generally in PC Settings and specifically in the dialog box below. So, if it's appropriate to use a sound at all, just choose the one that's closest to the nature of your toast and leave it at that.



Sound

A sound theme is a set of sounds applied to events in Windows and programs. You can select an existing scheme or save one you have modified.

Sound Scheme: Windows Default

To change sounds, click a program event in the following list and then select a sound to apply. You can save the changes as a new sound scheme.

Program Events:

New Text Message Notification
NFP Completion
NFP Connection
Notification
Open Program
Print Complete

☐ Play Windows Startup sound

Sounds: Windows Notify System Generic

Tea Time: Scheduled Toasts and Alarms

Issuing toasts from a running app is all well and good, but it's not actually a common scenario because the user is already looking at that same app. What's more interesting are cases where the app isn't necessarily running when a notification appears. This is why toasts are often used with push notifications and background tasks, as we'll see in the last two sections of this chapter, but the other means is a scheduled toast that will simply appear at some later time regardless of whether the app is running, which is especially important for alarms apps. Scheduled toasts are a great way to invite the user to activate the app again.

A basic scheduled toast (we'll come back to the special case of alarms) is created using the `ScheduledToastNotification` class instead of the usual `ToastNotification`. There are two forms of scheduled notification, as indicated by its pair of constructors:

- `ScheduledToastNotification(content, deliveryTime)` Creates a one-time scheduled toast with the toast's `XmlDocument` in `content` and the UTC `DateTime` when it should appear in `deliveryTime`.

- `ScheduledToastNotification(content, deliveryTime, snoozeInterval, maximumSnoozeCount)` Creates a recurring scheduled toast whose content will appear at `deliveryTime`. If the toast is dismissed either explicitly or by letting it disappear on its own, it continues to appear a total of `maximumSnoozeCount` times at intervals defined by the number of milliseconds in `snoozeInterval`. The `snoozeInterval` must be set between 60 seconds and 60 minutes; for longer intervals it's best to just schedule separate toasts altogether.

A `ScheduledToastNotification` also has an `id` property, a maximum 16-character string that's used to identify that toast. If you schedule a toast with the same `id` as an existing one, the new one will replace the old.

In all cases, the toast is scheduled by calling the `ToastUpdater.addToSchedule` method passing in the notification object. Here's the process in code, as found scenario 1 of the Scheduled notifications sample (js/scenario1.js), where `toastDOM` is the `XmlDocument` containing the content and `dueTime` is determined by a UI control in the sample. First, for a one-time notification:

```
var Notifications = Windows.UI.Notifications;

toast = new Notifications.ScheduledToastNotification(toastDOM, dueTime);
Notifications.ToastNotificationManager.createToastNotifier().addToSchedule(toast);
```

Second, for a notification that will repeat five times at 60-second intervals (the option that's exercised if you check the Repeat checkbox in the sample's UI):

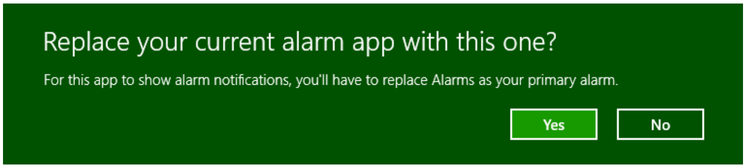
```
var Notifications = Windows.UI.Notifications;

toast = new Notifications.ScheduledToastNotification(toastDOM, dueTime, 60 * 1000, 5);
Notifications.ToastNotificationManager.createToastNotifier().addToSchedule(toast);
```

To enumerate currently scheduled toasts, call `ToastNotifier.getScheduledToastNotifications`. This returns a vector of `ScheduledToastNotification` objects, any of which can be canceled through `ToastNotifier.removeFromSchedule`. These methods are demonstrated in scenario 2 of the Scheduled notifications sample that I will leave you to examine more closely. Also, there are some debugging tips on the Guidelines for scheduled notifications topic in the documentation, mostly to note that the system has a limit of 4096 total notifications and to make sure you've set Toast Capable in the manifest to Yes.

Let's talk about alarms now. If you refer back to Figure 16-8 (PC Settings > PC and Devices > Lock Screen), you'll see an option at the bottom under Lock Screen Apps that says Choose An App to Show Alarms. This indicates that there is one app on the system that has the privilege to schedule alarms notifications and also have them appear on the lock screen, just like one calendar app can display detailed status. (That said, any alarms app that isn't chosen for the lock screen can still issue scheduled toasts while it's running and have them pop up more or less when they're supposed to.)

To be a privileged alarms app, you must meet the following requirements:

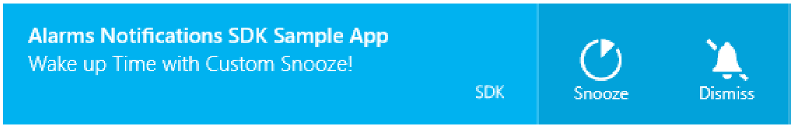
- The app is toast capable (of course!) and declares a lock screen notification in its manifest.
- The app declares a background task of along with the `windows.alarms` extension in its manifest (making it lock screen capable).
- The app calls [`Windows.ApplicationModel.Background.AlarmApplicationManager.requestAccessAsync`](), which prompts the user for consent to make this app the alarms app as shown below. Consent is necessary for alarms to appear on the lock screen; without it, scheduled toasts will still work as with any other app.



Replace your current alarm app with this one?

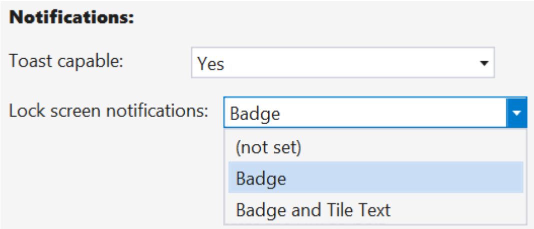
For this app to show alarm notifications, you'll have to replace Alarms as your primary alarm.

- The app then schedules toasts as always for its alarms, and can also include Snooze and Dismiss commands in the toast XML payload. These will appear with the toast only if the app is the one selected for the lock screen:



Alarms Notifications SDK Sample App
Wake up Time with Custom Snooze!
SDK

The Alarm toast notification sample in the SDK, from which the above images were taken, provides a simple demonstration of all these requirements. First is the declaration of toast capability and lock screen notification style on the manifest editor's Application tab:



Notifications:

Toast capable: Yes

Lock screen notifications: Badge

For the lock screen, of course, you'll also need to provide badge graphics on the Visual Assets tab.

Next, on the Declarations tab of the manifest, add a Background Task declaration and check one of the task types like Timer (the sample also has Audio checked, but this isn't necessary.) You also have to put *something* in the Entry Point field for the background task just to make the manifest XML schema happy. The sample actually includes a real entry point to a real file js/backgroundtask.js for this, but that code will never be invoked. You can just write "not used" and the app will still work fine.

What's most important in the manifest is an extension with the category `windows.alarms`, which you must add in the XML. Altogether, then, the applicable XML within the `Application` element looks like this, using Timer as the most benign background task type:

```
<Extensions>
  <Extension Category="windows.backgroundTasks" StartPage="not used" >
    <BackgroundTasks>
      <Task Type="timer" />
    </BackgroundTasks>
  </Extension>
  <m2:Extension Category="windows.alarm" />
</Extensions>
```

With all this in place, we ask to be the alarms app with one call (js/toast.js):

```
Windows.ApplicationModel.Background.AlarmApplicationManager.requestAccessAsync().done(
    function (status) {
    });
```

The result of `requestAccessAsync` is an `AlarmAccessStatus` value that indicates whether the user granted consent. The values are `denied`, `allowedWithWakeupCapability`, `allowedWithoutWakeupCapability`, and `unspecified` (status isn't known yet). The request can also be denied if the user has indicated through PC Settings to not allow alarms at all on the lock screen.

If at any time you want to check whether you are the privileged alarms app, the `getStatusAsync` method will return the current `AlarmAccessStatus` value.

Beyond this, it's now just a matter of scheduling toasts as before, except that you can include a `commands` element within the XML payload alongside the `visual` element. Under `commands` you can include one or more `command` elements for snooze or dismiss features. Here's the full payload as the sample builds in js/toast.js, shown as XML rather than the JavaScript string used to build it:

```
<toast duration="long">
  <visual>
    <binding template="ToastText02">
      <text id="1">Alarms Notifications SDK Sample App</text>
      <text id="2">Wake up Time with Default Snooze!</text>
    </binding>
  </visual>
  <commands scenario="alarm">
    <command id="snooze"/>
    <command id="dismiss"/>
  </commands>
  <audio src="ms-winsoundevent:Notification.Looping.Alarm2" loop="true" />
</toast>
```

Toast Events and Activation

As far as toasts are concerned, we have perhaps saved the best topic for last! The whole purpose of a toast is to get the user's attention and have them activate your app to take some kind of action. An app will commonly navigate to an appropriate page for whatever the content of the toast implies.

The most straightforward case of activation is with a scheduled toast, one that has been put up by a background task, or one that's come through a push notification. In all of these cases the app won't be running, so Windows will start it with the activation kind of launch, where the value of the payloads toast.launch attribute will be in the activated event's eventArgs.detail.arguments property. This is, once again, identical to the way secondary tiles work and you can process the arguments value however you wish.

If the app is not running when the toast is activated, it will still be launched even if the toast.launch attribute is empty. That is, toasts that occur under nonrunning conditions can be used to just launch the app, if desired. On the other hand, if a *running* app issues a toast with no toast.launch value, Windows will bring the app to the foreground but the app's activated event will *not* be fired at all. This is a way of saying that activation through a toast with no additional information would never cause the app to navigate in the first place, so what's the point of firing the activated event? None whatsoever. Thus, if a running app issues a toast with the intent that activating that toast will switch to a different part of the app, rather than just bringing the app to the foreground, a launch value is essential. (Of all the scenarios in the Toast notifications sample, only scenario 5 provides a launch value; set a breakpoint in the activated event of js/default.js, and you'll see that scenario 5 is the only time that event will fire when you tap a toast.)

Still, a running app might want to know when the user interacts with a toast, launch arguments aside. For this purpose it can listen to a ToastNotification object's activated, dismissed, and failed events, a few of which are demonstrated in scenario 5 of the sample. The activated event has no specific eventArgs, but dismissed comes with a ToastDismissalReason in eventArgs.reason (values are userCanceled, applicationHidden, and timedOut), and the failed event comes with an error code in eventArgs.errorCode. (These are all events from a WinRT object, so be sure to manage them with removeEventListener as appropriate. See "WinRT Events and removeEventListener" in Chapter 3.)

Note that a ScheduledToastNotification does not support any of these events because the assumption is that the app probably won't be running by that time anyway.

Push Notifications and the Windows Push Notification Service

We've now finally arrived in this chapter where we can leave running apps behind and look at more of the fun things that can happen behind the scenes. Earlier, in "Periodic Updates," we learned that the shortest interval you can use with that approach is pretty darn long by a computer's reckoning: 30 minutes. That's even long by many human standards, especially those of a user who really wants to know what's happening with whatever information source your app is connected to.

To update a tile, set a badge, or issue a toast as quickly as the system allows, and to personalize the content (as with calendar reminders and email alerts), it's necessary to be a little pushy and use *push notifications*. These are notifications that come to a system from an outside agent, typically a service that is monitoring some other source of information and detects conditions for which notifications are appropriate. We saw the mechanism in "The Four Sources of Updates and Notifications" and Figure 16-12 early in this chapter. To summarize:

- When launched, an app requests a channel URI for each of its live tiles and then sends those URIs to its associated backend service. An app should do this each time it's launched, as the expiration period for a WNS channel is 30 days and use of the app at least once a month is a good indicator that the service should maintain that channel. Each channel URI is unique for the combination of user, tile, and device.
- The service stores the channel URI and associates it with a user to customize the user's notification content (as again with email and calendar alerts, notifications from friends' activities, etc.).
- When needed, the service issues updates (XML payloads) to that channel.
- WNS then sends the notification to the client devices where the app acquired the channel URI. Those notifications can update tiles, update badges, issue toasts, and update the lock screen (with appropriate lock screen apps and background tasks).

It's also possible for the service and WNS to send a *raw notification*, which can contain any payload you want: there just needs to be someone listening as Windows won't know what to do with the data. A foreground app can listen through the `PushNotificationChannel.onpushnotificationreceived` event; a lock screen app can listen with a background task. In the latter case, raw notifications are generally used to deliver information to the background task that then issues other notifications in response and/or updates app data.

Design tip You've probably fallen victim of apps that display interesting content on their tiles or in toasts, only to activate the app and find that the content isn't yet available. The Windows News app is guilty of this: it sends new headlines from its backend through push notifications before the app can acquire the story. This disconnect between the notifications and the app content results in a poor user experience. To avoid this, use a raw notification to instruct a background task to acquire the content and store it in app data, and once that's done, have the *background task* issue the toast or tile update such that activating either one will navigate the app to that specific content. The Sports app, in my observation, does a much better job of this with its tile.

Before you do anything in your app for push notifications, you must follow the instructions on How to authenticate with the Windows Push Notification Service (WNS) on the Windows Developer Center (which is part of a whole series on Push notifications). This will walk you through the steps on the Windows Store Dashboard to obtain a Package Security Identifier (SID) and a client secret with which your app's web service authenticates itself with WNS.

In the sections that follow we'll first go through each of the steps in turn, using scenarios 1–3 of the same Push and periodic notifications client-side sample we used earlier for periodic updates. This will illustrate the nature of the work that's needed to make push notifications work. Then we'll take a look at how Windows Azure Mobile Services (AMS) and other third-party offerings can do most of the work for you—I suspect that you'll definitely want to employ such tools!

> **Tip** To use the SDK sample to test push notifications, create an app for it in your Store account (don't worry, you won't publish it), and then use the Store > Associate App With The Store command in Visual Studio to set its identity. You'll also need the SID and client secret from the Store dashboard so that your backend (including the example service we'll be using) can authenticate with WNS.

Let me also point out that because channel URIs are unique for an app+user+device, using push notifications can become an expensive proposition for your backend, which must record and maintain a channel for every unique tile on every user's device and then figure out when to send which notifications to which channels. If your app becomes popular, this will require scaling up your service to potentially manage thousands or even millions of channel URIs. A feature of AMS called *notification hubs* helps with this, and third-party services do as well. Still, it's worth taking the time to seriously evaluate whether periodic notifications would be sufficient for your scenario, especially for updates that aren't user-specific, because they will be much simpler on the server side of the picture and less expensive to sustain and maintain.

Requesting and Caching a Channel URI (App)

Requesting a channel URI is done through the PushNotificationChannelManager object. This manager has only two methods: createPushNotificationChannelForApplicationAsync and createPushNotificationChannelForSecondaryTileAsync. The first is clearly linked to the app tile as well as toast notifications; the second is clearly for use with secondary tiles and takes a tileId argument to identify the specific one.

The result of both async operations is a PushNotificationChannel object that will be passed to your completed handler, as shown in scenario 1 of the sample (start in js/scenario1.js, then go into js/notifications.js):

```
var channelOperation;

// Channel for the app tile
if (isPrimaryTile) {
    channelOperation = Windows.Networking.PushNotifications.PushNotificationChannelManager
        .createPushNotificationChannelForApplicationAsync();
} else {
    // Channel for a secondary tile
    channelOperation = Windows.Networking.PushNotifications.PushNotificationChannelManager
        .createPushNotificationChannelForSecondaryTileAsync(itemId);
}

channelOperation.done(function (newChannel) {
    // Send channel to service
}, /* error handler */
);
```

The PushNotificationChannel object (newChannel in the code above) is a simple object with just a few members, but they are important ones:

- expirationTime   A read-only property indicating when the channel expires—notifications sent to this channel after expiration will be rejected. Apps must be sure to refresh their channels when needed to avoid an interruption in notifications.
- uri   A read-only URI to which the backend service sends notifications to WNS.
- close   A method that explicitly invalidates the channel.
- pushnotificationreceived   An event that's fired when a push notification is received on the client device from this notification channel. This will be fired only for apps that are in the foreground and includes raw notifications as well as tile, badge, and toast payloads.

Your app should go through this short process to obtain the necessary channel URIs whenever it's launched *and* when it's resumed (especially if any channel's expirationTime has passed). It's unlikely that an app would stay suspended for that long, but it's still possible! Furthermore, if you're concerned that your app might not run for more than 30 days, you can implement a background task on a maintenance trigger for this purpose. See "Tasks for Maintenance Triggers" later in this chapter and scenario 2 of the sample.

Again note that you'll have multiple channel URIs if you're using push notifications for both secondary tiles and your app tile. In this case you'll be managing separate channel URIs for each tile. Also, save the channel URI for each tile in your local app state. This is so that you can check on subsequent runs if the URI is the same as one you've already obtained and sent to your service, and thus avoid unnecessary network traffic.

Sending the URI to your service can be done with a simple HTTP POST, as in the sample (inside channelOperation.done). Here we also see the checks for whether the URI is the same as before:

```
channelOperation.done(function (newChannel) {
    // _urls[] is an array of channel ids for primary and secondary tiles
    var tileData = that._urls[itemId];

    // Upload the channel URI if the client hasn't recorded sending the same
    // uri to the server
    if (tileData && newChannel.uri === tileData.channelUri) {
        // This saves the URI to local app data
        that._updateUrl(url, newChannel.uri, itemId, isPrimaryTile);
        completed(newChannel);
    } else {
        WinJS.xhr({
            type: "POST",
            url: url,  // Best practice: use https URIs
            headers: { "Content-Type": "application/x-www-form-urlencoded" },
            data: "channelUri=" + encodeURIComponent(newChannel.uri) +
                "&itemId=" + encodeURIComponent(itemId)
        }).done(function (request) {
            // Only update the data on the client if uploading the channel URI succeeds.
            // If it fails, you may consider setting another background task, trying
            // again, etc. (An exception will be thrown if it fails, ending up in the
            // error hander instead.)
            that._updateUrl(url, newChannel.uri, itemId, isPrimaryTile);
            completed(newChannel);
        }, failed);
    }
}, failed);
```

Managing Channel URIs (Service)

If you use the code in the previous section, your service (the one that generates push notifications) will receive HTTP POST requests with unique channel URIs for each and every tile. This isn't the only way to transport channel URIs, of course; in fact, because a channel URI might be used to transmit personal information through notifications, it should ideally be encrypted with a private key before it's sent to the server, and even sent over HTTPS. Otherwise someone could possibly intercept that URI and use it to redirect user-specific notifications.

In any case, the service must expect to receive—and then manage—a unique URI for each app/user/device combination. This underscores the fact that push notifications are best used for user-specific notifications rather than broadcast notifications. In the latter case, setting up a service for periodic updates is again a much easier solution.

Once the service receives a channel URI along with any data to identify the user and the purpose of the channel, it should securely save that information in persistent storage of some kind, such as a SQL Server database (for an ASP.NET service or Azure Mobile Service) or a MySQL (for a PHP service).

It's important that the service also removes obsolete channel URIs. If it receives a new URI for the same user and the same purpose, it should replace the old with the new. It should also remove any URIs if it receives an HTTP 404 or 410 error back from WNS, indicating an obsolete channel.

A simple ASP.NET service page that can receive a post from scenario 1 of the Push and periodic notifications client-side sample is the *receiveuri.aspx* page found in the HelloTiles website project in this chapter's companion content. To run this service, make sure you have the localhost established, as described earlier in "Debugging a Service Using the Localhost." You may also need to install ASP.NET on your localhost. An easy way to do this is to obtain the Background transfer sample, go into its Server folder, and then from an administrator command prompt run **powershell -ExecutionPolicy unrestricted -file serversetup.ps1**. If you then run the site in Visual Studio Express for Web as we did before, you'll have a localhost port for the service like *http://localhost:52568/HelloTiles/receiveuri.aspx*.

You can then set a breakpoint in the service code, paste the service URI into scenario 1 of the Push notifications sample, and press its Reopen Channel And Send To Server button. This should hit the breakpoint in the service and allow you to step through the code that processes the request. Here you'll find that the request contains *channelUri* and *itemId* values (along with *LOGON_USER*), which can be saved for when the service needs to send a notification to WNS.

Note that the receiveuri.aspx example page saves the URI into a text file with a hardcoded name (see its SaveChannel method), meaning that it's good for just one channel URI! If you use this example as a basis for your own service, be sure to replace that code with something that can handle any number of URIs (or use tools so that you don't have to do all the work manually).

Sending Updates and Notifications (Service)

Before a service can send updates, it must authenticate itself with WNS by sending the Package Security Identifier (SID) and client secret as obtained through the Windows Store Dashboard. This is a matter of sending a request to WNS (via HTTPS) that looks like this:

```
POST /accesstoken.srf HTTP/1.1
Content-Type: application/x-www-form-urlencoded
Host: https://login.live.com
Content-Length: 211
grant_type=client_credentials&client_id=ms-app%3a%2f%2fS-1-15-2-2972962901-2322836549-
3722629029-1345238579-3987825745-2155616079-
650196962&client_secret=Vex8L9WOFZuj95euaLrvSH7XyoDhLJc7&scope=notify.windows.com
```

where you must make sure the values of client_id and client_secret match the package SID and client secret. If the authentication works, you'll receive a 200 OK response with the access token you need for sending notifications:

```
HTTP/1.1 200 OK
Cache-Control: no-store
Content-Length: 422
Content-Type: application/json
{
    "access_token":"EgAcAQMAAAAALYAAY/c+Huwi3Fv4Ck10UrKNmtxRO6Njk2MgA=",
    "token_type":"bearer"
}
```

Tip For everything that can go into a request to WNS and come back in the response, see Push notification service request and response headers. This includes setting an expiration time, setting cache policy, and tagging tile payloads.

Code that accomplishes these steps for a service written in C# can be found on How to authenticate with the Windows Push Notification Service, where your service would use the GetAccessToken method shown there to obtain an OAuthToken object with the information from the response. Services written in other languages will obviously need to use the appropriate means to send the request and receive the response.

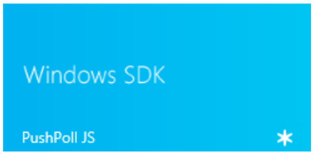
Whatever the case, once you have the access token, you're ready to start sending updates and notifications via HTTP requests to the channel URIs maintained by the service.

For tile updates, badge updates, and toast notifications, sending a notification means generating the XML payload as for any other update or notification, and then sending it to WNS with the previously acquired access token. The only real difference between these requests, besides the specific XML, is the value of `X-WNS-Type` in the request header: `wns/badge`, `wns/tile`, `wns/toast`, or `wns/raw` (see the next section). Otherwise the code is the same.

Generic code for a C# service can be found on Quickstart: Sending a push notification. I've included a badge update version in *sendBadgeToWNS.aspx* in the HelloTiles example site with this chapter, where the SID and client secret are old ones I once obtained for my copy of the Push notifications sample. You'll need to change these for your copy of the sample that you've registered with the Store.

To test this now, run the HelloTiles website in Visual Studio Express for Web and set a breakpoint at the beginning of sendBadgeToWNS.aspx. Assuming you're running scenario 1 of the Push notifications sample in Visual Studio Express for Windows to upload a channel URI to receiveuri.aspx, there should be a file called *channeluri_aspx.txt* in the website project that contains the uploaded data.

Now switch to scenario 3 of the Push Notifications sample, and press the button to start listening to the `pushnotificationreceived` event. In js/scenario3.js of that sample, set a breakpoint within the `pushNotificationReceivedHandler` function. With all this in place, open a browser and enter the address of sendBadgeToWNS.aspx on the Localhost, such as: *http://localhost:52568/HelloTiles/sendBadgeToWNS.aspx*. This should hit the breakpoint in Visual Studio Express for Web, where you can walk through that page's code and see it loading the channel URI from channeluri_aspx.txt, to which it then sends a badge update. When that happens, you should hit the breakpoint in the Push notifications app where you can walk through that code. Note that when you get to the line `e.cancel = true`, skip over it—right-click the line below it and select Set Next Statement—or just make sure the value is `false` before your return. This will allow Windows to process the notification and update the badge for the Push notifications sample, which should now look like the following with a * (alert) badge on the lower right:



Note again that if you receive an HTTP 404 or 410 error back from WNS, the channel URI is no longer valid and you should remove it from your list. It's also good for the app to notify your service whenever it no longer needs updates for a particular channel (no point in paying for unproductive bandwidth). And if WNS returns an error, avoid posting the update again unless it makes sense for your scenario.

404 or 410 errors are different, by the way, from an inability to deliver a notification because the client is offline. In this case WNS will cache the tile, badge, or raw notification until the client reconnects. In other words, it's not a condition that your service has to worry about. Send notifications as you always would, and let WNS handle the delivery details.

Raw Notifications (Service)

If you use `wns/raw` as the push notification type, the payload included with the push notification can be anything you want, not just XML, as long as it's under 5KB (see Guidelines for raw notifications). Of course, Windows cannot do anything with this payload directly, so an app has to provide a handler for receipt of the notification, as we're about to see.

Receiving Notifications (App)

A running app receives push notifications through the `PushNotificationChannel.onpushnotificationreceived` event. This is again required to process `wns/raw` payloads but can be used for any type. If the app is not running, of course, it won't receive this event. Instead, the app must be on the lock screen with a `PushNotificationTrigger` background task for this purpose. (See "Lock Screen Dependent Tasks and Triggers" later in this chapter.) That piece of code will then receive the XML payload, process it, and issue whatever tile/badge updates or toast notifications are necessary. Besides saving some state to the app data folders, this is really all that the background task can do, but it's enough to keep that sense of aliveness going as well as invite the user to launch the app in response.

In the running app, the `pushnotificationreceived` event is fired for the other notification types as well. Scenario 3 of the Push notifications sample shows this in its event handler—I've modified this code a little bit for simplicity:

```
function startListening() {
    // Assume channel has been obtained and validated
    channel.addEventListener("pushnotificationreceived", pushNotificationReceivedHandler);
}

function pushNotificationReceivedHandler(e) {
    // Extract notification payload for each notification type
    var notificationPayload;
    switch (e.notificationType) {
        case pushNotifications.PushNotificationType.toast:
            notificationPayload = e.toastNotification.content.getXml();
            break;

        case pushNotifications.PushNotificationType.tile:
            notificationPayload = e.tileNotification.content.getXml();
            break;

        case pushNotifications.PushNotificationType.badge:
            notificationPayload = e.badgeNotification.content.getXml();
            break;

        case pushNotifications.PushNotificationType.raw:
            notificationPayload = e.rawNotification.content;
            break;
    }

    // Process the notification: set e.cancel to true to suppress automatic handling.
}
```

The last bit in the comment above is important. When you receive this event in a running app, it wouldn't be necessary to display a toast unless it pertains to some other part of the app that isn't visible. For example, if you have an app that handles both email and a calendar, you might want to show email toasts when the user is looking at the calendar and calendar toasts when the user is looking at email. In this case, setting `e.cancel` to `true` will suppress the toast.

With the `pushnotificationreceived` event, the running app gets first crack at raw notifications. If the app doesn't process it, the notification will be sent to any lock screen background task configured for the `PushNotificationTrigger`. In either case, refer to the Raw notifications sample for details.

Debugging Tips

When using push notifications, experience shows that if notifications aren't getting through, it's typically not a problem with WNS. Here's a list of things to check (thanks to Hans Andersen):

- Check the return status of your HTTP POSTs to WNS. If it's returning an HTTP 200 response, check the `X-WNS-NotificationStatus` and other headers you get back. Look particularly for the status of "Received," which indicates that a notification has gone to the client.
- Lacking anything conclusive in the headers, run Event Viewer and check the events under *Application And Services Logs* > *Microsoft* > *Windows* > *Push Notifications Platform* > *Operational* to see the activity.
- Also look under *Application And Services Logs* > *Microsoft* > *Windows* > *Immersive-Shell* > *Microsoft-Windows-TWinUI* > *Operational* to see if there are error messages related to XML parsing about the same time you expected to receive a notification.
- Even if the XML is well-formed, an update might not show up if a referenced image is either too large (pixel dimensions or file size), if the image is the wrong format (for example, TIF), if the image is corrupt, if the server handling the image request can't handle the query parameters for the tile (scaling, contrast, language), or if the server is encountering other errors as might be revealed in its own logs.
- If updates appear but after a considerable delay, it could just mean internal timeouts or other network latency within the tile and notification infrastructure. If this happens, just accept that the world isn't always perfect and operations must sometimes be retried!

Tools and Providers for Push Notifications

Now that we've gone through the intricate details of working with push notifications, even omitting the question of storage, you're probably wondering, "Is there any way to make all this simpler?" Just imagine what it would take to manage potentially thousands or millions of channel URIs for a hopefully large and expanding customer base!

Fortunately, you're not the first to ask such questions. First of all, a number of third parties provide solutions for push notifications, which you can find listed on http://services.windowsstore.com/ and filtering on the "Push Notifications" service type. There you'll see providers such as Urban Airship, Push IO, and Parse, all of which I can recommend with confidence. You can also employ Windows Azure

Mobile Services (AMS) and its JavaScript client library for this purpose as described on Get started with push notifications in Mobile Services. Here's how it fulfills all the requirements:

- **App Registration with the Windows Store** Once you obtain the client secret and SID for your app from the Windows Store, go to the Push section of the mobile service and save those value under Windows Application Credentials.
- **Obtaining and Refreshing Channel URIs** Requesting and managing channel URIs in the app is purely a client-side concern and is the same as before.
- **Sending Channel URIs to the Service** This step becomes far easier with AMS. First you create a table in the mobile service under the Data section. Using the AMS client library, the app can then simply insert records with channel URIs and any other key information you need to attach. The client library takes care of the HTTP request behind the scenes and even updates the client-side record with any changes made on the server. Furthermore, AMS can automatically handle authentication through the user's Microsoft Account or through three other OAuth providers—Facebook, Twitter, or Google—if you've registered your app with one of them. See Get started with authentication in Mobile Services.
- **Sending the Notification** Within the mobile service, you can attach scripts (written in node.js) to database operations, and also create scheduled jobs. In these scripts, a simple call to the *push.wns* object with a channel URI and a payload generates the necessary HTTP request to the channel. It's also a simple matter to capture push failures and record the response with *console.log*. Those logs are easily reviewed on the Windows Azure portal.

For extensive details, see these two sample tutorials: Tile, Toast, and Badge Push Notifications using Windows Azure Mobile Services and Raw Notifications using Windows Azure Mobile Services. Also refer to my post on the Windows Developer Blog, Alive with Activity, Part 3: Push notifications and Windows Azure Mobile Services where I show the relevant highlights. The short of it is that you create an instance of the *MobileServiceClient* object in the AMS client library:

```
var mobileService = new Microsoft.WindowsAzure.MobileServices.MobileServiceClient(
    "https://{mobile-service-url}.azure-mobile.net/",
    "{mobile-service-key}");
```

This class encapsulates all the HTTP communication with the service, relieving you from all that low-level plumbing that you probably don't want to think about. For example, instead of sending a channel URI and other to the service, you can just store it in a data table. In the mobile service, a node.js script can pick up that insertion and then send a push notification in response.

In my post I also elaborate on how you might use push notifications in different real-world scenarios, such as using social networks, issuing updates and alerts for news and weather, and messaging. The basic idea in all of them relates back to Figure 16-15 at the end of "Periodic Updates," where you use a central database to gather information for your update services. With push notifications, you can use database inserts or other modifications as triggers to issue push notifications to the appropriate channels.

Background Tasks and Lock Screen Apps

At the end of the introduction to this chapter, I described how everything we've talked about so far helps to "create an environment that is constantly alive with activity while those apps are often not actually running or are *allowed to run just a little bit*." That last phrase—being allowed to run just a little bit through background tasks—is our last topic, and it relates to the lock screen because apps that are allowed to work behind the lock screen employ background tasks to do so.

Let me reiterate that we've already seen many scenarios in which users can experience app-related activity without apps having to run. Periodic tile updates, push notifications, scheduled toasts, and even sharing data through the Share contract all provide for activity when an app is suspended or not running. Apps can also configure background data transfers to occur while the app isn't running, as we've seen in Chapter 4. And background audio provides for that specific class of apps that need to continue running to maintain VoIP sessions, audio playback, online meetings, device firmware updates, and so forth.

What's left in the story are those little pieces of app code that Windows can run in response to specific *triggers*. Triggers in some cases can be further refined with optional *conditions* so that the background task runs only when it really needs to, thereby minimizing needless background activity that drain a device's battery. Some types of triggers, in fact, require that the user has placed the app on the lock screen to specifically limit the number of apps that can respond to those triggers. Windows also limits the amount of CPU time that background tasks can consume:

- **Lock screen background tasks** Two seconds of cumulative CPU time per 15 minutes.
- **Other background tasks** One second of cumulative CPU time every two hours.

Consumption of network bandwidth is also limited on battery power. What that limit is, exactly, I cannot say, because the system analyzes energy usage more so than bytes transferred. A means of estimating the limit can be found on Supporting your app with background tasks. (While we're at it, you might also be interested in Guidelines and checklists for background tasks, the Introduction to background tasks whitepaper, and Being productive in the background – background tasks on the Windows Developer Blog.)

"Whoa!" you're probably saying, "Is Windows really that restrictive?"

The short answer is yes, because Windows wants to save battery power for the foreground app with which a user is engaged and to help the foreground app deliver the best user experience. This means it isn't competing with background apps that would, if allowed, take up as many resources as they possibly can (like background services on the desktop seem to do!).

It's likely that you've experienced situations like this directly, where you've started an app but it takes for-EV-er to get going because some other dark and mysterious service is chewing on the hard drive, pounding the network, flaring up the CPU, and so forth. I, for one, have dug through Task Manager and the Resource Monitor to figure out—and kill off—whatever process is pouring molasses on my system, let come what will. This is the kind of user experience that Microsoft is trying to avoid with Windows Store apps.

"OK," you say (assuming that I've actually convinced you to some small degree), "does that mean that there isn't any way to do some background work like indexing data, creating picture thumbnails, processing video, and so on?"

Actually, there are ways to do this. For one, when an app is visible (and thus in for foreground, though it might not have the focus), it can do however much it wants of all these things because it's ultimately responsible for its own user experience.

Second, when a device is on AC power instead of battery, Windows allows apps to run background tasks in response to *maintenance triggers* on 15-minute or longer intervals (whatever is appropriate for the app). These are still limited in the total amount of CPU time they can consume, but tasks that don't involve UI—and background tasks are not allowed to alter UI—can burn through a few billion instructions in one or two seconds on a gigahertz CPU!

What we have in this whole story, then, are three distinct classes of background tasks and their associated triggers:

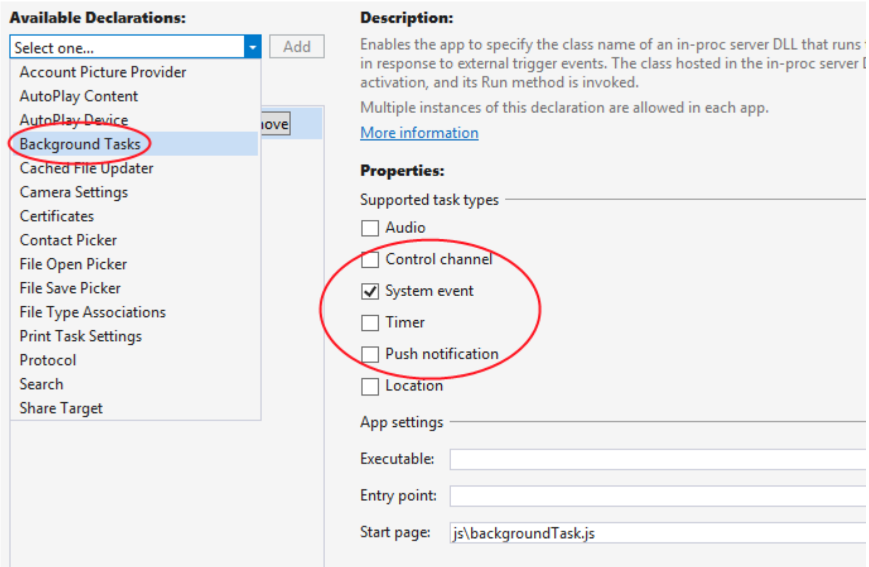
- Tasks for maintenance triggers that run on AC power only.
- Tasks for potentially conditioned system triggers that run on AC or battery and don't require being on the lock screen.
- Tasks for those privileged apps that the user has added to the lock screen.

We'll look at each of these in detail, but let's first examine a few aspects that all of them share: declaring background tasks in the manifest, the general process of building the task with the WinRT API, and applying conditions.

Lock screen personalization Configuring the lock screen image or slide show is not dependent on background tasks. This is discussed in Chapter 4 in "The User Profile (and Lock Screen Image)" and demonstrated through the Lock screen personalization sample.

Background Tasks in the Manifest

All background tasks for an app are declared in the manifest, where each declaration indicates the type of task as well as the code to execute for that task, as shown in Figure 16-18. We've seen this section of the manifest before in Chapter 12 and Chapter 13 where we checked Location and Audio, respectively, for background geolocation monitoring and background audio, respectively. As for the other options, System Event is used for maintenance and non–lock screen triggers; Control Channel, Timer, and Push Notification are used with lock screen apps.



Available Declarations:

Select one... ▾ Add

- Account Picture Provider
- AutoPlay Content
- AutoPlay Device
- Background Tasks
- Cached File Updater
- Camera Settings
- Certificates
- Contact Picker
- File Open Picker
- File Save Picker
- File Type Associations
- Print Task Settings
- Protocol
- Search
- Share Target

Description:

Enables the app to specify the class name of an in-proc server DLL that runs in response to external trigger events. The class hosted in the in-proc server activation, and its Run method is invoked.

Multiple instances of this declaration are allowed in each app.

More information

Properties:

Supported task types

☐ Audio
☐ Control channel
☑ System event
☐ Timer
☐ Push notification
☐ Location

App settings

Executable:

Entry point:

Start page: js\backgroundTask.js

FIGURE 16-18 The manifest editor for declaring background tasks, showing the option for Background Tasks in the drop-down list of Available Declarations (left), the background task types (center), and the Start Page field to indicate the JavaScript code to run for the task (bottom). Background tasks can also be written in other languages, in which case the Executable and Entry Point fields are used.

In all cases, the Start Page field is where you indicate the JavaScript file to execute for the task, but do note that because background tasks execute independently of the app itself, sharing state only through app data, you can choose whatever language you want. Given the quotas on CPU time, writing a background task in a language like C++ or C# will allow you to do some tasks more efficiently, in which case you'll use the Entry Point field (if the task is in a DLL in the package) and perhaps the Executable field (if the task is in another EXE in the package) to identify the code module and specific function to call.

It's also good to note that even though the Start Page field suggests a *page*, you always point to a piece of JavaScript that runs as a web worker, and thus no HTML or CSS can be loaded here. Indeed, issuing tile updates, badge updates, and toasts is as much UI work as a background task is allowed. For anything else, the background task must write values to app data that the main app can pick up within its handlers for background task events, as we'll see shortly.

You might also notice that triggers aren't represented in the manifest. This is done in code when you build the task, as we'll see next.

Building and Registering Background Tasks

The declaration of a background task in the manifest is only that—a declaration that tells the system that the app *intends* to use a background task. The app must still register the background task from code for it to execute at all, which is accomplished using the BackgroundTaskBuilder (whose namespace, Windows.ApplicationModel.Background, contains everything we'll be referring to in the context of background tasks). Simply said, you create an instance of the builder, set its name and taskEntryPoint properties, call its setTrigger and addCondition methods to specify exactly when the task should run, and then call register.

Generic code for this is found in the Background task sample within js/global.js. This module declares a global object BackgroundTaskSample that contains a number of properties and methods. The one that concerns us here is a method called registerBackgroundTask that registers a given entry point (the name of a JavaScript file or the name of a class in C#, Visual Basic, or C++), with a given name, and applying some trigger and condition:

```
var BackgroundTaskSample = {
    // Properties with names and entry points of the sample's tasks are omitted

    // Register a background task with the specified taskEntryPoint, taskName, trigger,
    // and condition (optional).
    "registerBackgroundTask": function (taskEntryPoint, taskName, trigger, condition) {
        var builder = new Windows.ApplicationModel.Background.BackgroundTaskBuilder();
        builder.name = taskName;
        builder.taskEntryPoint = taskEntryPoint;
        builder.setTrigger(trigger);

        if (condition !== null) {
            builder.addCondition(condition);
        }

        var task = builder.register();
        BackgroundTaskSample.attachProgressAndCompletedHandlers(task);

        // [Sample-specific code omitted]

        // Remove previous completion status from local settings.
        var settings = Windows.Storage.ApplicationData.current.localSettings;
        settings.values.remove(taskName);
    },
```

In this code, the BackgroundTaskBuilder.register method returns a BackgroundTaskRegistration object through which you manage a registered task. A registered task will have a name property and a system-assigned taskId property, the latter of which you can use to tag app data that's unique to the task. It also has an unregister method (which you would call for an obvious purpose) and two events: completed and progress. Handlers for those events are assigned in the usual manner with addEventListener, as seen within the BackgroundTaskSample.attachProgressAndCompletedHandlers function in the sample:

```
    "attachProgressAndCompletedHandlers": function (task) {
        task.addEventListener("progress",
            new BackgroundTaskSample.progressHandler(task).onProgress);
        task.addEventListener("completed",
            new BackgroundTaskSample.completeHandler(task).onCompleted);
    },
```

One of the key uses of these handlers—which are part of the running app—is to perform UI update tasks in response to data left behind by the background tasks. Those tasks themselves cannot work with UI, but they can save data to the app data areas that they share with the main app. The completed and progress events, then, are how the main app with the UI thread can pick up those events from the background task to read values from app data and do the necessary updates. The Background task sample does this in each of its scenarios.

There is also one static property, BackgroundTaskRegistration.allTasks, a MapView through which you can retrieve the BackgroundTaskRegistration object for each registered task.

It's very important to note that Windows allows you to register the same background task twice and will assign unique taskId values to both, so be careful to avoid duplicate tasks. Furthermore, background task registration will persist across app updates, so if you update your tasks be sure to check for and unregister old ones when your update is launched.

With this structure, the question now becomes: what do we provide for the triggers and the conditions? The answer is what differentiates the various kinds of background tasks.

Conditions

The specific conditions you can specify through BackgroundTaskBuilder.addCondition are instances of the SystemCondition class. A background task registered with one or more conditions—each call to addCondition is cumulative—will run only if the conditions are met. Each instance can be one of the following types as defined in the SystemConditionType enumeration:

Condition	Description
internetAvailable	Run only when the device is online.
internetNotAvailable	Run only when the device is offline.
sessionConnected	The user is logged in.
sessionDisconnected	The user is logged out (lock screen only, obviously)
userPresent	User has been recently active.
userNotPresent	User has not been active for a time.
freeNetworkAvailable	Network data is unlimited.
backgroundWorkCostNotHigh	Run only if the cost of background work is low.

Clearly, each complementary pair of these conditions is mutually exclusive: if you register a task with internetAvailable and internetNotAvailable, Windows will recognize that you never really wanted to run the task in the first place, so it will let it sit on the roadside, forever undisturbed! Otherwise, you can use these to make sure that your task is run only when needed. If you want to execute a background task that renews push notification channels, for example, there's no point in trying if there's no connectivity. (We'll see an example next in "Tasks for Maintenance Triggers.") On the other hand, if you have a background task that you want to make sure never interferes with the overall user experience, you can add the userNotPresent condition.

Note that because the sessionDisconnected condition implies that the user has logged out, it's useful only for background tasks that require the lock screen.

The backgroundWorkCostNotHigh condition relates to the static property BackgroundWorkCost.currentBackgroundWorkCost. This contains a hint of the current resource availability for background tasks that comes from the BackgroundWorkCostValue enumeration. The values are low, medium, and high, so the condition here applies when the current hint is low or medium.

Tasks for Maintenance Triggers

Background tasks that use a maintenance trigger are the most generic kind of task—they run any kind of code you want to run every now and then when the system is on AC power.[118] Such tasks are best for "checking up on something" or other activity that you want to run periodically but don't really care when. As such, maintenance triggers *aren't* appropriate for something like synchronizing data with a server because that should happen in a more timely manner and is best done with the background transfer API that we saw in Chapter 4.

A maintenance trigger—what you pass to BackgroundTaskBuilder.setTrigger—is an instance of the MaintenanceTrigger class. When creating the instance, you provide two parameters. The first is basically the refresh period you need (the freshnessTime property, in minutes), and you should use the longest period that's reasonable for your scenario; the system will always wait at least this long before first running the task. The second parameter is a flag that indicates if the task needs to be run only once (the oneShot property).

Scenario 2 of the Push and periodic notifications client-side sample demonstrates using a maintenance trigger to periodically refresh its WNS channels, as described earlier in "Requesting and Caching a Channel URI." The code here is condensed from js/scenario2.js, some of which is in an internal function called registerTask:

```
var background = Windows.ApplicationModel.Background;
var pushNotificationsTaskName = "UpdateChannels";
var maintenanceInterval = 10 * 24 * 60; // 10 days

var taskBuilder = new background.BackgroundTaskBuilder();
var trigger = new background.MaintenanceTrigger(maintenanceInterval, false);
taskBuilder.setTrigger(trigger);
taskBuilder.taskEntryPoint = "js\\backgroundTask.js";
taskBuilder.name = pushNotificationsTaskName;

var internetCondition = new
    background.SystemCondition(background.SystemConditionType.internetAvailable);
taskBuilder.addCondition(internetCondition);

taskBuilder.register();
```

[118] This is about the only API for which a clear distinction is made for battery vs. AC power; WinRT does not offer a specific API to detect the power source. What this means is that apps should design for running on the battery but assign AC-only tasks to maintenance triggers, allowing Windows to manage power on a systemwide basis.

Because the expiration period for channel URIs is 30 days, the sample creates a trigger on a recurring 10-day interval (10 days * 24 hours/day * 60 minutes/hour). It also wisely adds the internetAvailable condition because it's again pointless to attempt to renew channel URIs when there's no connectivity.

The task itself can be found in the js/backgroundTask.js file of the sample, as indicated in the taskEntryPoint property:

```
(function () {
    // Import the Notifier helper object
    importScripts("//Microsoft.WinJS.2.0/js/base.js");
    importScripts("notifications.js");

    var closeFunction = function () {
        close();
    };

    var notifier = new SampleNotifications.Notifier();
    notifier.renewAllAsync().done(closeFunction, closeFunction);
})();
```

This task code pulls in a couple of other script files using importScripts, the second of which, notifications.js, is the sample's set of helper functions for notifications where renewAllAsync refreshes the app's list of previously saved channel URIs.

Important Notice that the completed and error handlers given to the promise from renewAllAsync both go to closeFunction, which makes this mysterious call to close. What close is this? Well, it's not window.close but rather WorkerGlobalScope.close. Background tasks in an app written in JavaScript run as web workers, so the global scope within the code is WorkerGlobalScope rather than window. Calling this makes sure the independently running background task is shut down and guarantees that the resources that were allocated for the task are properly released.

Sidebar: The Task Instance and Background Task Deferrals

Within a JavaScript background task, the Windows.UI.WebUI.WebUIBackgroundTaskInstance.current property contains a WebUIBackgroundTaskInstanceRuntimeClass object with additional details about the running task: its instanceId, its associated BackgroundTaskRegistration object in the task property, a progress property in which the task can store a percentage value, a succeeded flag to indicate that the task has completed, a suspended count (when the task is suspended due to the resource quota being exceeded), and a canceled event that informs the task that the app as a whole has been terminated.

This object also provides a getDeferral method that, once again, returns a deferral object whose completed method you call when the task is complete. As always, you employ the deferral if you need to perform asynchronous operations within the background task. Just be sure to always call close when everything is finished.

Tasks for System Triggers (Non-Lock Screen)

The next class of background tasks contains those tied to a variety of system triggers, specifically instances of the SystemTrigger class. You again create the trigger object with new and pass two parameters: a SystemTriggerType value (available afterwards as the triggerType property) and a oneShot Boolean flag. The triggers that operate independently of the lock screen are described in the following table:

SystemTriggerType[119]	When Triggered and Usage Scenarios
internetAvailable	Internet becomes available. This is typically used for apps that need to start a synchronization process when connectivity is restored from an offline state. Note that this *trigger* is different from the *condition* with the same name.
lockScreenApplicationAdded	User has added the app to the lock screen, signaling that lock screen–dependent background tasks will now be executed.
lockScreenApplicationRemoved	User has removed the app from the lock screen, signaling that lock screen–dependent background tasks will no longer run.
networkStateChange	Change in network (cost, connectivity, etc.). A running app can detect the same event through NetworkInformation.onnetworkstatuschanged, and this provides a means for apps to execute a small piece of code when the app is suspended or otherwise not running. This trigger combined with the internetAvailable or internetNotAvailable *conditions* offers the app full awareness of connectivity states. For a review of connectivity, refer back to Chapter 4 and the Network status background sample.
onlineIdConnectedStateChange	The user's Microsoft account has changed. This is a relatively rare occurrence, but the trigger is essential for any app that caches any part of the Microsoft account for its own user identity.
servicingComplete	App has been updated from the Windows Store. Use this trigger to unregister obsolete background tasks and to migrate app data from one version to another as soon as the update happens. Refer to Chapter 10, "The Story of State, Part 1," for more on versioning app data.
timeZoneChange	A time zone or daylight savings time change has occurred. An app might refresh its locale settings at such a time, as well as adjust any internal timekeeping. This can be important to adjust scheduled notifications.

The Background task sample provides a few examples of these triggers. In scenario 1, js/sample-background-task-with-condition.js, we can see the use of timeZoneChange along with the userPresent condition (where BackgroundTaskSample is again a helper object in global.js):

```
BackgroundTaskSample.registerBackgroundTask(BackgroundTaskSample.sampleBackgroundTaskEntryPoint,
    BackgroundTaskSample.sampleBackgroundTaskWithConditionName,
    new Windows.ApplicationModel.Background.SystemTrigger(
        Windows.ApplicationModel.Background.SystemTriggerType.timeZoneChange, false),
    new Windows.ApplicationModel.Background.SystemCondition(
        Windows.ApplicationModel.Background.SystemConditionType.userPresent));
```

[119] There is an additional trigger called smsReceived that is only for apps provided by mobile operators.

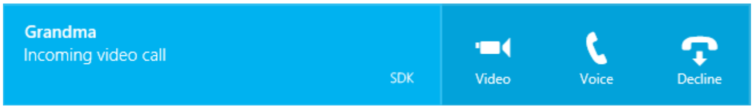
This is clearly a case where I'd use another variable to not type the Windows.ApplicationModel.Background namespace out every time, but at least you can't make a mistake in reading this code! In any case, the same sample, in scenario 4 and js/global.js, also shows use of the servicingComplete trigger within a helper function registerServicingCompleteTask, which also checks if the task is already registered:

```
"registerServicingCompleteTask": function () {
    // Check whether the servicing-complete background task is already registered.
    var iter =
        Windows.ApplicationModel.Background.BackgroundTaskRegistration.allTasks.first();
    var hascur = iter.hasCurrent;
    while (hascur) {
        var cur = iter.current.value;
        if (cur.name === BackgroundTaskSample.servicingCompleteTaskName) {
            BackgroundTaskSample.updateBackgroundTaskStatus(
                BackgroundTaskSample.servicingCompleteTaskName, true);
            return;
        }
        hascur = iter.moveNext();
    }

    // The servicing-complete background task is not already registered.
    BackgroundTaskSample.registerBackgroundTask(
        BackgroundTaskSample.servicingCompleteTaskEntryPoint,
        BackgroundTaskSample.servicingCompleteTaskName,
        new Windows.ApplicationModel.Background.SystemTrigger(
            Windows.ApplicationModel.Background.SystemTriggerType.servicingComplete, false),
        null);
},
```

In the sample, the tasks associated with these triggers are implemented in C#, within a WinRT component found in the Tasks project of the solution. I won't show the code here because we'll be looking at the general structure of WinRT components in Chapter 18. What it does demonstrate, though, is that you can use a mixed-language approach for background tasks. In these cases, the Entry Point field for the tasks in the manifest point to the C# class/method that implements the background task, such as Tasks.ServicingComplete. If you go to the Background task sample page, you can also download the C# and C++ versions of the sample to see even more structural variants.

Lock Screen–Dependent Tasks and Triggers

The last group of background tasks are those that require the app be also added to the lock screen. For this there are four applicable SystemTrigger options from SystemTriggerType, along with the four other distinct types that are represented in the manifest editor: LocationTrigger, TimeTrigger, PushNotificationTrigger, and Windows.Networking.Sockets.ControlChannelTrigger. These are described here, along with pointers to available samples that demonstrate their usage:

SystemTriggerType	When Triggered, Scenarios, and Samples
`controlChannelReset`	See `ControlChannelTrigger` below.
`sessionConnected`	User has logged in from the lock screen.
`userAway`	Device has become inactive (e.g., blank screen) due to user inactivity.
`userPresent`	User has returned; device wakes up from an inactive state.
`backgroundWorkCostChange`	The cost of background work (`BackgroundWorkCost.currentBackgroundWorkCost`) has changed
Other lock screen triggers	**When Triggered, Scenarios, and Samples**
`LocationTrigger`	The device has entered or exited a geofence region; see scenario 5 of the Geolocation sample.
`TimeTrigger`	A period of time as configured in the trigger has elapsed; see scenario 5 of the Background task sample demonstrates and scenario 3 of the Geolocation sample.
`PushNotificationTrigger`	A raw push notification for the app has arrived from WNS. Because Windows cannot handle a raw notification directly, this kind of background task is necessary to take action on a raw notification when the app isn't running. A running app, on the other hand, can use the `pushnotificationreceived` event, as described earlier in "Receiving Notifications (App)." For an example, refer to the Raw notifications sample.
`Windows.Networking.Sockets.ControlChannelTrigger`	Real-time notifications have been received through the *control channel*—that is, a networking channel typically using sockets or another networking transport, if it's not possible to use WNS and raw notifications for the same purpose. This trigger is used for real-time communication apps such as VoIP, IM, and Mail so that they are "always reachable" if the user places them on the lock screen. A deep dive on this subject is beyond the scope of this book, so refer to the How to set background connectivity options in the documentation along with the following samples, all of which employ C# or C++ and are not available in JavaScript: • ControlChannelTrigger StreamSocket sample • ControlChannelTrigger XmlHttpRequest sample • ControlChannelTrigger StreamWebSocket sample • ControlChannelTrigger HTTP client sample The `SystemTriggerType.controlChannelReset` is used to manage a background task for changes in the control channel rather than events on the channel itself. Where channel events like calls are concerned, such apps will typically make use of toast notifications with the `incomingCall` commands in the toast XML as shown below. See the Lock screen call SDK sample for a demonstration.  Grandma Incoming video call

Note Working with the lock screen is not supported in the Visual Studio simulator. To debug lock screen apps and background tasks, use the Local Machine or Remote Machine debugging options.

Background tasks for these triggers are created and registered as we've already seen. A TimeTrigger, for example, is created with its freshnessTime interval (in minutes) and a oneShot flag, as seen in scenario 5 of the Background task sample (js/time-trigger-background-task.js):

```
BackgroundTaskSample.registerBackgroundTask(
    BackgroundTaskSample.sampleBackgroundTaskEntryPoint,
    BackgroundTaskSample.timeTriggerTaskName,
    new Windows.ApplicationModel.Background.TimeTrigger(15, false), null);
```

A TimeTrigger is also used in scenario 3 of the Geolocation sample to allow the user to add a navigation app to the lock screen for more continuous tracking (scenario 5 uses the LocationTrigger for geofencing). Generally speaking, though, a navigation app isn't particularly useful on the lock screen in the first place, since it wouldn't be able to show a map! Better, then, to again use the Windows.System.Display.DisplayRequest API to prevent going to the lock screen at all.

Creating a PushNotificationTrigger is even simpler because there are no parameters. This can be seen in the Raw notifications sample, scenario 1 (js/scenario1.js):

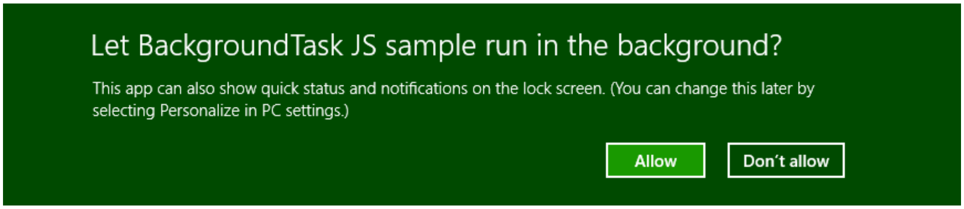
```
function registerBackgroundTask() {
    // Register the background task for raw notifications
    var taskBuilder = new background.BackgroundTaskBuilder();
    var trigger = new background.PushNotificationTrigger();
    taskBuilder.setTrigger(trigger);
    taskBuilder.taskEntryPoint = sampleTaskEntryPoint;
    taskBuilder.name = sampleTaskName;

    var task = taskBuilder.register();
    task.addEventListener("completed", backgroundTaskComplete);
}
```

Although the call to BackgroundTaskBuilder.register might succeed, the task itself will not execute until the user adds the app to the lock screen, as we saw earlier in Figure 16-8. This latter action is never under the app's control—all it can do is make sure it's *available* for the user to select on that section of PC Settings, which is what asking for access is all about.

The request is made through the BackgroundExecutionManager.requestAccessAsync method; this call should be made prior to registering the background task (see scenario 5 of the Background task sample again):

```
Windows.ApplicationModel.Background.BackgroundExecutionManager.requestAccessAsync();
```

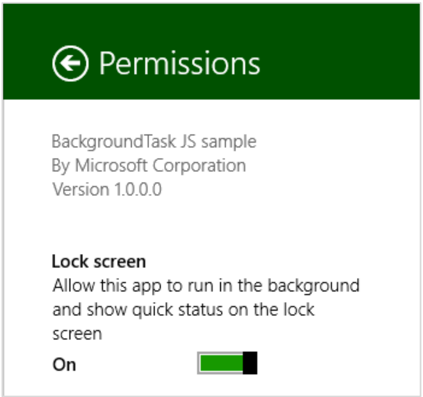
When this is called the first time in an app, it will generate a user consent prompt, as shown in Figure 16-19. If the user chooses Allow, the app will appear in PC Settings as an option for the lock screen, otherwise it won't. As with other permissions, users can change their minds later on through the Permissions settings, as shown in Figure 16-20.



Let BackgroundTask JS sample run in the background?

This app can also show quick status and notifications on the lock screen. (You can change this later by selecting Personalize in PC settings.)

FIGURE 16-19 The user consent prompt when an app requests lock screen access.



Permissions

BackgroundTask JS sample
By Microsoft Corporation
Version 1.0.0.0

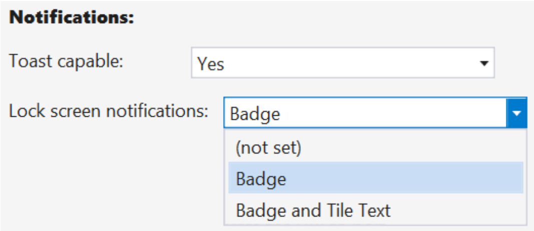
Lock screen
Allow this app to run in the background and show quick status on the lock screen

On

FIGURE 16-20 The lock screen option on the Permissions settings panel for apps that request access.

For a complete demonstration, refer to the Lock screen apps sample. Scenario 1 shows how to again request access to the lock screen and check the result, which is a value from the `BackgroundAccessStatus` enumeration. It also shows querying for and removing that access with the `getAccessStatus` and `removeAccess` methods of `BackgroundExecutionManager`.

Scenario 2 then demonstrates sending badge updates to the lock screen, along with a text tile update if the app happens to be the single one selected for that privilege. There is nothing particular in this process where the lock screen is concerned, however: such updates happen exactly as they do for the primary app tile. It's just that those updates are also reflected on the lock screen as determined by the setting in the Application > Notifications section of the manifest, as shown below. Remember that the badge graphic must have white or transparent pixels and the three scale sizes are 24x24 (100%), 33x33 (140%), and 43x43 (180%).



Notifications:

Toast capable: Yes

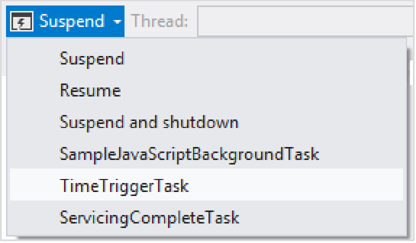
Lock screen notifications: Badge

Scenario 3, finally, demonstrates that secondary tiles can be added to the lock screen as well, irrespective of the app tile. To make a secondary tile available for the lock screen, assuming that the app has requested lock screen access already, you need to set those two properties of [Windows.UI.StartScreen.SecondaryTile]() that we mentioned long ago: `lockScreenBadgeLogo` and `lockScreenDisplayBadgeAndTileText`. If the secondary tile is on the Start screen, these properties will also make it available on the PC Settings page for the lock screen.

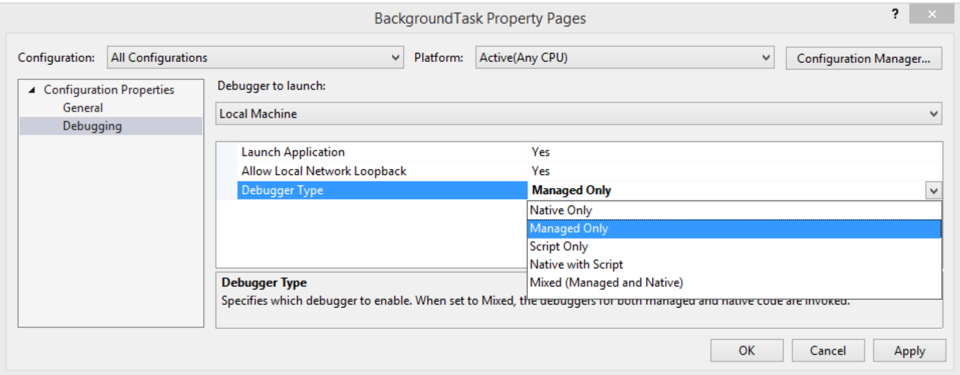
Debugging Background Tasks

By this time you might have run the `TimeTrigger` background task in scenario 5 of the Background tasks sample, and unless it's been more than 15 minutes since that time (maybe up to 30 minutes if you just missed the 15-minute window when timers are coalesced), you might still be waiting for that period to elapse. Is this, then, your destiny for debugging background tasks: to wait, wait, wait?

Fortunately, the answer is no, no, and maybe! That is, Visual Studio's debugger is mostly aware of registered background tasks and provides a list of them on its Suspend drop-down toolbar menu:



Selecting one of these will immediately trigger the background task, so you won't have to wait or otherwise attempt to activate the trigger for real. One caveat is that if the trigger had `oneShot` set to `true` and already fired, it won't fire again. A second caveat is that if you're running a JavaScript app with background tasks written in other languages, you'll need to change the debugger type for the main app project from Script Only to any of the others that list Managed or Native, as shown below, otherwise you can't set breakpoints in those other modules:


BackgroundTask Property Pages

Configuration: All Configurations
Platform: Active(Any CPU)

Debugger to launch:
Local Machine

Launch Application: Yes
Allow Local Network Loopback: Yes
Debugger Type: Managed Only

Debugger Type
Specifies which debugger to enable. When set to Mixed, the debuggers for both managed and native code are invoked.

A third caveat is that background tasks using `PushNotificationTrigger`, `ControlChannelTrigger`, and `SystemTriggerType.SmsReceived` will not appear on the drop-down menu. You might need to rely on the tried-and-true methods of outputting diagnostic information to figure out what's going on with your task and checking events in the Event Viewer for activation failures. More on these methods can be found on How to debug a background task.

Finally, note one more time that background tasks are not supported in the Visual Studio simulator, as is also true of live tiles, notifications, and much else we've covered in this chapter. You'll need to use the Local Machine or Remote Machine options instead.

What We've Just Learned (Whew!)

- Tile updates, on both the app's primary and secondary tiles, along with badges, toast notifications, and background tasks are how an app contributes to the overall aliveness of the system even while the app isn't running.
- Tile updates and notifications can be sent from a running app but there are other methods to deliver those updates when the app isn't running. Updates can be scheduled to appear at a later time, and the app can configure the system to periodically ask a service for tile/badge updates. Apps can also configure push notifications that are raised from a service and sent to clients through the Windows Push Notification Service (WNS).
- Tile updates are issued using an XML payload based on predefined templates. Typical payloads include medium, wide, and large tile updates so that the user can choose how the tile is displayed on the Start screen. The XML can reference images from both local and remote sources, so long as the images are 1024x1024 pixels or smaller and less than 200KB in size.
- A tile can cycle through up to five updates at any given time for each tile size; each update can be replaced separately.
- Apps that have specific content that is interesting to bookmark as secondary tiles to the Start screen provide Pin and Unpin commands to the user for that purpose. Secondary tiles, which launch the app with specific startup arguments, can also receive live tile updates.
- Badges are small glyphs or numbers that can appear on any given tile. Badge updates are sent through the same mechanism as tile updates, but they operate independently.
- Toasts are popup notifications that appear for a time to alert the user of new information, reminders, and so on. They can play sounds, recur on a given interval, be scheduled to appear in the future, and for alarms and incoming calls can display relevant commands. Like secondary tiles, activating a toast launches the app with specific startup arguments.
- Periodic updates for tiles and badges means providing Windows the URIs of REST endpoints from which it will request updates at selected intervals between 30 minutes and 24 hours.

Periodic updates are the easiest and lower-cost means to update a tile from a service.

- Push notifications for tiles, badges, toast notifications, and raw notifications (whatever data an app wants to manage) can be used for higher-frequency, user-specific updates. This involves creating services that communicate with WNS to issue those notifications to specific channel URIs, a process that is much more involved and expensive than periodic updates.
- Windows Azure Mobile Services can be used for many update activities, including the implementation of periodic update endpoints and backends for push notifications.
- Background tasks are small pieces of code that an app configures to run when certain triggers occur, such as changes in connectivity, timers, geofencing, receipt of push notifications, and app updates. Apps should never depend on background tasks, however, because they are always under the user's control.
- Background task triggers can be refined through specific conditions to avoid running tasks when it's not necessary (such as when there is no connectivity).
- Some triggers require that the app is also added to the lock screen. Such apps must first request access, which is subject to user consent, and the user must specifically add the app through PC Settings. Given that privilege, apps can issue badge updates and tile text to the lock screen.
- Through maintenance triggers, apps can also set up tasks to run periodically when a device is on AC power.

Chapter 17

Devices and Printing

I sometimes marvel at all the stuff that's hanging off the humble laptop with which I've been writing this book. Besides the docking station that is currently also serving well as a monitor stand and rather efficient dust collector, there's a mouse (wired), a keyboard (wireless), two large monitors, speakers, a USB thumb drive, a headset, an Xbox game controller, and the occasional external hard drive. Add to that a couple of printers and media receivers hanging off my home network and, well, I probably don't come close to what the majority of my readers probably have around their home and workplace!

For all that might be going on within one computer itself, and for all the information it might be obtaining from online sources, the world of external devices is another great realm, especially those that apps can work with directly. Indeed, we've spent most of our time in this book talking about what's going on within an app and its host system and about using networks and services to gather data. It's time that we take a look at the other ways we can draw on external hardware to make a great app experience.

We've already encountered a few of these areas:

- In Chapter 11, "The Story of State, Part 2," we learned about the *Removable Storage* capability (in the manifest) that enables an app to work with files on USB sticks and other pluggable media. When a device is connected, those folders become available through `Windows.Storage.KnownFolders.removableDevices`, a `StorageFolder` object whose subfolders are each connected to a particular storage device. See the Removable storage sample.
- In Chapter 2, "Quickstart," along with Chapter 13, "Media," we took full advantage of the `Windows.Media.Capture` API to effortlessly obtain audio, images, and video from an attached camera (see the CameraCaptureUI Sample). This included the ability to select a specific capture device through scenario 2 of the Media capture using capture device sample, for which we used the API in `Windows.Devices.Enumeration`.
- Also in Chapter 13 we looked at the `Windows.Media.PlayTo` API to connect media to a PlayTo capable receiver, as demonstrated in the Media Play To sample.
- In Chapter 12, "Input and Sensors," we explored pointer input devices (touch, mouse, and stylus), keyboard input, the APIs for reading sensor data, and using a GPS device (for geolocation and geofencing). Refer to that chapter for applicable samples.

In this chapter we're ready to expand the story more generally, even though we'll only be scratching the surface of the world of hardware! Our topics include:

- **Declaring device access** in the app manifest.
- **Device enumeration and watcher APIs** that you use in scenarios where you wouldn't just use the default device of a given class. Enumerating available devices and watching them as they're added to or removed from a system are processes that are generally applicable, so we'll discuss these before looking at any device-specific APIs.
- **Device scenario APIs** work with known types of devices, namely printers (2D and 3D), scanners, point-of-service devices (barcode readers and magnetic stripe readers), fingerprint readers, virtual smartcards, and Bluetooth call control devices. Each class of devices responds to a set of standard commands that are hidden behind the API abstraction (see Figure 17-1) so that the same app code works for any device of that class that the user might attach.

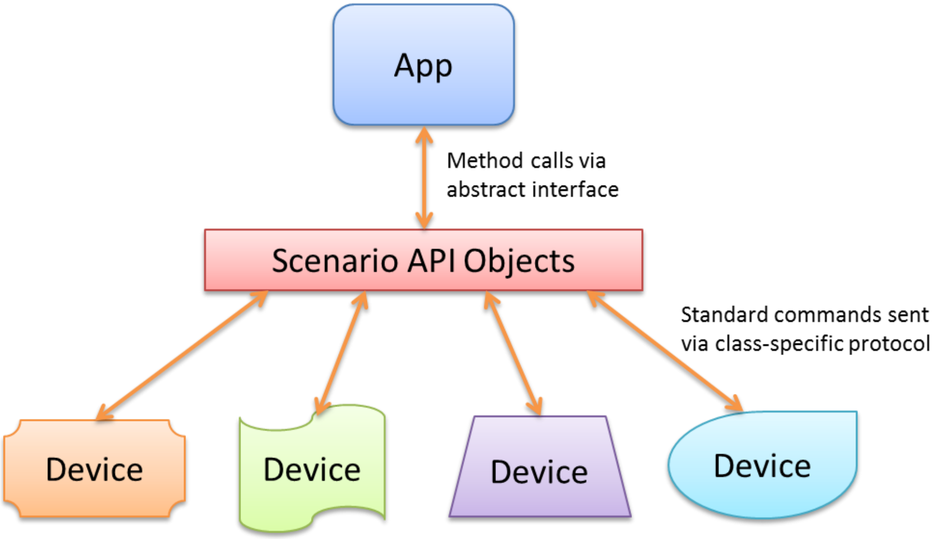

App
Method calls via abstract interface
Scenario API Objects
Standard commands sent via class-specific protocol
Device
Device
Device
Device

FIGURE 17-1 Scenario APIs abstract different device classes and the standard commands used to communicate with them. Through that API, an app can work with any number of connected devices that conform to that class.

- **Protocol APIs** allow you to communicate with a wide variety of devices through specific protocols, namely Bluetooth, HID, USB, and Wi-Fi Direct. In this cased the APIs handle the details of the underlying transport and the app sends device-specific commands through that protocol (see Figure 17-2). Such apps, therefore, are written to work with a specific device or a set of such devices. Note also that protocol APIs specifically block access to devices that are otherwise supported through device scenario APIs or other more general WinRT APIs like those for pointer events, keyboards, and sensors.

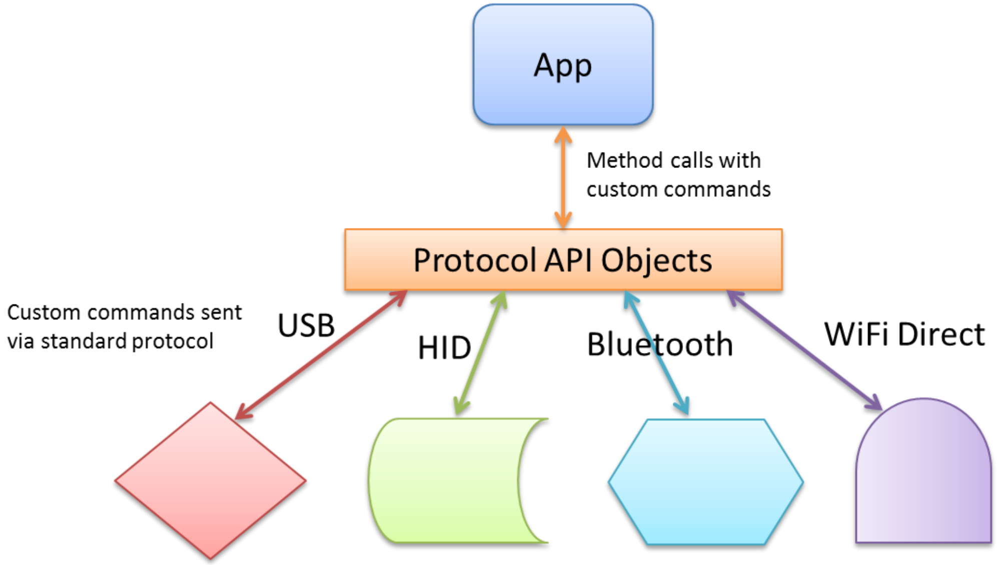

App
Method calls with custom commands
Protocol API Objects
Custom commands sent via standard protocol
USB
HID
Bluetooth
WiFi Direct

FIGURE 17-2 Protocol APIs abstract the details of transport protocols—namely USB, HID, Bluetooth, or Wi-Fi Direct—allowing apps to easily send device-specific commands to a wide variety of devices.

- Finally, **NFC (near-field communications)** connects with devices in the vicinity of the one your app is running on, including inexpensive NFC tags. The initial NFC handshakes happen over Bluetooth or Wi-Fi Direct protocols, and once the connection is made, apps communicate using sockets, as shown in Figure 17-3.

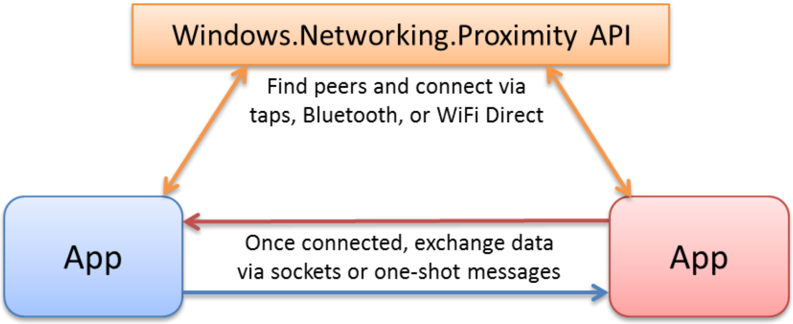

Windows.Networking.Proximity API
Find peers and connect via taps, Bluetooth, or WiFi Direct
App
Once connected, exchange data via sockets or one-shot messages
App

FIGURE 17-3 Near-field communications (NFC) use taps, Bluetooth, or Wi-Fi Direct to establish a connection, then apps exchange data directly via sockets or messages.

Note For wireless devices (like those using Bluetooth and Wi-Fi Direct), Windows has a pairing UI already built in, meaning that apps don't need any pairing features themselves.

Another special group of devices are those accessible through Win32/COM APIs, for which you can then basically create your own scenario API as a Windows Runtime Component. For example, I could use the HID protocol API to talk to an Xbox 360 game controller (and will in this chapter), or I can use the XInput API (part of DirectX) through a WinRT component, whose interface is projected into JavaScript like any other WinRT API. We'll see how to do this in Chapter 18, "WinRT Components."

Clearly, you can build an app around many devices, and you can enable many creative scenarios by building around multiple devices. It's important to note that with the protocol APIs you'll typically be declaring access to specific devices and their features, so you won't so much be thinking about general-purpose apps that can work with random devices. With Windows Store apps it's good to be focused anyway, and the nature of devices pretty much forces this.

All that said, there are a few things we won't be covering in this chapter. One topic is 3D printing, which, due to its intense graphical requirements, is designed to work with DirectX and is therefore applicable to Windows Store or desktop apps written with C++. For more on this subject, refer to 3D Printing in the documentation, the 3D printing sample (C++), and the //build 2013 session, 3D Printing with Windows.

We also won't be covering what are called Windows Store Device Apps, the ones that can be automatically acquired from the Windows Store when their associated devices are attached into a system. This also includes mobile broadband apps that deal with provisioning a mobile account. Such dedicated device apps are associated with their hardware through device metadata, and also have more privileges than other apps such as launch on connect, updating firmware, background sync, and much more generous budgets for background CPU time and I/O traffic.

For more details on this subject, refer to the Windows Store Device App Workshop (Channel 9 videos), along with the Windows 8.1 Device Experience area of the documentation. The latter includes sections on AutoPlay, automatic installation, device sync/firmware update, apps for printers, apps for cameras, apps for internal devices, and mobile broadband/mobile operator concerns. Here are also a number of other device app samples to draw on:

Windows Store device app for camera sample	Firmware Update USB device sample
Device apps for printers sample	AppContainer mobile broadband pin, connection, and device management sample
Print settings and print notifications sample	Mobile broadband account and device management sample
Print job management and printer maintenance sample	Mobile broadband account provisioning sample
USSD message management sample	Portable device services sample

A much more specific one is the Custom driver access sample, which works with a device called FX2 in the OSR USB FX2 Learning Kit (from Open System Resources). This is a piece of hardware that those learning how to develop device drivers use to understand the intricacies. Let me also mention that many device-oriented sessions can be found in the //build 2013 conference content on Channel 9, filtered for Device Apps. I'll be referring to some of these as we go along, but you might at this point be interested in the overview session, Building Apps the Connect with Devices.

Declaring Device Access

Many devices require that you have a DeviceCapability declaration in your manifest—for instance:

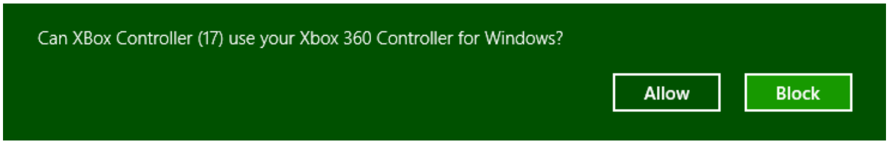
```
<Capabilities>
  <m2:DeviceCapability Name="pointOfService" />
</Capabilities>
```

where Name is an appropriate value for the type of device in question and the m2 namespace signifies the Windows 8.1 manifest schema, which is necessary for most of the devices we'll be covering in this chapter. I also show such a capability in XML because Visual Studio's manifest editor does not expose device-related features in its UI. When working with devices, then, you'll get used to right-clicking package.appxmanifest in Visual Studio and selecting View Code to edit the XML manually.

Some hardware, such as printers, scanners, and fingerprint readers, do not require a manifest declaration, but nearly every other device does. We'll see the detailed entries in the appropriate context for each, which sometimes include more child elements under DeviceCapability. For example:

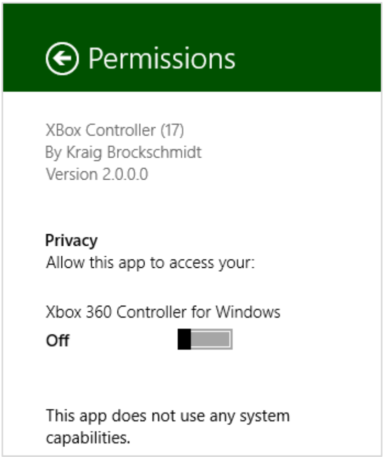
```
<m2:DeviceCapability Name="humaninterfacedevice">
  <m2:Device Id="vidpid:045E 0610">
    <m2:Function Type="usage:FFAA 0001" />
  </m2:Device>
</m2:DeviceCapability>
```

As with other manifest capabilities, these declarations are necessary to access the hardware at all (otherwise you'll always see Access Denied exceptions) and will generate a user consent prompt like this one when you first attempt access:


Can XBox Controller (17) use your Xbox 360 Controller for Windows?

It's important to note whatever API call might trigger the prompt should always happen on the UI thread. This is a given for apps written in JavaScript, but if you're working in other languages you'll need to take this into account. Also, because the user can deny access at this point, you must always be prepared to respond accordingly.

As with all other capabilities, the user can change their consent at any time through the app's Settings > Permissions (below left) or PC Settings > Privacy > Other Devices (below right), so also be prepared for device access to change at any time:



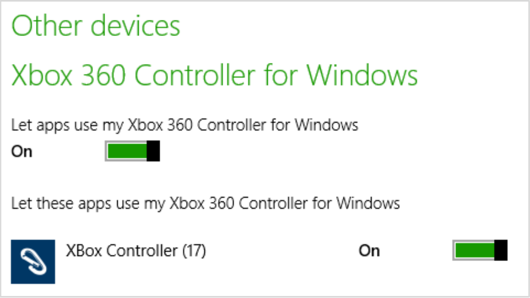
Permissions

XBox Controller (17)
By Kraig Brockschmidt
Version 2.0.0.0

Privacy
Allow this app to access your:

Xbox 360 Controller for Windows
Off

This app does not use any system capabilities.



Other devices

Xbox 360 Controller for Windows

Let apps use my Xbox 360 Controller for Windows
On

Let these apps use my Xbox 360 Controller for Windows

XBox Controller (17) On

Enumerating and Watching Devices

To talk to any device through its associated WinRT API, it's necessary to acquire a device-specific API object, such as a BarcodeScanner, ImageScanner, HidDevice, and so on. In a few cases, namely BarcodeScanner and MagneticStripeReader, the object provides a static getDefaultAsync method that shortcuts the whole process:

```
Windows.Devices.PointOfService.BarcodeScanner.getDefaultAsync().then(function (scanner) {
// scanner is a BarcodeScanner instance.
}

Windows.Devices.PointOfService.MagneticStripeReader.getDefaultAsync().then(function (reader) {
// reader is a MagneticStripeReader instance.
}
```

Most other device APIs, however, don't have this shortcut, and so it's necessary to *enumerate* devices of that type. Enumeration means to obtain a list of all devices that match a particular *selector*, as we'll see in a moment. Each device in the list is represented by a DeviceInformation object (in Windows.Devices.Enumeration), whose id property is needed by another ubiquitous static method, fromIdAsync (e.g., HidDevice.fromIdAsync), to connect with that specific piece of hardware.

To connect with the default device in the enumerations, use the id from the first DeviceInformation object in the list, call fromIdAsync, and go from there. The DeviceInformation objects you get through the process of enumeration makes displaying a list of available devices in your UI straightforward so that the user can choose which camera, microphone, scanner, printer, or game controller they want to play with. For all the details, refer to Enumerating Devices in the documentation, but let me give an overview here before we look at device-specific APIs.

Again, enumeration works with something called a *selector*, specifically an Advanced Query Syntax (AQS) string, as we encountered in Chapter 11 in "Custom Queries." A device selector typically looks something like this:

```
System.Devices.InterfaceClassGuid:="{E5323777-F976-4F5B-9B55-B94699C46E44}" AND
System.Devices.InterfaceEnabled:=System.StructuredQueryType.Boolean#True
```

where the interface class GUID shown here is the particular one for webcams.

In the simplest cases, common selectors have numerical shortcuts obtained from the `DeviceClass` enumeration: `audioCapture`, `audioRender`, `portableStorageDevice`, `videoCapture`, `imageScanner`, and `location` (as well as `all`). You can pass one of these values to one variant of the static `DeviceInformation.findAllAsync` method, for example:

```
var wde = Windows.Devices.Enumeration;

wde.DeviceInformation.findAllAsync(wde.DeviceClass.videoCapture)
   .done(function (devices) {
      //devices is a DeviceInformationCollection, essentially
      //an array of DeviceInformation objects
   });
```

In this example, the `videoCapture` shortcut maps to the same AQS string for webcams shown earlier. The result of `findAllAsync` is then a `DeviceInformationCollection`, which in JavaScript can basically be treated as an array of `DeviceInformation` objects.

Another way to obtain a selector is to ask for one from a device-specific class in WinRT, typically through a method called `getDeviceSelector`.[120] You'll find such a method on quite a number of classes, including `ProximityDevice`, `StorageDevice`, `SmsDevice`, `UsbDevice`, `HidDevice`, `SmartCardReader`, `BarcodeScanner`, `ImageScanner`, `MagneticStripeReader`, and so on. We'll encounter these throughout this chapter. The one odd duck of the group is the `MediaDevice` object, which has `getAudioCaptureSelector`, `getAudioRenderSelector`, and `getVideoCaptureSelector` methods. With the latter, for example:

```
var mediaDevice = Windows.Media.Devices.MediaDevice;
selectorString =  mediaDevice.getVideoCaptureSelector();
```

you'll get back the same AQS string as before.

Thirdly, you can also construct a selector from scratch if you have a known GUID that isn't otherwise represented in the WinRT API. Scenario 1 of the Device enumeration sample does this with known GUIDs for printers, webcams, and portable devices (html/interfaces.html):

[120] Note that in the code snippet in the Quickstart: enumerating commonly used devices topic in the docs (at the time of writing) uses `GetDeviceSelector` in a couple of cases, which will throw exceptions. The camel-cased `getDeviceSelector` is correct, and in the case of the `ServiceDevice` object it also requires an additional argument that the code in the docs does not provide.

```
<select size="3" id="selectInterfaceClass" aria-labelledby="listLabel">
    <option selected value="{0ECEF634-6EF0-472A-8085-5AD023ECBCCD}">Printers</option>
    <option value="{E5323777-F976-4F5B-9B55-B94699C46E44}">Webcams</option>
    <option value="{6AC27878-A6FA-4155-BA85-F98F491D4F33}">Portable Devices</option>
</select>
```

When you select one of these from the sample's drop down and click the button to enumerate, it builds a selector string like so (js/interfaces.js):

```
var deviceInterfaceClass = deviceInterfaceClassText.value;
var selector = "System.Devices.InterfaceClassGuid:=\"" + deviceInterfaceClass + "\"";
// + " AND System.Devices.InterfaceEnabled:=System.StructuredQueryType.Boolean#True";
```

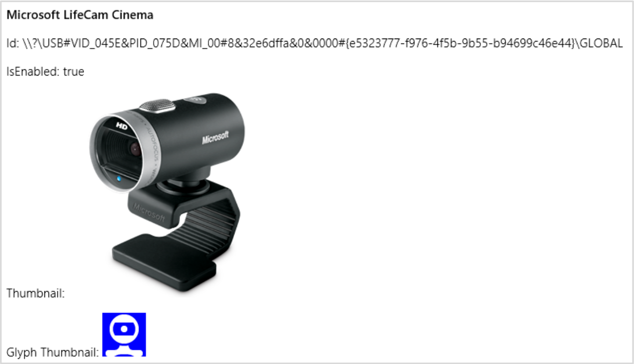
where the *InterfaceEnabled* part of the string is optional but generally a good idea.

However you obtain the selector string, the next step is to use the other variant of `DeviceInformation.findAllAsync` that accepts a string selector along with an array of additional property strings (such as sub-containers in a device; pass `null` if you don't have anything else to specify):

```
Windows.Devices.Enumeration.DeviceInformation.findAllAsync(selector, null).done(
    function(devinfoCollection) {
        var numDevices = devinfoCollection.length;
        for (var i = 0; i < numDevices; i++) {
            displayDeviceInterface(devinfoCollection[i], id("scenario1Output"), i);
        }
    });
```

Really important for JavaScript! I can't stress the following point enough because it's caused me some hours of grief. The `DeviceInformation` class has *two* `findAllAsync` methods that accept one argument, but the only one that's projected into JavaScript accepts a `DeviceClass` object and *not* a selector string. Unfortunately, that method accepts a string without complaint and proceeds to enumerate every device on your system! Therefore, if you're using a selector string, *always pass* `null` *as the second argument* (as shown above) so that you get the right variant of `findAllAsync`.

In any case, the result of an enumeration is just an array of `DeviceInformation` objects. As noted before, to use the first enumerated device as the default, just grab the first item in the array and pass its id property to the appropriate object's `fromIdAsync`. Otherwise you can use the other properties and methods in `DeviceInformation` to create an attractive list of devices in your UI such as those shown in Figure 17-4 and Figure 17-5 (using the sample). Here the device image is obtained from `DeviceInformation.getThumbnailAsync` and the glyph from `getGlyphThumbnailAsync`.



Microsoft LifeCam Cinema

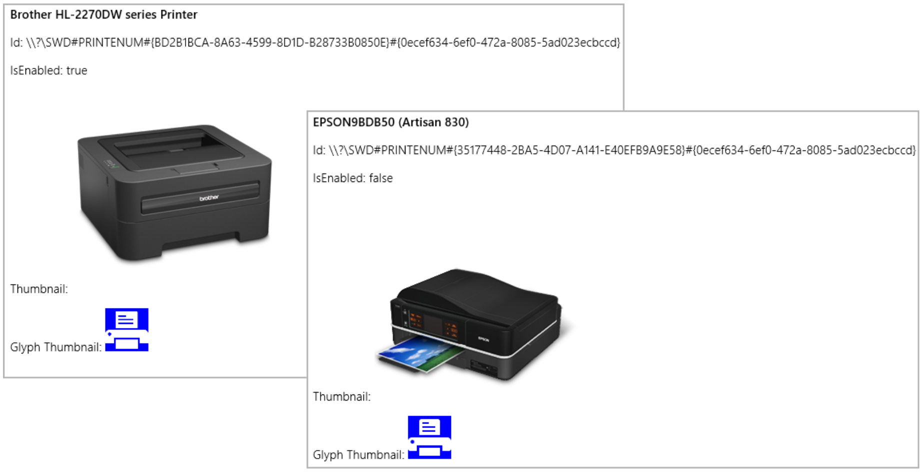
Id: \\?\USB#VID_045E&PID_075D&MI_00#8&32e6dffa&0&0000#{e5323777-f976-4f5b-9b55-b94699c46e44}\GLOBAL

IsEnabled: true

Thumbnail:

Glyph Thumbnail:

FIGURE 17-4 Sample device enumeration output for a webcam—which perfectly represents the one sitting on top of my monitor.



Brother HL-2270DW series Printer

Id: \\?\SWD#PRINTENUM#{BD2B1BCA-8A63-4599-8D1D-B28733B0850E}#{0ecef634-6ef0-472a-8085-5ad023ecbccd}

IsEnabled: true

Thumbnail:

Glyph Thumbnail:

EPSON9BDB50 (Artisan 830)

Id: \\?\SWD#PRINTENUM#{35177448-2BA5-4D07-A141-E40EFB9A9E58}#{0ecef634-6ef0-472a-8085-5ad023ecbccd}

IsEnabled: false

Thumbnail:

Glyph Thumbnail:

FIGURE 17-5 Sample device enumeration output for printers—showing two that look exactly like those sitting next to my desk.

Scenario 2 of the sample executes the same process (with plain text output) for Plug and Play (PnP) object types using `Windows.Devices.Enumeration.Pnp.PnpObject.findAllAsync`. This API lets you enumerate devices by interface, interface class, and container (the visible and localized aspects of a piece of hardware, like manufacturer and model name):

```
Windows.Devices.Enumeration.Pnp.PnpObject.findAllAsync(deviceContainerType,
    propertiesToRetrieve).done(function (containerCollection) {
        var numContainers = containerCollection.length;
        for (var i = 0; i < numContainers; i++) {
            displayDeviceContainer(containerCollection[i], id("scenario2Output"));
        }
    });
```

In the call above, the propertiesToRetrieve variable contains an array of strings that identify the Windows properties you're interested in. The sample uses these:

```
var propertiesToRetrieve = ["System.ItemNameDisplay", "System.Devices.ModelName",
    "System.Devices.Connected"];
```

The result of the enumeration—the containerCollection variable in the code above—is a PnpObjectCollection that contains PnpObject instances. The sample just takes the information from these (which might take a while to produce) and displays a text output for each.

Note that there is a variant of findAllAsync that accepts an AQS string as a filter. This is a string that you obtain from APIs like Windows.Devices.Portable.StorageDevice.getDeviceSelector that makes enumeration of those particular devices easier.

The findAllAsync methods that we've seen so far enumerate devices on demand. At other times you want to actively watch for devices as they are added or removed from the system. This is done with the DeviceInformation.createWatcher and PnpObject.createWatcher static methods, the former of which has variants that accept DeviceClass and selector strings as with findAllAsync.

The watcher objects that you get back, DeviceWatcher and PnpObjectWatcher, respectively, have various events to tell you about when the available devices change: added, removed, and updated, each of which supplies the appropriate DeviceInformation objects. These are clearly what you'd use to maintain a display of active devices in your app (as a number of the samples do), rather than using a timer to periodically enumerate the devices with findAllAsync. Just be sure to call the watcher's start method to enable periodic scanning, otherwise you'll be severely disappointed in the results!

As a final note, the API for near-field communication provides similar enumeration and watching capabilities, but the structure is slightly different. We'll see these in "Finding Your Peers (No Pressure!)."

Sidebar: Enumerating "Portable Devices"

Another group of devices are known as *portable devices*. Enumerating these is a third option in scenario 1 of the Device enumeration sample and is also demonstrated in the Portable device service sample. In both contexts, what we're talking about here opens the doors to the subject of Windows Portable Devices (or WPD), a driver technology that supports things like phones, digital cameras, portable media players, Bluetooth devices, and so on, where the need is primarily to transfer data between the device and the system. WPD supplies an infrastructure for this, and the Windows.Devices.Portable API lets you interact directly with WPD. Here you'll find the ServiceDevice and StorageDevice classes, both of which simply provide methods that return selector strings and id properties. In the former case, such information is meaningful only to the device app associated with the hardware. In the latter case, however, the StorageDevice.fromId method provides a StorageFolder through which you can enumerate its contents. This is demonstrated in scenario 3 of the Removable storage sample that we noted in Chapter 11, where it will create a list of removable storage devices to choose from and then display the first image found on the one you select.

Scenario API Devices

As described in the introduction, the *scenario* APIs work with different classes of devices for which both the underlying protocol and command structures are standardized. As a result, WinRT provides an abstract interface for different device types and apps don't need to have any knowledge of specific makes and models. Printing is the primary example of this, so much so that we'll come back to that concern (2D printing) in "Printing Made Easy." Input devices, sensors, and geolocation devices also fall into this category, and we've already given those a chapter unto themselves (Chapter 12).

What we'll explore here are a handful of other device types: image scanners, point of service devices (barcode scanners and magnetic stripe readers), smart cards, fingerprint readers, and Bluetooth call control devices. Most of these are represented by APIs within the `Windows.Devices` namespace, although fingerprint readers come from `Windows.Security` and Bluetooth call control from `Windows.Media`. In those cases too you don't need to worry about manifest declarations or device discovery. But let's not quibble over such concerns but move right into fun parts! And to make it all the more fun, I hope you have at least some of these devices on hand so that you can try them out through the SDK samples.

Image Scanners

If printing is a way to take digital information and render it in physical form, scanners serve the opposite purpose: to take physical manifestations of information and render them digitally. From there, such digital information can be shared, archived, edited, and otherwise manipulated in ways that are much less difficult or expensive than with the physical forms.

Image scanners in particular serve to digitize the many pictures and documents (from receipts and check stubs to legal documents and medical records) that we so often still have piling up around our homes and workplaces (especially if you work in a legal firm!). I know I once spent many hours converting several thousand 35mm color slides that filled a shelf worth of three ring binders into digital form, now taking up only a small amount of space on the 2TB drives in my home server. And I'm always happy to empty a file drawer through the document scanner on my Epson Artisan 830 shown earlier in Figure 17-5, offering the remains to my trust recycling bin.

Image scanners are also distinct from cameras in that they eliminate shadows, eliminate distortions, provide consistent lighting and color quality, often have document feeders, and often provide much higher pixel resolutions and therefore image quality than cameras do.

Recognizing the unique role of image scanners, you can start to identify the many different ways that your apps can help customers digitize the information that flows through their lives. When you do, the image scanning API in `Windows.Devices.Scanners` (which doesn't require any special capabilities in your manifest) allows you to work with the hardware, specifically any scanner that bears the Windows logo. (And telemetry shows that something like two-thirds of all printers connected to Windows devices are also scanners, so there are many of them out there!)

Background scanning Because scanning typically takes a lot more time than users are willing to wait, the API automatically prevents an app from being suspended during scanning operations. Once the operations are complete, an app that's still in the background will then be suspended. However, in your completed handler for the operation you can issue a toast notification to alert the user that the job is done. That way he knows to switch back to your app to see those results.

As with many device interactions, you'll typically start by enumerating available scanners or just using the default. Through this you obtain the appropriate ImageScanner object, after which you can set configuration options and enable a preview, if desired. Then you call its scanFilesToFolderAsync method, which creates files in the designated folder for which you can obtain the StorageFile objects. In other words, the scanner API is set up to generate user data files, because typically the user will want to do something else with all their scans and not have them hidden away in your app data.

To enumerate scanners, use DeviceInformation.findAllAsync or createWatcher with DeviceClass.imageScanner. The Scan runtime API sample, where all this is demonstrated, uses a watcher (js/scannercontext.js):

```
scannerWatcher = Windows.Devices.Enumeration.DeviceInformation.createWatcher(
    Windows.Devices.Enumeration.DeviceClass.imageScanner);

// Register to know when devices are added or removed, and when the enumeration ends
scannerWatcher.addEventListener("added", onScannerAdded);
scannerWatcher.addEventListener("removed", onScannerRemoved);
scannerWatcher.addEventListener("enumerationcompleted", onScannerEnumerationComplete);

scannerWatcher.start();
```

This code is run from scenario 1, where the event handlers simply maintain a list of available devices (in the variable ScannerContext.scannerList) that are used in the sample's other scenarios, where you always need to select a scanner from a drop-down list. Each scanner has a device id assigned to it, which you pass to the static ImageScanner.fromIdAsync method to obtain the ImageScanner *instance* for that particular device:

```
Windows.Devices.Scanners.ImageScanner.fromIdAsync(deviceId).then(function (myScanner) {
    // myScanner is an ImageScanner
});
```

From here we have a number of different paths to take with the ImageScanner, which are demonstrated in the other scenarios of the sample. Scenario 2, for instance, just does a default scan with the selected scanner (ScannerContext.currentScannerId), saving the result in the Pictures library (for which we need the capability selected in the manifest). This code is condensed from js/scenario2JustScan.js—I've removed some code so that we can see the core interaction path:

```
scanToFolder(ScannerContext.currentScannerId, Windows.Storage.KnownFolders.picturesLibrary);

function scanToFolder(deviceId, folder) {
    // Get the scanner object for this device id
    Windows.Devices.Scanners.ImageScanner.fromIdAsync(deviceId).then(function (myScanner) {
        // Scan API call to start scanning
```

```
            return myScanner.scanFilesToFolderAsync(
                Windows.Devices.Scanners.ImageScannerScanSource.default, folder);
        }
    }).done(function (result) {
        if (result) {
            if (result.scannedFiles.size > 0) {
                displayResults(result.scannedFiles);
            }
        }
    }, function (error) {
        if (error.name === "Canceled") {
            // Operation was canceled
        } else {
            // Other errors
        }
    }, function(progress) {
        // Display progress
    });
}
```

Once we have the ImageScanner object, we can ask it to make a scan with its default *source* into the given folder by using scanFilesToFolderAsync. The result of this (result in the code) is an ImageScannerScanResult object that contains a single property, scannedFiles, which is a vector view (therefore read-only) of StorageFile objects. With this you can generate whatever kind of display you want, just like you would had you enumerated files from some other folder.

Note If nothing is scanned, scanFilesToFolderAsync will succeed but the scannedFiles vector will be empty and its size property set to zero.

Other errors can occur that prevent the scan from completing, in which case the promise's error handler is invoked. If the promise from scanFileToFolderAsync is canceled (through the promise's cancel method), the error handler is invoked with error.name set to "Canceled", as you can see above. Other errors can be paper jams, cover open, user intervention needed, and so on. Refer to the reference page on scanFilesToFolderAsync for a list of possible errors.

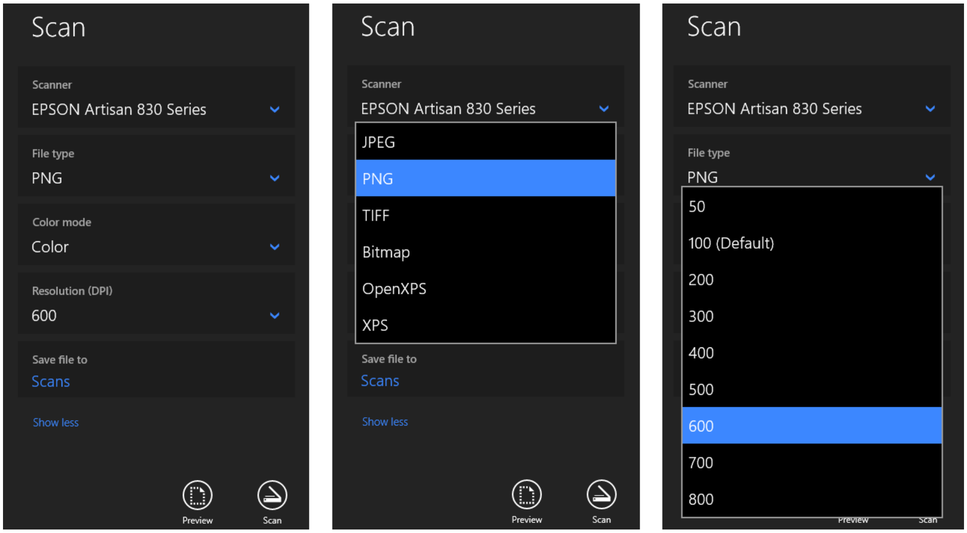
You'll notice too that scanFileToFolderAsync supports progress, so you can attach a progress handler to the promise to receive the number of files that have been scanned. This is most helpful when you are running multiple scans from some kind of feeder (see scenario 7 of the sample for a demonstration), because you can display a thumbnail of each scan file as its scan completes. On the other hand, if you want to provide a scan preview without generating a file, you use the scanPreviewToStreamAsync method, which we'll see in a little bit.

First, let's talk about the other options for the ImageScanner. In the scanFileToFolderAsync call we pass a *scan source* value from ImageScannerScanSource. The options here are default (auto-select), autoConfigured, flatbed, and feeder, all of which are used in various scenarios in the sample. The scanner's default can be retrieved through the ImageScanner.defaultScanSource property, and to determine whether a certain source is available, as when you want to present the user with the choice, call ImageScanner.isScanSourceSupported for the applicable options.

If you're using a scan source other than the default, you'll typically configure that source prior to starting the scan. Each of the specific scan sources has an associated configuration property on the ImageScanner object:

Property	Object Type	Description
autoConfiguration	ImageScannerAutoConfiguration	Specifies the file format to use with the scan but leaves all other properties as auto-configured by the device or through its dedicated control panel app, if one exists. For more, see Auto-Configured Scanning.
flatbedConfiguration	ImageScannerFlatbedConfiguration	Specifies resolution, cropping mode, brightness, color mode, contrast, format, and scan region, with properties for default, minimum, and maximum values for these.
feederConfiguration	ImageScannerFeederConfiguration	Specifies the same properties as for the flatbed source and then includes settings for page size, scan ahead, duplexing, page dimensions, and orientation.

The flatbed and feeder configuration objects are rather involved, so I won't list all the options here. For the most part, you use the min/max/default settings and their various is*Supported methods to build a configuration UI that specifically reflects the scanner's actual capabilities, as shown in Figure 17-6 for the Windows Scan app. For example, the ImageScannerAutoConfiguration object has just format and defaultFormat properties, which come from the ImageScannerFormat enumeration (with values like jpeg, gif, png, tiff, pdf, etc.). Its isFormatSupported method tells you which formats it can scan to, so when building a selection list, you iterate the formats and call isFormatSupported to determine whether that setting should be included.


Scan

Scanner
EPSON Artisan 830 Series

File type
PNG

Color mode
Color

Resolution (DPI)
600

Save file to
Scans

FIGURE 17-6 How the Windows Scan app displays various scanner properties and capabilities.

Let's see a few concrete examples in code. Scenario 4 of the Scan runtime API sample works with auto configuration (see js/scenario4DeviceAutoConfiguredScan.js). It checks first that auto configuration is supported and then sets the file format to PNG, if that's supported. Then it starts the scan with the `autoConfigured` source (I've added the namespace variable `wds` for brevity):

```
var wds = Windows.Devices.Scanners;

if (myScanner.isScanSourceSupported(wds.ImageScannerScanSource.autoConfigured)) {
    // Set the scan file format to PNG, if available
    if (myScanner.autoConfiguration.isFormatSupported(wds.ImageScannerFormat.png)) {
        myScanner.autoConfiguration.format = wds.ImageScannerFormat.png;
    }

    return myScanner.scanFilesToFolderAsync(wds.ImageScannerScanSource.autoConfigured, folder);
}
```

Scenario 5 does more or less the same thing for the flatbed configuration, setting the format to a device-independent bitmap (js/scenario5ScanFromFlatbed.js):

```
if (myScanner.isScanSourceSupported(wds.ImageScannerScanSource.flatbed)) {
    myScanner.flatbedConfiguration.format = wds.ImageScannerFormat.deviceIndependentBitmap;
    return myScanner.scanFilesToFolderAsync(wds.ImageScannerScanSource.flatbed, folder);
}
```

Scenario 6 uses the feeder source and shows the additional steps of setting a grayscale color mode and duplex scanning. I've taken the liberty here of modifying the code to set PDF as a primary format, using JPEG as a backup and shortening a few lines (js/scenario6ScanFromFeeder.js):

```
var wds = Windows.Devices.Scanners;

if (myScanner.isScanSourceSupported(wds.ImageScannerScanSource.feeder)) {
    var fc = myScanner.feederConfiguration;

    fc.format = fc.isFormatSupported(wds.ImageScannerFormat.pdf) ?
        wds.ImageScannerFormat.pdf | wds.ImageScannerFormat.jpeg;

    if (fc.isColorModeSupported(wds.ImageScannerColorMode.grayscale)) {
        fc.colorMode = wds.ImageScannerColorMode.grayscale;
    }

    fc.duplex = fc.canScanDuplex;
    fc.maxNumberOfPages = 0;  // Zero means scan all the pages that are present in the feeder

    return myScanner.scanFilesToFolderAsync(wds.ImageScannerScanSource.feeder, folder);
}
```

And that's pretty much it for the scanning source. As you can see, there's a definite pattern that you follow for all the options.

The last part of the API is the ability to get a preview of a scan prior to scanning to a file. First, check `ImageScanner.isPreviewSupported` to discover whether the device is capable of previews, and if so, call `ImageScanner.scanPreviewToStreamAsync` to do the job. This method takes a scan source, as we've seen, and also a RandomAccessStream rather than a folder. To use the example from scenario 3 of the sample (code shortened from js/scenario3PreviewFromFlatbed.js):

```
var stream = new Windows.Storage.Streams.InMemoryRandomAccessStream();

scanPreviewToStreamAsync(wds.ImageScannerScanSource.flatbed, stream).done(function (result) {
   if (result && result.succeeded) {
      displayResult(stream);
   }
}
```

As with any other stream, you can call MSApp.createBlobFromRandomAccessStream, pass the blob to URL.createObjectURL, and assign that URL to an img element:

```
function displayResult(stream) {
   var image = document.getElementById("displayImage");
   var blob = window.MSApp.createBlobFromRandomAccessStream("image/bmp", stream);
   var url = window.URL.createObjectURL(blob);
   image.src = url;
}
```

Note that scanPreviewToStreamAsync does not support a progress handler nor interim results for a scan in progress, so you need to wait until the operation is complete before displaying the preview. In the meantime, you'll generally want to include some kind of progress ring or such to show that the scanner is working, and again remember that scanning will continue when the app is moved to the background.

Barcode and Magnetic Stripe Readers (Point-of-Service Devices)

Without having thought about it much, I'm sure you've encountered many *point-of-service* (PoS) devices in recent months. When you go to an airport and scan a receipt, a credit card, or some other identification, you're interacting with PoS devices, as does the boarding agent when scanning your boarding pass. When you swipe a card at a retailer to make payment, you're using a PoS device. When you go through self-checkout at a grocery store or a clerk scans the UPC codes of your items for you, PoS devices are involved. (PoS is sometimes called "point-of-sale," but many interactions have nothing to do with sales, which is why we use the more generic "service.")

These devices are quite diverse in their forms and include everything from scanners and card readers to cash drawers, change machines, receipt printers, NFC card scanners, and do on. In many cases you'll work with those devices through one of the protocol APIs discussed later in this chapter. For the two most common ones, however—barcode readers and magnetic stripe readers (MSRs)—Windows provides dedicated APIs that are found in Windows.Devices.PointOfService: BarcodeScanner and MagneticStripeReader. The job of these APIs is to convert data that's represented as barcodes or data that's written onto a magnetic card into a form that apps can consume. They work over USB HID connectivity with devices that adhere to the Unified POS standard, which has been adopted internationally for such common peripheral devices. If you want a little more background, watch the beginning of //build 2013 session 3-029, How to Use Point-of-Sale Devices in

Your App. In this chapter I'll just focus on the basic device interactions.

To use either device, you must first declare a DeviceCapability in your app manifest as follows (editing the XML directly):

```
<Capabilities>
  <DeviceCapability Name="pointOfService" />
</Capabilities>
```

The next step is to acquire a device. You can, of course, enumerate available devices and give the user a choice, or watch for devices being added and removed from the system, but for specifically engineered PoS systems it's much more likely that there is only one device. This means you can ask for the default device and be done with it. The BarcodeScanner and MagneticStripeReader classes make this easy by providing static getDefaultAsync methods:

```
Windows.Devices.PointOfService.BarcodeScanner.getDefaultAsync().then(function (scanner) {
// scanner is a BarcodeScanner instance.
}

Windows.Devices.PointOfService.MagneticStripeReader.getDefaultAsync().then(function (reader) {
// reader is a MagneticStripeReader instance.
}
```

Notice that I'm showing both classes in parallel, drawing code from the Barcode scanner sample and the Magnetic stripe reader sample (or MSR sample for show). Because they share a common paradigm, we can talk about their basic operations together. A little later we'll discuss each one's specifics.

At this point the device objects can report only on their capabilities (which are rich and varied), health (checkHealthAsync), statistics (retrieveStatisticsAsync), and status (statusupdated events). With the BarcodeScanner you can also retrieve and check for profiles along with supported symbologies (via getSupportedSymbologiesAsync and isSymbologySupportedAsync, with the BarcodeSymbologies class identifying dozens of these). With the MagneticStripeReader you also check the deviceAuthenticationProtocol and supportedCardTypes.

Now, acquiring an instance of the device object is not enough to obtain readings. For that you need to *claim* exclusive use of the device (see js/scenario1.js in both samples):

```
scanner.claimScannerAsync().done(function (claimedScanner) {
   // claimedScanner is an instance of ClaimedBarcodeScanner
}

reader.claimReaderAsync().done(function (claimedReader) {
   // claimedReader is an instance of ClaimedMagneticStripeReader
}
```

This step of claiming the device is what allows you to attach event handlers to it and to start receiving data (you'll find events in the ClaimedBarcodeScanner and ClaimedMagneticStripeReader classes, which I'll refer together to as Claimed<Device>). Why this extra step? It's because there can be multiple apps on the same system wanting to access the device. The APIs here thus work on a cooperative claim-and-release model. Let's say app A has a claim and thus has the Claimed<Device>

object. When app B makes a request, app A will receive the Claimed<Device>.releaseDeviceRequested event. By default, this will release the claim from app A and give it to app B. However, because app A might be in the middle of a transaction, it can deny the request by calling Claimed<Device>.retainDevice. Code for this can be found in scenarios 1 and 2 of the Barcode scanner sample, but to show it simply:

```
claimedScanner.addEventListener("releasedevicerequested", function () {
    claimedScanner.retainDevice();
});
```

The MSR sample doesn't show handling of the event, but it works identically. In the end, you almost always want to have such a handler because otherwise another app can come along and take the claim away at any time.

> **Note** If app A has a claim that is suspended and then app B makes a request, app B will be granted the claim automatically because the system will not resume app A nor fire the event. When app A resumes, it must request its claim again.

Assuming that you have a claim, you can then tell the device to decode encrypted data automatically by setting Claimed<Device>.isDecodeDataEnabled to true. If you want to do the decryption in the app, set this to false.

Next you register for the events of interest, which, apart from releasedevicerequested and erroroccurred, are unique for each class. Be mindful that all these are WinRT events and must be properly released to avoid memory leaks.

ClaimedBarcodeScanner event	Description
datareceived	Receives a BarcodeScannerDataReceivedEventArgs with a single report property (a BarcodeScannerReport), which contains scanData, scanDataLabel, and scanDataType (the symbology).
imagepreviewreceived	Receives a BarcodeScanningImagePreviewReceivedEventArgs whose single preview property is a RandomAccessStream that you can then pass through the MSApp.createBlobFromRandomAccessStream to URL.createObjectUrl to an img.src for display.
triggerpressed	The barcode scanner's button was pressed (no other data).
triggerreleased	The barcode scanner's button was released (no other data).

ClaimedMagneticStripeReader event	Description
aamvacarddatareceived	Receives a MagneticStripeReaderAamvaCardDataReceivedEventArgs object. AAMVA is the America Association Motor Vehicles Administrators and is used for cards like driver's licenses. The properties of this object are licenseNumber, expirationDate, firstName, surname, suffix, gender, height, weight, eyeColor, hairColor, address, city, state, postalCode, birthDate, endorsements, restrictions, and report (the raw data).
bankcarddatareceived	Receives a MagneticStripeReaderBankCardDataReceivedEventArgs from a bank card, e.g. credit and debit cards, which contains these properties: accountNumber, expirationDate, title, firstName, middleInitial, surname, suffix, serviceCode, and report (the raw data).

vendorspecificdatareceived	Receives a MagneticStripeReaderVendorSpecificCardDataReceivedEventArgs for cards that aren't otherwise identified as an AAMVA or a bank card. The only property in this object is report, which contains vendor-specific data.

Once you're ready to receive events, the last step in the process is to *enable* the device to receive data. "My, my," you say, "I already had to claim a device. Why must I now also enable it?" The reason for this is that doing something meaningful with the scanned data can take a considerable amount of time, during which you don't necessarily want to get another reading. For example, scanning an item at a self-checkout station just gets you the UPC number, which isn't enough to show any meaningful data to the customer who just waved that product through the laser field. The app will typically need to do an async database lookup with the UPC and display the results before allowing the customer to scan their next item.[121] Similarly, an MSR doesn't need to send more data while the app is busy authenticating the last credit card swipe.

Therefore, the app needs a way to tell the device, "Please wait a bit until I'm ready to receive data again," and this is what disable-enable does. By calling the Claimed<Device>.enableAsync call, you say that you're ready to receive data via events. When you need to hold off for a bit, call Claimed<Device>.disableAsync before starting your other async lookups. When you're ready to receive more data, call enabledAsync. And to simplify matters, you can instruct the Claimed<Device> to automatically call disableAsync after each data event by setting its isDisabledOnDataReceived property to true. (And the isEnabled property will return the status at any time.)

In short, claiming a device establishes a pipeline and disable-enable provides a quick way to open and close the valve on that pipe so you can control the flow of information in your app.

Beyond this, you can explore many other details with these devices, such as statistics, authentication, encryption, track IDs on MSRs, and more. I won't go into those details; if you're creating a PoS system, you'll necessarily know what more you need to explore within the Windows.Devices.PointOfService namespace.

Smartcards

If you're working with or have worked with a company that has grown large enough to take security seriously, you have encountered a smart card. As a Microsoft employee, I've had one for many years, which serves a dual purpose of giving me access to buildings on various corporate campuses as well as authenticating me when I connect to the corporate network over VPN. For the latter purpose I have both my physical smartcard and a virtual smartcard. The virtual smartcard is wholly convenient because I don't have to plug my physical card into a reader to connect over VPN.

[121] This is true even considering just how *slowly* some people seem to work through their baskets in the self-checkout lines!

This book is not the place to go into details about security, so let me briefly outline the API in Windows.Devices.SmartCards. First, the SmartCardReader object lets you find connected cards through its static findAllCardsAsync, fromIdAsync, and getDeviceSelector methods, and you can watch for cards through its cardadded and cardremoved events. Each reader (which represents a reader device, not a card) also has a kind (SmartReaderKind), a name, a deviceId, and status (getStatusAsync, SmartCardReaderStatus).

Each individual card is then represented by a SmartCard object, which provides basic info through getStatusAsync, reader, and getAnswerToResetAsync.

The richest piece of the API is the SmartCardProvisioning class, which is what you can use to create virtual smart cards (requestVirtualSmartCardCreationAsync), reset or change the card's PIN (requestPinChangeAsync, requestPinResetAsync), and delete cards (requestVirtualSmartCard-DeletionAsync). All this is the focus of the Smart card sample in the SDK. The only other bit I'll mention is that the sample declares the Shared User Certificates capability in its manifest, which is necessary for using the smart card APIs.

Fingerprint (Biometric) Readers

This particular scenario API isn't actually found in Windows.Devices but rather in Windows.-Security.Credentials.UI.UserConsentVerifier. This is what works with fingerprint readers on suitably equipped devices and does not involve any capabilities nor enumeration.

The name of the object here, UserConsentVerifier, underscores the fact that a fingerprint reader specifically validates the *physical presence* of the logged in user. This is much more certain than a PIN or a password, because you just can't share a fingerprint with anyone else! (Biometric devices have many ways to defeat fake fingers too.) It also allows for an additional level of security when a user shares their tablet or laptop with another person without logging out. So, if your app does any kind of high-value transaction for which you should double-check user consent (such as stock trades or financial exchanges), using a fingerprint adds a deeper layer of authentication. What's more, it's much easier for the user as well, especially on touch-only devices. For more background, refer to //build 2013 session 2-9110, Biometrics – Fingerprints for Apps.

The API is also super-simple: the UserConsentVerifier object has just two methods. First, checkAvailabilityAsync returns a UserConsentVerifierAvailability value: available, deviceNotPresent, disabledByPolicy, notConfiguredForUser, and deviceBusy:

```
var uci = Windows.Security.Credentials.UI;

uci.UserConsentVerifier.checkAvailabilityAsync().done(function (consentAvailability) {
    // Take action on availability
});
```

This is demonstrated in scenario 1 (js/s1-check-availability.js) of the UserConsentVerifier sample.[122] Assuming there is availability, scenario 2 (js/s2-request-consent.js) requests verification via `requestVerificationAsync`. The *message* argument for this method will be displayed in a `MessageDialog`, and assuming the user does a scan, the result of the operation is a `UserConsentVerificationResult` value (code simplified from the sample):

```
var uci = Windows.Security.Credentials.UI;
var ucvr = uci.UserConsentVerificationResult;

uci.UserConsentVerifier.requestVerificationAsync(message).done(function (consentResult) {
    switch (consentResult) {
        case ucvr.verified:
            // User's presence verified.
            break;
        case ucvr.deviceNotPresent:
            // No biometric device found.
            break;
        case ucvr.disabledByPolicy:
            // Biometrics is disabled by policy.
            break;
        case ucvr.retriesExhausted:
            // Too many failed attempts.
            break;
        case ucvr.notConfiguredForUser:
            // User has no fingerprints registered.
            break;
        case ucvr.deviceBusy:
            // Biometric device is busy.
            break;
        case ucvr.canceled:
            // Consent request prompt was canceled.
            break;
        default:
            // Consent verification with fingerprints is currently unavailable.
            break;
    }
});
```

For both methods, the only cases that invoke an error handler for the promise is when there's a problem unrelated to device availability or its status.

Bluetooth Call Control

Although we'll deal with generic Bluetooth devices in "Protocol APIs," another scenario API exists for a special case: Bluetooth communications devices that can handle calls, such as headsets and other telephony devices. These devices typically handle audio transfer and provide one or more buttons to

[122] Scenario 1 of the sample also has a (misspelled) `retriesExhaused` case for availability, but this is not a member of the availability enumeration. It is included in the `UserConsentVerificationResult` enumeration.

perform actions like redial, hang up, and so on. For these there exists an API in Windows.Media.Devices.CallControl and no manifest declarations are needed.

To obtain a device instance, call the static methods CallControl.getDefault or fromId. With that object you either instruct it to reflect call status or respond to events raised by the device. In all cases the app is the one that's doing the actual calling, such as doing real-time communications over sockets: the device is just a piece of peripheral hardware that can participate in the process, in much the same way a keyboard participates in text entry.

The CallControl object has four methods to set indicators on the device to reflect call status: indicateNewIncomingCall, indicateNewOutgoingCall, indicateActiveCall, and endCall methods. How the device responds depends on its design.

The events that the device can raise also depends on its design, but the CallControl object can receive the following: answerrequested, audiotransferrequested, hanguprequested, redialrequested, and keypadpressed. Your event handlers for these will typically invoke the CallControl methods. For example, if you respond to answerrequested by starting a call, you'd invoke indicateActiveCall.

The CallControl.hasRinger property also tells you if the device has its own ringer.

If you have a suitable device to play with, you can see the API in action through the Bluetooth Call Control sample. A walkthrough of the sample can be found on How to manage calls on the default Bluetooth communications device.

Printing Made Easy

An embarrassingly long time ago, when I was first working in the computer industry, I remember hearing excited talk about the "paperless office" and how very soon now we wouldn't need things like printers because everything would be shuttled around digitally. Decades later, we do find ourselves shuttling around plenty of digital content, and yet printing is still alive and well (except for this present book, of course, where early on we decided on an ebook format so that we could use extensive hyperlinks and color!). Maybe we still like paper for how it feels, how it uses our eyes differently, how it's cheap and disposable (unlike your Windows tablet), how it can be used to start fires in a pinch or make airplanes, and how it makes good use of all the small trees that get thinned out of commercial tree farms (at least here in the western United States). Maybe too it's just part of the human experience—after all, as much as we play with our computers, we do still live in a physical world with physical objects, so it makes sense that we continue to appreciate placing information onto physical media. And when we make printouts, we also give ourselves a good excuse to exercise our scanners!

Sometimes I wonder whether the idea of the paperless office wasn't fueled in part by the fact that many apps didn't implement printing very well, an artifact of it being a difficult task to begin with. (And then there were printer drivers of dubious quality, connection difficulties, and many other challenges.)

But gradually the whole world of printing has improved, both for consumers and for developers. More recently this has also seen the rise of 3D printing, a topic that is beyond the scope of this book because it requires that an app is written in C++.

Of course, printing isn't always about going to paper either. I frequently use a PDF "printer" to create read-only copies of documents that are more suitable for sharing in many cases than my originals. Occasionally I print to a fax machine (to send a fax), and more occasionally I'll print an email or web page directly to Microsoft OneNote for filing. In fact, while you're working on printing features in an app, I highly recommend setting your default printer to a digital target. That way you'll avoid producing copious amounts of scratch paper in the process, unless you happen to own a tree farm that you'll be thinning in a couple of years!

> **Get the backstory** If you want to know more about how printing as a whole has been reimagined, check out Simplifying printing in Windows 8 on the Building Windows 8 Blog, a post that provides deep soul satisfaction knowing that your printing future includes fewer drivers.

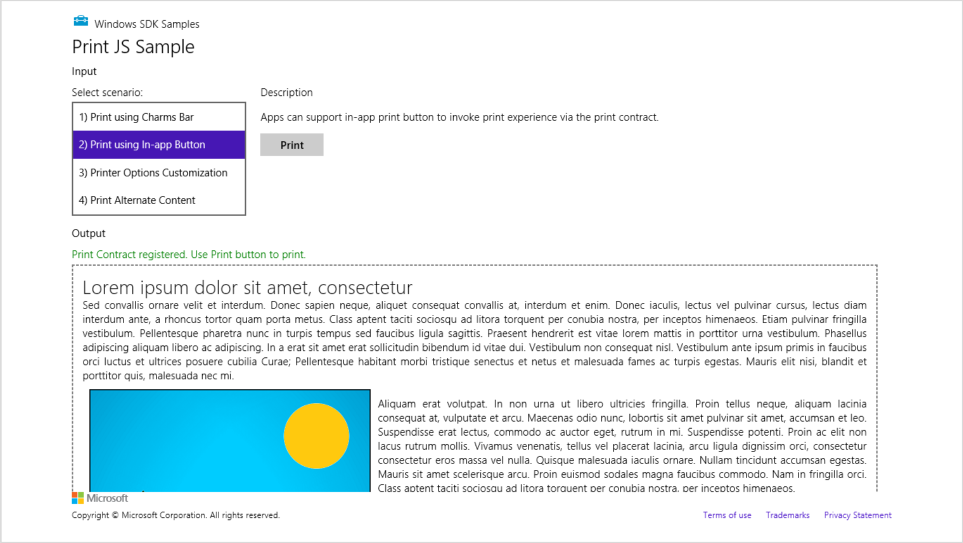
To understand how to implement printing in an app, let's first see what it looks like to the user. Then we'll see how to prepare content for printing and how to handle the printing-related events that you receive from Windows.

> **Note** A Windows Store app written in JavaScript can use the `window.print` method to print with default settings. It's not recommended, however, because it doesn't work with the print UI and doesn't always produce the best output. Windows Store apps should give the user the full Windows experience as described here.

The Printing User Experience

Printing typically starts in an app where the user is looking at something she wants to print and invokes an appropriate command. In scenario 2 of the refreshingly short-named Print sample, for instance, whose code we'll be looking at later in the chapter, we see a big block of content along with a Print button, as shown in Figure 17-7. Note that such a Print button would normally be on the app bar and not on the app canvas, but this is a sample.

To start printing, the user can either tap this Print button or open the Charms bar and select Devices. Either way, if the app is registered for printing—that is, it's listening for the event that's raised from the Devices charm and provides suitable content—the user will see a list of print targets, as shown on the left side of Figure 17-8. If the app doesn't have printable content—that is, it doesn't listen for the event or provides no content in response—the user will see a panel like that on the right side of Figure 17-8. This is very much the same experience that a user sees with the Share charm depending on whether the app provides data for that feature. You've likely seen the epic fail message of "This app can't share." Printing supplies a similar disappointment for apps that lack the capability. Don't let your app be one of them.



Windows SDK Samples

Print JS Sample

Input

Select scenario:

1) Print using Charms Bar
2) Print using In-app Button
3) Printer Options Customization
4) Print Alternate Content

Description

Apps can support in-app print button to invoke print experience via the print contract.

Print

Output

Print Contract registered. Use Print button to print.

Lorem ipsum dolor sit amet, consectetur

Sed convallis ornare velit et interdum. Donec sapien neque, aliquet consequat convallis at, interdum et enim. Donec iaculis, lectus vel pulvinar cursus, lectus diam interdum ante, a rhoncus tortor quam porta metus. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos himenaeos. Etiam pulvinar fringilla vestibulum. Pellentesque pharetra nunc in turpis tempus sed faucibus ligula sagittis. Praesent hendrerit est vitae lorem mattis in porttitor urna vestibulum. Phasellus adipiscing aliquam libero ac adipiscing. In a erat sit amet erat sollicitudin bibendum id vitae dui. Vestibulum non consequat nisl. Vestibulum ante ipsum primis in faucibus orci luctus et ultrices posuere cubilia Curae; Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris elit nisi, blandit et porttitor quis, malesuada nec mi.

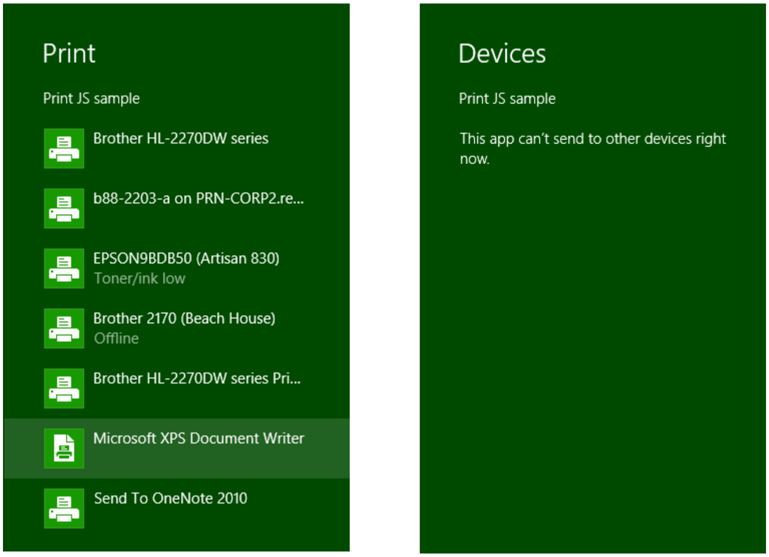
Aliquam erat volutpat. In non urna ut libero ultricies fringilla. Proin tellus neque, aliquam lacinia consequat at, vulputate et arcu. Maecenas odio nunc, lobortis sit amet pulvinar sit amet, accumsan et leo. Suspendisse erat lectus, commodo ac auctor eget, rutrum in mi. Suspendisse potenti. Proin ac elit non lacus rutrum mollis. Vivamus venenatis, tellus vel placerat lacinia, arcu ligula dignissim orci, consectetur consectetur eros massa vel nulla. Quisque malesuada iaculis ornare. Nullam tincidunt accumsan egestas. Mauris sit amet scelerisque arcu. Proin euismod sodales magna faucibus commodo. Nam in fringilla orci. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos himenaeos.

Microsoft

Copyright © Microsoft Corporation. All rights reserved.

Terms of use Trademarks Privacy Statement

FIGURE 17-7 Scenario 2 of the Print sample shows a typical app with something ready to print.



Print

Print JS sample

Brother HL-2270DW series

b88-2203-a on PRN-CORP2.re...

EPSON9BDB50 (Artisan 830)
Toner/ink low

Brother 2170 (Beach House)
Offline

Brother HL-2270DW series Pri...

Microsoft XPS Document Writer

Send To OneNote 2010

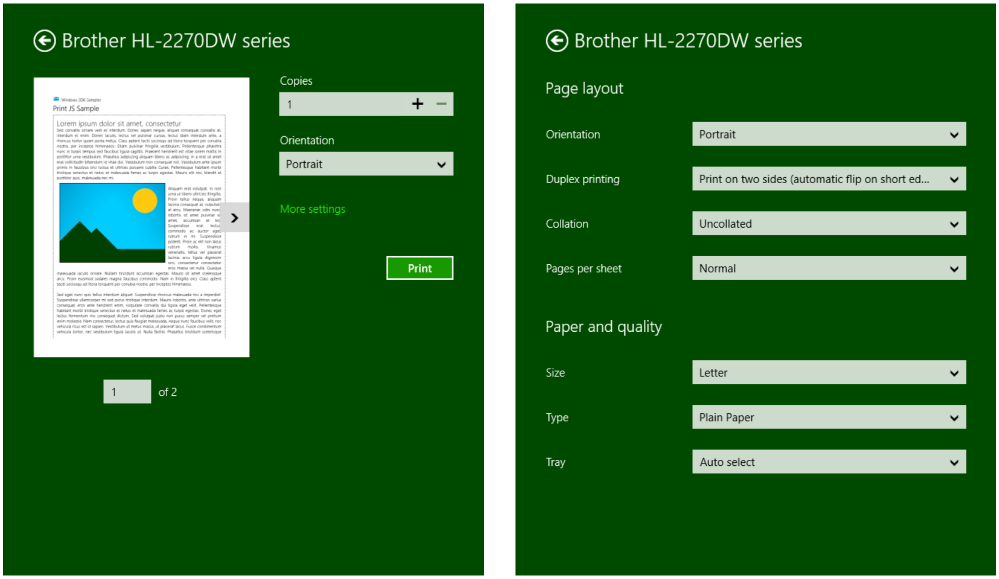
Devices

Print JS sample

This app can't send to other devices right now.

FIGURE 17-8 The Devices charm when an app has available print content (left) and when it doesn't (right).

From this point on, the system is just taking whatever content the app provides and displaying UI based on the capabilities of the printer driver, as shown in Figure 17-9 for my trusty Brother (actually, my real brother's name is Kevin). From the app's point of view, it thankfully gets all of this for free! The app can also indicate additional options to customize the UI, such as paper size and duplex printing, as shown in Figure 17-10, which comes from scenario 3 of the sample.



Copies: 1

Orientation: Portrait

Orientation: Portrait

Duplex printing: Print on two sides (automatic flip on short ed...

Collation: Uncollated

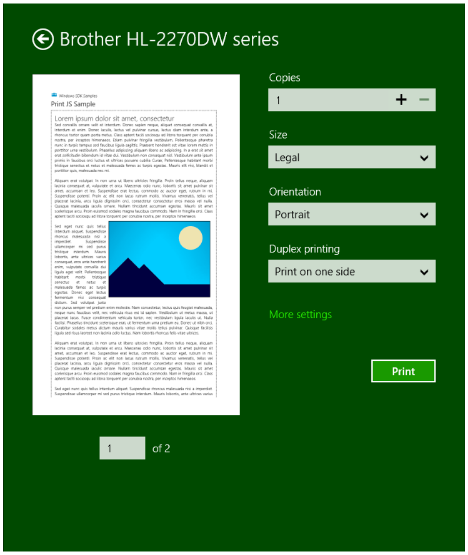
Pages per sheet: Normal

Size: Letter

Type: Plain Paper

Tray: Auto select

FIGURE 17-9 Print preview and printer options are shown once the user selects a printer. The More Settings link on the left is what opens the options pane on the right.



Copies: 1

Size: Legal

Orientation: Portrait

Duplex printing: Print on one side

FIGURE 17-10 The Print pane reflecting customization options indicated by the app.

Sidebar: Device Apps for Printers

Apps that are specifically associated with printers have the ability to extend the print settings in the UI and also participate in issuing print notifications to the user. For more details, refer to Windows Store device apps for printers along with the Print settings and print notifications sample in the SDK.

Print Document Sources

No matter where the user might want to print content, the important thing is to make that content ready for printing. The key API you need to know about here is not found in WinRT but in the `MSApp` object: `MSApp.getHtmlPrintDocumentSource`. I like the way the documentation once put it (since changed): "This method is used as the bridge between HTML and [Windows app] style printing. In other words, this is how an app dev says 'give me some stuff to print'." What you give it is an HTML *document* that contains your content.

I emphasize the word *document* here because what you pass to `getHtmlPrintDocumentSource` cannot be any arbitrary element in the DOM. It must be the same kind of thing that the `document` variable always points to, or else you'll see a run-time exception with "no such interface supported."

So where do you get such an object?

If what your app is showing on the screen is exactly what you want to print, you can just use the `document` object directly. This is what scenarios 1–3 of the Print sample do:

```
MSApp.getHtmlPrintDocumentSource(document);
```

Of course, you don't necessarily want to print everything on the screen; you can see that what's on the screen in Figure 17-7 and what appears in the print preview of Figure 17-9 and Figure 17-10 is different. This is where the `print` media query in CSS comes into play:

```
@media print {
   /* Print-only styles */
}
```

Simply said, if there's anything you don't want to show up in the printed output, set the `display: none` style within this media query. An alternate strategy, one that the sample employs, is to create a separate CSS file, such as css/print.css, and link it in your HTML file with the `media` attribute set to *print* (see html/scenario1.html):

```
   <link rel="stylesheet" type="text/css" href="/css/print.css" media="print" />
```

Print styles need not be limited to visibility of content: you can also use them however you like to arrange your content for more printer-friendly output. In a way, printing is like another view state where you're not adding to or changing the content, you're simply changing the visibility and layout. There are also some events you can use to do more specific formatting before and after printing has happened, as we'll see later.

But what if the content you want to print isn't your document object at all? How do you create another? There are several options here:

- In the document.body.onbeforeprint event handler, append additional child elements to the document and use the document.body.onafterprint event to remove them (the structure of such handlers is shown in scenario 2 of the Print sample). If your print CSS leaves only those newly added elements visible, that's all that gets printed. This very effectively controls the entire print output, such as adding additional headers and footers that aren't visible in the app. You might have a place in the app, in fact, where the user can configure those headers and footers.

- Call document.createDocumentFragment to obtain a document fragment and then populate it with whatever elements you want to print. getHtmlPrintDocumentSource accepts such a fragment. Note that if you want to drop in an HTML string (such as a piece acquired from a webview), insert it using outerHTML rather than innerHTML.

- If you have an iframe whose src is set to an SVG document (one of the tips we discussed for SVG's in Chapter 13), obtain that SVG document directly through the iframe element's contentDocument property. This too can be passed directly to getHtmlPrintDocumentSource and will print just that SVG, for example:

  ```
  <!-- in HTML -->
  <iframe id="diagram" src="/images/diagram.svg"></iframe>

  //In JavaScript
  var frame = document.getElementById("diagram");
  args.setSource(MSApp.getHtmlPrintDocumentSource(frame.contentDocument));
  ```

- If you want to print the contents of an altogether different HTML page, create a link element in the document head that points to that other page for print media (see below). This will redirect getHtmlPrintDocumentSource to process that page's content instead.

The latter is demonstrated in scenario 4 of the Print sample, where a link element is added to the document with the following code (js/scenario4.js):

```
var alternateLink = document.createElement("link");
alternateLink.setAttribute("id", "alternateContent");
alternateLink.setAttribute("rel", "alternate");
alternateLink.setAttribute("href", "http://go.microsoft.com/fwlink/?LinkId=240076");
alternateLink.setAttribute("media", "print");
document.getElementsByTagName("head")[0].appendChild(alternateLink);
```

Here the rel attribute indicates that this is alternate content, the media attribute indicates that it's only for print, and href points to the alternate content (id is optional). Note that if the target page has any print-specific media queries, those are certainly applied when creating the print source.

Providing Print Content and Configuring Options

Now that we know how to get a source for print content, it's straightforward to provide that content to Windows for printing. First, obtain the `Windows.Graphics.Printing.PrintManager` object:

```
var printManager = Windows.Graphics.Printing.PrintManager.getForCurrentView();
```

and then listen for its `printtaskrequested` event (a WinRT event), either through `addEventListener` or by assigning a handler as done in the sample:

```
printManager.onprinttaskrequested = onPrintTaskRequested;
```

If you don't add a handler for this event, the user will see the message on the right side of Figure 17-7 when invoking the Devices charm, unless you've also registered for other device-related events such as `Windows.Media.PlayTo.PlayToManager.sourceRequested` (see Chapter 13).

If you want to directly invoke printing from an app command, such as the Print button in scenario 2 of the sample, call the `PrintManager.showPrintUIAsync` method. This is equivalent to the user invoking the Devices charm when the app has registered for the `printtaskrequested` event.

The `printtaskrequested` event is fired when the Devices charm is invoked. In response, your handler creates a `PrintTask` object with a callback function that will provide the content document when needed. Here's how that works. First, your handler receives a `PrintTaskRequest` object that has just three members:

- `deadline` The date and time that indicates how long you have to fulfill the request.
- `getDeferral` Returns a `PrintTaskRequestedDeferral` object in case you need to perform any async operations to fulfill the request. As with all deferrals, you call its `complete` method when the async operation has finished.
- `createPrintTask` Creates a `PrintTask` with a given title and a function that provides the source document when requested.

The structure of `createPrintTask` is slightly tricky. Although it returns a `PrintTask` object through which you can set options and listen to task-related events, as we'll see shortly, its `source` property is read-only. So, instead of creating a task and storing your content document in this property, you provide a callback function that does the job when requested. The function itself is simple: it just receives a `PrintTaskSourceRequestedArgs` object whose `setSource` method you call with what you get back from `MSApp.getHtmlDocumentPrintSource`.

This is typically where you can also do other work to configure the task, so let's take an example from scenario 3 of the Print sample (where I've added a namespace variable for brevity):

```
function onPrintTaskRequested(printEvent) {
    var printTask = printEvent.request.createPrintTask("Print Sample", function (args) {
        args.setSource(MSApp.getHtmlPrintDocumentSource(document));

        // Choose the printer options to be shown. The order in which the options are
        // appended determines the order in which they appear in the UI
        var options = Windows.Graphics.Printing.StandardPrintTaskOptions;
        printTask.options.displayedOptions.clear();
        printTask.options.displayedOptions.append(options.copies);
        printTask.options.displayedOptions.append(options.mediaSize);
        printTask.options.displayedOptions.append(options.orientation);
        printTask.options.displayedOptions.append(options.duplex);

        // Preset the default value of the printer option
        printTask.options.mediaSize =
           Windows.Graphics.Printing.PrintMediaSize.northAmericaLegal;

        // Register the handler for print task completion event
        printTask.oncompleted = onPrintTaskCompleted;
    });
}
```

Note that PrintTaskSourceRequestedArgs also contains a getDeferral method, should you need it, along with a deadline.

Tip If you step through the code in your printtaskrequested handler but you pass the deadline, the print UI will time out and say there's nothing available to print. This might not be an error in the app at all—remove or disable the breakpoints and run the app again to check.

You can exercise some control over the appearance of the print UI through PrintTask.options, in which context you should review Guidelines for print-capable apps. The options object here, of type PrintTaskOptions, has a number of properties. A few obvious numerical ones are maxCopies, minCopies, and numberOfCopies. You can also call getPageDescription with a page number to obtain a PrintPageDescription with resolution information for that page.

Then there is a host of properties whose values come from various printing enumerations:

PrintTaskOptions Property	Windows.Graphics.Printing Enumeration
binding	PrintBinding
collation	PrintCollation
colorMode	PrintColorMode
duplex	PrintDuplex
holePunch	PrintHolePunch
mediaSize	PrintMediaSize
mediaType	PrintMediaType
orientation	PrintOrientation
printQuality	PrintQuality
staple	PrintStaple

PrintTaskOptions.displayedOptions, for its part, is a vector of strings that must come from the StandardPrintTaskOptions class, as shown in the earlier code. Each of these controls the visibility of the option in the print UI if, of course, the printer supports it (otherwise the option will not be shown). The full list of options is binding, collation, colorMode, copies, duplex, holePunch, inputBin, mediaSize, mediaType, nUp, orientation, printQuality, and staple.

Take special note of the mediaSize property, for which there are literally 172 different values in the PrintMediaSize enumeration that reflect all the sizes of paper, envelopes, and so forth that we find around the world. When you intend to market a print-capable Windows Store app in different regions, you might want to include mediaSize in displayedOptions and set its value to something that's applicable to the region (as the code above is doing for legal size paper). Even so, the media size is typically available in the More Settings panel in the print UI, depending on what the printer in question supports, so users will have access to it.

The final bit to mention in the code above is that a PrintTask has a completed event, along with previewing, progressing, and submitting. You can use these to reflect the status of print tasks in your app should you choose to do so. More information about the task itself is also available through its properties, which will typically contain the title you gave to the print job along with a unique ID. In all of this, however, you might have noticed a conspicuous absence of any method in PrintTask that would cancel a print job—in fact, there is none. This is because the HTML print model, as presently used by Windows Store apps written in JavaScript, is an all-or-nothing affair: once the job gets into the print engine, there's no programmatic means to stop it. The user can still go to the printer control panel on the desktop and cancel the job there, or revert to the old-school method of yanking out the paper tray, but at present an app isn't able to provide such management functions itself.

Protocol APIs: HID, USB, Bluetooth, and Wi-Fi Direct

Whereas the scenario APIs in WinRT that we saw earlier in this chapter provide a focused interface to a set of specific devices, the *protocol APIs* give you access to the much broader ecosystem of gadgets and gizmos. So long as a device adheres to one of the supported standards—Bluetooth (RFCOMM/SPP), Bluetooth Smart (LE/GATT), USB HID, USB (custom devices), and Wi-Fi Direct—you can use one of these APIs to communicate with it. The only exceptions are for those devices that already have a dedicated scenario API: where such support exists, as with barcode scanners, magnetic stripe readers, printers, sensors, mice, keyboards, and so on, Windows automatically blocks access through the protocol APIs.

To be honest, one could (and I hope someone does!) write a whole book on the subject of these protocol APIs, so in this chapter my intent is to give you enough of an introduction to familiarize you with the possibilities and point you to additional resources. Toward that end I start each of the following sections with a summary list of the device capability name, the primary namespace and object class in WinRT for that protocol, applicable samples in the SDK, background documentation, and the applicable talk from //build 2013 where all of these APIs were introduced. Where I happen to have some suitable hardware available to me I'll go into more detail about the protocol API in question; in other cases I'll just give a shorter overview.

For the sake of efficiency, let me also point out the common characteristics of the protocol APIs so that I don't have to bore you later by repeating myself five times!

The first commonality is, of course, declaring device access in the manifest. For all protocol APIs other than Wi-Fi Direct, this takes the following form:

```
<m2:DeviceCapability Name="[capability]">
  <m2:Device Id="[id]">
      <m2:Function Type="[usage]" />
  </m2:Device>
</m2:DeviceCapability>
```

where `m2` is the Windows 8.1 manifest namespace, *[capability]* identifies the protocol, such as *humaninterfacedevice, [id]* is a string such as *any* (for "broad" devices) or *vidpid:045E 0610* to identify a specific ("narrow device) vendor/product id, and *[usage]* identifies the specific operational interface to that device.

If you are declaring access to multiple devices, you can have multiple `m2:Device` entries under the same capability. If you are declaring access to multiple devices with different capabilities, you'll have multiple `m2:DeviceCapability` entries.

Device discovery is also the same as we're seen earlier, where each protocol API class has a static `getDeviceSelector` method that you pass to the `DeviceInformation.findAllAsync` or `createWatcher` to obtain a specific `DeviceInformation` object. You then pass its `id` property to the static `fromIdAsync` method of the protocol API class to instantiate the object. Here's an example using the Wi-Fi Direct API, but the pattern is the same for all protocol APIs:

```
var namespace = Windows.Devices.WiFiDirect;
var deviceClass = Windows.Devices.WiFiDirect.WiFiDirectDevice;

var deviceSelector = deviceClass.getDeviceSelector();

Windows.Devices.Enumeration.DeviceInformation.findAllAsync(deviceSelector, null).done(
    function (devices) {
    // Set up selection UI
}

// When a device is selected from your UI list, create an object instance using its id.
// This will prompt the user for consent. If consent is denied, fromIdAsync will succeed
// but the deviceInstance variable will be null. (Other errors can also occur that will
// invoke your error handler.
deviceClass.fromIdAsync(selectedId).done(
    function (deviceInstance) {
        // Do your stuff!
    }
);
```

Beyond this there is one more common characteristic: in your `suspending` handler you must call a device object's `close` method and null out your variable, and then reacquire the object in your `resuming` handler. For example:

```
Windows.UI.WebUI.WebUIApplication.addEventListener("suspending", function () {
    // Stop all existing I/O

    // Close the device instance
    deviceInstance && deviceInstance.close();
    deviceInstance = null;
});

Windows.UI.WebUI.WebUIApplication.addEventListener("resuming", function () {
    // Reacquire the device object
    deviceInstance = acquireDeviceInstance();
})
```

The *acquireDeviceInstance* function would call `deviceClass.fromIdAsync` again, as shown in the previous bit of code, and reset the *deviceInstance* variable. The reason for this suspending behavior that Windows will automatically invalidate and turn off a device when it's not being used by a foreground app. By closing it and nulling out your variable, you remind yourself that you must reacquire the device when you resume and thus avoid exception.

It's also worth mentioning that if you're building devices of your own that apps could work with through these protocol APIs, consider building a WinRT component that supplies a focused scenario API of your own. The component, in other words, would provide a higher-level interface to your device's specific capabilities and make it much easier for developers to write apps for your device. We'll be looking at WinRT components in Chapter 18 and implementing an example for this purpose.

Human Interface Devices (HID)

Manifest capability name: `humaninterfacedevice`
Protocol API namespace: `Windows.Devices.HumanInterfaceDevice`
Primary class: `HidDevice`
SDK Sample(s): Custom HID device access sample, XBoxController and CircusCannon examples in the companion content
Background docs: Human Interface Devices (MSDN) and Human interface device (Wikipedia)
Applicable //build 2013 session: Apps for HID Devices (2-924b)

It's highly likely you've been using many HID devices already for a long time—this category includes most mice and other pointing devices, keyboards, front-panel controls, game controllers, and sensors. It's just that those devices already have dedicated APIs of their own, such as those in Chapter 12, so you don't need to (and cannot, in fact) talk to them on a lower level. Barcode readers and magnetic stripe readers are also HID devices, but they too already have a scenario API. In short, this protocol API is intended for custom external peripheral devices, and it blocks a number of HID usages, as listed on Supporting human interface devices.

Technically speaking, HID is a device class over the more generic USB protocol (and others like I2C), but it has become specialized enough to be treated as its own thing. One reason for this is that HID devices generally work with small bits of data, as opposed to general USB devices like external hard drives that can deal with much larger amounts.

Through HID, you communicate with a device using *reports*, which are simply the HID term for blobs of binary data that go back and forth. The three types of reports are *input* (data coming from the device), *output* (data going to the device), and *feature* (for configuration and settings). Generally speaking, a report provides values for Boolean and Numeric *controls*, which essentially translate to pieces of the hardware that have on/off states (like buttons) and variable states (like joysticks or trackballs), respectively. A *report descriptor* is then what describes the format and the meaning of those blobs of data should you want to interpret reports dynamically. Most frequently, however, you'll write your code to work directly with specific report structures.

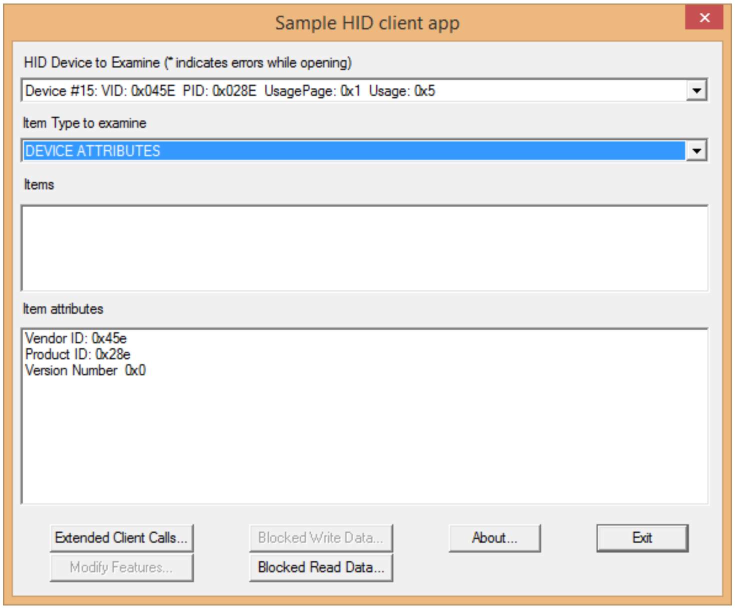
Beyond that, this book is not the place to give you all the background, which you can find in the section of the documentation. Here, we just want to look at the `HumanInterfaceDevice` API and its usage, namely that of the `HidDevice` class.

With manifest declarations you use the `humaninterfacedevice` capability, beneath which you must declare the specific device ID and its usage (function). For example, in the Custom HID device access sample in the SDK, it declares access to a test device known as SuperMutt (see MUTT (Microsoft USB Test Tool) devices on MSDN):

```
<m2:DeviceCapability Name="humaninterfacedevice">
  <!--SuperMutt Device-->
  <m2:Device Id="vidpid:045E 0610">
    <m2:Function Type="usage:FFAA 0001" />
  </m2:Device>
</m2:DeviceCapability>
```

The device ID is always in the format *vidpid:<vid> <pid>* (meaning vendor ID and product ID); the function type is always in the format *usage:<usagepage> <usageid>*. See How to specify device capabilities for HID.

Although you can find some of these hexadecimal values through Device Manager, the best way is to use the HCLIENT tool that's a sample in the Windows Driver Development Kit (WDK). (Note that the WDK requires Visual Studio 2013 Ultimate and does not work with the express editions; see How to get the WDK for installation instructions.) Once you build and run this tool, plug in your device and see what gets added to the topmost drop-down. In the image below, I plugged in an Xbox 360 game controller and saw that Device #15 appeared:



HID Device to Examine (* indicates errors while opening)

Device #15: VID: 0x045E PID: 0x028E UsagePage: 0x1 Usage: 0x5

Item Type to examine

DEVICE ATTRIBUTES

Items

Item attributes

Vendor ID: 0x45e
Product ID: 0x28e
Version Number 0x0

Therefore, to access this device, I need the following in my manifest:

```
<m2:DeviceCapability Name="humaninterfacedevice">
  <m2:Device Id="vidpid:045E 028E">
    <m2:Function Type="usage:0001 0005" />
  </m2:Device>
</m2:DeviceCapability>
```

Similarly, I have a Dream Cheeky Circus Cannon on my desk, whose IDs and usage are as follows (see the CircusCannon example):[123]

```
<m2:DeviceCapability Name="humaninterfacedevice">
  <m2:Device Id="vidpid:1941 8021">
    <m2:Function Type="usage:FFA0 0001" />
  </m2:Device>
</m2:DeviceCapability>
```

In a number of cases, you can write an app to work with a group of devices that all use the same commands. Then you use `any` instead for the device ID and a "top level" usage of `*`:

```
<m2:DeviceCapability Name="humaninterfacedevice">
  <m2:Device Id="any">
    <m2:Function Type="usage:0005 *" />
  </m2:Device>
```

[123] This is a fun device, though I had to disassemble it and unjam two of the three motors to get it to work. Nevertheless, understanding the internal mechanisms and how they related to the input reports was also very helpful when creating a WinRT component as a device library, which we'll see in Chapter 18.

```
</m2:DeviceCapability>
```

With your manifest declaration in place, use HidDevice.getDeviceSelector as the basis for enumeration. This method has two variants, one that takes the usage page and usage value only (as when your device ID in the manifest is any) and one that takes usage value plus a *vid* and *pid* (for when you declare a specific device). Either way, send the selector to DeviceInformation.findAllAsync or createWatcher to get a specific DeviceInformation object, whose id property you can then pass to HidDevice.fromIdAsync to instantiate the object. You can see this in scenario 1 of the Custom HID device access sample, although you have to step through quite a bit of the code to see the sequence because it uses a somewhat involved object model. So let me instead offer something simpler from the XboxController example in this chapter's companion content, where we just use the first device of this type that we find (if any; js/default.js):

```
function acquireController() {
    var xbcSelector = hid.getDeviceSelector(0x0001, 0x0005, 0x045E, 0x028E);
    var id = null;
    var name = null;

    //Notice the second null argument (required in JavaScript when using a selector)
    Windows.Devices.Enumeration.DeviceInformation.findAllAsync(xbcSelector, null)
        .then(function (devInfo) {
        //If no devices are found, throw an error out to our chain's handler
        if (devInfo.size == 0) {
            throw "no devices found";
        }

        //If we find any, assume we just have one such device on the system and use it.
        id = devInfo[0].id;
        name = devInfo[0].name;
        return hid.fromIdAsync(id, Windows.Storage.FileAccessMode.read);
    }).done(function (device) {
        if (device == null) {
            //Device could be enumerated but not accessed; user likely denied consent.
            var status = Windows.Devices.Enumeration.DeviceAccessInformation.createFromId(id);
            txtOutput.innerText = "Device exists but not acquired. Status: " +
                statusString(status.currentStatus);  //statusString is a helper (not shown)
        } else {
            controllerDevice = device;
            txtOutput.innerText = "Device aquired: " + name;
        }
    }, function (e) {
        //Some other error happened
        txtOutput.innerText = "Error acquiring device: " + e;
    });
}
```

I use nearly identical code in the CircusCannon example, just changing the arguments to gerDeviceSelector to match that device. Note also that the FileAccessMode value you pass to fromIdAsync determines whether the device can be shared (read) or is open for exclusive access (readwrite). Exclusive access is necessary if you want to send output reports (that is, write) to the device, as with the Circus Cannon.

Now that we have a HidDevice instance for this device in hand, its usageId, usagePage, vendorId, productId, and version are all available as properties. We also must remember to close and release the instance on suspend/resume. Most importantly, though, we can start talking to it through its methods and events:

Method	Description
getInputReportAsync	Retrieves an input report from the device, with the result of HidInputReport.
getFeatureReportAsync	Retrieves a feature report from the device with the result of HidFeatureReport.
createOutputReportAsync, sendOutputReportAsync	Creates and sends a HidOutputReport object to the device, respectively.
createFeatureReportAsync, sendFeatureReportAsync	Creates and sends a HidFeatureReport object to the device, respectively.
getBooleanControlDescriptions getNumericControlDescriptions	Retrieves descriptions for Boolean and numeric controls for the device. With both calls you specify a HidReportType (input, output, feature), the usagePage, and usageId. Return values are vectors of HidBooleanControlDescription and HidNumericControlDescription objects, respectively.
Event	**Description**
inputreportreceived	Fires when the device generates an input, if supported, providing a HidInputReportReceivedEventArgs whose single property, report, is a HidInputReport object.

In short, if you want to read information from the device, use getInputReportAsync and/or the inputreportreceived event; the XboxController example with this chapter uses the event, for instance, to watch for any changes in the hardware. If you want to send specific reports to the device, use createOutputReportAsync or createFeatureReportAsync, followed by sendOutputReportAsync or sendFeatureReportAsync.

What exactly you can do with input, output, and feature reports depends on the device you're working with, of course, but there are some general characteristics. The HidInputReport, HidOutputReport, and HidFeatureReport objects all share most of their properties and methods in common, with the input report having two extra properties:

Report Property	Description
id	The report identifier.
data	A WinRT Buffer object containing the raw data associated with the report. You read information from the Buffer using the DataReader object, and write information using the DataWriter object.
activatedBooleanControls	HidInputReport only: a vector view of HidBooleanControl objects, identifying those Boolean controls on the device that are turned on.
transitionedBooleanControls	HidInputReport only: a vector view of HidBooleanControl objects, identifying those Boolean controls on the device that have changed since the last report.
Report Method	**Description**
getBooleanControl getBooleanControlByDescription	Returns a HidBooleanControl object for a given usage page and usage id, or a

	HidBooleanControlDescription.
getNumericControl getNumericControlByDescription	Returns a HidNumericControl object for a given usage page and usage id, or a HidNumericControlDescription.

If a device raises interrupts that cause the HidDevice object to fire its inputreportreceived event, your event handler will receive a HidInputReport object in eventArgs.report. For the Xbox 360 controller, as demonstrated in the XboxController example for this chapter, the input report contains all the button states, stick positions, and so forth. Some of these states come through in a single bit in the report, so it's important to know the exact report structure. To help illustrate this, the example's code in the inputReportReceived event handler (js/default.js), shows both the raw hex dump of the report along with a more human-readable deconstruction. I'll leave you to look at that code for the details.

The CircusCannon also issues ongoing inputreportreceived events to report the status of its internal switches. Four bits in the second and third bytes of the report represent switches that indicate that left, right, up, or down motion has reached its mechanical limit (thereby closing a switch). Similarly, there's a switch that's closed and then opened each time a missile fires, which shows up as a bit in the third byte of the report. We can use these bits, then, to automatically stop the movement or firing mechanisms as shown below, where the CircusCannon namespace contains the various constants and bit masks for the device (js/default.js):

```
function inputReportReceived(e) {
    var reader = Windows.Storage.Streams.DataReader.fromBuffer(e.report.data);
    var report = new Uint8Array(e.report.data.length);
    reader.readBytes(report);

    var upDown = report[CircusCannon.offset.upDown];
    var leftRight = report[CircusCannon.offset.leftRight];

    var atLimit = (upDown & (CircusCannon.status.topLimit | CircusCannon.status.bottomLimit))
        || (leftRight & (CircusCannon.status.leftLimit | CircusCannon.status.rightLimit));

    var missileFired = report[CircusCannon.offset.missile] & CircusCannon.status.missileFired;

    //Stop movement if we're at a limit, or stop the firing after one shot.
    if (atLimit || missileFired) {
        sendCommand(CircusCannon.commands.stop);
    }
}
```

In working with the device, I found that the limit switches can stay closed even after movement in an opposite direction has started, so turning this kind of input into a good user experience means ignoring some of the reports until the switch is reset. The example for this chapter does not do such debouncing, so you generally have issue multiple movement commands through the UI until the switch is opened. We'll correct this shortcoming in the control component we create in Chapter 18.

Controlling the motion and firing mechanisms of the cannon is done with a simple 9 byte output report, with report id of 0 and the relevant command in the second byte:

```
WinJS.Namespace.define("CircusCannon", {
    commands: {
        stop: 0x00,
        up: 0x01,
        upSlow: 0x0D,
        down: 0x02,
        downSlow: 0x0E,
        left: 0x04,
        leftSlow: 0x07,
        right: 0x08,
        rightSlow: 0x0B,
        fire: 0x10,
        upLeft: 0x01 + 0x04,
        upRight: 0x01 + 0x08,
        downLeft: 0x02 + 0x04,
        downRight: 0x02 + 0x08,
        nop: 0xFF
    },

    //Other members omitted
}

function sendCommand(command) {
    if (launcher == null) {
        return;
    }

    var reportId = 0x00;
    var report = launcher.createOutputReport(reportId);

    var dataWriter = new Windows.Storage.Streams.DataWriter();
    var packet = new Uint8Array([reportId, command, 0, 0, 0, 0, 0, 0, 0]);
    dataWriter = writeBytes(packet);
    report.data = dataWriter.detachBuffer();

    try {
        launcher.sendOutputReportAsync(report);
    } catch (e) {
        console.log(e);
    }
}
```

With this code, controlling the device is as easy as calling `sendCommand` with a value like `CircusCannon.commands.fire`. To create a good user experience out of it, however, it's helpful to build a higher-level API on top of the input and output reports so that we don't have to deal with such low-level minutiae. Again, we'll come back to this in Chapter 18.

Finally, as an example of a feature report, the SuperMutt device supports an LED blink pattern report, as demonstrated in scenario 2 of the Custom HID device access sample. Here you can also see the use of the DataReader and DataWriter objects to work with the data buffers in the report (js/scenario2_featureReports.js). :

```
getLedBlinkPatternAsync: function () {
    return SdkSample.CustomHidDeviceAccess.eventHandlerForDevice.current
    .device.getFeatureReportAsync(SdkSample.Constants.superMutt.ledPattern.reportId)
    .then(function (featureReport) {
        if (featureReport.data.length === 2) {
            var reader = Windows.Storage.Streams.DataReader.fromBuffer(featureReport.data);

            // First byte is always report id
            var reportId = reader.readByte();
            var pattern = reader.readByte();

            WinJS.log && WinJS.log("The Led blink pattern is " + pattern, "sample", "status");
        }
    });
},

// Set the blink pattern (see full sample comments for details). Note that creating
// a feature report nulls out all data in the report.
setLedBlinkPatternAsync: function (pattern) {
    var featureReport = SdkSample.CustomHidDeviceAccess.eventHandlerForDevice.current
        .device.createFeatureReport(SdkSample.Constants.superMutt.ledPattern.reportId);

    var writer = new Windows.Storage.Streams.DataWriter();
    writer.writeByte(featureReport.id);
    writer.writeByte(pattern);

    featureReport.data = writer.detachBuffer();
    return SdkSample.CustomHidDeviceAccess.eventHandlerForDevice.current
        .device.sendFeatureReportAsync(featureReport).then(function (bytesSend) {
    });
}
```

The other scenarios in the sample, as you would expect, exercise additional capabilities.

Custom USB Devices

Manifest capability name: usb
Protocol API namespace: Windows.Devices.Usb
Primary class: UsbDevice
SDK Sample(s): Custom USB device access sample, USB CDC Control sample (also Firmware Update USB device sample for dedicated device apps)
Background docs: Windows Store app for a USB device, Writing Apps for USB devices
Applicable //build 2013 session: Apps for USB Devices (3-924a)

Whereas HID covers a wide range of devices that transfer small amounts of data, the USB protocol API covers a much wider swath of device classes, namely those that don't belong to a device class or those that rely on the generic WinUSB driver (which is a requirement for this API). The specific list of supported class codes (along with subclass and protocol codes) is found on [How to add USB device capabilities to the app manifest](). Those classes include CDC control, Physical, PersonalHeathCare, ActiveSync, PalmSync, DeviceFirmwareUpdate (for dedicated device apps), IrDA (infrared), Measurement, and vendor-specific classes. Audio/video, imaging, printing, storage, smart card, and wireless controller classes are not supported through this API because they already have scenario APIs of their own. The API is also intended solely for peripherals and thus blocks access to internal devices.

Overall, the subject of general USB devices is quite deep, so I very much recommend the documentation linked to earlier for a complete treatment of the subject. Here I'll just summarize.

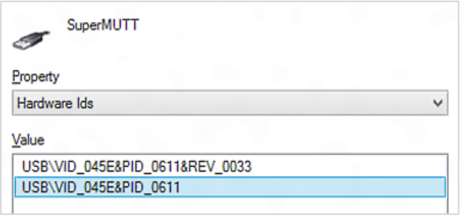
In the manifest, you can declare either a name, class id, or a WinUsbId GUID for the `m2:Function` element. The Custom USB device access sample shows the first two:

```
<m2:DeviceCapability Name="usb">
  <!--OSRFX2 Device-->
  <m2:Device Id="vidpid:0547 1002">
    <m2:Function Type="classId:ff * *"/>
  </m2:Device>
  <!--SuperMutt Device (USB interface, not HID) -->
  <m2:Device Id="vidpid:045E 0611">
    <m2:Function Type="name:vendorSpecific"/>
  </m2:Device>
</m2:DeviceCapability>
```

The *classId* function string maps to the class, subclass, and protocol for the device. It can be in one of these formats: *classId:nn * **, *classId:nn 00 **, or *classId:nn 00 00*, where *nn* is a two-digit hexadecimal value. A name is whatever the device uses, and a GUID is specified with the form *winUsbId:<GUID>*.

The device ID can be a *vidpid*, as we see above, or *any*. The latter, *any*, can be used only when the *classId* also identifies a specific (or narrow) device class, namely those whose class id's start with 02 (CDC), 0A (CDC-data), EF (miscellaneous), FE (application-specific), and FF (vendor specific). It cannot be used for *winUsbId*, serial-over-USB, miscellaneous, app-specific, or vendor-specific device classes.

To find all of this information, you can simply open the desktop Device Manager and view the properties for the device. The Hardware Ids (below left) give you what you need for the `m2:Device` element, and the Compatible Ids (below right) show the values you can use for *classId*:

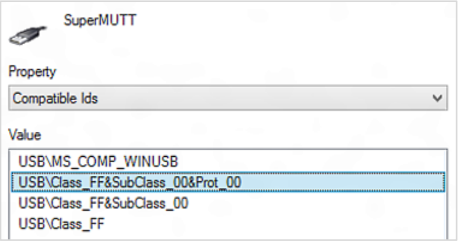


SuperMUTT

Property: Hardware Ids

Value:

USB\VID_045E&PID_0611&REV_0033
USB\VID_045E&PID_0611



SuperMUTT

Property: Compatible Ids

Value:

USB\MS_COMP_WINUSB
USB\Class_FF&SubClass_00&Prot_00
USB\Class_FF&SubClass_00
USB\Class_FF

Connecting to a device through the process of enumeration and `UsbDevice.fromIdAsync` gets you an instance of the [`UsbDevice`]() class. With that in hand, you can retrieve the device's configuration descriptor set through its `deviceDescriptor` property, which conveniently parses the descriptor information into a `UsbDeviceDescriptor` object. Custom descriptors will be included here, if they exist. Standard descriptors can be decoded using the static `tryParse` methods of three other specific classes: `UsbConfigurationDescriptor`, `UsbInterfaceDescriptor`, and `UsbEndpointDescriptor`. See How to get USB descriptors.

You then have three supported ways to communicate with the device, as described in the following table (isochronous transfers are not supported). I especially encourage you to read the linked documentation—those topics include detailed diagrams and step-by-step guides to the communication model.

Transfer Type	Core UsbDevice members	Description
Control (scenario 2 in the SDK sample)	`sendControlInTransferAsync` (read) `sendControlOutTransferAsync` (write) `selectSettingAsync`	Read or write configuration information or perform control-specific functions. See How to send a USB control transfer. Information is send within `UsbSetupPacket` objects with optional `Buffer` objects (for which you use the `DataReader` and `DataWriter` classes).
Interrupt (scenario 3)	`defaultInterface.interruptInPipes` (read) `configuration.interruptInPipes` (read) `defaultInterface.interruptOutPipes` (write) `configuration.interruptOutPipes` (write) `datareceived` event on pipes	Notifications that arise from periodic polling in the device. See How to send a USB interrupt transfer request. Transfers happens through `UsbInterruptInPipe` and `UsbInterruptOutPipe` objects. Data is transferred using their respective `inputStream` and `outputStream` properties using the `DataReader` and `DataWriter` classes.
Bulk (scenario 4)	`defaultInterface.bulkInPipes` (read) `configuration.bulkInPipes` (read) `defaultInterface.bulkOutPipes` (write) `configuration.bulkOutPipes` (write)	Perform high volume data transfer with error detection and limited retries. Data is transferred when there is unused bandwidth on the bus, so should not be used for time-critical transfers. See How to send a USB bulk transfer request. Bulk endpoints (which are unidirectional) are represented by `UsbBulkInPipe` and `UsbBulkOutPipe` objects; data is transferred using their respective `inputStream` and `outputStream` properties using the `DataReader` and `DataWriter` classes.

For further information on USB devices, again refer to the documentation I've linked here along with Concepts for app USB developers and Talking to USB devices, start to finish.

Bluetooth (RFCOMM)

Manifest capability name: `bluetooth.rfcomm` or `vidpid: <vid> <pid> bluetooth`
Protocol API namespace: `Windows.Devices.Bluetooth.Rfcomm`
Primary class: `RfcommDeviceService`
SDK Sample(s): Bluetooth Rfcomm chat sample; also see the SpheroColorController example in the companion content
Background docs: Supporting Bluetooth devices, Bluetooth Devices
Applicable //build 2013 session: Apps for Bluetooth, HID, and USB Devices (3-026)

When we talk about Bluetooth devices we have to be clear about what specification they support. Bluetooth RFCOMM or SPP (Serial Port Protocol), means the 1.x, 2.x, and 3.x specifications (sometimes also referred to as Bluetooth "classic" or "legacy"). This is for high-speed streaming devices with higher power consumption, such as a pair of wireless speakers, a Bluetooth printer, or connecting your phone to your car's audio system. Bluetooth Smart, LE (low energy), and GATT (Generic Attribute Profile using LE as a transport), on the other hand, use the 4.0 specification for small or slow data transfers with lower power devices, such as heart rate monitors that you charge up only once a week. In short, the two are really separate things, something like the Java and JavaScript programming languages! (And to make it even more confusing, the Bluetooth Smart Ready logo means that the device supports all Bluetooth specs, whereas Bluetooth Smart is just 4.0. There will be a quiz next week!)

Here, we'll talk about working with Bluetooth RFCOMM devices, such as the Sphero, and we'll look at Bluetooth Smart (LE) after that. In both cases, the APIs work with devices that have already been paired with Windows, and fortunately such pairing capabilities are already built into the system. That is, if the user has gone to PC Settings > PC and Devices > Bluetooth and paired the device, your app will find it in the process of enumeration.

As indicated above, the capability name for Bluetooth RFCOMM is, not surprisingly, `bluetooth.rfcomm`, or you can use the `vidpid: <vid> <pid> bluetooth` form. In the manifest, the device id is generally *any* and the function is either *serviceId:<GUID>* where <GUID> is specific to the device or a descriptive form like *name:serialPort*.

To talk to the Sphero device, for instance, you'd use the following declaration, as shown in the SpheroColorController example in this chapter's companion content:

```
<m2:DeviceCapability Name="bluetooth.rfcomm">
  <m2:Device Id="any">
    <m2:Function Type="serviceId:00001101-0000-1000-8000-00805F9B34FB" />
  </m2:Device>
</m2:DeviceCapability>
```

You can also use the serial port profile by name (which maps to the GUID above):

```
    <m2:Function Type="name:serialPort" />
```

The Bluetooth Rfcomm chat sample in the SDK provides another example:

```
<m2:DeviceCapability Name="bluetooth.rfcomm">
  <m2:Device Id="any">
    <m2:Function Type="serviceId:34B1CF4D-1069-4AD6-89B6-E161D79BE4D8" />
  </m2:Device>
</m2:DeviceCapability>
```

In this case the *serviceId* GUID is just an arbitrary GUID that's been assigned to this particular app, because the sample doesn't work with an external device. It instead sets up a chat session between two PCs running the same sample.

For the full list of supported devices for Bluetooth RFCOMM, see [How to specify device capabilities for Bluetooth](). This also includes a list of devices that aren't supported through this protocol API, because other APIs already exist for them.

Enumeration and discovery, as usual, starts with a selector from `RfcommDeviceService.getDeviceSelector` and ends with `RfcommDeviceService.fromIdAsync` to get a specific instance of `RfcommDeviceService`. That instance has the following methods and properties:

Property	**Description**
`serviceId`	Retrieves the `RfcommServiceId` for the device.
`connectionHostName` `connectionServiceName`	The `HostName` endpoints with which to create a socket connection to the device.
`protectionLevel` `maxProtectionLevel`	Retrieves the current and maximum `SocketProtectionLevel` that describes the level of security in the Bluetooth link layer.
Method	**Description**
`getSdpRamAttributesAsync`	Retrieves Service Discovery Protocol (SDP) attributes for the device.

As suggested by its connection properties, the `RfcommDeviceService` instance has no dedicated communication methods of its own: you instead use the WinRT sockets APIs that are described in Appendix D, "Provider-Side Contracts." Here's a generic piece of code where `deviceObj` is the device instance, derived from the Bluetooth Rfcomm sample (js/s1-chat-client.js):

```
var sockets = Windows.Networking.Sockets;
var streams = Windows.Storage.Streams;

socket = new sockets.StreamSocket();
socket.connectAsync(deviceObj.connectionHostName, deviceObj.connectionServiceName,
    sockets.SocketProtectionLevel.plainSocket)
    .done(function () {
        // Device is connected and ready to use.

        // Send data to the device through writer.writeBytes, etc. and storeAsync.
        writer = new streams.DataWriter(socket.outputStream);

        // Send data to the device through reader.readerBytes, etc. and loadAsync.
        var reader = new streams.DataReader(socket.inputStream);
    });
```

As a more specific example, here's how the SpheroColorController example generates a random color packet and sends it to the device, using a socket and `DataWriter` as done in the code above (js/default.js):

```
// writer is a DataWriter instance for the socket
function changeColor() {
    if (writer == null) {
        return;
    }
    var packet = generateRandomColorPacket();
    writer.writeBytes(packet);
    writer.storeAsync().done(function () {
    }
}

function generateRandomColorPacket() {
    var r = Math.floor(Math.random() * 256);
    var g = Math.floor(Math.random() * 256);
    var b = Math.floor(Math.random() * 256);

    //Checksum is the lower 8 bits of the packet contents minus the 0xFFFE prefix
    var checksum = (0x02 + 0x20 + 0x01 + 0x05 + r + g + b + 0x01) % 256;

    return new Uint8Array([0xFF, 0xFE, 0x02, 0x20, 0x01, 0x05, r, g, b, 0x01, ~checksum]);
}
```

Tip When I first converted Ellick Sung's C# Sphero sample (from his //build 2013 talk linked at the beginning of this section) to JavaScript, I was hitting an exception in DataWriter.storeAsync that told me only "The operation identifier is not valid." Say what? The problem, which was really hard to see, was a simple casing error: I'd used socket.OutputStream (which resolved to undefined) when constructing the DataWriter instead of the valid socket.outputStream. However, the DataWriter does not validate this backing stream during construction, and the stream isn't otherwise touched until storeAsync, which is when I saw the exception. So, if you're seeing odd behavior like this, check your constructor arguments! (Thanks to Matthew Beaver and Nigel D'Souza for helping track this down!)

Finally, as a reminder of properly cleaning up our objects when the app is suspended, here's the code that disconnects from the device, the socket, and the DataWriter (js/default.js):

```
var wui = Windows.UI.WebUI.WebUIApplication;

wui.addEventListener("suspending", function (e) {
    disconnect();
});

function disconnect() {
    output.innerText = "";

    writer && writer.close();
    writer = null;

    socket && socket.close();
    socket = null;

    device = null;
}
```

For good measure the example saves the last device id that it connected to so that we can automatically reconnect when the app is resumed:

```
var rfc = Windows.Devices.Bluetooth.Rfcomm;
```

```
wui.addEventListener("resuming", function (e) {
    //Attempt to reconnect to previously enumerated device
    if (lastDeviceId != 0) {
        connectToDevice(lastDeviceId);
    }
})

function connectToDevice(id) {
    rfc.RfcommDeviceService.fromIdAsync(id).done(function (foundDevice) {
        device = foundDevice;
        openSocket();
    });
}
```

Bluetooth Smart (LE/GATT)

Manifest capability name: `bluetooth.genericAttributeProfile`
Protocol API namespace: [`Windows.Devices.Bluetooth.GenericAttributeProfile`]
Primary class: [`GattDeviceService`]
SDK Sample(s): [Bluetooth Generic Attribute Profile – Heart Rate Service sample]
Background docs: [Bluetooth Low Energy Overview], [Bluetooth low energy (Wikipedia)]
Applicable //build 2013 session: [Apps for Bluetooth Smart Devices (3-9028)]

Having previously introduced the various flavors of Bluetooth—which you can go back and read if you want!—we'll waste no more time here and just dive straight into the considerations for Bluetooth Smart, LE (Low Energy), and GATT (Generic Attribute Profile) devices. These are again those that work with small amounts of data transfer in exchange for long battery life, and as with other Bluetooth devices, Windows provides a built-in pairing UI so that apps needn't worry about that part of the story. Of course, user consent is involved whenever an app attempts to connect to a device the first time.

The communication model for GATT devices centers around *characteristics*, which is a data value transferred between the device and a client, such as a heart rate reading, battery voltage, temperature, and so forth. Optional *descriptors* provide information about a characteristic, such as units, min/max values, etc. A GATT service is then a hierarchal collection of related characteristics and descriptors; most devices will provide more than one characteristic, for instance, a battery level along with a specific reading like heart rate or thermometer. Collectively, services, characteristics, and descriptors are all referred to as *attributes* that each have some kind of GUID assigned to it.

In the app manifest, the general device capability name is `bluetooth.genericAttributeProfile`, where the device id is typically `any`, `model`, or a *vidpid*. The service id can be a string with `serviceId:<GUID>` or `name:<service_name>`, same as with Bluetooth RFCOMM. The SDK sample, for instance, asks to work with Bluetooth LE heart rate monitors:

```
<Capabilities>
  <m2:DeviceCapability Name="bluetooth.genericAttributeProfile">
    <m2:Device Id="any">
      <m2:Function Type="name:heartRate" />
    </m2:Device>
  </m2:DeviceCapability>
</Capabilities>
```

For the full list of supported devices for Bluetooth RFCOMM, see How to specify device capabilities for Bluetooth. Note that the GATT service for HID is not supported, as you'd use the HID API for that.

Enumeration and discovery starts with a selector from `GattDeviceService.getDeviceSelector` and ends with `GattDeviceService.fromIdAsync` to get a specific instance of `GattDeviceService`. This is the object representing the device's service, and through it you can get the `deviceId` as well as the service's `uuid`. Two methods, `getIncludedServices` and `getCharacteristics`, let you then explore the device's capabilities, along with the `attributeHandle` property.

Characteristics, which is what you're typically most interested in, are represented by `GattCharacteristic` objects, through which you exchange data (`readValueAsync` and `writeValueAsync` methods and a `valuechanged` event), control encryption (`protectionLevel`), work with descriptors (`getDescriptors` and `[read | write]CharacteristicConfigurationDescriptorAsync`), `characteristicProperties`, and `presentationFormats`.

You can see usage examples in the SDK sample linked above, which as its name suggests deals with heart rate monitors. Some code is shared between its scenarios in js/heart-rate-service.js, such as connecting to a device, setting up a `valuechanged` event handler, and doing some configuration. Scenario 1 monitors the event for ongoing heart rate readings, where the event args object (a `GattValueChangedEventArgs`) contains a `timestamp` and `characteristicValue`.

The characteristic value here is a WinRT `Buffer` object, as is any reading you get from `GattCharacteristic.readValueAsync`. You also use a buffer to send information to the device through `GattCharacteristic.writeValueAsync`. As such, you use the `DataReader` and `DataWriter` objects to put data into the buffers. Thus the event handler in js/heart-rate-service.js processes the reading as follows:[124]

```
function onHeartRateMeasurementValueChanged(args) {
    var heartRateData = new Uint8Array(args.characteristicValue.length);

    Windows.Storage.Streams.DataReader.fromBuffer(args.characteristicValue)
        .readBytes(heartRateData);

    // Interpret the Heart Rate measurement value according to the Heart Rate Bluetooth Profile
    var heartRateMeasurement = processHeartRateMeasurementData(heartRateData);

    // Save the processed data in our namespace
    data[data.length] = {
        timestamp: args.timestamp,
        value: heartRateMeasurement.heartRateValue,
        expendedEnergy: heartRateMeasurement.energyExpended,
```

[124] There's also an event handler in js/s1-eventing.js, but this does not handle the WinRT event directly: the handler in js/heart-rate-service.js dispatches its own event once it has processed the raw data so that the scenario 1 code can consume a higher level structure.

```
        toString: function () {
            return this.value + ' bpm @ ' + this.timestamp;
        }
    };

    // Code to dispatch a custom event to the scenario code omitted.
}
```

One of the other typical characteristics of a heart rate monitor describes the location of the monitor, such as the chest, wrist, finger, and so on. Scenario 2 of the sample shows how to retrieve this bit of information (js/s2-read-characteristic-value.js):

```
var gatt = Windows.Devices.Bluetooth.GenericAttributeProfile;
var bodySensorLocationCharacteristics = HeartRateService.getHeartRateService()
    .getCharacteristics(gatt.GattCharacteristicUuids.bodySensorLocation);

if (bodySensorLocationCharacteristics.length > 0) {
    bodySensorLocationCharacteristics[0].readValueAsync().done(function (readResult) {
        if (readResult.status === gatt.GattCommunicationStatus.success) {
            var bodySensorLocationData = new Uint8Array(readResult.value.length);
            Windows.Storage.Streams.DataReader.fromBuffer(readResult.value)
                .readBytes(bodySensorLocationData);

           // Process the location data (see hs/heart-rate-service.js)
            var bodySensorLocation = HeartRateService.processBodySensorLocation(
                bodySensorLocationData);
    });
}
```

Similarly, scenario 3 writes a characteristic value to the device to reset the expended energy status value, and also watches this characteristic through the `valuechanged` event. Here's the code to reset the status (condensed from js/s3-write-characteristic-value.js):

```
var heartRateControlPointCharacteristics = HeartRateService.getHeartRateService()
    .getCharacteristics(gatt.GattCharacteristicUuids.heartRateControlPoint);

if (heartRateControlPointCharacteristics.length > 0) {
    characteristic = heartRateControlPointCharacteristics[0];

    var writer = new Windows.Storage.Streams.DataWriter();
    writer.writeByte(1);

    characteristic.writeValueAsync(writer.detachBuffer()).done(function (status) {
        if (status === gatt.GattCommunicationStatus.success) {
            // Value written
        }
    });
}
```

It's worth noting that most apps that connect to a Bluetooth LE device will acquire values for various status characteristics like location and battery life and then display those in a way to keep the user informed. This is especially useful if the device itself doesn't make such information readily available otherwise. For a demonstration and a few other advanced topics, refer to //build 2013 session 3-9028 linked at the beginning of this section.

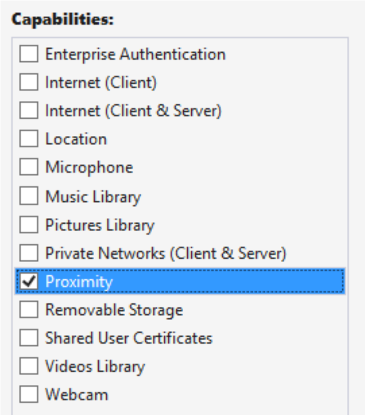
Wi-Fi Direct

Manifest capability name: use the top level *Proximity* capability
Protocol API namespace: `Windows.Devices.WiFiDirect`
Primary class: `WiFiDirectDevice`
SDK Sample(s): WiFiDirectDevice sample
Background docs: WiFi Direct (Wikipedia)
Applicable //build 2013 session: Building Windows Apps that Use Wi-Fi Direct (3-9030)

Generally speaking, Wi-Fi Direct means connecting two endpoints, like a phone and a laptop, or your phone and your automobile's audio system, without the need for an access point. Working with Wi-Fi Direct devices is very similar to what we've seen already with other protocols. As mentioned in the introduction to this chapter, pairing of wireless devices is something that Windows handles at the OS level, so all you really need to do from an app is declare the necessary capability in your manifest, enumerate the devices you care about, pick the one you want to use, and then use the API in `Windows.Devices.WiFiDirect` to communicate with it. In this case, communication happens through sockets, for which you use the WinRT APIs discussed in Appendix D.

The WiFiDirectDevice sample that we'll draw from here demonstrates the API, as usual, and also provides a good app for testing connections with Wi-Fi Direct devices. If you find that your own app is having troubles with a device, try it also with this sample and compare the results.

As noted above, the sample shows that the only capability you need for Wi-Fi Direct is *Proximity*, which you can set through Visual Studio's manifest editor directly:



Capabilities:

□ Enterprise Authentication
□ Internet (Client)
□ Internet (Client & Server)
□ Location
□ Microphone
□ Music Library
□ Pictures Library
□ Private Networks (Client & Server)
☑ Proximity
□ Removable Storage
□ Shared User Certificates
□ Videos Library
□ Webcam

When you enumerate Wi-Fi Direct devices, you'll enumerate only those that have been paired. Once you have the ID you want, `WiFiDirectDevice.fromIdAsync` gets you the `WiFiDirectDevice` instance for that specific hardware (assuming the user consents). At that point, register a handler for the `WiFiDirectDevice.onconnectionstatuschanged` event:

```
var wfd = Windows.Devices.WiFiDirect;

wfd.WiFiDirectDevice.fromIdAsync(selectedId).done(
    function (wfdDevice) {
        // Listen for status changes (assuming wfdDevice is non-null)
        wfdDevice.onconnectionstatuschanged = statusHandler;

        // Start communications
    }
);
```

This event tells you when the device object's connectionStatus property changes. Its values are simply connected or disconnected. Apps typically use the connection status to show some kind of visual indicator, stop data transfers, and/or prompt the user to select a different device if the one they've been using is unavailable.

The other key thing to do when you lose a connection is to close any sockets you've opened for it. As mentioned earlier, communication with a Wi-Fi Direct device happens through sockets. This starts by calling WiFiDirectDevice.getConnectionEndpointPairs to retrieve the endpoints and then open and close as many sockets as you desire using DatagramSocket and StreamSocket objects and their connectAsync methods (setting the EndpointPair.remoteServiceName to the desired TCP port). See Appendix D for details on working with sockets, and watch the //build 2013 session linked to earlier for a live demonstration (it starts at about 12m 15s in).

Near Field Communication and the Proximity API

Connecting with devices that are near to the one on which your app is running is one area that I suspect will see much creative innovation in the coming years as PCs are increasing equipped with the requisite hardware. In this case we're speaking of "devices" more generally than we have been. In some cases there will be a separate discrete device, most notably Bluetooth devices or NFC tags. But then we're also speaking of an app running on one machine connecting with itself or another that's running on a different machine. In this sense, apps can communicate with each other as if they were themselves separate "devices."

> **Caveat** Although it is possible for different apps to know about each other and communicate, the Windows Store certification requirements do not allow them to be interdependent. Approach such communication scenarios as a way to extend the functionality of the app, but be sure to provide value when the app is run in isolation.

Near Field Communication (NFC) is one of the key ways for apps to connect to devices and across devices. NFC works with electromagnetic sensors (including unpowered NFC tags) that resonate with each other when they get close, usually within 3–4 centimeters. Practically speaking, this means that the devices actually make physical contact, a tap that effectively initiates a digital handshake to open a conversation. When this pairing happens between the apps running on both devices that share a communication protocol (which is easy if they're the same app), those apps can have an ongoing conversation.

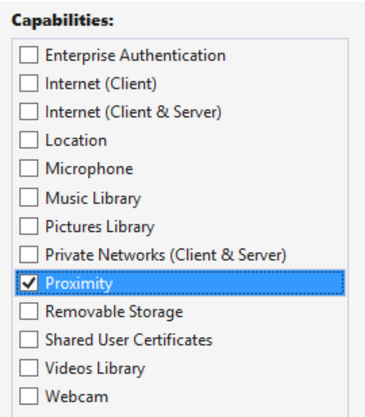
When you think about "devices" in this context, though, they can vary tremendously. The devices that are making the connection don't need to be at all similar. One device might be your tablet PC, and the other might by anything from a large all-in-one PC display to a simple NFC tag mounted in a poster or name badge.

Apps can also learn about each other on different devices through Wi-Fi Direct (if the wireless adapter supports it) and Bluetooth. In these cases it's possible for one app to browse for available (advertised) connections on other devices, which might or might not be coming from the same app. The devices don't need to be as physically close as with a tap, just within range of their wireless signals.

Whatever the case, working with *proximity*—as all of this is collectively referred to—is useful for many scenarios, such as sharing content, setting up multiplayer experience, broadcasting activity, and really anything else where some kind of exchange might happen, including both one-time data transfers and setting up more persistent connections.

Three main conditions exist for using proximity (see Guidelines for proximity):

- The app must declare the *Proximity* capability in its manifest.



Capabilities:

☐ Enterprise Authentication
☐ Internet (Client)
☐ Internet (Client & Server)
☐ Location
☐ Microphone
☐ Music Library
☐ Pictures Library
☐ Private Networks (Client & Server)
☑ Proximity
☐ Removable Storage
☐ Shared User Certificates
☐ Videos Library
☐ Webcam

- Communications are supported only for foreground apps (there are no background tasks to maintain a conversation).

- The app must ask for user consent to enter into a multiuser relationship. An app should show waiting connections, connections being established, and connections that are active, and it should allow the user to disconnect at any time. Note that using the APIs to make a connection will automatically prompt the user.

The API for working with proximity is in the appropriately named `Windows.Networking.Proximity` namespace, as is the Proximity sample that we'll be working with here. It almost goes without saying that doing any deep exploration of proximity will require two machines that are suitably equipped or one device plus an NFC tag. For NFC between two machines there is also a driver sample in the Windows Driver Kit that simulates NFC over a network connection. To use it, you'll also need Visual Studio 2013 Ultimate Edition; the Express version does not support driver development. In that case it might make more sense to just acquire an NFC-capable device!

To install the Windows Driver Kit, follow the instructions on How to get the WDK *after* installing Visual Studio 2013 Ultimate. To start the download, you run the small the installer and then select the option to acquire the kit for use on another computer. Then, according to the Proximity sample's description page:

> *After you have installed the WDK and samples, you can find the proximity driver sample in the src\nfp directory in the location where you installed the WDK samples. See the NetNfpProvider.html file in the src\nfp\net directory for instructions on building and running the simulator.* [Note: be sure to specify a Windows 8.1 target when you build.] *After you start the simulator, it runs in the background while your proximity app is running in the foreground. Your app must be in the foreground for the tap simulation to work.*

For more complete instructions, refer to How to use Near-Field Proximity API without NFC hardware on Stephanié Hertrich's MSDN blog. There you can also find a C# library for Transferring a file between 2 peers using Wifi-Direct and Proximity API, which you might find helpful or instructive.

Assuming that you have an environment in which proximity can at least be simulated, let's look at the two mainline scenarios now. The first uses the `PeerFinder` class to create a socket connection between two peer apps for ongoing communication; the second uses the `ProximityDevice` class to send data packets between two devices.

Finding Your Peers (No Pressure!)

To find peers, an app on one device can advertise its availability such that apps on other devices can browse advertised peers and initiate a connection over Wi-Fi Direct (Windows devices) or Bluetooth (Windows Phone). The second way to find a peer is through a direct NFC tap (which works on Windows and Windows Phone). Both methods are shown in scenario 1 of the Proximity sample, but these three distinct functions—advertise, browse, and tap to connect—are somewhat intermixed, because they each use some distinct parts of the `PeerFinder` class and some parts in common.

One commonality is the static property `PeerFinder.supportedDiscoveryTypes`, which indicates how connections can be made. This contains a combination of values from the `PeerDiscoveryTypes` enumeration and depends on the available hardware in the device. Those values are `browse` (Wi-Fi Direct is available), `triggered` (NFC tapping is available), and `none` (`PeerFinder` can't be used). You can use these values to selectively enable or disable certain capabilities in your app as needed. Scenario 1 of the Proximity sample, for instance, checks the discovery types to set some flags and enable buttons for NFC activities. Otherwise, it just shows disappointing messages (this code is condensed somewhat from the page's `ready` method js/PeerFinder.js, and note the namespace variable at the top):

```
var ProxNS = Windows.Networking.Proximity;

var supportedDiscoveryTypes = ProxNS.PeerFinder.supportedDiscoveryTypes;

// Enable triggered (tap) related UI only if the hardware support is present
if (supportedDiscoveryTypes & ProxNS.PeerDiscoveryTypes.triggered) {
    triggeredConnectSupported = true;
} else {
    peerFinderErrorMsg = "Tap based discovery of peers not supported \n";
}

// Enable browse related buttons only if the hardware support is present
if (supportedDiscoveryTypes & ProxNS.PeerDiscoveryTypes.browse) {
    browseConnectSupported = true;
    // [Add listeners to buttons, code omitted]
    } else {
    // [Show messages, code omitted]
    }

if (triggeredConnectSupported || browseConnectSupported) {
    // [Set up additional UI]
}

// ...
}
```

Now let's tease apart the distinct areas.

Advertising a Connection

Making yourself available to others through advertising has two parts: putting out the word and listening for connections that are made. Assuming that some form of communication is possible, the first step in all of this is to configure the `PeerFinder` with the `displayName` that will `appear to other devices when you advertise and to set allowBluetooth`, `allowInfrastructure`, and `allowWiFi-Direct` as desired to allow discovery over additional networks (infrastructure refers to TCP/IP). Setting none of these flags will still enable connections through NFC tapping, which is always enabled.

Next, set up a handler for the `PeerFinder.onconnectionrequested` event, followed by a call to the static method `PeerFinder.start` (again, `ProxNS` is a namespace variable):

```
ProxNS.PeerFinder.onconnectionrequested = connectionRequestedEventHandler;
ProxNS.PeerFinder.start();
```

Note `connectionrequested` is an event that originates within WinRT. Because this is perhaps an event you might listen to only temporarily, be sure to call `removeEventListener` or assign `null` to the event property to prevent memory leaks. See "WinRT Events and removeEventListener" in Chapter 3, "App Anatomy and Performance Fundamentals."

The connectionrequested event is triggered when other devices pick up your advertisement and call your toll-free hotline, so to speak, specifically over Wi-Fi Direct or Bluetooth. The event receives a ConnectionRequestedEventArgs object that contains a single property, peerInformation, which is an instance of—not surprisingly—the PeerInformation class. This object too is simple, containing nothing but a displayName, but that is enough to make a connection.

```
function connectionRequestedEventHandler(e) {
    requestingPeer = e.peerInformation;
    ProximityHelpers.displayStatus("Connection Requested from peer: "
        + requestingPeer.displayName);
    // Enable Accept button (and hide Send and Message) [some code omitted]
    ProximityHelpers.id("peerFinder_AcceptRequest").style.display = "inline";
}
```

A connection is established by passing that PeerInformation object to PeerFinder.connectAsync. This will prompt the user for consent, and, given that consent, your completed handler will receive a Windows.Networking.Sockets.StreamSocket, which is explained in Appendix C, "Additional Networking Topics."

```
function peerFinder_AcceptRequest() {
    ProxNS.PeerFinder.connectAsync(requestingPeer).done(function (proximitySocket) {
        startSendReceive(proximitySocket);
    });
}
```

From this point on, you're free to send whatever data with whatever protocols you'd like, on the assumption, of course, that the app on the other end will understand what you're sending. This is clearly not a problem when it's the same app on both ends of the connection; different apps, of course, will need to share a common protocol. In the sample, the "protocol" exchanges only some basic values, but the process is all there.

If at any time you want to stop advertising, call PeerFinder.stop. To close a specific connection, call the socket's close method.

Making a Connection

On the other side of a proximity relationship, an app can look for peers that are advertising themselves over Wi-Fi Direct or Bluetooth. In the Proximity sample, a Browse Peers button is enabled if the browse discovery type is available. This button triggers a call to the following function (js/PeerFinder.js) that uses PeerFinder.findAllPeersAsync to populate a list of possible connections, including those from different apps:

```
function peerFinder_BrowsePeers() {
    // Empty the current option list [code omitted]

    ProxNS.PeerFinder.findAllPeersAsync().done(function (peerInfoCollection) {
        // Add newly found peers into the drop down list.
        for (i = 0; i < peerInfoCollection.size; i++) {
            var peerInformation = peerInfoCollection[i];
            // Create and append option element using peerInformation.displayName
            // to the peerFinder_FoundPeersList control [code omitted]
        }
    });
}
```

When you select a peer to connect to, the sample takes its PeerInformation object and calls PeerFinder.connectAsync as before (during which the user is prompted for consent):

```
function peerFinder_Connect() {
    var foundPeersList = ProximityHelpers.id("peerFinder_FoundPeersList");
    var peerToConnect = discoveredPeers[foundPeersList.selectedIndex];

    ProxNS.PeerFinder.connectAsync(peerToConnect).done(
        function (proximitySocket) {
            startSendReceive(proximitySocket);
        });
}
```

Once again, this provides a StreamSocket as a result, which you can use as you will. To terminate the connection, call the socket's close method.

Tap to Connect and Tap to Activate

To detect a direct NFC tap—which again works to connect apps running on two devices—listen to the PeerFinder.onTriggeredConnectionStateChanged (a WinRT event that I spell out in camel casing so that it's readable!). In response, start the PeerFinder:

```
ProxNS.PeerFinder.ontriggeredconnectionstatechanged =
    triggeredConnectionStateChangedEventHandler;
ProxNS.PeerFinder.start();
```

The process of connecting through tapping will go through a series of state changes (including user consent), where those states are described in the TriggeredConnectState enumeration: listening, connecting, peerFound, completed, canceled, and failed. Each state is included in the event args sent to the event (a TriggeredConnectionStateChangedEventArgs…some of these names sure get long!), and when that state reaches completed, the socket property in the event args will contain the StreamSocket for the connection:

```
function triggeredConnectionStateChangedEventHandler€ {
    // [Other cases omitted]

    if (e.state === ProxNS.TriggeredConnectState.completed) {
        startSendReceive(e.socket);
    }
}
```

Again, from this point on, it's a matter of what data is being exchanged through the socket—the NFC tap is just a means to create the connection. And once again, call the socket's close when you're done with it.

When tapping connects the same app across devices, it's possible for the tap to launch an app on one of those devices. That is, when the app is running on one of the devices and has started the PeerFinder, Windows will know the app's identity and can look for it on the other device. If it finds that app, it will launch it (or activate it if it's already running). The app's activated handler is then called with an activation kind of launch, where eventArgs.detail.arguments will contain the string "*Windows.Networking.Proximity.PeerFinder:StreamSocket*" (see js/default.js in the Proximity sample; the code here is modified for clarity):

```
var tapLaunch = ((eventObject.detail.kind ===
    Windows.ApplicationModel.Activation.ActivationKind.launch) &&
    (eventObject.detail.arguments ===
    "Windows.Networking.Proximity.PeerFinder:StreamSocket"));

if (tapLaunch) {
    url = scenarios[0].url; // Force scenario 0 if launched by tap to start the PeerFinder.
}

return WinJS.Navigation.navigate(url, tapLaunch);
```

The code in scenario 1 picks up this condition (the tapLaunch parameter to WinJS.Navigation.Navigate is true) and calls PeerFinder.start automatically instead of waiting for a button press. In the process of startup, the app also registers its own triggeredConnectionStateChanged handler so that it will also receive a socket when the connection is complete.

Watching for Peers

In addition to deliberately browsing for peers in response to user interaction, for which you use PeerFinder.findAllPeersAsync, an app can dynamically watch for peers as they become visible. This is helpful for implementing multipeer scenarios.

After calling PeerFinder.start, a watcher—an instance of the PeerWatcher class—is created through PeerFinder.createWatcher, as demonstrated in scenario 2 of the Proximity sample (js/PeerWatcher.js):

```
peerWatcher = ProxNS.PeerFinder.createWatcher();
```

When you're ready to watch, call PeerWatcher.start:

```
peerWatcher.start();
```

If you want to stop watching, call the stop method. But while watching is active, the PeerWatcher fires relevant events for peer activity:

- added A peer has entered into the proximity space.
- removed A peer has exited the proximity space.
- updated A peer in the proximity space has changed either its displayName or discoveryData property.

- enumerationcompleted The PeerWatcher has completed a scan of the proximity space. Note that the watcher will continue to scan periodically, so this event will be fired repeatedly.

- stopped PeerWatcher.stop was called, or the watcher was stopped for some other reason. To determine the cause, check the PeerWatcher.status property.

With the added, removed, and updated events, your handler receives the PeerInformation object for the appropriate peer. If, for example, you wanted to watch for and automatically connect to a peer that you've connected with before, have your added handler check the PeerInformation properties of new peers as they show up, and call PeerFinder.connectAsync on those are in your auto-connect list.

The PeerWatcher.status property, as mentioned for the stopped event, contains a value from PeerWatcherStatus. This will be one of created (ready to watch), started (actively watching), stopping (in the process of shutting down), stopped (no longer watching), aborted (stopped due to failure), and enumerationCompleted (set after the first scan).

Sending One-Shot Payloads: Tap to Share

Although the PeerFinder sets up a StreamSocket and is good for scenarios involving ongoing communication, other scenarios—like sharing a photo, a link, or really any kind of information including data from an NFC tag—need only send some data from one device to another and be done with it. For such purposes we have the ProximityDevice class, which you obtain as follows:

```
var proximityDevice = Windows.Networking.Proximity.ProximityDevice.getDefault();
```

An app that has something to share "publishes" that something as a *message* in the form of a string, a URI, or a binary buffer. NFC tags publish their messages passively; an app, on the other hand, uses the ProximityDevice class and its publishMessage, publishUriMessage, and publishBinaryMessage methods (and a matching stopPublishing method). For example, drawing from scenario 3 of the Proximity sample (js/ProximityDevice.js, there publishText contains the contents of an edit control):

```
var publishedMessageId = proximityDevice.publishMessage("Windows.SampleMessageType",
    publishText);
```

On the other side, an app that would like to receive such a message calls ProximityDevice.-subscribeForMessage, passing the name of the message it expects along with a handler for when messages arrive:

```
var subscribedMessageId = proximityDevice.subscribeForMessage("Windows.SampleMessageType",
    messageReceived);

function messageReceived(receivingDevice, message) {
    // Process the message
}
```

If the app is no longer interested in messages, it calls stopSubscribingForMessage.

With this simple protocol, you can see that an app that supports "tap to share" (as it's called) would publish messages whenever it has appropriate content in hand. It can also use the ProximityDevice object's devicearrived and devicedeparted events to know when other tap-to-share peers are in proximity such that it's appropriate to publish (these are WinRT events, demonstrated in scenario 4 of the Proximity sample). The devicearrived event is also what you use to discover that an NFC tag has come into proximity (see below).

What's interesting to think about, though, is what kind of data you might share. Consider a travel app in which you can book flights, hotels, rental cars, and perhaps much more. It can, of course, publish messages with basic details but could also publish richer binary messages that would allow it to transfer an entire itinerary to the same app running on another device, typically to another user. This would enable one person to set up such an itinerary and then share it with a second person, who could then just tap a Book It button and be done! This would be far more efficient than emailing that itinerary as text and having the second person re-enter everything by hand.

On a simpler note, publishing a URI makes it super-simple for one person to tap-and-share whatever they're looking at with another person, again avoiding the circuitous email route or other forms of sharing. A quick tap, and you're seeing what I'm seeing. Again, though, there's so much more than can be shared that it's a great thing to consider in your design, especially if you're targeting mobile devices. "What do people near each other typically do together?" That's the question to ask and to answer in the form of a great proximity app.

Do note that the URIs you share don't have to be http:// references to websites but can contain any URI scheme. If there's an app associated with that URI scheme, tap-to-share also becomes tap-to-activate because Windows will launch the default app for that association. And if there's no association, Windows will ask if you want to acquire a suitable app from the Store. You can also consider using a Windows Store URI that will lead a user to directly install an app. Those URIs are described on Creating links with the Windows Store protocol.

Such URIs make it possible for NFC tags, whose messages are basically hardcoded into the device, to support tap-to-share and tap-to-activate scenarios. When you tap an NFC tag to an NFC-capable device, the ProximityDevice object will fire a devicearrived event. An app can then receive the tag's message through ProximityDevice.subscribeForMessage. This means that the app will need to know what type of message might be sent from that tag—it might be a standard type, or the app might be written specifically for tags with specific programming. For example, an art gallery could place tags near every piece it displays and then make an app available in the Windows Store for that gallery (or any other galleries that work in cooperation) that knows what messages those tags will send. If the message has an appropriate URI scheme in it, tapping on an NFC tag can help the user acquire an app and enjoy a rich experience.

For more, check out Develop a cutting edge app with NFC on the Windows Developer Blog.

What We've Just Learned

- The WinRT device APIs allow apps to access a wide range of peripheral devices. One app can also be registered as the dedicated app for a device, which gives it special permissions like performing firmware updates.
- Accessing devices begins with declaring appropriate capabilities in the app manifest. In a few cases the capabilities are supported through Visual Studio's manifest editor, but most of the time you need to edit the XML directly.
- The `Windows.Devices.Enumeration` API allows an app to discover what hardware is connected to a machine. Enumeration results in one or more `DeviceInformation` objects, and an app can either use the first (or default) device found or present the list to the user and let him or her choose the specific device to use.
- The enumeration API also provides for *watchers* that raise events when a suitable device is connected or disconnected from a system.
- Scenario APIs in WinRT provide for access to specific types of hardware that are backed by standards commands and communication protocols. Supported devices in Windows 8.1 include image scanners, barcode scanners, magnetic stripe readers, smartcards, and fingerprint readers.
- Printing (a scenario API), having been reimagined for Windows as a whole, is relatively easy to implement in a Windows Store app. It involves listening for the printing event when the Devices charm is invoked and providing HTML content to Windows.
- Printable content can come from the app's document, a document fragment, an SVG document, or a remote source. Such content can be customized using a CSS media query for print, and Windows takes care of the layout and flow of the information on the target printer.
- Protocol APIs in WinRT provide the means to communicate with devices through supported communication mechanisms, namely HID, USB, Bluetooth RFCOMM, Bluetooth Smart (LE/GATT), and Wi-Fi Direct. In these cases the app must tailor its commands to the devices in question, but the APIs take care of low-level transport concerns.
- The `Windows.Networking.Proximity` API supports peer browsing (over Wi-Fi Direct and Bluetooth) as well as *tap-to-connect* and *tap-to-share* scenarios with near field communication (NFC)–capable machines.
- Proximity connections can employ sockets for ongoing communication (like a multiplayer game) or can simply send messages from one device to another through a publish-and-subscribe mechanism, as is typical with tap-to-share scenarios, including NFC tags.

Chapter 18

WinRT Components: An Introduction

At the very beginning of this book, in the first two pages of Chapter 1, "The Life Story of a Windows Store App," we learned that apps can be written in a variety of languages and presentation technologies. For the better part of this book we've spoken of this as a somewhat singular choice: you choose the model that best suits your experience and the needs of your app and go from there.

At the same time, we've occasionally encountered situations where some sort of mixed language approach is possible, even necessary. In Chapter 1, in "Sidebar: Mixed Language Apps," I introduced the idea that an app can actually be written in multiple languages. In Chapter 10, "The Story of State, Part 1," I mentioned that gaining access to database APIs beyond IndexedDB could be accomplished with a WinRT component. In Chapter 13, "Media," we saw that JavaScript might not necessarily be the best language in which to implement a pixel-crunching routine. In Chapter 16, "Alive with Activity" we encountered the Notifications Extensions Library, a very useful piece of code written in C# that made the job of constructing an XML payload much easier from JavaScript. We also saw that background tasks can be written in languages that differ from that of the main app. And finally, in Chapter 17, "Devices and Printing," I mentioned that some devices can be accessed through Win32 APIs when other WinRT APIs don't exist and that you could use a WinRT component to create your own device-specific scenario API instead on top of the general purpose protocol APIs. Doing so can make it much easier for others to write apps for your device.

With the primary restriction that an app that uses HTML and CSS for its presentation layer cannot share a drawing surface with WinRT components written in other languages, the design of WinRT makes the mixed language approach possible. As discussed in Chapter 1, WinRT components written in any language are made available to other languages through a projection layer that translates the component's interface into what's natural in the target language. All of WinRT is written this way, and custom WinRT components simply take advantage of the same mechanism. (We'll see the core characteristics of the JavaScript projection later in this chapter.)

What this means for you—writing an app with HTML, CSS, and JavaScript—is that you can implement various parts of your app in a language that's best suited to the task or technically necessary. As a dynamic language, JavaScript shows its strength in easily gluing together functionality provided by other components that do the heavy lifting for certain kinds of tasks (like camera capture, background transfers, multi-threaded async work, etc.). Those heavy-lifting components are often best written in other language such as C++, where the compiled code runs straight on the CPU (instead of going through runtime layers like JavaScript) and has access to thread pools.

Indeed, when we speak of *mixed language apps*, you truly can use a diverse mix.[125] You can write a C# component that's called from JavaScript, and that C# component might invoke a component written in C++. You can use the best language for any particular job, including when you need to create your own async operations (that is, to run code on concurrent threads that don't block the UI thread). In this context it's helpful to also think through what this means in relationship to web workers, something that Windows Store apps can also employ.

> **Don't forget** You don't always need to offload work to other threads: you can use the `WinJS.Utilities.Scheduler` API that we discussed in Chapter 3, "App Anatomy and Performance Fundamentals."

> **And don't forget** Events that are generated from within a custom WinRT component work just like those from built-in WinRT APIs. This means that apps written in JavaScript must prevent memory leaks by calling `removeEventListener` for temporary listeners. Refer to "WinRT Events and removeEventListener" in Chapter 3.

In this chapter, we'll first look at the different reasons why you might want to take a mixed language approach in your app. We'll go through a couple of quickstarts for C# and C++ so that we understand the structure of these components, how they appear in JavaScript, and the core concepts and terminology. The rest of the chapter will then primarily give examples of those different scenarios, which means we won't necessarily be going deep into the many mechanical details. I've chosen to do this because there is very good documentation on those mechanics, which you can find in Creating Windows Runtime Components in the documentation.

As you peruse the subsections of that topic, don't let the words "basic" and "simple" in the walkthrough titles deter you: all these topics are comprehensive cookbooks that cover the fine details of working with data types like vectors, maps, and property sets; declaring events; creating async operations; and how all this shows up in JavaScript. We'll see some of these things in the course of this introduction, but with great topics covering the *what*, we'll be spending our time here on *why* we'd want to use such components in the first place and the problems they can help solve. Plus, I want you to save some energy for the book's finale in the next two chapters, where we'll talk about getting your app out to the world once you solve those problems!

> **Note** By necessity I have to assume in this chapter that you have a little understanding of the C# and C++ languages. If these languages are entirely new to you, spending a few hours familiarizing yourself with them will improve your mileage with this chapter.

[125] Mixed language apps have occasionally been referred to as "hybrid apps," a term I've chosen to avoid because it's typically used to describe apps that combine native functionality with hosted content in a webview control.

Choosing a Mixed Language Approach (and Web Workers)

Here are some of the reasons to take a mixed language approach in your app, which can again include any number of WinRT components written in C#, Visual Basic (hereinafter just VB), and/or C++:

- You can accomplish certain tasks faster with higher performance code. This can reduce memory overhead and also consume less CPU cycles and power, an important consideration for background tasks for which a CPU quota is enforced—you might get much more done in 2 CPU seconds in C++ than with JavaScript, because there's no language engine involved. However, the JavaScript language engine is highly optimized and could perform far better than you think. In other words, don't think that JavaScript is inherently slow, especially when you factor in the costs of crossing language boundaries. We'll see more later in "Comparing the Results."

- C#, Visual Basic, and C++ have access to a sizable collection of additional APIs that are not available to JavaScript. These include .NET, Win32, and COM (Component Object Model) APIs, including the non-UI features of DirectX such as XAudio and XInput. We'll see a number of examples later in "Access to Additional APIs."

- Access to other APIs might be necessary for utilizing third-party .NET/Win32/COM libraries and gives you the ability to reuse existing code that you might have in C#, VB, or C++. The Developing Bing Maps Trip Optimizer, a Windows Store app in JavaScript and C++ topic shows a complete scenario along these lines, specifically that of migrating an ActiveX control to a WinRT component so that it can be used from an app, because ActiveX isn't directly supported. (We won't cover this scenario further in this chapter.)

- It can be easier to work with routines involving many async operations by using the `await` keyword in C# and Visual Basic, because the structure is much cleaner than with promises. An example of this can be found in the Here My Am! app of Chapter 19, "Apps for Everyone, Part 1," where the `transcodeImage` function written in JavaScript for Chapter 16 is rewritten in C#. (See `Utilities.ImageConverter.TranscodeImageAsync` in the Utilities project.)

- A WinRT component written in C++ is more effective at obfuscating sensitive code than JavaScript and .NET languages. Although it won't obfuscate the interface to that component, its internal workings are more painstaking to reverse-engineer.

- A WinRT component is the best way to write a non-UI library that other developers can use in their chosen language or that you can use in a variety of your own projects, like the Notifications Extensions library we saw in earlier chapters, a custom scenario API for a device, or a library to hide the details of interacting with a REST API on the web. In this context, see Creating a software development kit, which includes details how the component should be structured to integrate with Visual Studio. Also refer to //build 2013 session 4-142, Building a Great Windows Store Library/SDK.

- Although you can use web workers in JavaScript to execute code on different threads, a WinRT component can be much more efficient for custom asynchronous APIs. Other languages can also make certain tasks more productive, such as using language-integrated queries (LINQ) from C#/VB, creating parallel loops in C#/C++, using C++ Accelerated Massive Parallelism (AMP) to take advantage of the GPU, and so on.

Again, components can also make use of one another—the component system has no problem with that. I reiterate this point because it becomes relevant in the context of the last bullet above—web workers—and running code off the UI thread.

You are wholly free to use web workers (or just *workers*, as I'll call them) in a Windows Store app. Visual Studio provides an item template for this purpose: right-click your project in Visual Studio's solution explorer, select Add > New Item, and choose Dedicated Worker. I'll also show an example later in "JavaScript Workers." The real drawback here, compared with WinRT components, is that communication between the worker and the main app thread is handled entirely through the singular async `postMessage` mechanism; data transferred between the app and the worker must be expressed as properties of a message. In other words, workers are set up for a client-server architecture more so than one of objects with properties, methods, and events, though you can certainly create structures for such things through messages.

What this means is that using workers with multiple asynchronous operations can get messy. By comparison, the methods we've seen for working with async WinRT operations—namely WinJS promises—is much richer, especially when you need to chain or nest async operations, raise exceptions, and report progress. Fortunately, you can easily wrap a worker within a promise, as we'll see in "JavaScript Workers."

What also interesting to think about is how you might use a worker written in JavaScript to act as an agent that delegates work to WinRT components. The JavaScript worker then serves as the glue to bring all the results from those components together, reporting a combined result to the main app through `postMessage`.

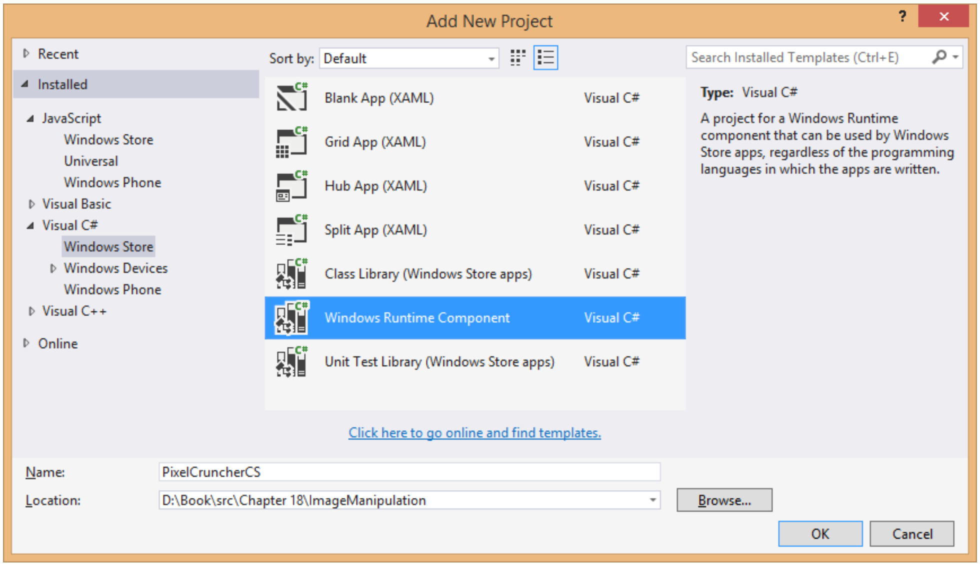
Along these same lines, if you have some experience in .NET languages like C# and Visual Basic along with C++, you'll know that their respective programming models have their own strengths and weaknesses. Just as you can take advantage of JavaScript's dynamic nature where it serves best, you can use the strongly typed nature of .NET where it helps your programming model and then use C++ where you want the most performant code.

You can have your main app's UI thread in JavaScript delegate a task to a worker in JavaScript, which then delegates some tasks to a WinRT component written in C#, which might then delegate its most intensive tasks to still other components written in C++. Truly, the combination of workers with WinRT components gives you a great deal of flexibility in your implementation.

Pros and cons One complication with WinRT components written in C# or Visual Basic is that they require loading the Common Language Runtime (CLR), which incurs additional memory overhead when used with an app written in JavaScript or C++. With a JavaScript app, it also introduces a second garbage collector, which can result in a consistently higher memory footprint for the app. On the flip side, a potential disadvantage of WinRT components written in C++ is that while JavaScript and .NET languages (C#/VB) are architecture-neutral and can target any CPU, C++ components must be separately compiled for x86, x64, and ARM. This means that your app will have at least two separate packages in the Windows Store (maybe three if you want to build an x64 target specifically). In the end, you need to decide what's best for your project and your customers.

Quickstarts: Creating and Debugging Components

When you set out to add a WinRT component to your project, the easiest place to start is with a Visual Studio item template. Right-click your *solution* (not your project) in Visual Studio's solution explorer, select Add > New Project, and choose Windows Runtime Component as listed under the Visual Basic >, Visual C# >, or Visual C++ > Windows Store nodes, as shown in Figure 18-1 (for a C# project).


Add New Project
?
✕
Recent
Installed
JavaScript
Windows Store
Universal
Windows Phone
Visual Basic
Visual C#
Windows Store
Windows Devices
Windows Phone
Visual C++
Online
Search Installed Templates (Ctrl+E)
Blank App (XAML) Visual C#
Grid App (XAML) Visual C#
Hub App (XAML) Visual C#
Split App (XAML) Visual C#
Class Library (Windows Store apps) Visual C#
Windows Runtime Component Visual C#
Unit Test Library (Windows Store apps) Visual C#
Click here to go online and find templates.
Browse...
OK
Cancel

Sort by: Default

Type: Visual C#

A project for a Windows Runtime component that can be used by Windows Store apps, regardless of the programming languages in which the apps are written.

Name: PixelCruncherCS

Location: D:\Book\src\Chapter 18\ImageManipulation

FIGURE 18-1 Visual Studio's option for creating a C# WinRT Component project; similar options appear under Visual Basic > Windows Store and Visual C++ > Windows Store.

We'll look at both the C# and C++ options as we fulfill a promise made in Chapter 13 to see whether we can improve the performance of the Image Manipulation example in this book's companion content. That earlier sample performed a grayscale conversion on the contents of an image file and displayed the results on a `canvas`. We did the conversion in a piece of JavaScript code that performs quite well, actually, but it's good to ask if we can improve that performance and perhaps make the conversion more memory-efficient. Our first two WinRT components, then, are equivalent implementations written in C# and C++. Together, all three give us the opportunity to compare relative performance, as we'll do in "Comparing the Results."

> **Warning** Experience has shown that it's generally better to write components in C++ rather than C# or Visual Basic because you avoid the overhead of garbage collection, especially with an app written in JavaScript where there's another garbage collector at work already. In my tests with the Image Manipulation code, I've found that the C# variants have a much higher memory footprint both initially and over time than the C++ version, especially with the large amounts of data that's being marshaled across the component boundary. However, for less data-intensive components, writing them in C# will incur some added overhead when first loading the component, which can be a small price to pay for the added productivity of developing in C#.

One problem with all these implementations is that they still run on the UI thread, so we'll want to look at making the operations asynchronous. We'll come back to that later in "Key Concepts and Details" so as to avoid making these quickstarts less than quick!

Note also that I introduce terminology and tooling considerations in the C# quickstart, so read it even if you plan on working in C++.

Sidebar: WinRT Components vs. Class Libraries (C#/VB) vs. Dynamic-Link Libraries

In the Add New Project dialog of Visual Studio, you'll notice that an option for a Class Library is shown for Visual C# and Visual Basic and an option for a DLL (dynamic-link library) is shown for C++. These effectively compile into assemblies and DLLs, respectively, which bear resemblances to WinRT components. The difference is that these types of components can be used from only those same languages. A Class Library (.NET assembly) can be used by apps written in .NET languages but not from JavaScript. A DLL can be called from C++ and .NET languages (the latter only through a mechanism called P/Invoke) but is not available to JavaScript. A WinRT component is the only choice that works across language boundaries, thanks to the WinRT projection layers.

Sometimes a simple DLL is required, as with media extensions that provide custom audio/video effects or a codecs. These are not WinRT components because they lack metadata that would project them into other languages, nor is such metadata needed. Details on DLLs for the media platform can be found in Using media extensions and Media extensions sample.

Quickstart #1: Creating a Component in C#

As shown above in Figure 18-1, I've added a WinRT Component in C# to the Image Manipulation example, calling it PixelCruncherCS. Once this project has been added to the solution, we'll have a file called class1.cs in that project that contains a namespace `PixelCruncherCS` with one class:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace PixelCruncherCS
{
    public sealed class Class1
    {
    }
}
```

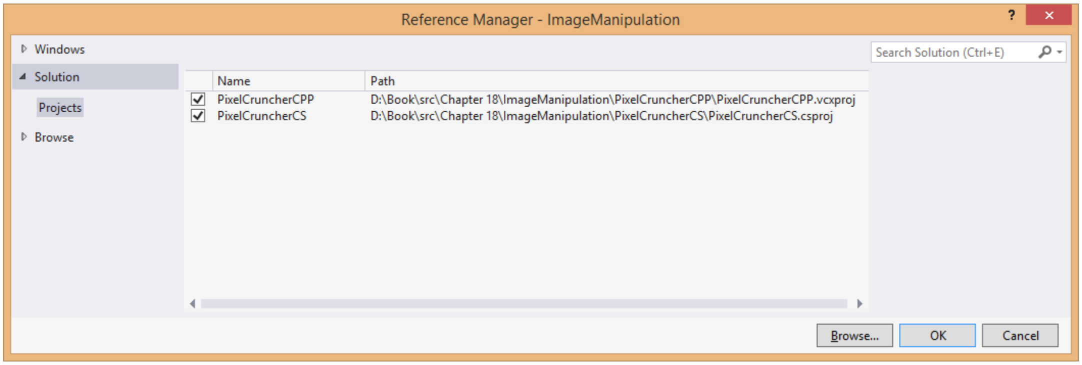
Not particularly exciting code at this point but enough to get us going. You can see that the class is marked `public`, meaning it will be visible to apps using the component. It is also marked `sealed` to indicate that other classes cannot be derived from it (because of current technical constraints). Both keywords are required for WinRT components. (These two keywords are `Public` and `NotInheritable` in Visual Basic.)

To test the interaction with JavaScript, I'll give the class and its file a more suitable name (*Tests* and grayscale.cs, because we'll be adding more to it) and create a test method and a test property:

```
public sealed class Tests
{
    public static string TestMethod(Boolean throwAnException)
    {
        if (throwAnException)
        {
            throw new System.Exception("Tests.TestMethod was asked to throw an exception.");
        }

        return "Tests.TestMethod succeeded";
    }

    public int TestProperty { get; set; }
}
```

If you build the solution at this point (Build > Build Solution), you'll see that the result of the PixelCruncherCS project is a file called PixelCruncher.winmd. The .winmd extension stands for Windows Metadata: a WinRT Component written in C# is a .NET assembly that includes extra metadata referred to as the component's Application Binary Interface or ABI. This is what tells Windows about everything the component makes available to other languages (those `public` classes), and it's also what provides IntelliSense for that component in Visual Studio and Blend.

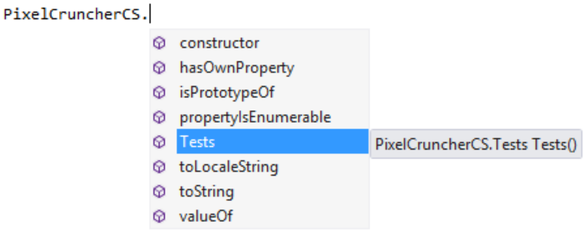
In the app you must add a reference to the component so that it becomes available in the JavaScript namespace, just like the WinRT APIs. To do this, right-click References within the JavaScript app project and select Add Reference. In the dialog that appears, select Solution on the side and then check the box for the WinRT component project as shown in Figure 18-2.



	Name	Path
☑	PixelCruncherCPP	D:\Book\src\Chapter 18\ImageManipulation\PixelCruncherCPP\PixelCruncherCPP.vcxproj
☑	PixelCruncherCS	D:\Book\src\Chapter 18\ImageManipulation\PixelCruncherCS\PixelCruncherCS.csproj

FIGURE 18-2 Adding a reference to a WinRT component within the same solution as the app.

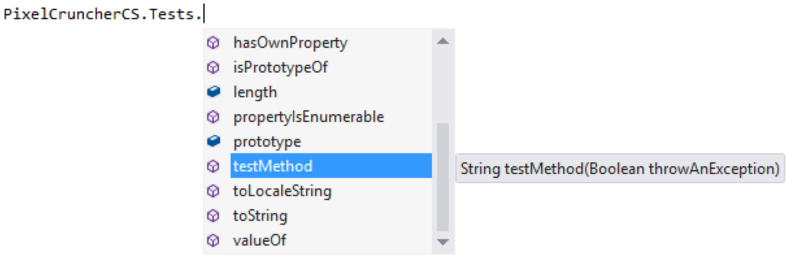
When writing code that refers to the component, you always start with the namespace, `PixelCruncherCS` in our case. As soon as you enter that name and a dot, IntelliSense will appear for available classes in that namespace:



```
PixelCruncherCS.
```

PixelCruncherCS.Tests Tests()

Once you add the name of a class and type another dot, IntelliSense appears for its methods and properties:



```
PixelCruncherCS.Tests.
```

String testMethod(Boolean throwAnException)

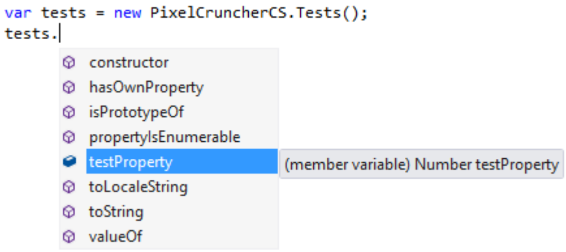
Note If you've made changes to namespace, class, and other names in your WinRT component project, you'll need to run Build > Build Solution to see the updated names within IntelliSense.

Here you can see that the method in the C# code is `TestMethod`; it's projected into JavaScript as `testMethod`, matching typical JavaScript conventions. This casing conversion is done automatically through the JavaScript projection layer for all WinRT components, including those in your own app.

Notice also that IntelliSense is showing only `testMethod` here but not `testProperty` (whose casing is also converted). Why is that? It's because in C# `TestMethod` is declared as `static`, meaning that it can be executed without first instantiating an object of that class:

```
var result = PixelCruncherCS.Tests.testMethod(false);
```

On the other hand, `testProperty`, but not `testMethod`, is available on a specific instance:



```
var tests = new PixelCruncherCS.Tests();
tests.
```

I've set up `TestMethod`, by the way, to throw an exception when asked so that we can see how it's handled in JavaScript with a `try`/`catch` block:

```
try {
    result = PixelCruncherCS.Tests.testMethod(true);
} catch (e) {
    console.log("PixelCruncherCS.Tests.testMethod threw: '" + e.description + "'.");
}
```

Let's try this code. Attaching it to some button (see the `testComponentCS` function in the example's js/default.js file), set a breakpoint at the top and run the app in the debugger. When you hit that breakpoint, step through the code using Visual Studio's Step Into feature (F11 or Debug > Step Into). Notice that you do not step into the C# code: Visual Studio isn't presently able to debug across the script/managed (C#) boundary, although it can do so across script/native (C++). What this means is that within a single instance of Visual Studio you can debug either the JavaScript or the C# side of your code, but not both at the same time. Console output can help here, but it's also possible to use two instances of the debugger, as we'll see later.

Having the basic mechanics worked out, we're now ready to add our real functionality. The first ste p is to understand how to get the canvas pixel data arrays in and out of the WinRT component. In the JavaScript code (within the `copyGrayscaleToCanvas` method) we have an array named `pixels` with the original pixel data and another empty one in `imgData.data`, where `imgData` is obtained as follows:

```
var imgData = ctx.createImageData(canvas.width, canvas.height);
```

We can pass both these arrays into a component directly. A limitation here is that arrays passed to a WinRT component can be used for input *or* output, but not both—a component cannot just manipulate an array in place. The topic [Passing arrays to a Windows Runtime component]() has the fine details. To make the story short, we fortunately already have an input array, `pixels`, and an `output` array, `imgData.data`, that we can pass to a method in the component:

```
var pc = new PixelCruncherCS.Grayscale();
pc.convert(pixels, imageData.data);    //Note casing on method name
```

> **Note** The techniques shown here and in the article linked to above apply only to *synchronous* methods in the WinRT component; arrays *cannot* be used with asynchronous operations. See "Key Concepts for WinRT Components" later in this chapter for more on this topic.

To receive this array in C#, both parameters must to be appropriately marked with their directions. Such marks in C# are called *attributes*, not to be confused with those in HTML, and they appear in square brackets ([ ]) before the parameter name. In this particular case, the attributes appear as `[ReadOnlyArray()]` and `[WriteOnlyArray()]` preceding the parameters. (The `ReadOnlyArray` and `WriteOnlyArray` methods are found in the `System.Runtime.InteropServices.WindowsRuntime` namespace.) So the declaration of the method in the component, which again must be public, looks like this, just using a `Boolean` as a return type for the time being:

```
public Boolean Convert([ReadOnlyArray()] Byte[] imageDataIn,
    [WriteOnlyArray()] Byte[] imageDataOut)
```

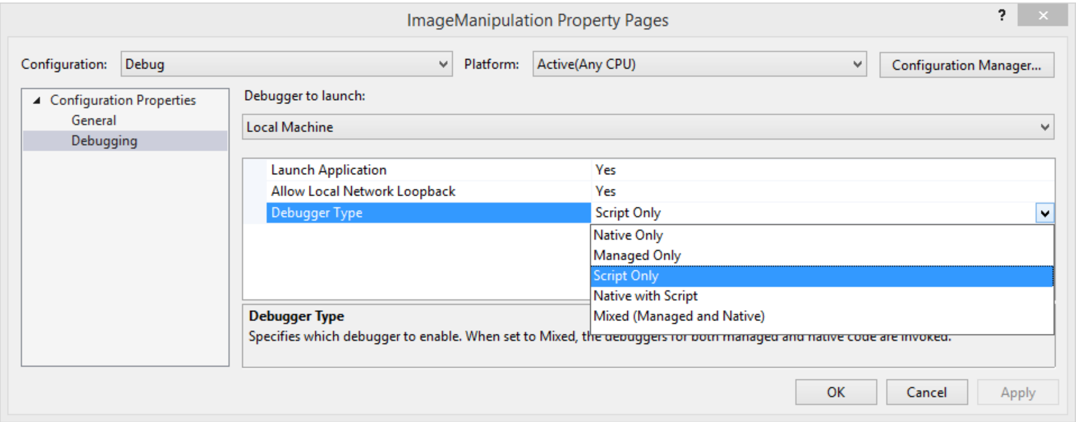
With this in place, it's a simple matter to convert the JavaScript grayscale code to C#:

```
public Boolean Convert([ReadOnlyArray()] Byte[] imageDataIn,
    [WriteOnlyArray()] Byte[] imageDataOut)
{
    int i;
    int length = imageDataIn.Length;
    const int colorOffsetRed = 0;
    const int colorOffsetGreen = 1;
    const int colorOffsetBlue = 2;
    const int colorOffsetAlpha = 3;

    Byte r, g, b, gray;

    for (i = 0; i < length; i += 4)
    {
        r = imageDataIn[i + colorOffsetRed];
        g = imageDataIn[i + colorOffsetGreen];
        b = imageDataIn[i + colorOffsetBlue];

        //Assign each rgb value to brightness for grayscale
        gray = (Byte)(.3 * r + .55 * g + .11 * b);

        imageDataOut[i + colorOffsetRed] = gray;
        imageDataOut[i + colorOffsetGreen] = gray;
        imageDataOut[i + colorOffsetBlue] = gray;
        imageDataOut[i + colorOffsetAlpha] = imageDataIn[i + colorOffsetAlpha];
    }

    return true;
}
```

Simultaneously Debugging Script and Managed/Native Code

One of the challenges in writing a mixed language app with WinRT components is debugging code that's written in different languages, especially as you step across the component boundary. Fortunately it's not too difficult to make this work.

The key is changing the debugger type in Visual Studio. Right-click your app project in solution explorer, select Debugging on the left, and choose a Debugger Type on the right, as shown in Figure 18-3. In this drop-down list, Script refers to JavaScript, Managed refers to C# or Visual Basic, and Native refers to C++, and you can see that you can choose from different combinations.



ImageManipulation Property Pages

Configuration: Debug

Platform: Active(Any CPU)

Debugger to launch: Local Machine

Launch Application: Yes

Allow Local Network Loopback: Yes

Debugger Type: Script Only

Debugger Type
Specifies which debugger to enable. When set to Mixed, the debuggers for both managed and native code are invoked.

FIGURE 18-3 Debugger types in Visual Studio.

If you choose any of the "Only" options, it means that you can set breakpoints and step through code only in that language. These have their place, as we'll see in a moment.

If you choose the Native With Script option (which was introduced with Visual Studio 2013 editions) and have a component written in C++, you can easily step from JavaScript into the C++ component and back—and set breakpoints in both—for a seamless debugging experience. This is one great reason to write C++ components!

If you have components written in both C#/VB and C++ in your project, choosing Mixed will allow you to debug all your component code but not your JavaScript. So what can you do here? That is, if you have only a component written in C#/VB like the one we worked with previously, is there a way to debug both the app and the component at the same time?

Fortunately, the answer is yes—you just need to use two instances of Visual Studio. Here's how (a tip of the hat to Rob Paveza for this):

4. Launch the app (F5) with a Script debugging option. This is the instance of Visual Studio in which you'll be stepping through JavaScript (and possibly C++ component code if you use Native With Script).

5. Load your project in a second instance of Visual Studio. Ignore warnings about IntelliSense files being already open, and prepare to wait a little while as Visual Studio builds a new copy of that database for this second instance.
6. In this second instance, set the debugging mode to an option that includes Managed so that you can debug C# or VB component code.
7. Select the Debug > Attach to Process menu command in the second instance. This displays a list of other running processes.
8. Find and select the line for WWAHost.exe (the app host for JavaScript) in the process list that has your app name in the title column.
9. Above the list, check the Attach To value. If it says "Automatic: Script code", press the Select button and indicate Managed and Native instead. Close that dialog.
10. Click the Attach button.
11. Now you can set breakpoints and step through your component code in this second instance of Visual Studio. By setting breakpoints at the beginning of the component's public methods, you'll stop there whenever the component is called from JavaScript.

Note that this method also works for debugging a C++ component in 2012 editions of Visual Studio if you have a project that's still targeting Windows 8.

Quickstart #2: Creating a Component in C++

To demonstrate a WinRT component written in C++, I've also added a project to the Image Manipulation example, calling it PixelCruncherCPP. The core code is the same as the C# example—manipulating pixels is a rather universal experience! The necessary code structure for the component, on the other hand, is unique to the language—C++ has ceremony all its own.

As we did with C#, let's start by adding a new project using the Visual C++ > Windows Runtime Component item template, using the PixelCruncherCPP name. After renaming *Class1* to *Tests* in the code and renaming the files, we'll have the following code in the header (which I call grayscale.h, and omitting a compiler directive):

```
namespace PixelCruncherCPP
{
   public ref class Tests sealed
   {
   public:
      Tests();
   };
}
```

where we see that the class must be `public ref` and `sealed`, with a public constructor. These together make the object instantiable as a WinRT component. In Tests.cpp we have the following:

```
#include "pch.h"
#include "Grayscale.h"

using namespace PixelCruncherCPP;
using namespace Platform;

Tests::Tests()
{
}
```

Again, not too much to go on, but enough. (Documentation for the Platform namespace, by the way, is part of the Visual C++ Language Reference.) To follow the same process we did for C#, let's add a static test method and a test property. The class definition is now:

```
public ref class Tests sealed
{
public:
    Tests();

    static Platform::String^ TestMethod(bool throwAnException);
    property int TestProperty;
};
```

and the code for `TestMethod` is this:

```
String^ Tests::TestMethod(bool throwAnException)
{
    if (throwAnException)
    {
        throw ref new InvalidArgumentException;
    }

    return ref new String(L"Tests.TestMethod succeeded");
}
```

When you build this project (Build > Build Solution), you'll see that we now get PixelCruncherCPP.dll and PixelCruncherCPP.winmd files. Whereas a C# assembly can contain both the code and the metadata, a C++ component compiles into separate code and metadata files. The metadata is again used to project the component's ABI into other languages and provides IntelliSense data for Visual Studio and Blend. If you now add a reference to this component in your app project—right-click the project > Add Reference > Solution, and choose PixelCruncherCPP, as in Figure 18-2—you'll find that IntelliSense works on the class when writing JavaScript code.

You'll also find that the casing of the component's property and method names have also been changed. In fact, with the exception of the namespace, PixelCruncherCPP, everything we did to use the C# component in JavaScript looks exactly the same, as it should be: the app *consuming* a WinRT component does not need to concern itself with the language used to *implement* that component. And remember that by choosing Native With Script debugging in Visual Studio, as described previously, you get a seamless experience across the component boundary.

Now we need to do the same work to accept arrays into the component, using Array and WriteOnlyArray as a reference. In C++, an input array is declared with `Platform::Array<T>^` and an output array as `Platform::WriteOnlyArray<T>^`, where we use `uint8` as the type here instead of `Byte` in C#:

```
bool Grayscale::Convert(Platform::Array<uint8>^ imageDataIn,
   Platform::WriteOnlyArray<uint8>^ imageDataOut)
```

The rest of the code is identical except for this one type change and for how we obtain the length of the input array, so we don't need to spell it out here. The code to invoke this class from JavaScript is also the same as for C#:

```
var pc2 = new PixelCruncherCPP.Grayscale();
pc2.convert(pixels, imgData.data);
```

Sidebar: The Windows Runtime C++ Template Library

Visual Studio includes what is called the Windows Runtime C++ Template Library (or WRL), which helps you write low-level WinRT components in C++. It's really a bridge between the raw COM level and what are called the C++/CX component extensions that we've been using here. If you have any experience with the Active Template Library (ATL) for COM, you'll find yourself right at home with WRL. For more information, see the documentation linked to above along with the Windows Runtime Component using WRL sample.

Comparing the Results

The Image Manipulation example in this chapter's companion content contains equivalent code in JavaScript, C#, and C++ to perform a grayscale conversion on image pixels. Taking a timestamp with `new Date()` around the code of each routine, I've compiled a table of performance numbers:[126]

	Average milliseconds (five samples; dual-core 2.5GHz processor)			**Temporary/residual increase in working set (MB)***		
Image Size	**JavaScript**	**C#**	**C++**	**JavaScript**	**C#**	**C++**
14.8K	8.4	7.2	6.4			
231K	45.2	40.0	33.8	10.7/6.0	15.5/10.1	6.1/2.7
656K	76.6	65.8	54.4			
1.98MB	798	728	598			
4.57MB	796	750	637	176.2/71.5	252.0/147.6	176.7/72.3

* Part of the residual increase comes from displaying the newly converted image, which is more or less the residual number for C++. The larger residual increases for JavaScript and C# come from allocations that are not yet garbage collected.

[126] You might be interested in a series of blog posts by David Rousset about building a camera app using HTML and JavaScript. In Part 4 of that series he offers much more in-depth performance analysis for a variety of devices, using pixel manipulation components much like we've been working with here.

A couple of notes and observations about these numbers and measuring them:

- When doing performance tests like this, be sure to set the build target to Release instead of Debug. This makes a tremendous difference in the performance of C++ code, because the compiler inserts all kinds of extra run-time checks in a debug build.
- When taking measurements, also be sure to run the Release app outside of the debugger (in Visual Studio select Debug > Start Without Debugging). If you've enabled script debugging, JavaScript will run considerably slower even with a Release build and could lead you to think that the language is far less efficient than it really is in production scenarios. Again, the JavaScript engine is highly optimized and performs better than you might think!
- If you run similar tests in the app itself, you'll notice that the time reported for the conversion is considerably shorter than the time it takes for the app to become responsive again. This is because the `canvas` element's `putImageData` method takes a long time to copy the converted pixels. Indeed, the majority of the time for the whole process is spent in `putImageData` and not the grayscale conversion.
- Assuming the CPU load for the grayscale conversion is roughly identical between the implementations, you can see that a higher performance component reduces the amount of time that the CPU is carrying that load. Over many invocations of such routines, this can add up to considerable power savings.
- The first time you use a WinRT component for any reason, it will take a little extra time to load the component and its metadata. The numbers above do not include first-run timings. Thus, if you're trying to optimize a startup process in particular, the extra overhead to load a component could mean that it's best to just do the job in JavaScript.

Now let's think about what these numbers mean for "performance." I put that word in quotes because *performance is ultimately something that is judged by your users*, not by a diagnostic tool that's giving you a measurement. What this means is that we have to look not just at a single measurement but at the larger picture of impact on the entire system. This is why I've also included a few measurements for working set, because for large images you can see that the impact is very significant and could affect the system in such a way as to hurt overall performance even though some particular routine in your app is running faster!

In short, always keep in mind that there is much more to measuring and improving app performance than just offloading computationally intensive routines to a WinRT component. Analyzing the performance of Windows Store apps and Analyzing the code quality of Windows Store apps with Visual Studio code analysis in the documentation will help you make a more thorough assessment of your app, as will *High-Performance Windows Store Apps* by Brian Rasmussen (Microsoft Press, 2014).

I also want to add that when I first ran these tests with the example program, I was seeing something like 100% improvements in C#/C++ over JavaScript. The reason for that came from the nature of the `canvas` element's `ImageData` object (as returned by the canvas's `createImageData` method) and had little to do with JavaScript. In my original JavaScript code (since corrected), I was dereferencing the `ImageData.data` array to set every *r*, *g*, *b*, and *a* value for each pixel. When I learned how dreadfully slow that particular dereference operation actually is, I changed the code to cache the array reference in another variable and suddenly the JavaScript version became amazingly faster. Indeed, minimizing identifier references is generally a good practice for better performance in JavaScript. For more on this and other performance aspects, check out *High Performance JavaScript*, by Nicholas C. Zakas (O'Reilly, 2010).

Anyway, looking at just the raw CPU story with my measurements, we can see that C# runs between 6–21% faster than the equivalent JavaScript and C++ 25–46% faster. C++ also runs 13–22% faster than C#. This shows that for noncritical code, *writing a component* ***won't*** *necessarily give you a good return on the time you invest*. It will be more productive to just stay in JavaScript most of the time and perhaps use a component in a few places where raw CPU performance really matters.

Comparing C# to C++, we can also see that the gain is incremental in terms of CPU, which means it's not always worth incurring the extra complexity of writing in C++. But when we add in the memory footprint numbers, the equation looks somewhat different.

For one thing, just loading a component written in C# or Visual Basic means loading the CLR into your process, which I've found incurs a 6-to-8-megabyte hit initially. More importantly, though, is the fact that the CLR has its own garbage collection process that's now running alongside that of JavaScript. With our grayscale conversion routines, the fact that we're marshaling large arrays of data across the component boundary means that we're allocating a bunch of memory in JavaScript, sharing it with the .NET Common Language Runtime (CLR), and then having to wait for both garbage collectors to figure out what, exactly, they can clean up (which might not happen until there is more demand for memory elsewhere). For this reason you can see that the C# component has a *much* higher residual memory footprint than the same routine written in C++.[127]

To further evaluate the effects of garbage collection and take the array allocations out of the picture, I added some simple counting routines in JavaScript, C#, and C++ to the Image Manipulation example and its components (they all look about the same as this JavaScript code):

```
function countFromZero(max, increment) {
    var sum = 0;

    for (var x = 0; x < max; x += increment) {
        sum += x;
    }

    return sum;
```

[127] It's also possible to avoid marshaling arrays (and the associated memory spike) altogether by just sending a `StorageFile` to the component, which I'll discuss later.

```
}
```

Running a count with a max of 1000 and an increment of 0.000001 (only use this increment outside the debugger—otherwise you might be waiting a while!), the timings I got averaged 2112ms for JavaScript, 1568ms for C#, and 1534ms for C++. Again, the difference between JavaScript and the other languages is significant (35–38% gain), but it's hardly significant between C# and C++. However, I occasionally found that, after loading a number of images and running the grayscale tests, counting in JavaScript and/or C# can take considerably longer than before, due to garbage collection falling behind my active use of the app.

Whatever the case, all of this becomes very important when your app is running on a device with all of 1GB of memory—the extra overhead could easily drag down overall system performance. If you're writing a component for just your app alone, you can decide how important this is for your particular customers. On the other hand, if you're writing a library for use by many other apps, some of which will be written in JavaScript, or if you simply want consistent and reliable performance for your component, we *highly* recommend implementing the component in C++ and thus minimizing the overhead.

Key Concepts for WinRT Components

The WinRT components we've just seen in "Quickstarts: Creating and Debugging Components" demonstrate the basic structure and operation of such components, but clearly there is much more to the subject. Because exploring all the nuances is beyond the scope of this chapter, I'll refer you again to the references given in this chapter's introduction. Here I'll offer a summary of the most relevant points, followed by separate sections on asynchronous methods and the projection of WinRT into JavaScript.

Component Structure

- The output of a C#/VB component project is a .NET assembly with Windows metadata in a single .winmd file; the output of a C++ component is a DLL with the code and a separate .winmd file with the metadata.
- Apps that use components must include the .winmd/DLL files in their projects and add a reference to them; it's not necessary to include the component source.
- Component classes that can be used by other languages are known as *activatable* classes. In C# these must be marked as `public sealed`, in Visual Basic as `Public NotInheritable`, and in C++ as `public ref sealed`. A component must have one activatable class to be usable from other languages.
- Classes can have `static` members (methods and properties) that are usable without instantiating an object of that class.
- A component can contain multiple public activatable classes as well as additional classes that

are internal only. All public classes must reside in the same root namespace, which has the same name as the component metadata file.

- It is possible to create a data source in a WinRT component and bind it to controls in JavaScript. The simplest way is to use the WinJS.Binding.oneTime initializer, the reasons for which are described in Chapter 6, "Data Binding, Collections, and Templates," under "Sidebar: Binding to WinRT Objects." More work is needed in JavaScript if you want to do one-way or two-way binding, but it is possible. See the WinMD and Databinding post on the MSDN forum. Furthermore, if you need to also use converter in the binding relationship, then it's necessary to write your own binding initializer and duplicate some of the functionality of oneTime.

- By default, all public classes in a component are visible to all other languages. A class can be hidden from JavaScript by applying the WebHostHiddenAttribute (that is, prefix the class declaration with [Windows.Foundation.Metadata.WebHostHidden] in C# or [Windows::Foundation::Metadata::WebHostHidden] in C++). This is appropriate for classes that work with UI (that cannot be shared with JavaScript, such as the whole of the Windows.Xaml namespace in WinRT) or others that are redundant with intrinsic JavaScript APIs (such as Windows.Data.Json which is redundant with JSON.*).

- For some additional structural options, see the following samples in the Windows SDK (all of which use the WRL; see "Sidebar: The Windows Runtime C++ Template Library (WRL)" under "Quickstart #2"):

    - Creating a Windows Runtime in-process component sample (C++/CX)
    - Creating a Windows Runtime EXE component with C++ sample
    - Creating a Windows Runtime DLL component with C++ sample
    - Windows Runtime in-process component authoring with proxy/stub generation sample

Types

- Within a component, you can use native language types (that is, .NET types and C++ runtime types). At the level of the component interface (the Application Binary Interface, or ABI), you must use WinRT types or native types that are implemented with WinRT types. Otherwise those values cannot be projected into other languages. In C++, WinRT types exist in the Platform namespace, and see Type System (C++/CX); in C#/VB, they exist in the System namespace, and see .NET Framework mappings of Windows Runtime types.

- A component can use structures created with WinRT types, which are projected into JavaScript as objects with properties that match the struct members.

- Collections must use specific WinRT types found in Windows.Foundation.Collections, such as IVector, IMap (and IMapView), and IPropertySet. This is why we've often encountered vectors, maps, and property sets throughout this book.

- Arrays are a special consideration because they can be passed in only one direction, as we saw in the quickstarts; each must therefore be marked as read-only or write-only. See Passing arrays to a Windows Runtime component. Furthermore, arrays cannot be effectively used with async methods, because an output array will not be transferred back to the caller when the async operation is complete. We'll talk more of this in "Implementing Asynchronous Methods" below.

Component Implementation

- When creating method overloads, make the *arity* (the number of arguments) different for each one because JavaScript cannot resolve overloads by type only. If you do create multiple overloads with the same arity, one *must* be marked with the `DefaultOverloadAttribute` so that the JavaScript projection knows which one to use. A side-effect of this, however, is that the other overloads with the same arity are then inaccessible to JavaScript altogether. In this case, be sure to provide distinct overloads for each critical method.
- A *delegate* (an anonymous function in JavaScript parlance) is a function object. Delegates are used for events, callbacks, and asynchronous methods. Declaring a delegate defines a function signature.
- The `event` keyword marks a public member of a specific delegate type as an event. Event delegates—the signature for a handler—can are typically a `Windows.Foundation.EventHandler<TResult>` or a `TypedEventHandler<TSender, TResult>`. In all cases, support for JavaScript requires a little extra work to ensure that the event is raised on the UI thread. The details for this can be found on Raising events in Windows runtime components and in the four samples linked to above in "Component Structure." We'll see a concrete example later in "Library Components."
- Throwing exception: use the `throw` keyword in C#, VB, and C++. In C#/VB, you throw a new instance of an exception type in the System namespace. In C++, you use `throw ref new` with one of the exception types within the `Platform` namespace, such as `Platform::InvalidArgumentException`. These appear in JavaScript with a stack trace in the message field of the exception; the actual message from the component will appear in Visual Studio's exception dialog.

Implementing Asynchronous Methods

For as fast as the C# and C++ routines that we saw in the quickstarts might be, the fact of the matter is that they still take more than 50ms to complete while running on the UI thread. This is the recommended threshold at which you should consider making an operation asynchronous. This means running that code on other threads so that the UI thread isn't blocked at all. To cover the basics, I'll now show how to implement asynchronous versions of the simple `countFromZero` function we saw earlier in "Sidebar: Managed vs. Native." We'll do it first with a worker and then in C# and C++.

For C#/VB and C++ there is quite extensive documentation on creating async methods. The cookbook topics we've referred to already cover this in the subsections called Asynchronous operations and Exposing asynchronous operations for C#, and the "To add the public members" section in the C++ walkthrough. There is also Creating Asynchronous Operations in C++ for Windows Store apps, along with a series of comprehensive posts on the Windows Developer Blog covering both app and component sides of the story: Keeping apps fast and fluid with asynchrony in the Windows Runtime, Diving Deep with WinRT and await, and Exposing .NET tasks as WinRT asynchronous operations. Matching the depth of these topics here would be a pointless exercise in repetition, so the sections that follow focus on creating async versions of the pixel-crunching methods from the quickstarts and the lessons we can glean from that experience.

This turns out to be a fortuitous choice. The particular scenario that we've worked with—performing a grayscale conversion on pixel data and sending the result to a canvas—just so happens to reveal a number of complications that are instructive to work through and are not addressed directly in the other documentation. These include troubles with passing arrays between the app and a component, which introduces an interesting design pattern that is employed by some WinRT APIs. Even so, the solution brings us to something of a stalemate because of the limitations of the HTML `canvas` element itself. This forces us to think through some alternatives, which is a good exercise because you'll probably encounter other difficulties in your own component work.

JavaScript Workers

For pure JavaScript, workers are the way you offload code execution to other threads, where those tasks do not need access to the `document`, `window`, or `parent` objects, or anything else in the DOM.[128] A key point to understand here is that communication between the main app (the UI thread) and workers happens through the singular `postMessage` method and the associated `message` events. Workers are not like components in which you can just call methods and get results back. If you want to invoke methods inside that worker with specific arguments, you must make those calls through `postMessage` with a message that contains the desired values. On the return side, a function that's invoked inside the worker sends its results to the main app through its own call to `postMessage`.

Various examples can be found in the JavaScript Web Workers app multitasking sample in the SDK. Here let me offer a simple one from the Image Manipulation example in this chapter—which is growing beyond its original intent for sure! I placed the `countFromZero` function into js/worker_count.js along with a message handler that serves as a simple method dispatcher:

```
onmessage = function (e) {
    switch (e.data.method) {
        case "countFromZero":
            countFromZero(e.data.max, e.data.increment);
            break;
```

[128] There are also a few libraries that build on workers to provide parallel processing. Just do a web search for "parallel JavaScript" and you'll find them.

```
        default:
            break;
    }
};

function countFromZero(max, increment) {
    var sum = 0;
    max = 10;

    for (var x = 0; x < max; x += increment) {
        sum += x;
    }

    postMessage({ method: "countFromZero", sum: sum });
}
```

When this worker is started, the only code that executes is the `onmessage` assignment. When that handler receives the appropriate message, it then invokes `countFromZero`, which in turn posts its results. In other words, setting up a worker just means converting method calls and results into messages.

Invoking this method from the app now looks like this:

```
var worker = new Worker('worker_count.js');
worker.onmessage = function (e) { //e.data.sum is the result }

//Call the method
worker.postMessage({ method: "countFromZero", max: 1000, increment: .00005 });
```

Keep in mind that `postMessage` is itself an asynchronous operation—there's no particular guarantee about how quickly those messages will be dispatched to the worker or the app. Furthermore, when a worker is created, it won't start executing until script execution yields (as when you call `setImmediate`). This means that workers are not particularly well suited for async operations that you want to start as soon as possible or for those where you want to get the results as soon as they are available. For this reason, workers are better for relatively large operations and ongoing processing; small, responsive, and high-performance routines are better placed within WinRT components.

The `postMessage` mechanism is also not the best for chaining multiple async operations together, as we're easily able to do with promises that come back from WinRT APIs. To be honest, I don't even want to start thinking about that kind of code! I prefer instead to ask whether there's a way that we can effectively wrap a worker's messaging mechanism *within* a promise, such that we can treat async operations the same regardless of their implementation.

We need to get the result from within the `worker.onmessage` handler and send it to a promise's completed handler. To do that, we use a bit of code in the main app that's essentially what the JavaScript projection layer uses to turn an async WinRT API into a promise itself:

```
// This is the function variable we're wiring up.
var workerCompleteDispatcher = null;
var promiseJS = new WinJS.Promise(function (completeDispatcher, errorDispatcher,
    progressDispatcher) {
    workerCompleteDispatcher = completeDispatcher;
});

// Worker would be created here and stored in the 'worker' variable

// Listen for worker events
worker.onmessage = function (e) {
    if (workerCompleteDispatcher != null) {
        workerCompleteDispatcher(e.data.sum);
    }
}

promiseJS.done(function (sum) {
    // Output for JS worker
});
```

To repeat a few points from Appendix A, "Demystifying Promises," a promise is a separate thing from the async operation itself. (It has to be, because WinRT APIs and components know nothing of promises.) In many ways a promise is just a tool to manage a bunch of listener functions on behalf of an async operation, like our worker here. When an async operation detects certain events—namely, completed, error, and progress—it wants to notify whoever has expressed an interest in those events. Those whoevers have done so by calling a promise's `then` or `done` methods and providing one or more handlers.

Within `then` or `done`, a promise just saves those functions in a list (unless it knows the async operation is already complete, in which case it can call the completed or error functions immediately). This is why you can call `then` or `done` multiple times on the same promise—it just adds your completed, error, and progress handlers to the appropriate list within the promise. Of course, those lists are useless without some way to invoke the handlers they contain. For this purpose, a promise has three functions of its own to iterate each list and invoke the registered listeners. That's again the core purpose of a promise: maintain lists of listeners and call those listeners when asked.

The code that starts up an async operation, then, will want to use a promise to manage those listeners, hence the call to `new WinJS.Promise`. When that promise is initialized, this function you pass to the constructor is called with references to the *dispatcher* functions that notify the listeners. The async operation code saves the dispatchers it needs later. In our worker's case, we're interested only in notifying the completed handlers, so we save that dispatcher in `workerCompleteDispatcher`.

When we then detect that the operation is complete—that is, when we receive the appropriate message from the worker—we check to make sure `workerCompleteDispatcher` is a good reference and then call it with the result value. That dispatcher will again loop through all the registered listeners and call them with that same result. In the code above, the only such listener is the anonymous function we gave to `promiseJS.done`.

Truth be told, it's really just mechanics. To handle errors and progress, we'd simply save those dispatchers as well, add more specific code inside the `onmessage` event handler that would check e.data for other status values from the worker, and invoke the appropriate dispatcher in turn. Such relationships are illustrated in Figure 18-4.

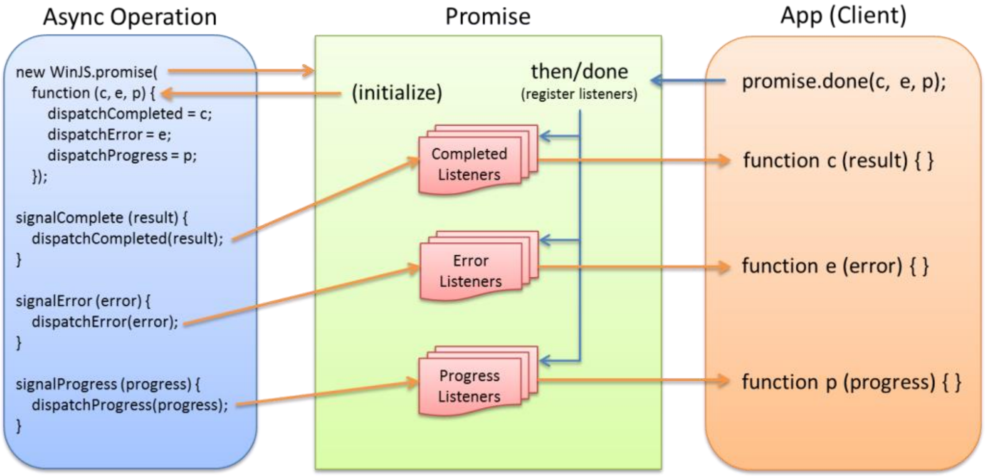


FIGURE 18-4 A promise manages and invokes listeners on behalf of an async operation.

Again, everything you see here with the exception of the call to done (which is client code and not part of the async operation) is what the JavaScript projection layer does for an async operation coming from WinRT. In those cases the async operation is represented by an object with an IAsync* interface instead of a worker. Instead of listening for a worker's message event, the projection layer just wires itself up through the IAsync* interface and creates a promise to manage connections from the app.

The code above is included in the Image Manipulation example accompanying this chapter. It's instructive to set breakpoints within all the anonymous functions and step through the code to see exactly when they're called, even to step into WinJS and see how it works. In the end, what's meaningful is that this code gives us a promise (in the promiseJS variable) that looks, feels, and acts like any other promise. This will come in very handy when we have promises from other async operations, as explained later in "Sidebar: Joining Promises." It means that we can mix and match async operations from WinRT APIs, WinRT components, and workers alike.

Async Basics in WinRT Components

Within a WinRT component, there are three primary requirements to make any given method asynchronous. First, append *Async* to the method name, a simple act that doesn't accomplish anything technically (and isn't technically required) but clearly communicates to callers that their need to treat the method differently from synchronous ones.

Second, the return value of the method must be one of the following Windows.Foundation interfaces, shown in the table below, each one representing a particular combination of async behaviors, namely whether the method produces a result and whether the method is capable of reporting progress:

Interface (in Windows.Foundation)	Use Case
IAsyncAction	Use for an async method that produces no results (no arguments are sent to the completed handler) and reports no progress.
IAsyncActionWithProgress<TProgress>	Use for an async method that produces no results but does report progress, where <TProgress> is the data type of the argument sent to the progress handler.
IAsyncOperation<TResult>	Use for an async method that produces results of type <TResult> but reports no progress.
IAsyncOperationWithProgress<TResult, TProgress>	Use for an async method that produces results of type <TResult> and reports progress with type <TProgress> to a progress handler.

Having chosen the type of async method we're creating, we now have to run the method's code on another thread. It is possible here to utilize threads directly, using the thread pool exposed in the Windows.System.Threading API, but there are higher level constructs in both C#/VB and C++ that make the job much easier.

Async Methods in C#/Visual Basic In C# and Visual Basic we have the System.Threading.Tasks.Task class for this purpose. A Task is created through one of the static Task.Run methods. To this we give an anonymous function (called a *delegate* in .NET, defined with a *lambda operator* =>) that contains the code to execute. To then convert that Task into an appropriate WinRT async interface, we call the task's AsAsyncAction or AsAsyncOperation extension method. Here's what this looks like in a generic way:

```
public IAsyncOperation<string> SomeMethodAsync(int id)
{
    var task = Task.Run<string>( () =>  // () => in C# is like function () in JS
    {
        return "Here is a string.";
    });

    return task.AsAsyncOperation();
}
```

If the code inside the task itself performs any asynchronous operations (for which we use the C# await keyword as described in the blog posts linked earlier), the delegate must be marked with async:

```
public IAsyncOperation<string> SomeMethodAsync(int id)
{
    var task = Task.Run<string>(async () =>
    {
        var idString = await GetMyStringAsync(id); // await makes an async call looks sync
        return idString;
    });

    return task.AsAsyncOperation();
}
```

Note that Task.Run does not support progress reporting and the AsAsyncAction and

AsAsyncOperation extension methods don't support cancellation. In these cases you need to use the System.Runtime.InteropServices.WindowsRuntime.AsyncInfo class and one of its Run methods as appropriate to the chosen async behavior. The Task.AsAsyncOperation call at the end is unnecessary here because AsyncInfo.Run already provides the right interface:

```
public IAsyncOperation<string> SomeMethodAsync(int id)
{
    return AsyncInfo.Run<string>(async (token) =>
    {
        var idString = await GetMyStringAsync(id);
        token.ThrowIfCancellationRequested();
        return idString;
    });
}
```

In this code, AsyncInfo.Run provides the delegate with an argument of type System.Threading.CancellationToken. To support cancellation, you must periodically call the token's ThrowIfCancellationRequested method. This will pick up whether the original caller of the async method has cancelled it (for example, calling a promise's cancel method). Because cancellation is typically a user-initiated action, there's no need to call ThrowIfCancellationRequested inside a very tight loop; calling it every 50 milliseconds or so will keep the app fully responsive.

Alternately, if a method like *GetMyStringAsync* accepted the CancellationToken, you could just pass the token to it. One strength of the CancellationToken model is that it's highly composable: if you receive a token in your own async call, you can hand it off to any number of other functions you call that also accept a token. If cancellation happens, the request will automatically be propagated to all those operations.

Note that WinRT methods can accept a token because of an AsTask overload. Instead of this:

```
await SomeWinRTMethodAsync();
```

you can use this:

```
await SomeWinRTMethodAsync().AsTask(token);
```

Anyway, given these examples, here's a noncancellable async version of CountFromZero:

```
public static IAsyncOperation<double> CountFromZeroAsync(double max, double increment)
{
    var task = Task.Run<double>(() =>
    {
        double sum = 0;

        for (double x = 0; x < max; x += increment)
        {
            sum += x;
        }

        return sum;
    });
    return task.AsAsyncOperation();
}
```

The IAsyncOperation interface returned by this method, like all the async interfaces in Windows.Foundation, gets projected into JavaScript as a promise, so we can use the usual code to call the method and receive its results (asyncVars is just an object to hold the variables):

```
asyncVars.startCS = new Date();
var promiseCS = PixelCruncherCS.Tests.countFromZeroAsync(max, increment);
promiseCS.done(function (sum) {
    asyncVars.timeCS = new Date() - asyncVars.startCS;
    asyncVars.sumCS = sum;
});
```

With code like this, which is in the Image Manipulation example with this chapter, you can start the async counting operations (using the "Counting Perf (Async)" button) and then immediately go open an image and do grayscale conversions at the same time.

Async Methods in C++ To implement an async method in C++, we need to produce the same end result as in C#: a method that returns one of the IAsync* interfaces and runs its internal code on another thread.

The first part is straightforward—we just need to declare the method with the C++ types (shown here in the C++ code; the class declaration in Grayscale.h is similar):

```
using namespace Windows::Foundation;
IAsyncOperation<double>^ Tests::CountFromZeroAsync(double max, double increment)
```

The C++ analogue of the AsyncInfo class is a task found in what's called the Parallel Patterns Library for C++, also known as PPL or the Concurrency Runtime, whose namespace is concurrency. (Use a #include <ppltasks.h> and using namespace concurrency; in your C++ code, and you're good to go.) The function that creates a task is called create_async. All we need to do is wrap our code in that function:

```
IAsyncOperation<double>^ Tests::CountFromZeroAsync(double max, double increment)
{
    return create_async([max, increment]()
    {
        double sum = 0;

        for (double x = 0; x < max; x += increment)
        {
            sum += x;
        }

        return sum;
    });
}
```

As with C#, there are additional structures for when you're nesting async operations, supporting cancellation, and reporting progress. I will leave the details to the documentation. See Asynchronous Programming in C++ and Task Parallelism, but one short excerpt from the CircusCannon2 example for this chapter will illustrate that async chaining in C++ looks a lot like promise chains in JavaScript:

```
IAsyncOperation<bool>^ Controller::ConnectAsync()
{
    return create_async([=]()
    {
        Platform::String ^ ccSelector = HidDevice::GetDeviceSelector(
            DeviceProperties::usage_page, DeviceProperties::usage_id,
            DeviceProperties::vid, DeviceProperties::pid);
        IAsyncOperation<DeviceInformationCollection^>^ deviceOp =
            DeviceInformation::FindAllAsync(ccSelector);

        return create_task(deviceOp).then([this](DeviceInformationCollection^ devices) {
            if (devices->Size == 0) {
                //This error message doesn't get through...not sure what the solution is.
                throw "No Circus Cannon devices found";
            }

            //Take the first one
            m_lastDevice = devices->GetAt(0);

            //Do the connect and return its result
            return _connectAsync();
        });
    });
}

task<bool> Controller::_connectAsync()
{
    if (nullptr == m_lastDevice) {
        throw "No device was enumerated so last device id is undefined.";
    }

    IAsyncOperation<HidDevice^>^ deviceOp = HidDevice::FromIdAsync(m_lastDevice->Id,
        FileAccessMode::ReadWrite);

    auto task = create_task(deviceOp);

    return task.then([this](HidDevice^ device) {
        if (nullptr == device) {
            m_isConnected = false;
            m_status = _statusString(m_lastDevice->Id);
        } else {
            m_status = "OK";
            m_isConnected = true;
            m_device = device;
            _sendCommand(Commands::Stop);
            _registerEvents();
        }

        return m_isConnected;
    });
}
```

Here the first create_async call wraps a block of code to run it on a separate thread. Within that code we make another async call (FindAllAsync) and create a task from it. To chain, we use that task's then method, provide a function (lambda) to receive the results, and return another task from another function whose ultimate return value is a Boolean.

Sidebar: Joining Promises

There's one detail from the Image Manipulation example that takes advantage of having all the async operations managed through promises. In the app, we show a horizontal progress indicator before starting all the async operations with the Counting Perf (Async) button:

```
function testPerfAsync() {
    showProgress("progressAsync", true);
    //...
}
```

We want this control to stay visible while any of the async operations are still running, something that's easily done with WinJS.Promise.join. What I wanted to point out is that because you can call a promise's then or done as many times as you want, it's just fine to attach handlers on each of the individual promises and to attach separate handlers to join:

```
promiseJS.done(function (sum) {
    // Output for JS worker
}

promiseCS.done(function (sum) {
    // Output for C# component
})

promiseCPP.done(function (sum) {
    // Output for C++ component
});

WinJS.Promise.join([promiseJS, promiseCS, promiseCPP]).done(function () {
    // Hide progress indicator when all operations are done
    showProgress("progressAsync", false);
});
```

Now we see how much we simplify everything by wrapping a worker's message mechanism within a promise! Without doing so, we'd need to maintain one flag to indicate whether the promises were fulfilled (set to true inside the join) and another flag to indicate if the worker's results had been received (setting it to true inside the worker's message handler). Inside the join, we'd need to check if the worker was complete before hiding the progress indicator; the worker's message handler would do the same, making sure the join was complete. This kind of thing might be manageable on a small scale but would certainly get messy with more than a few parallel async operations—which is the reason promises were created in the first place!

Arrays, Vectors, and Other Alternatives

Now that we've seen the basic structure of asynchronous methods in WinRT components, let's see how we might create an asynchronous variant of the synchronous `Convert` methods we implemented earlier. For the purpose of this exercise we'll just stick with the C# component though, again, such things are generally best written in C++.

It would be natural with `Convert` to consider `IAsyncAction` as the method's type, because we already return results in an output array. This would, in fact, be a great choice if we were using types *other* than an array. However, arrays present a variety of problems with asynchronous methods. First, although we can pass the method both the input and output arrays and the method can do its job and populate that output array, its contents won't actually be transferred back across the async task boundary at present. So the completed handler in the app will be called as it would expect, but the output array passed to the async method will still be empty.

The next thing we can try is to turn the async action into an operation that produces a result. We might consider a return type of `IAsyncOperation<Byte[]>` (or an equivalent one using progress), where the method would create and populate the array it returns. The problem, however, is that the app receiving this array wouldn't know how to release it—clearly some memory was allocated for it, but that allocation happened inside a component and not inside JavaScript, so there's no clear rule on what to do. Because this is a sure-fire recipe for memory leaks, returning arrays like this isn't supported.

An alternative is for the async method to return a specific WinRT collection type (where there are clear rules for deallocation), such as an `IList<Byte>`, which will be converted to a vector in JavaScript that can also be accessed as an array. (Note that `IList` is specific to .NET languages; the C++ walkthrough topic shows how to use a vector directly with the `concurrent_vector` type.) Here's a simple example of such a method:

```
public static IAsyncOperation<IList<Byte>> CreateByteListAsync(int size)
{
    var task = Task.Run<IList<Byte>>(() =>
    {
        Byte [] list = new Byte[size];

        for (int i = 0; i < size; i++)
        {
            list[i] = (Byte)(i % 256);
        }

        return list.ToList();
    });

    return task.AsAsyncOperation();
}
```

Applying this approach to the grayscale routine, we get the following `ConvertPixelArrayAsync` (see PixelCruncherCS > ConvertGrayscale.cs), where the `DoGrayscale` is the core code of the routine broken out into a separate function, the third parameter of which is a periodic callback that we can use to handle cancellation). This is the function that's called if you select Async w/ Vector from the drop-down list in the Image Manipulation app's UI:

```
public IAsyncOperation<IList<Byte>> ConvertPixelArrayAsync([ReadOnlyArray()]
    Byte[] imageDataIn)
{
    //Use AsyncInfo to create an IAsyncOperation that supports cancellation
    return AsyncInfo.Run<IList<Byte>>((token) => Task.Run<IList<Byte>>(() =>
    {
        Byte[] imageDataOut = new Byte[imageDataIn.Length];
        DoGrayscale(imageDataIn, imageDataOut, () =>
            {
                token.ThrowIfCancellationRequested();
            });

        return imageDataOut.ToList();
    }, token));
}
```

A fourth approach is to follow the pattern used by the `Windows.Graphics.Imaging.PixelDataProvider` class, which we're already using in the Image Manipulation example. In the function `setGrayscale` (js/default.js), we open a file obtained from the file picker and then decode it with `BitmapDecoder.getPixelDataAsync`. The result of this operation is the `PixelDataProvider` that has a method called `detachPixelData` to provide us with the pixel array (some code omitted for brevity):

```
function setGrayscale(componentType) {
    imageFile.openReadAsync().then(function (stream) {
        return Imaging.BitmapDecoder.createAsync(stream);
    }).then(function (decoderArg) {
        //Configure the decoder ... [code omitted]
        return decoder.getPixelDataAsync();
    }).done(function (pixelProvider) {
        copyGrayscaleToCanvas(pixelProvider.detachPixelData(),
                decoder.pixelWidth, decoder.pixelHeight, componentType);
    });
}
```

A similar implementation of our grayscale conversion routine is in PixelCruncherCS > ConvertGrayscale.cs in the function `ConvertArraysAsync`. Its type is `IAsyncAction` because it operates against the `Grayscale.inputData` array (which must be set first). The output is accessed from `Grayscale.detatchOutputData()`. Here's how the JavaScript code looks, which is used if you choose the Async w/ Properties option in the Image Maniupulation app's UI:

```
pc1.inputData = pixels;
pc1.convertArraysAsync().done(function () {
    var data = pc1.detachOutputData()
    copyArrayToImgData(data, imgData);
    updateOutput(ctx, imgData, start);
});
```

You might be wondering about that copyArrayToImgData function in the code above. I'm glad you are, because it points out a problem that forces us to take a different approach altogether, one that leads us to an overall better solution!

All along in this example we've been loading image data from a file, using the BitmapDecoder, and then converting those pixels to grayscale into an array provided by the canvas element's createImageData method. Once the data is inside that image data object, we can call the canvas's putImageData method to render it. All of this was originally implemented to show interaction with the canvas, including how to save canvas contents to a file. That was fine for Chapter 13, where graphics were our subject. But if we're really looking to just convert an image file to grayscale, using a canvas isn't necessarily the best road to follow!

The key issue that we're encountering here is that the canvas's putImageData method accepts *only* an ImageData object created by the canvas's createImageData method. The canvas does not allow you to create and render a separate pixel array, nor insert a different array in the ImageData.data property. The only way it works is to write data directly into the ImageData.data array.

In the synchronous versions of our component methods, it was possible to pass ImageData.data as the output array so that the component could perform a direct write. Unfortunately, this isn't possible with the async versions. Those methods can provide us with the converted data all right, but because we can't point ImageData.data to such an array, we're forced to use a routine like copyArrayToImageData function to copy those results into ImageData.data, byte by byte. Urk. That pretty much negates any speed improvement we might have realized by creating components in the first place!

Let me be clear that this is a limitation of the canvas element, not of WinRT or components in general. Moving arrays around between apps and components, as we've seen, works perfectly well for other scenarios (remembering, of course, that having multiple garbage collectors running between JavaScript and C# arrays can incur memory overhead). Still, the limitation forces us to ask whether we even have the right approach at all.

Taking a step back, the whole purpose of the demonstration is to convert an image file to grayscale and show that conversion on the screen. Using a canvas is just an implementation detail—we can achieve the same output in other ways. For example, instead of converting the pixels into a memory array, we could create a temporary file by using the Windows.Graphics.Image.BitmapEncoder class instead, just like we use in the SaveGrayscale function that's already in the app. We'd just give it the converted pixel array instead of grabbing those pixels from the canvas again. Then we can use a thumbnail, URL.createObjectURL or an ms-appdata:/// URI to display it in an img element. This would likely perform much faster because the canvas's putImageData method actually takes a long time to run, much longer than the conversion routines in our components.

Along these same lines, there's no reason that we couldn't place more of the whole process inside a component. Only those parts that deal with UI need to be in JavaScript: the rest can be written in another language. For example, why bother shuttling pixel arrays between JavaScript and a WinRT component? Once we get a source StorageFile from the file picker we can pass *that* to a component method directly. The component can then use the BitmapDecoder to obtain the pixel stream, convert it, and then create the temporary file and write the converted pixels back out using the `BitmapEncoder`, handing back a `StorageFile` for the temp file from which we can set an `img.src` or grab a thumbnail. The pixels, then, never leave the component and never have to be copied between memory buffers. This should result in both faster throughput as well as a smaller memory footprint.

To this end the PixelCruncherCS project in the Image Manipulation example has another async method called `ConvertGrayscalFileAsync` that does exactly what I'm talking of here:

```
public static IAsyncOperation<StorageFile> ConvertGrayscaleFileAsync(StorageFile file)
{
    return AsyncInfo.Run<StorageFile>((token) => Task.Run<StorageFile>(async () =>
    {
        StorageFile fileOut = null;

        try
        {
            //Open the file and read in the pixels
            using (IRandomAccessStream stream = await file.OpenReadAsync())
            {
                BitmapDecoder decoder = await BitmapDecoder.CreateAsync(stream);
                PixelDataProvider pp = await decoder.GetPixelDataAsync();
                Byte[] pixels = pp.DetachPixelData();

                //We know that our own method can convert in-place,
                //so we don't need to make a copy
                DoGrayscale(pixels, pixels);

                //Save to a temp file.
                ApplicationData appdata = ApplicationData.Current;

                fileOut = await appdata.TemporaryFolder.CreateFileAsync(
                    "ImageManipulation_GrayscaleConversion.png",
                    CreationCollisionOption.ReplaceExisting);

                using (IRandomAccessStream streamOut =
                    await fileOut.OpenAsync(FileAccessMode.ReadWrite))
                {
                    BitmapEncoder encoder = await BitmapEncoder.CreateAsync(
                        BitmapEncoder.PngEncoderId, streamOut);

                    encoder.SetPixelData(decoder.BitmapPixelFormat, decoder.BitmapAlphaMode,
                        decoder.PixelWidth, decoder.PixelHeight,
                        decoder.DpiX, decoder.DpiY, pixels);

                    await encoder.FlushAsync();
                }
            }
        }
        catch
        {
            //Error along the way; clear fileOut
            fileOut = null;
        }
        //Finally, return the StorageFile we created, which makes it convenient for the
        //caller to copy it elsewhere, use in a capacity like URL.createObjectURL, or refer
        //to it with "ms-appdata:///temp" + fileOut.Name
        return fileOut;
    }));
}
```

One thing we can see when comparing the equivalent JavaScript code with this is that the C# `await` keyword very much simplifies dealing with asynchronous methods—making them appear like they're synchronous. This is one potential advantage to writing code in a component! The other important detail is to note the `using` statements around the streams. Streams, among other types, are *disposable* (they have an `IDisposable` interface) and must be cleaned up after use or else files will remain open and you'll see access denied exceptions or other strange behaviors. The `using` statement encapsulates that cleanup logic for you.

In any case, with this method now we need only a few lines of JavaScript to do the job:

```
PixelCruncherCS.Grayscale.convertGrayscaleFileAsync(imageFile).done(function (tempFile) {
    if (tempFile != null) {
        document.getElementById("image2").src = "ms-appdata:///temp/" + tempFile.name;
    }
});
```

The line with the URI could be replaced with these as well (which I'm using only because I'm displaying the full image at its original size—if you don't need that full display, remember to use a thumbnail from the `StorageFile`):

```
var uri = URL.createObjectURL(tempFile, { oneTimeOnly: true });
document.getElementById("image2").src = uri;
```

Running tests with this form of image conversion, the app shows a much better response, so much so that the progress ring that's normally shown while the operation is running doesn't even appear! Furthermore, we reduce the overall memory spike for the operation by half, because we're loading the pixels into memory just once, inside the component, rather than populating arrays in both the app and the component.

All this illustrates the final point of this whole exercise: if you're looking for optimizations, think beyond just the most computationally intensive operations, especially if it involves moving lots of data around. As we've seen here, challenging our first assumptions can lead to a much better solution.

Projections into JavaScript

Already in this chapter we've seen some of the specific ways that a WinRT component is projected into JavaScript. In this section I'll offer a fuller summary of how this world of WinRT looks from JavaScript's point of view.

Let's start with naming. We've seen that a JavaScript app project must add a component as a reference, at which point the component's namespace becomes inherently available in JavaScript; no other declarations are required. The namespace and the classes in the component just come straight through into JavaScript. What does change, however, are the names of methods, properties, and events. Although namespaces and class names are projected as they exist in the component, method and property names (including members of `struct` and `enum`) are converted to camel casing: *TestMethod* and *TestProperty* in the component become *testMethod* and *testProperty* in JavaScript. This casing change can have some occasional odd side effects, as when the component's name starts with two capital letters such as *UIProperty*, which will come through as *uIProperty*.

Event names, on the other hand, are converted to all lowercase as befits the JavaScript convention. An event named *SignificantValueChanged* in the component becomes *significantvaluechanged* in JavaScript. You'd use that lowercase name with `addEventListener`, and the class that provides it will also be given a property of that name prefixed with *on*, as in *onsignificantvaluechanged.* An important point with events is that it's sometimes necessary to explicitly call `removeEventListener` to prevent memory leaks. For a discussion, refer back to "WinRT Events and removeEventListener" in Chapter 3. In the context of this chapter, WinRT events include those that come from your own WinRT components.

Static members of a class, as we've seen, can just be referred to directly using the fully qualified name of that method or property using the component's namespace. Nonstatic members, on the other hand, are accessible only through an instance created with `new` or returned from a static member.

Next are two limitations that we've mentioned before but are worth repeating in this context. First is that a WinRT component cannot work with the UI of an app written in JavaScript. This is because the app cannot obtain a drawing surface of any kind that the component could use. Second is that JavaScript can resolve overloaded methods by arity (number of parameters) only and not by type. If a component provides overloads distinguished only by type, JavaScript can access only the one that's marked as the default.

Next we come to the question of data types, which is always an interesting subject where interoperability between languages is concerned. Generally speaking, what you see in JavaScript is naturally aligned with what's in the component. A WinRT `DateTime` becomes a JavaScript `Date`, a `TimeSpan` (based on 100ns units) becomes a number expressed in milliseconds, numerical values become a `Number`, `bool` becomes `Boolean`, strings are strings, and so on. Some WinRT types, like `IMapView` and `IPropertySet`, just come straight through to JavaScript as an object type (with the interface's methods) because there are no intrinsic equivalents. Then there are other conversions that are, well, more interesting:

- Asynchronous operations in a component that return interfaces like `IAsyncOperation` are projected into JavaScript as promises.
- Because JavaScript doesn't have a concept of `struct` as does C#, VB, and C++, structs from a WinRT component appear in JavaScript as objects with the struct's fields as members. Similarly, to call a WinRT component that takes a `struct` argument, a Javascript app creates an object with the fields as members and passes that instead. Note that the casing of `struct` members is converted to camel casing in JavaScript.

- Some collection types, like IVector, appear in JavaScript as an array but with different methods (refer to Chapter 6 in "Collection Data Types"). That is, the collection can be accessed using the array operator [ ], but its methods are different. Be careful, then, passing these to JavaScript manipulation functions that assume those methods exist.
- Enums are translated into objects with camel-cased properties corresponding to each enum value, where those values are JavaScript Number types.
- WinRT APIs sometimes return Int64 types (alone or in structs) for which there is no equivalent in JavaScript. The 64-bit type is preserved in JavaScript, however, so you can pass it back to WinRT in other calls. However, if you modify the variable holding that value, even with something as simple as a ++ operator, it will be converted into a JavaScript Number. Such a value will not be accepted by methods expecting an Int64.
- If a component method provides multiple output parameters, these show up in JavaScript as a single object with those different values as properties. No clear standard for this exists in JavaScript; it's best to avoid in component design altogether.

The bottom line is that the projection layer tries to make WinRT components written in any other language look and feel like they belong in JavaScript, without introducing too much overhead.

Scenarios for WinRT Components

Earlier in "Choosing a Mixed Language Approach" I briefly outlined a number of scenarios where WinRT components might be very helpful in the implementation of your app. In this section we'll think about these scenarios a little more deeply, and I'll point you to demonstrations of these scenarios in the samples, where such are available.

Higher Performance (Perhaps)

Increasing the performance of a Windows Store app written in HTML, CSS, and JavaScript is one of the primary scenarios for offloading some work to a WinRT component, but do so carefully.

When evaluating the performance of your app, keep an eye open for specific areas that are computationally intensive or involve moving a lot of data around. For example, if you found it necessary to implement an extended splash screen for those exact reasons, perhaps you can reduce the time the user has to wait (especially on first launch) before the app is active. Any other situation where the user might have to wait—and have better things to do than watch a progress indicator!—is a great place to use a high-performance component if possible. Clearly, in some scenarios the performance of the app isn't so much the issue as is network latency, but once you get data back from a service you might be able to pre-process it faster in a component than in JavaScript.

As another example, an app package might include large amounts of compressed data to minimize the size of its download from the Store, and it needs to decompress that data on first run. A WinRT component might significantly shorten initialization time. If the component uses WinRT APIs to write to your app data folders, all that data will also be accessible from JavaScript through those same APIs.

One challenge is that writing a component to chew on a bunch of data might mean passing JavaScript arrays into that component and getting arrays back out. As we saw in the quickstarts, this works just fine with synchronous operations but is not presently supported for async, which is how you'd often want to implement potentially long-running methods. Fortunately, there are ways around this limitation, either by transferring results for an async operation through synchronous properties or by using other collection types such as vectors.

That said, marshaling data across language boundaries has its costs, and in various cases the performance gain of code written in C++ will be lost in the overhead. For this reason, investigate whether you can keep the work centered around app data, which can be shared between the app and components without the transfer cost.

One place where performance is very significant is with background tasks. As explained in Chapter 16, background tasks are limited to a few seconds of CPU time every 15 minutes. Because of this, you can get much more work accomplished in a background task written in a higher-performance language than one written in JavaScript.

The structure of a component with such tasks is no different than any other, as demonstrated in the C# Tasks component included with the Background task sample. Each of the classes in the Tasks namespace is marked as `public` and `sealed`, and because the component is brought into the JavaScript project as a reference, those class names (and their public methods and properties) are in the JavaScript namespace. As a result, their names can be given to the `BackgroundTaskBuild.taskEntryPoint` property without any problems.

Another example of the same technique can be found in the Network status background sample.

Something that we didn't discuss in Chapter 16, but which is appropriate now, is that when you create a WinRT component for this purpose, the class that implements the background task must derive from `Windows.ApplicationModel.Background.IBackgroundTask` and implement its singular `Run` method. That method is what gets called when the background task is triggered. We can see this in the Network status background sample where the whole C# implementation of the component comprises just a few dozen lines of code (see BackgroundTask.cs in the sample's NetworkStatusTask project; some comments and debug output omitted):

```
namespace NetworkStatusTask
{
   public sealed class NetworkStatusBackgroundTask : IBackgroundTask
   {
      ApplicationDataContainer localSettings = ApplicationData.Current.LocalSettings;

      // The Run method is the entry point of a background task.
      public void Run(IBackgroundTaskInstance taskInstance)
      {
         // Associate a cancellation handler with the background task.
         taskInstance.Canceled += new BackgroundTaskCanceledEventHandler(OnCanceled);
         try
         {
            ConnectionProfile profile =
               NetworkInformation.GetInternetConnectionProfile();
            if (profile == null)
            {
               localSettings.Values["InternetProfile"] = "Not connected to Internet";
               localSettings.Values["NetworkAdapterId"] = "Not connected to Internet";
            }
            else
            {
               localSettings.Values["InternetProfile"] = profile.ProfileName;

               var networkAdapterInfo = profile.NetworkAdapter;
               if (networkAdapterInfo == null)
               {
                  localSettings.Values["NetworkAdapterId"] =
                     "Not connected to Internet";
               }
               else
               {
                  localSettings.Values["NetworkAdapterId"] =
                     networkAdapterInfo.NetworkAdapterId.ToString();
               }
            }
         }
         catch (Exception e)
         {
         // Debug output omitted
         }
      }

      // Handles background task cancellation.
      private void OnCanceled(IBackgroundTaskInstance sender,
         BackgroundTaskCancellationReason reason)
      {
         // Debug output omitted
      }
   }
}
```

You can see that the Run method receives an argument through which it can register a handler for cancelling the task. The code above doesn't do anything meaningful with this because the task itself executes only a small amount of code. The C# tasks in the Background tasks sample, on the other hand, simulate longer-running operations, in which case it uses the handler to set a flag that will stop those operations.

Access to Additional APIs

Between the DOM API, WinJS, third-party libraries, and JavaScript intrinsics, JavaScript developers have no shortage of APIs to utilize in their apps. At the same time, there is a whole host of .NET and Win32/COM APIs that are available to C#, VB, and C++ apps that are not directly available to JavaScript, including the DirectX, ESE "Jet" database, and Media Foundation APIs.

With the exception of APIs that affect UI or drawing surfaces (namely Direct2D and Direct3D), WinRT components can make such functions—or, more likely, higher-level operations built with them—available to apps written in JavaScript.

The Building your own Windows Runtime components to deliver great apps post on the Windows Developer Blog gives some examples of this. It shows how to use the `System.IO.Compression` API in .NET to work with ZIP files and the XAudio APIs (part of DirectX) to bypass the HTML `audio` element and perform native audio playback. In the latter case, you might remember from Chapter 13, in "Playing Sequential Audio," that no matter how hard we tried to smooth the transition between tracks with the `audio` element, there is always some discernible gap. This is because of the time it takes for the element to translate all of its operations into the native XAudio APIs. By going directly to those same APIs, you can circumvent the problem entirely. (That said, Microsoft knows about the behavior of the `audio` element and will likely improve its performance in the future.)

The DirectWrite font enumeration sample similarly uses the DirectWrite Win32 APIs through a C++ component to enumerate fonts, a capability not otherwise provided in WinRT.

This approach can also be used to communicate with external hardware that's not represented in the WinRT APIs (except perhaps through a protocol API) but is represented in Win32/COM. We'll take a closer look at the XInput and JavaScript controller sketch sample soon.

Another very simple example would be creating a component to answer an oft-heard question: "How do I create a GUID in JavaScript?" Although you can implement a routine to construct a GUID string from random numbers, it's not a proper GUID in that there is no guarantee of uniqueness (GUID stands for Globally Unique Identifier, after all). To do the job right, you'd want to use the Win32 API `CoGreatGuid`, for which you can create a very simple C++ wrapper.

> **Overkill?** Some developers have commented that going to all the trouble to create a WinRT component just to call one method like `CoCreateGuid` sounds like a heavyweight solution. However, considering the simplicity of a basic WinRT component as we've seen in this chapter, all you're really doing with a component is setting up a multilanguage structure through which you can use the full capabilities of each language. The overhead is really quite small: a Release build of the C++ component in "Quickstart #2" produces a 39K DLL and a 3K .winmd file, for example.

Using a WinRT component in this way applies equally to COM DLLs that contain third-party APIs like code libraries. You can use these in a Windows Store app provided they meet three requirements:

- The DLL is packaged with the app.
- The DLL uses only those Win32/COM APIs that are allowed for Windows Store apps. Otherwise the app will not pass certification.
- The DLL must implement what is called *Regfree COM*, meaning that it doesn't require any registry entries for its operation. (Windows Store apps do not have access to the registry and thus cannot register a COM library.) The best reference I've found for this is the article Simplify App Deployment with ClickOnce and Registration-Free COM in MSDN Magazine.

If all of these requirements are met, the app can then use the `CoCreateInstanceFromApp` function from a component to instantiate objects from that DLL.

The XInput API and Game Controllers

In Chapter 17 and its XboxController example, I demonstrated how to work with this particular device through the `Windows.Devices.HumanInterfaceDevice.HidDevice` protocol API. That's certainly one way to go about it, but has a few drawbacks. For one, it might not work with similar controllers because we were interpreting the input reports according to a very specific byte (and bit!) pattern. Furthermore, some of the controls like the triggers are somewhat odd in how they report their state and could use a little better interpolation.

The XInput API, part of DirectX, is a Win32 API that works with a range of game controllers and is on the list of allowable Win32/COM APIs. The most commonly used function here is `XInputGetState`, which returns an `XINPUT_STATE` structure that describes the position of the various thumb controllers, how far throttle or other triggers are depressed (with interpolation to remove the oddities), and the on/off states of all the buttons. It's basically meant to be polled with every animation frame in something like a game; the API doesn't itself raise events when the controller state changes.

The XInput and JavaScript controller sketch sample in the Windows SDK demonstrates exactly this. Because the XInput API is not accessible directly through JavaScript, it's necessary to create a WinRT component for this purpose. First, here's the header file from the GameController project (Contoller.h):

```
namespace GameController
{
    public value struct State
    {
    // [Omitted--just contains the same values as the Win32 XINPUT_STATE structure
    };

    public ref class Controller sealed
    {
        ~Controller();

        uint32              m_index;
        bool                m_isControllerConnected;  // Do we have a controller connected
        XINPUT_CAPABILITIES  m_xinputCaps;           // Capabilites of the controller
        XINPUT_STATE         m_xinputState;      // The current state of the controller
        uint64              m_lastEnumTime;     // Last time a new controller connection
```

```
                                          // was checked

    public:
        Controller(uint32 index);

        void SetState(uint16 leftSpeed, uint16 rightSpeed);
        State GetState();
    };
}
```

The implementation of `GetState` in Controller.cpp then just calls `XInputGetState` and copies its properties to an instance of the component's public `State` structure:

```
State Controller::GetState()
{
    // defaults to return controllerState that indicates controller is not connected
    State controllerState;
    controllerState.connected = false;

    // An app should avoid calling XInput functions every frame if there are
    // no known devices connected as initial device enumeration can slow down
    // app performance.
    uint64 currentTime = ::GetTickCount64();
    if (!m_isControllerConnected && currentTime - m_lastEnumTime < EnumerateTimeout)
    {
        return controllerState;
    }

    m_lastEnumTime = currentTime;

    auto stateResult = XInputGetState(m_index, &m_xinputState);

    if (stateResult == ERROR_SUCCESS)
    {
        m_isControllerConnected = true;
        controllerState.connected = true;
        controllerState.controllerId = m_index;
        controllerState.packetNumber = m_xinputState.dwPacketNumber;
        controllerState.LeftTrigger = m_xinputState.Gamepad.bLeftTrigger;
        controllerState.RightTrigger = m_xinputState.Gamepad.bRightTrigger;

    // And so on [copying all the other properties omitted.]
    }
    else
    {
        m_isControllerConnected = false;
    }

    return controllerState;
}
```

The constructor for a Controller object is also very simple:

```
Controller::Controller(uint32 index)
{
    m_index = index;
    m_lastEnumTime = ::GetTickCount64() - EnumerateTimeout;
}
```

In a JavaScript app—once the reference to the component has been added—the `GameController` namespace contains the component's public API, and we can utilize it like we would other scenario APIs. In the case of the sample, it first instantiates a `Controller` object (with index of zero) and then kicks off animation frames (program.js):

```
app.onactivated = function (eventObj) {
    if (eventObj.detail.kind ===
        Windows.ApplicationModel.Activation.ActivationKind.launch) {
        // [Other setup omitted]

        // Instantiate the Controller object from the WinRT component
        controller = new GameController.Controller(0);

        // Start rendering loop
        requestAnimationFrame(renderLoop);
    };
};
```

The `renderLoop` function then just calls the component's `getState` method and applies the results to a canvas drawing before repeating the loop (also in program.js, though much code omitted):

```
function renderLoop() {
    var state = controller.getState();

    if (state.connected) {
        controllerPresent.style.visibility = "hidden";

        // Code added to the sample to extend its functionality
        if (state.leftTrigger) {
            context.clearRect(0, 0, sketchSurface.width, sketchSurface.height);
            requestAnimationFrame(renderLoop);
            return;
        }

        if (state.a) {
            context.strokeStyle = "green";
        } else if (state.b) {
            context.strokeStyle = "red";
        } else if (state.x) {
            context.strokeStyle = "blue";
        } else if (state.y) {
            context.strokeStyle = "orange";
        }

        // Process state and draw the canvas [code omitted]
    };

    // Repeat with the next frame
    requestAnimationFrame(renderLoop);
};
```

The output of this sample is shown in Figure 18-5, reflecting the features I added to the modified sample in the companion content (which is what I'm showing above) to make it more interesting to my young son: changing colors with the A/B/X/Y buttons and clearing the canvas with the left trigger. As you can see, my own artwork with this app isn't a whole lot different from his!

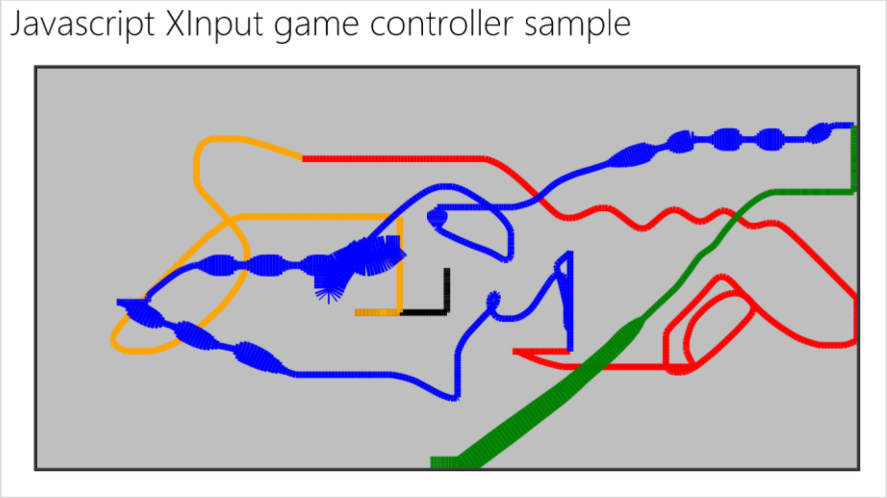


FIGURE 18-5 The XInput and JavaScript controller sketch sample with some modifications to change colors. The varying line width is controlled by the position of the right trigger.

Obfuscating Code and Protecting Intellectual Property

Back in Chapter 1 in "Playing in Your Own Room: The App Container," we saw how apps written in HTML, CSS, and JavaScript exist on a consumer's device as source files, which the app host loads and executes. By now you've probably realized that for as much as Windows tries to hide app packages from casual access, all that code is there on their device where a determined user can gain access to it. In other words, assume your code is just as visible in an app package as it is on the web through a browser's View Source command.

It's certainly possible—and common, in fact—to embed web content in a webview element to keep some of your code on a server. Still, there will be parts of an app that must exist and run on the client, so you are always running the risk of someone taking advantage of your generosity!

Ever since developers started playing with Windows Store apps written in HTML, CSS, and JavaScript, they've been asking about how to protect their code. In fact, developers using C# and Visual Basic ask similar questions because although those languages are compiled to IL (intermediate language), plenty of decompilers exist to produce source code from that IL just as other tools circumvent JavaScript minification. Neither JavaScript nor .NET languages are particularly good at hiding their details.

Code written in C++, being compiled down to machine code, is somewhat harder to reverse-engineer, although it's still not impossible for someone to undertake such a task (in which case you have to ask why they aren't just writing their own code to begin with!). Nevertheless, it's the best protection you can provide for code that lives on the client machine.

If the rest of the app is written in a language other than C++, especially JavaScript, know that it's a straightforward manner to also reverse-engineer the interface to a component. The issue here, then, is whether a malicious party could use the knowledge of a component's interface to employ that component in their own apps. The short answer is yes, because your code might show them exactly how. In such cases, a more flexible and nearly watertight solution would be for the component vendor to individually manage licenses to app developers. The component would have some kind of initialization call to enable the rest of its functionality. Within that call, it would compare details of the app package obtained through the `Windows.ApplicationModel.PackageId` class against a known and secure registry of its own, knowing that the uniqueness of app identity is enforced by the Windows Store. Here are some options for validation:

- **Check with an online service** This would require network connectivity, which might not be a problem in various scenarios. Just remember to encrypt the data you send over the network to avoid snooping with a tool like Fiddler!
- **Check against information encrypted and compiled into the component itself** That is, the component is compiled for each licensee uniquely. This is the most difficult to hack.
- **Check against an encrypted license file distributed with the component that is unique to the licensee** (contains the app name and publisher, for instance) Probably the easiest solution, because even if the license file is copied out of the licensed app's package, the info contained would not match another app's package info at run time. The encryption algorithm would be contained within the compiled component, so it would be difficult to reverse-engineer in order to hack the license file—not impossible, but difficult. Another app could use that component only if it used the same package information, which couldn't be uploaded to the Store but could still possibly be side-loaded by developers or an unscrupulous enterprise.

In the end, though, realize that Windows itself cannot guarantee the security of app code on a client device. Further protections must be implemented by the app itself, or you have to keep the code on a server and use it from there, either via HTTP requests or within a webview.

Concurrency

We've already seen that web workers and WinRT components with asynchronous methods can work hand in hand to delegate tasks to different threads. If you really want to exercise such an option, you can architect your entire app around concurrent execution by using these mechanisms, spinning up multiple async operations at once, possibly across multiple web workers. A WinRT component can also use multiple `Task.Run` or `create_async/task` calls, not just the single ones we've seen.

Deeper still, a WinRT component can utilize the APIs in `Windows.System.Threading` to get at the *thread pool*, along with those APIs in `Windows.System.Threading.Core` that work with semaphores and other threading events. The details of these are well beyond the scope of this book, but I wanted to mention them because many of the built-in WinRT APIs make use of these and your components can do the same. And although it doesn't demonstrate components, necessarily, the Thread pool sample provides a good place to start on this topic.

Tip To check if your C#, VB, or C++component code is running on the UI thread, use `Window.UI.Core.CoreWindow.GetForCurrentThread`, which returns `null` on non-UI threads.

Library Components

A library component is a piece of code that's designed to be used by any number of other apps written in the language of their choice. You might be looking to share such a library with other developers (perhaps as a commercial product), or you might just be looking to modularize your own apps.

You could, of course, create a .js library for JavaScript apps, a .NET assembly for C# and VB, and a DLL for C++, and then ship and maintain each one separately. A single WinRT component, on the other hand, works with all languages, including any new language projections that might be added to Windows in the future. This greatly reduces development and maintenance costs over time.

We've seen some examples of library components already in some of the Windows SDK samples. For example, the HttpClient sample provides two sample filters for HTTP requests. Because the authors of this sample needed to provide JavaScript, C#, VB, and C++ variants, it made sense to implement the filters that were common to all of them as WinRT components in C++.

Another example is the Notifications Extensions Library that we saw in various samples of Chapter 16: App tiles and badges sample, Lock screen apps sample, Scheduled notifications sample, Secondary tiles sample, and Toast notifications sample. This library was designed to be incorporated you're your own apps as-is, as we've done with the Here My Am! app for that chapter. (Note that because all the methods in the library's classes are small and fast, they're all designed to be synchronous.)

If you plan to create a commercial library, you can ship source code for every app to recompile, but it's better to follow the How to: Create a software development kit documentation so that you can provide only the compiled DLL and/or WinMD file to your customers. Customers will add these libraries to their projects, so they're packaged directly with the app.

Note When shipping a DLL or WinMD written in C++, be sure to provide separate builds x86, x64, and ARM targets. Providing standard compilations for each target is important to enable multiple apps to share the same library, a feature that the Store automatically manages on your behalf, as we'll see in Chapter 20, "Apps for Everyone, Part 2."

A primary use case for library components is to simplify interaction with a backend service, where the library can abstract complex HTTP interactions. Another is to create a simplified interface for a custom device. Although the protocol APIs that we explored in Chapter 17 make these devices accessible, you're still typically working on a very low level where details are concerned: creating packets, setting bits in output reports, and so forth. Such details are very sensitive to errors, as even a single bit out of place can cause a malfunction.

With a WinRT component you can effectively create your own scenario API for a device or a set of devices (abstracting their differences underneath the API). The code in the component presents a simplified or higher level interface to its clients, which it then translates into lower-level interactions with the protocol APIs. Such a component can also smooth out the eccentricities of a device so that app logic doesn't have to concern itself with such details.

Consider the Dream Cheeky Circus Cannon that we worked with in Chapter 17, a device that I can now say from experience has its eccentricities. These come from the nature of its internal mechanics: it has internal switches that are closed when movement in some direction reaches its limit. When these switches are closed, specific bits in one byte of the input report change from 0 to 1, and when the switches are opened again due to movement in the opposite direction, they change from 1 to 0. The device, however, only reports the state of the switches—it does not automatically stop itself when the limit is reached. Unless you issue a command to stop or move in the opposite direction, the cannon will sit there and grind its gears rather unpleasantly.

In the code we saw in Chapter 17, I used changes in the limit switches to automatically stop movement, thereby preventing the gear grinding (from js/default.js in that example):

```
function inputReportReceived(e) {
    //Read input report from e.report.data (omitted)

    //Stop if a limit is hit to prevent grinding.
    var upDown = report[CircusCannon.offset.upDown];
    var leftRight = report[CircusCannon.offset.leftRight];

    var atLimit = (upDown & (CircusCannon.status.topLimit | CircusCannon.status.bottomLimit))
        || (leftRight & (CircusCannon.status.leftLimit | CircusCannon.status.rightLimit));

    if (atLimit) {
        sendCommand(CircusCannon.commands.stop);
    }
}
```

So far so good. However, the device *constantly* sends input reports with current status, not when status changes, and it takes some movement in the opposite direction before a closed limit switch opens again. As a result, issuing a command to move the cannon in the direction opposite a closed limit switch doesn't open that switch automatically. With the code above, movement stops right away because the limit switch remains closed until we send enough commands to make the device move far enough. It's therefore necessary to debounce the limit switches, which means ignoring a closed limit switch until it first opens again. This way we can convert the constant input reports into discrete events for hitting a limit.

Along similar lines, issuing a command to fire missiles just turns on a motor for the firing mechanism, and that motor continues to spin until you issue a stop command. In the mechanism there's another switch that closed *about* the time that a missile is fired, and opened again once the firing mechanism has rotated past the trigger point. I say "about" because it's not really that precise: the switch can close before a missile has actually been released—in my tests it's more like the switch has to first close and *then* open again to be sure that firing took place. Furthermore, I found that it's good to let the mechanism keep moving about another 100ms past the point where the switch gets re-opened to be sure that it won't suddenly report being closed again. All of these characteristics make it difficult to tell the cannon to "fire one shot" because what that means, exactly, is a bit tricky to determine.

For UI purposes, I also wanted to be able to tell the cannon to move a little in any given direction. However, the device has no concept of step-wise movement: you tell it to start and stop, nothing more. (It does have slow movement commands, but they continue until a stop is issued.) To create such behavior, it's necessary to use timers with something like 500ms intervals.

All together, then, you can see that mapping simple user-level commands like "move in a direction," "move a little in a direction," "fire a missile," and "fire all three missiles" requires some intricate communication with the device and some additional device-level logic. Putting all this behind a custom scenario API in a WinRT component makes a lot of sense.

You can see such an implementation in the CircusCannon2 example in this chapter's companion content, where the CircusCannonControl project contains a C++ WinRT component that does everything I've described: it debounces the limit switches and raises singular events for any given limit, it works with the missile-fired bit in the input report to provide reliable "fire one" and "fire all" commands, and it implements step movements using internal timers. This all makes it very straightforward to write a decent app UI around the device (though the UI of the front-end test app in the example falls well short of "decent"!).

One part of this component that bears a little more discussion is properly raising events such that single-threaded JavaScript apps can consume them. Again, the Raising events in Windows Runtime Components topic in the documentation has all the details, so let me just show you what I used in the example. First of all, I used a simple `EventHandler` type for each event along with the C++ `event` keyword (see CircusCannonControl project > CannonControl.h):

```
event Windows::Foundation::EventHandler<bool>^ LeftLimitChanged;
event Windows::Foundation::EventHandler<bool>^ RightLimitChanged;
event Windows::Foundation::EventHandler<bool>^ TopLimitChanged;
event Windows::Foundation::EventHandler<bool>^ BottomLimitChanged;
event Windows::Foundation::EventHandler<bool>^ MissileFired;
```

Here, each event provides a Boolean argument to indicate the status of the limit switch (debounced, of course). In theory, raising an event as I do within the `inputreportreceived` handler (CannonControl.cpp) just means calling the event name with the sender object and handler args:

```
LeftLimitChanged(this, true);
```

The caveat is that unless the event is raised on the UI thread, you'll see exceptions in a JavaScript app and other complications in C#, VB, and C++ apps. The topic on raising events linked above discusses the options to handle this: (a) be sure to raise the event on the UI thread, (b) raise the event such that Windows can provide an automatic proxy and stub for the event, which loses type information in the arguments, and (c) provide your own proxy and stub.[129] For the purposes of my component, I chose to issue the events on the UI thread directly using the following code:

```
using namespace Windows::ApplicationModel::Core;
using namespace Windows::UI:Core;

CoreApplication::MainView->CoreWindow->Dispatcher->RunAsync(CoreDispatcherPriority::Normal,
    ref new DispatchedHandler([this]() {
        LeftLimitChanged(this, true);
    }))
```

Because I use the same structure for all five events, I implemented it as a C++ macro in CannonControl.h, and with this little bit of wrapper the events work great with JavaScript.

What We've Just Learned

- Windows Store apps need not be written in just a single language; with WinRT components, apps can effectively use the best language for any given problem. Although components can share app data with the host app, they cannot work with UI on behalf of an app written in HTML, CSS, and JavaScript.
- Reasons for using a mixed language approach include improving performance, gaining access to additional APIs (including third-party libraries) that aren't normally available to JavaScript, obfuscating code to protect intellectual property, creating modular library components that can be used by apps written in any other language, and effectively managing concurrency.
- For computationally intensive routines, a component written in C#/VB can realize on the order of a 15% improvement over JavaScript and a component written in C++ an improvement on the order of 25%. When testing performance, be sure to build a Release version of the app and run outside of the debugger, otherwise you'll see very different results for the different languages. Be mindful, though, that the overhead of marshaling data across language

[129] The custom proxy/stub approach is demonstrated in the Windows Runtime in-process component authoring with proxy\stub generation sample.

boundaries could be more costly than the savings in execution time, so you'll need to evaluate each case individually.

- Windows Store apps can employ web workers for creating asynchronous routines that run separately from the UI thread, and you can wrap that worker within a WinJS promise to treat it like other async method in WinRT.
- Async methods can also be implemented in WinRT components by using task, concurrency, and thread pool APIs. Compared to web workers, such async methods are more responsive because they are directly structured as methods.
- No matter what language a component is written in, the JavaScript projection layer translates some of its structures into forms that are natural to JavaScript, including the casing of names and conversion of data types.

Chapter 19

Apps for Everyone, Part 1: Accessibility and World-Readiness

The shared title of this chapter and the next, "Apps for Everyone"—especially the "Everyone" part—has several shades of meaning. First is the vitally central role that the Windows Store plays in the whole Windows experience. As first mentioned in Chapter 1, "The Life Story of a Windows Store App," the Store is *the* place where you distribute apps to customers (outside of the enterprise and sharing with other developers). Everyone, in other words, gets their apps from the Store.

In this same way, everyone who does business with apps does business with the Store. To define your app's relationship to the Store is in many ways to define your business itself, and that relationship affects all stages of the app lifecycle, from planning and development to distribution and servicing. As I recommended in Chapter 1, you might want to read the first part of Chapter 20, "Apps for Everyone, Part 2," even before starting your first coding experiments! Truly, the Windows Store is like a pair of bookends to the whole app development process: you think about the Store when planning the business of your app, and when all is said and done, you go to the Store's developer portal itself to make your app available to others.

Those "others" are the context for the additional meanings of "everyone," which is the focus of this present chapter. In general, when you set out to offer a product to customers, you want to broaden your reach to include as many potential customers as you can. There are, of course, cases where you might want to specifically limit your audience, but for most apps, being able to reach more customers is certainly an attractive opportunity. And if you don't, your competitors will!

One way to broaden your reach is to cover your bases where *accessibility* is concerned. Though accessibility has its origins in serving people with serious disabilities, research has shown that a majority of people—nearly 60%—use accessibility features in some capacity, even though there's no disability involved. For one, being able to accommodate limited input models—like keyboard-only or mouse/pointer-only—is inherent in dealing with touch-only devices. Resolution-scaling, similarly, serves the needs of the visually impaired alongside the desires of the financially *un*impaired (that is, those customers who splurge for a high-DPI device just to get sharper graphics). An app that works well with a screen reader for the visually impaired can also work rather well for the mobile customer whose otherwise sound eyes need to be focused elsewhere—like the road they're ostensibly driving on! And providing for high-contrast color schemes helps not only those whose eyes don't do well with subtle colorations but also those who might be working with a mobile device in bright sunlight.

It therefore behooves app developers to take accessibility concerns seriously, especially as the Store will specifically mark fully accessible apps. As we will see, this primarily involves adding the appropriate `aria-*` attributes to your HTML markup, adapting your layout to different screen sizes, and making sure to provide image variations for different contrast settings.

The second way to extend your reach is to make your app world-ready—that is, to utilize localized resources within the app so that it adapts itself to each user's language, regional conventions, date and time formats, currency formats, and so on. Fortunately, Windows enables you to structure your resources—images and strings, primarily—so that the right variations show up automatically, just like they do for resolution scales and contrasts. The Windows Runtime also contains a number of APIs to help an app be world-ready, and the app itself can take additional steps to localize the web services from which it's drawing data, how it works with live tiles and notifications, and so on. And additional tools like the Multilingual App Toolkit make it all the easier to translate your resources.

The reward for all this effort, of course, is that users who search in the Windows Store for apps in their regional language will see *your* app and not those that are available only in a single language. Those users are more likely to express their appreciation in your app's reviews and ratings.

One of the great things about the Windows Store is the access it gives you to global markets from wherever you happen to be working. In the past, learning to do business around the world has been a tedious and expensive process, sometimes requiring that you understand local tax laws, manage currency conversions, and so forth. No longer—this is really what the Windows Store is doing on your behalf. Once the Store becomes available in a market, it means that Microsoft has done the work to embed local policies into the Store itself. Put another way, whatever small fee you pay to upload apps to the Store has made it possible for you to do business in those markets with little or no effort! This is good. The Store also automatically adjusts your pricing tier—if you charge for the app or in-app purchases—because standards of living do vary widely around the world. This is also good.

In this penultimate chapter then, we'll take a tour of accessibility followed by another through the world of world-readiness. That will set us up well for Chapter 20, in which we'll come full circle to where we started in Chapter 1: uploading your app to the Store and what you can expect there. Now that you've brought your app this far, let's get it ready for everyone to enjoy!

Accessibility

As I mentioned, nearly 60% of users employ accessibility features in some capacity—sometimes because of a disability, sometimes due to personal preference, and sometimes just to make the device easier to use within certain environments. In many countries, accessibility is a legal requirement, so it will be necessary if you plan to make an app available in those regions. In the United States, for example, the 21st Century Communications and Accessibility Act contains regulations for communications apps.

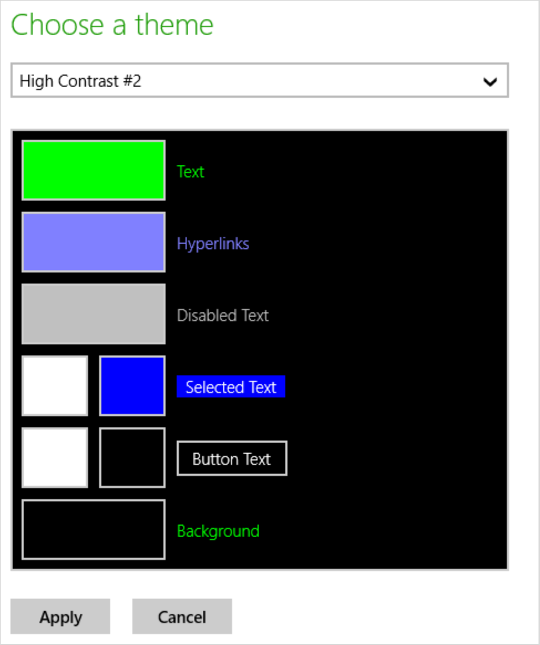
In short, supporting accessibility is something that every app should do and do well, and fortunately this isn't the onerous task you might think it to be. (For reference, see Accessibility for Windows Store Apps. Also see Guidelines and checklist for Accessibility, Practices to avoid for accessible apps, and Implementing accessibility for particular content types.)

Accessibility might feel like a lot of work because developers are relatively unfamiliar with what it means. To remedy this, take a few minutes to give yourself some direct experience. But before you do anything else:

Go to PC Settings OneDrive > Sync Settings and ***turn off*** *the options for Desktop Personalization.* Otherwise the effects of your tinkering will roam to other devices you might have. I learned this the hard way when I was playing around with contrast settings on my main laptop after which a game my son wanted to play on a tablet came up mostly black! Clearly, that app didn't handle high contrast well, but it also took me a while to figure out what was going on!

Now that we've taken care of that detail, try the following:

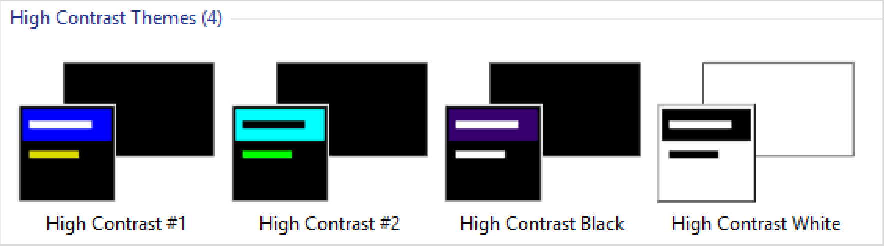
- Press Left Shift+Alt+Print Screen or go to PC Settings > Ease of Access > High Contrast > Choose a Theme (see below), and then select different options and tap Apply. How does the app respond? Are all of the critical elements visible? A mode like this is important for users who have difficulty distinguishing subtle colors and those using a device in bright sunlight.



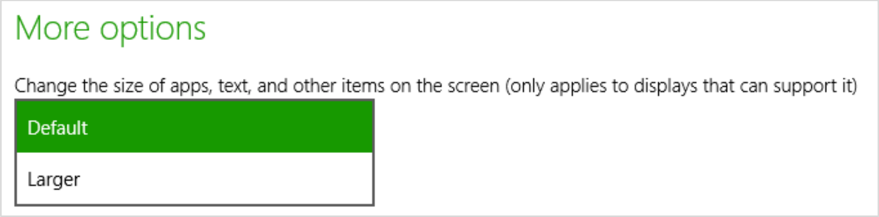
Choose a theme

High Contrast #2

- You can also select a high-contrast theme through Control Panel > Appearance And Personalization > Personalization, where you have the same choices as in PC Settings: three black background themes and one white-background theme; the latter is the same one that's activated through Left Shift + Alt + Print Screen:



- In PC Settings > PC & Devices > Display, the More Options drop-down (below) can change the effective scaling on the display. How does the app respond to the new screen dimensions? As discussed in Chapter 8, "Layout and Views," turning this on will activate the 140% resolution scaling, even if you're not using a high pixel density device.


More options

Change the size of apps, text, and other items on the screen (only applies to displays that can support it)

- Press Win+Ctrl+U to start (and stop) the built-in *screen reader* called Narrator. Win+Enter also starts it, and you can go to PC Settings > Ease of Access > Narrator. (Note that Narrator is a desktop application that starts minimized; you can close that application to stop Narrator or use PC Settings again.) Now turn off the monitor. Can you still use the app? What happens when you navigate around with the keyboard? Do you hear an audible indication of where the focus is? This is clearly important for users who are blind or visually impaired. (Tip: If you run Narrator for your app inside the Visual Studio simulator, it will apply to only that session and not everything else on your desktop, such as Visual Studio itself.)

- If you have a mouse, disconnect it and try *keyboard-only navigation* (you can open your eyes now). This is important for users with mobility issues and those who rely on speech recognition.

- With your mouse connected, go to PC Settings > Ease of Access > Keyboard and activate the On-Screen Keyboard to try *mouse/touch-only navigation*. This special on-screen keyboard is different from the one activated for touch with input fields (as we saw in Chapter 12, "Input and Sensors") because it always remains visible. In this same page of PC Settings you can also turn sticky keys on and off.

Through this experience I hope you've gained some understanding of what accessibility means. Simply said, four key scenarios exist for accessibility support: screen readers, keyboard-only or mouse-only input, high contrast, and resolution scaling.

Be very clear that supporting accessibility is something that starts at design time—it's a common mistake to implement an app and then think about accessibility only at the end of the process. But then you'll be going back and fixing many things in your code and UI layer that would have been easier to handle up front. Think about all the scenarios I just listed as part of your app design!

We've already covered two scenarios in this book. In Chapter 8 we saw how to work with different resolution scales, how to handle varying screen sizes (which can occur as a result of scaling), and how to provide raster graphics for different scales so that they always look their best. For a quick review, you might want to revisit the Scaling according to DPI sample.

Input considerations were also covered in Chapter 12, and I'll remind you again that the Store certification policy (section 6.13.4) requires that apps disclose when they lack keyboard and mouse support. Typically, the real work to be done is making sure that your app can be used with nothing but a keyboard; see Implementing keyboard accessibility for full details. Testing your app with Narrator turned on will also reveal whether you've paid attention to keyboard navigation, because no matter how well your elements are labeled for that purpose, those labels don't do any good if the user can never set the focus to them!

It's also worth mentioning that including closed captions in video will assist users who are hearing-impaired. Doing so, however, is a detail for the video data itself or can be implemented via text overlays on a `video` element. See the HTML5 and Accessibility (MSDN Magazine) for more on video accessibility.

One other important component to accessibility is respecting when users have disabled animations. We discussed this in Chapter 14, "Purposeful Animations," in "Systemwide Enabling and Disabling of Animations." If you're using the animations library, all this is taken care of automatically, but if you do any custom animations, be sure to check the `Windows.UI.ViewManagement.UISettings.animationsEnabled` property and respond accordingly:

```
var settings = new Windows.UI.ViewManagement.UISettings();
var enabled = settings.animationsEnabled;
```

Let's now look at how we support screen readers and contrast variations.

Sidebar: Accessibility Test Tools

The Windows SDK includes two tools to help you verify your implementation of accessibility. The first is Inspect, a UI automation tool that checks through the accessibility information you've made available to screen readers and lets you know what you've missed. The second is AccChecker, which runs a series of verifications on the rest of the app and validates that you'll work with a variety of third-party screen readers in addition to Narrator. (AccChecker is somewhat aggressive in its reporting: it flags warnings about a host of possible issues, a number of which you don't need to address especially if they occur within hosted web content. It also doesn't check app bars, nav bars, flyouts, and settings panes.)

You'll find these tools in the Windows SDK install folder, typically *c:\Program Files (x86)\Windows Kits\8.1\bin\x86*. You might also be interested in the Accessible Event Watcher and the UI Automation Verify tools. For usage details on all of these, see Automation Testing Tools in the documentation as well as Testing your app for accessibility. Of course, for the most complete kind of testing, find yourself a few users who regularly work with assistive technologies, set them up with a developer license (so that you can share your app package), and let them put your app through its paces!

Sidebar: Narrator and tabindex Attributes

When making your app navigable by keyboard, be careful to not overuse `tabindex` attributes on elements that don't need them, thinking that this will help Narrator for noninteractive elements. It doesn't. Narrator has its own keyboard commands (like CapsLock+arrows) and its own navigation modes that skilled users employ to read anything on a page, irrespective of `tabindex`. For this reason, setting `tabindex` properties on static elements *decreases* Narrator's usability; you should set the property only on interactive elements. In other words, understand that using an app through the keyboard and using it through Narrator are different processes, and think of `tabindex` only in the context of keyboard navigation.

Screen Readers and Aria Attributes

Screen readers like the built-in Narrator work best if the app provides some kind of information that properly describes elements in the UI. For Windows Store apps written in HTML, CSS, and JavaScript, this is achieved through `aria-*` attributes on your UI elements.

Tip If you need separate text to speech capabilities, remember that you have the `Windows.Media.SpeechSynthesis` API available to you, as discussed in Chapter 13, "Media" and demonstrated in the Speech synthesis sample.

ARIA stands for Accessible Rich Internet Applications, a standard that's spelled out in the WAI-ARIA specifications. WIA itself stands for the W3C Web Accessibility Initiative. Two other W3C documents of interest are the WAI-ARIA Primer and WAI-ARIA Authoring Practices.

What it really boils down to is that assistive technologies like Narrator are first able to automatically derive what they need from certain elements, like headers, paragraphs, the `title` attribute, `label` elements associated with focusable elements (using the label's `for` attribute), button text, input elements, the `caption` attribute of a table, and the `alt` attribute of `img` elements. The `role` attribute also comes into play here.

For everything else, like `div` elements (including custom controls) whose role cannot be inferred, the specs define a number of attributes, starting with `aria-`, to indicate the role that a particular element plays in the app. Full details can of course be found in the specifications linked above, along with the ARIA reference on the Windows Developer Center. Another good reference topic is also Exposing basic information about UI elements in the documentation, which shows examples of a number of the core `aria-*` attributes. That page makes a special note about the `canvas` element. Because a `canvas` is just a pixel bucket, it doesn't have content that is accessible to screen readers, even though it might appear as text. Make a special effort, then, to give a canvas appropriate attributes as you would with other custom elements (again see the HTML5 and Accessibility article for more on `canvas`).

All of the controls in WinJS are fully stocked with ARIA attributes and other bits that work with assistive technologies, so by using them you get lots of accessibility for free. (An exception is the SemanticZoom control that specifically does not use `aria-label` because it's a container for other controls that should have such labels.) That said, it's still necessary for you to properly adorn other elements, including HTML controls like `progress`. Here's a summary of the core `aria-*` attributes:

- `aria-label` Directly provides text for screen readers.
- `aria-labelledby` (Note the spelling with two L's.) Specifies the identifier of another element that contains the appropriate label for an element. The specifications state that `aria-labelledby` should be used instead of `aria-label` if that text is already on the screen.
- `aria-describedby` Similar to `aria-labelledby`, identifies an element (which doesn't have to be visible) that contains fuller description instead of just label text. Narrator reads that text when the user presses the Win+Alt+F key on the element with this attribute. This is a good option to use with a `canvas` that contains drawn text if it doesn't work to use `aria-label` directly. (The Here My Am! app as updated for this chapter uses `aria-label`.)
- `aria-valuemin`, `aria-valuemax`, and `aria-valuenow` For `div` elements whose `role` is set to *slider*, *progressbar*, or *spinbutton*, these indicate the values within the control. `aria-valuetext` can also provide text that corresponds to the value of `aria-valuenow`.
- `aria-selected`, `aria-checked`, `aria-disabled` and `aria-hidden` Indicate the state of an element.
- `aria-live` Needed for "live region" content that changes dynamically, such as master-detail views, chat, RSS feeds, fragment loading, and so forth. This is also appropriate for transitions and on `progress` controls.

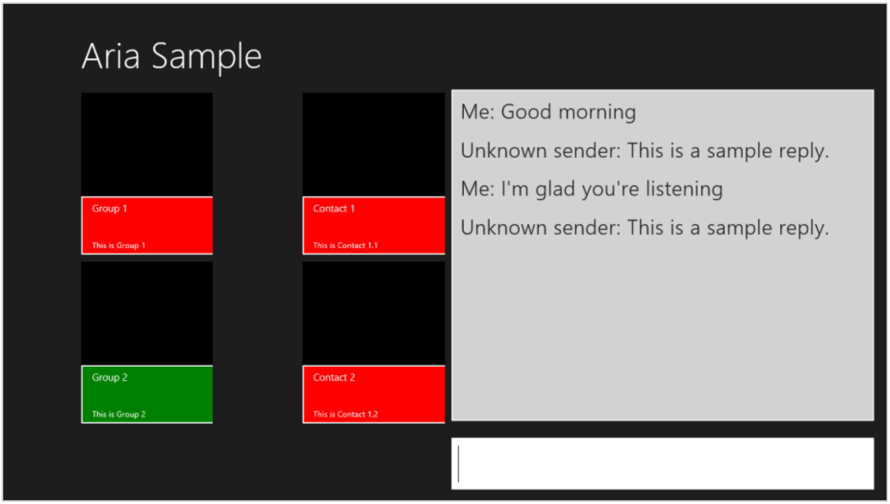
Back in Chapter 6, "Data Binding, Templates, and Collections," we saw that a special syntax was necessary to do data-binding on attributes of target elements where there are no associated JavaScript properties. The `aria-*` attributes are the primary example of this, because of their hyphenated names. For these we use `this[ ]` along with special WinJS initializers in `data-win-bind`:

```
<div data-win-bind="this['aria-label']: title WinJS.Binding.setAttribute"></div>
<div data-win-bind="this['aria-label']: title WinJS.Binding.setAttributeOneTime"></div>
```

It's probably more typical, though, that you'll provide localized ARIA labels in your app's resources, and for this there is a different declarative syntax that we'll see later on "World Readiness and Localization."

The ARIA Sample

To see the various `aria-*` attributes and Narrator in action, the best place to turn is a unique sample in the Windows SDK, the [ARIA sample](). One of its unique characteristics (besides having the shortest name of all the samples!) is that is doesn't at all *look* like an SDK sample, as you can see in Figure 19-1. This was done to intentionally represent the content of a typical app, without the normal SDK décor. In this case the sample emulates a simple chat app with a kind of master-detail view.



Me: Good morning

Unknown sender: This is a sample reply.

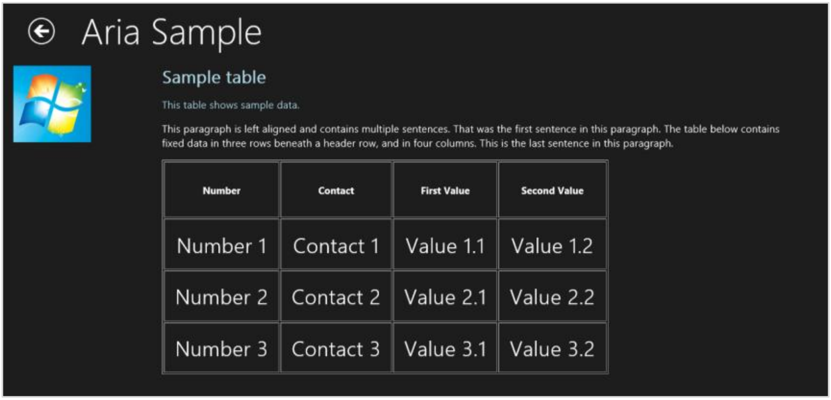
Me: I'm glad you're listening

Unknown sender: This is a sample reply.

FIGURE 19-1 The ARIA sample's main page.

When you run this sample, be sure to turn on Narrator to hear what it has to say. (Again, I highly recommend doing this in the Visual Studio simulator because then Narrator is running in that session only and not for your entire machine!) You'll find that it's accurately reflecting what's happening on the screen, especially as you Tab or Shift+Tab between controls, press Enter or the spacebar to select items, and enter chat text. Again, turn off your monitor, close your eyes, get a blindfold, or have your five-year-old come up behind you and cover your eyes to get the full experience.

Pressing Enter on an item in the left-hand list will update the contacts shown in the middle. Selecting a contact and pressing enter will then open another page containing a table—a contrived table, certainly, but one that shows the `aria-*` attributes that apply there. The page, shown in Figure 19-2, also provides an opportunity to experience page navigation through the keyboard and Narrator.



Aria Sample

Sample table

This table shows sample data.

This paragraph is left aligned and contains multiple sentences. That was the first sentence in this paragraph. The table below contains fixed data in three rows beneath a header row, and in four columns. This is the last sentence in this paragraph.

Number	Contact	First Value	Second Value
Number 1	Contact 1	Value 1.1	Value 1.2
Number 2	Contact 2	Value 2.1	Value 2.2
Number 3	Contact 3	Value 3.1	Value 3.2

FIGURE 19-2 The ARIA sample's secondary page (cropped a bit).

Apart from the small bits of code in pages/chat/chat.js to work the chat window, the really interesting parts of this sample are all contained in the markup, specifically pages/chat/chat.html (the main page) and pages/table/ table.html (the secondary page). In the first we can see `aria-label` on most of the controls (with the text you hear as you tab around) and much more extensive roster of attributes for the chat output `div` near the bottom:

```
<div class="chatpage fragment">
    <header aria-label="Header content" role="banner">
        <button class="win-backbutton" aria-label="Back" disabled></button>
        <h1 class="titlearea win-type-ellipsis">
            <span class="pagetitle">Aria Sample</span>
        </h1>
    </header>
    <section aria-label="Main content" role="main">
        <div class="chat">
            <div class="groupslist" aria-label="List of groups"
                data-win-control="WinJS.UI.ListView"
                data-win-options="{ selectionMode: 'none'}"></div>
            <div class="contactslist" aria-label="List of contacts"
                data-win-control="WinJS.UI.ListView"
                data-win-options="{ selectionMode: 'none' }"></div>
            <div class="chatTextContainer">
                <div class="chatTextEchoContainer" aria-label="Chat text area"
                    aria-live="assertive" aria-multiline="true" aria-readonly="true"
                    aria-relevant="additions" role="log"
                    tabindex="0"></div>
                <input class="chatTextInput" accesskey="i"
                    aria-label="Chat input area" type="text"
                    value="Type here..."/>
            </div>
        </div>
    </section>
</div>
```

That div, with the *chatTextEchoContainer* class, is updated at run time to contain child div elements for each text entry. This makes it a "live region" and thus has the aria-live attribute, whose values are described as a "politeness level" in the W3C spec. The value of assertive says "communicate the change right away," which is appropriate for chat but should be used carefully. The other value, polite (the default for role="log" elements), indicates a lower priority such that Narrator won't interrupt the current task. The aria-relevant="additions" attribute is related to this, indicating what kind of changes are relevant to the live area. Its values are additions, removals, text, and all. With additions, if we happened to add an image to the chat window with an alt attribute, that would be communicated; if we set this to text, only text elements would be read.

The aria-multiline attribute indicates that the chat window is a multiline textbox such that the Enter key is taken as text input rather than as a button press that would submit a form (as with the single-line textbox). The aria-readonly attribute then indicates that this control cannot be edited, to distinguish it from those marked with aria-disabled.

If you play with the sample, you'll notice that when you tab to the chat window, Narrator reads the entire contents. When you enter a line of text in the single line control, on the other hand, Narrator reads only the new element that's been added. This is due to a default value of false for the aria-atomic attribute (not present in the markup). When used on an aria-live element, this tells the screen reader to read only the changed node in that element. If you set aria-atomic to true, a change to any child element is considered a change to the whole element such that all the contents will be read. This can apply on multiple levels, mind you, so that if you add a child element that is atomic and add grandchild elements within it, only that atomic child element would be read if the parent element is not atomic.

As for the markup in pages/table/table.html, this gives us an example of aria-describedby. Here's the relevant section, omitting the table contents:

```
<div class="detail">
   <h2 id="title" role="heading" aria-level="2">Sample table</h2>
   <p id="subtitle" role="note">This table shows sample data.</p>
   <p class="generaltext">...</p>
   <table class="tabledetail" aria-describedby="subtitle"
      aria-labelledby="title" border="1">
      <!-- Contents omitted -->
   </table>
</div>
```

When you set the focus to the table in the running sample (you have to use the mouse for this unless you add a tabindex to the table), you'll initially hear "Sample table" according to the aria-labelledby attribute. Then press Win+Alt+F, and you'll hear "Item described by…" followed by the aria-describedby text. (And yes, go ahead and change it so that Narrator says some silly things. You know you want to!)

Note, finally, that it's essential that the *title* and *subtitle* elements also have some aria-related attributes, such as role. Otherwise aria-labelledby and aria-describedby won't work.

Handling Contrast Variations

Working with high-contrast modes is primarily one of accommodating changes to the Windows color theme and making sure that you apply graphics that meet high-contrast requirements. Technically speaking, high contrast is defined by the W3C as a minimum luminosity ratio of 4.5 to 1. A full explanation including how to measure this ratio can be found on http://www.w3.org/TR/WCAG20-TECHS/G18.html. A Contrast Analyzer (from the Paciello Group) is also available to check your images (some of mine in Here My Am! originally failed the test). Do note, however, that creating high-contrast graphics isn't required for noninformational content such as logos and decorative graphics. At the same time, full-color graphics might look out of place in a high-contrast mode, so be sure to evaluate your entire end-to-end user experience under such conditions.

An app handles high contrast through four means. The first is to use built-in controls (both HTML and WinJS) and let the system do the work! To see what happens, run a few of the controls samples, such as the HTML essential controls sample and the HTML Rating control sample, and switch between the different high-contrast themes in PC Settings > Ease of Access > High Contrast.

Of course, an app will almost always have some layout of its own, such as `div` elements with custom color schemes defined in CSS. You'll want to make sure you have appropriate style rules for high-contrast settings, for which we have the `-ms-high-contrast` media feature for media queries. This feature can have the values of `active` (all high-contrast themes), `black-on-white` (the white background theme), `white-on-black` (a black background theme), and `none`. Clearly, `none` is implied when you don't use `-ms-high-contrast` to group any rules; `active` is also implied if you use `-ms-high-contrast` without a value. We'll take a closer look at all this coming up.

As with other queries, you can use media query listeners and `matchMedia` to pick up contrast themes in code. This is useful for updating `canvas` elements, as we'll see shortly. There is also the `-ms-high-contrast-adjust` CSS style, which indicates whether to allow the element's normal CSS properties to be overridden for high contrast. The default value, `auto`, allows this; the value of `none` prevents this behavior. Again, we'll see more shortly.

Next, WinRT surfaces the current contrast settings through the `Windows.UI.ViewManagement.AccessibilitySettings` class. This has two properties: `highContrast`, a Boolean indicating if any high-contrast theme is active, and `highContrastScheme`, a string with the name of the specific theme exactly as it appears in PC Settings, which is also localized for the user's current language. In English, the black on white theme will be "High Contrast White"; for the other three themes the strings will be "High Contrast #1", "High Contrast #2", and "High Contrast Black" (going from left to right). You can see these results through scenario 2 of the UI contrast and settings sample, where the code is very simple:

```
var accessibilitySettings = new Windows.UI.ViewManagement.AccessibilitySettings();
id("highContrast").innerHTML = accessibilitySettings.highContrast;
id("highContrastScheme").innerHTML = accessibilitySettings.highContrast ?
    accessibilitySettings.highContrastScheme : "undefined";
```

WinRT also provides detailed color information through the `Windows.UI.ViewManagement.UISettings.uIElementColor` method. (Note the odd casing on `uIElementColor`, an artifact of WinRT names projecting into JavaScript.) This returns a `Windows.UI.Color` object for an element identified with a `UIElementType`. Scenario 1 of the UI contrast and settings sample shows all these possibilities with a piece of instructive but otherwise uninspiring code that I won't duplicate here!

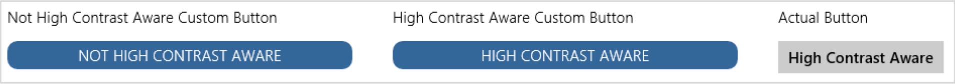
The `AccessibilitySettings` object also supports one event, `highcontrastchanged`, which lets you know when high contrast is turned on or off; its `eventArgs.target` is the updated `AccessbilitySettings` object. You can use this event to trigger any programmatic updates you need to make in your UI, such as redrawing a `canvas` with high-contrast colors if you're not using a media query listener for that purpose. I use this in Here My Am! to redraw placeholder images appropriately.

Finally, with both raster and vector images, there are file-naming conventions that you use in conjunction with the *.scale-100*, *.scale-140*, and *.scale-180* suffixes for pixel density. For contrast, the appropriate suffixes are *.contrast-standard*, *.contrast-high*, *.contrast-black* (black-on-white), and *.contrast-white* (white on black), where the *black* and *white* names are mutually exclusive with *high*. We'll see all this in action later in "High-Contrast Resources," and we'll see how to combine both the scaling and contrast suffixes in "Scale + Contrast = Resource Qualifiers."

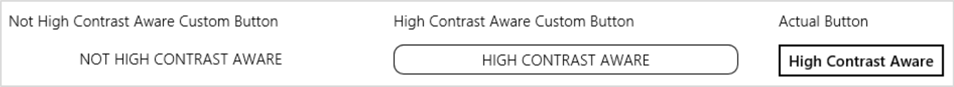
CSS Styling for High Contrast

The CSS styling for high contrast mode sample provides a valuable look at dealing with high-contrast modes where media queries and image files are concerned. As you might expect, most of these features are demonstrated declaratively in CSS and through the app's resources; only one scenario actually has any JavaScript code at all!

Scenario 1 shows the difference between elements that are and are not aware of contrast. With a normal color scheme in effect, its three buttons appear as follows, where the first two are `div` elements and the third a true `button`:



When high contrast is turned on (Left Shift + Alt + Print Screen is handy to toggle the setting for this sample), they appear as shown here for a black on white theme:

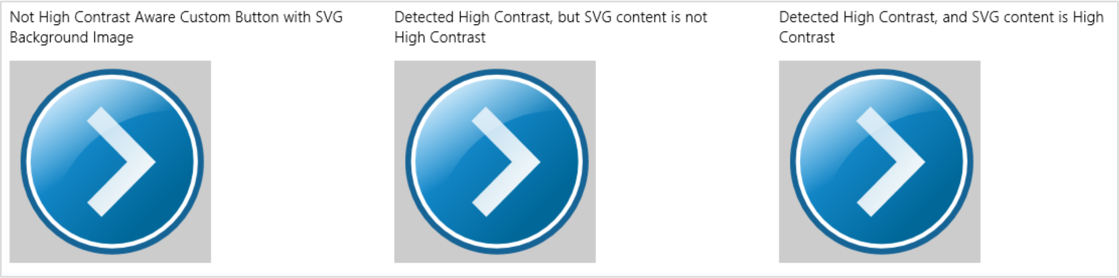


The first control, lacking contrast awareness, is still using white for its border, which of course disappears against a white background. The second button, on the other hand, has styles that use system-defined colors associated with a high-contrast media query, so the button works well with any theme (css/scenario1.css):

```
@media (-ms-high-contrast) {
    .s1-hc {
        background-color: ButtonFace;
        color: ButtonText;
        border: 1px solid ButtonText;
    }
    /* ... */
}
```

> **Tip** If you just stick with system colors entirely, both in CSS and in SVGs, you won't need to use media queries or different SVG files at all, because those colors will be adjusted for high-contrast modes automatically. See User-defined system colors for a reference. You can also use the `currentColor` value in SVGs for `fill`, `stroke`, `stop-color`, `flood-color`, and `lighting-color` properties to reflect contrast settings.

Scenario 2 shows similar effects with button elements that use SVGs for their background images. With normal settings, those buttons appear as follows:

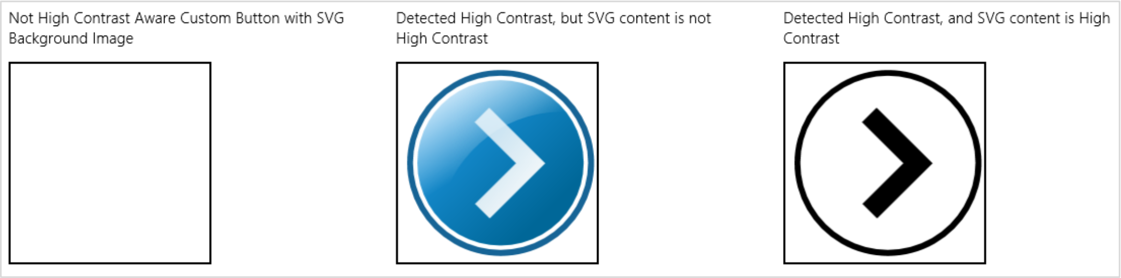


Not High Contrast Aware Custom Button with SVG Background Image

Detected High Contrast, but SVG content is not High Contrast

Detected High Contrast, and SVG content is High Contrast

With high contrast turned on, they appear like this:



Not High Contrast Aware Custom Button with SVG Background Image

Detected High Contrast, but SVG content is not High Contrast

Detected High Contrast, and SVG content is High Contrast

All that's happening here is that a media query sets a high-contrast background image for the button when necessary:

```
.s2-button-hc-bg-svg {
    background-image: url(../button-not-aware.svg);
    background-size: 100% 100%;
    width: 200px;
    height: 200px;
}
```

```
@media (-ms-high-contrast) {
    .s2-button-hc-bg-svg {
        background-image: url(../button.contrast-high.svg);
        background-repeat: no-repeat;
        background-size: cover;
    }
}
```

If you look in button-not-aware.svg, you'll see that its gradient colors have many different values; in button.contrast-high.svg, on the other hand, those colors are generally set to *black* or *ButtonFace*, the latter reflecting the system color setting as is appropriate. (It would probably be better, in fact, to replace *black* with *ButtonText*, a system color that will automatically adjust to contrast settings.)

What's going on with the first and second buttons? If you look in the CSS (css/scenario2.css), the only difference is that the style class for the first button, *.s2-button*, lacks a rule within the high-contrast media query, whereas the second, *.s2-button-hc*, has a rule there that just specifies the exact same background image. What's the deal? What's happening is that because the first button lacks any applicable style rule within the media query, its styles are automatically overridden with high-contrast values. Turning on high contrast overrides most color styles as well as `background-image`, in the latter case simply removing those images. This is why the first button shows up blank. The second button has a rule to define a `background-image` within the media query, so that image appears.

This brings us to the purpose of the `-ms-high-contrast-adjust` style. By default this is set to `auto`, allowing CSS properties to be overridden. Setting this to `none` prevents those styles from being overridden or adjusted. Thus, if you add `-ms-high-contrast-adjust: none;` to the *.s2-button* rule in css/scenario2.css, you'll see that the first and second buttons behave exactly the same. You can see this change in the copy of the sample included with this chapter's companion content.

Moving now to scenario 3, it normally draws the Internet Explorer 'e' on a `canvas` in color (below left), whereas in high-contrast mode it draws the logo in black and white (below right):

In this case, high contrast is picked up in JavaScript (js/scenario3.js) using a media query listener; no CSS is involved (this code is simplified for clarity; the sample also detects high contrast on startup):

```
var fillStyleOuterColor = "rgb(9, 126, 196)";
var fillStyleInnerColor = "rgb(255, 255, 255)";

var mql = matchMedia("(-ms-high-contrast)");
mql.addListener(updateColorValues);

function updateColorValues(listener) {
    if (listener.matches) {
        fillStyleOuterColor = "ButtonText";
        fillStyleInnerColor = "ButtonFace";
        draw();
    }
    else {
        fillStyleOuterColor = "rgb(9, 126, 196)";
        fillStyleInnerColor = "rgb(255, 255, 255)";
        draw();
    }
```

Note that the `AccessibilitySettings.onhighcontrastchanged` event could be used here instead of the media query listener.

> **Canvas a better choice?** In my original versions of the Here My Am! app, I used images to provide messages in `img` elements when a photograph or the map isn't available. When considering the needs for contrast and localized variations of those images, it was ultimately easier to generate the images on the fly by drawing text onto a `canvas`, as we've been doing since Chapter 2, "Quickstart." This eliminates the need for many different image files and makes the app package smaller, while still fully addressing both accessibility and localization needs.

High-Contrast Resources

In the previous section with the CSS styling for high contrast mode sample we saw a bit of the filename conventions that the Windows resource loader uses for high contrast: button.contrast-high.svg, for example. Scenario 4 of that sample shows how this lookup can happen automatically. In the project there is a file named button.svg alongside one named button.contrast-high.svg, with an `img` element declared in html/scenario4.html as follows:

```
<img src="../button.svg" />
```

If the system is running with normal contrast, the resource loader resolves the URI here to button.svg. (The `../` is because the scenario page is one level down in the HTML folder—generally it's better to give the full in-package path like `/images/...`.) When high contrast is in effect, the resource loader instead looks for that same filename but with one of three qualifiers in the filename. If you have distinct resources for black-on-white and white-on-black themes, name those files with *.contrast-white* and *.contrast-black,* respectively. If, on the other hand, you have just a single high-contrast variant, use *.contrast-high*. This is because Windows will ignore any *high* variants if you provide *black* and *white* files with the same root name.

> **Note** If you're using custom app bar icons, as discussed in "Custom Icons" in Chapter 9, "Commanding UI," remember to include high-contrast variants of your source images by using this naming scheme.

If you like having more parallel filenames, you can also name the normal contrast file with *.contrast-standard*, as in button.contrast-standard.svg. If you do this in the sample project, leaving the HTML as is, you'll see no difference in the output. At the same time, because of behavior nuances with contrast handling, it's recommended to use *.contrast-standard* only if you also supply *.contrast-white* and *.contrast-black* variants.

As noted before, contrast variants are applied automatically for black-on-white (white background) and white-on-black (black background) themes, respectively. To see this, rename button.contrast-high.svg in the SDK sample to button.contrast-white.svg, and then make a copy named button.contrast-black.svg. In that copy, modify the gradient colors in the CDATA block by exchanging *black* with *ButtonFace*. When you then switch on a black background theme, you'll see a button that's white on black, as it should be. (All these changes can be found in the copy of the sample included with this chapter's companion content.)

The one caveat with the `img` element in scenario 4 is that it won't be updated when contrast is changed while the app is running, as happens with media queries in scenarios 1–3. That is, the app host will not re-render the `img` element in response to a contrast switch. To change this behavior, we have to trick the app host into thinking that the source URI has changed by appending some dummy URI parameters. We can do this inside `AccessibilitySettings.onhighcontrastchanged` with `eventArgs.target.highContrastScheme` providing a decent variable for the URI (see js/scenario4.js in the modified sample; you can also set `src` to `""` and then reassign):

```
var page = WinJS.UI.Pages.define("/html/scenario4.html", {
    ready: function (element, options) {
        var accSet = new Windows.UI.ViewManagement.AccessibilitySettings();

        accSet.addEventListener("highcontrastchanged", function (e) {
            var image = document.getElementById("buttonImage");

            //Use the scheme name (sans whitespace) as the dummy URI parameter
            var params = e.target.highContrast ?
                "?" + e.target.highContrastScheme.replace(/\s*/g, "") : "";
            image.src = "../button.svg" + params;
        });
    }
});
```

One significant advantage to `highcontrastchanged` over media query listeners is that the latter will be fired very soon after the change happens, at which point the resource loader might not have picked up the change by the time you set the `img.src` attribute. This results in the wrong image being displayed. `highcontrastchanged` is fired much later, so the code above generally works. That said, my experiments along these lines (with the sample running and making changes in the desktop contro l panel) show that it's still not 100% reliable: changing contrasts is an expensive operation that triggers many events throughout the system, and there's no guarantee when the resource loader will get reset. For this reason you can consider just bypassing the whole matter and explicitly setting the `src` attribute to a known file with a specific name. The modified sample contains code like this in comments, or you can just use media queries!

Scale + Contrast = Resource Qualifiers

Because the graphics we just worked with are SVGs, there is no need to supply separate files for different pixel densities. But what if we have raster graphics? How do we combine scaling and contrast? This will also come up when we look at localization later, because that will introduce language variants as well. This brings us to the matter of resource *qualifiers*, a topic that's discussed in its fullest extent on How to name resources using qualifiers. Qualifiers include scale and contrast as we've seen, along with language, layout direction, home region, and a few other obscure variants.

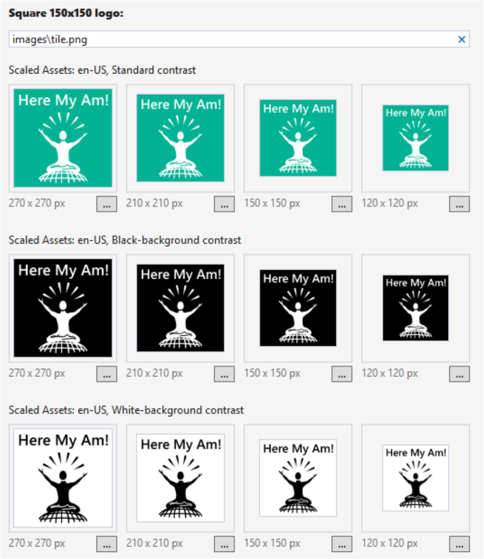
To combine qualifiers within a single filename, append them together with underscores. The general form is *filename.qualifiername-value_qualifiername-value.ext*. So, a graphic named *logo.png* can have variants like *logo.contrast-standard_scale-180.png* and *logo.scale-100_contrast-white.png* (the order of qualifiers doesn't matter). Clearly, with the full set of three or four scales (accounting for the few scale-80 cases) and two or three possible contrasts, you might have as many as 12 distinct graphics files for that one resource. Indeed, for all the core images in your manifest alone—the ones we talked about in Chapter 3, "App Anatomy and Performance Fundamentals," you can have up to 52 total images (for standard/high contrast) or 78 (for standard/black/white). For a few examples of this, load the Application resources and localization sample into Visual Studio and look in the *images* folder. (Although the sample shows only contrast+100% scale examples, be sure to provide at 140% and 180% scales as well in your own app; Here My Am! for this chapter does so with its splash screen, tile, and other logo graphics.)

As we get into the topic of world readiness, we'll find that localized image resources will require a set of scale and contrast variants *for each language*. As you can guess, the file-naming conventions here could get really messy as the file count increases! Fortunately, the resource loader also allows qualifiers in folder names, so localized resources are typically placed within language-specific folders. We'll see more of this later in "Part 2: Structuring Resources for the Default Language." We'll also avoid this complexity entirely in Here My Am! by using a `canvas` instead of discrete images for those graphics that contain text messages (the logos aren't localized).

High-Contrast Tile and Toast Images

Like any other images in your app, tile images in your manifest, images sent to the tile through updates, and images used in toast notifications all respect contrast settings. (Badges are not an issue because they are already monochromatic and adapt automatically.) In the manifest, naming images with resource qualifiers works for scale, contrast, and language.

> **Tip** For your manifest images, Visual Studio's manifest editor shows all the contrast variants alongside scale variants, which gives you a way to quickly see if you have all the filename qualifiers spelled correctly, because misspelled files will not appear in the editor:



Square 150x150 logo:

images\tile.png

Scaled Assets: en-US, Standard contrast

270 x 270 px ... 210 x 210 px ... 150 x 150 px ... 120 x 120 px ...

Scaled Assets: en-US, Black-background contrast

270 x 270 px ... 210 x 210 px ... 150 x 150 px ... 120 x 120 px ...

Scaled Assets: en-US, White-background contrast

270 x 270 px ... 210 x 210 px ... 150 x 150 px ... 120 x 120 px ...

XML payloads for tiles and toasts can refer to local images using `ms-appx:///` URIs, and the resource loader will look for the appropriately qualified file. This does not apply to `ms-appdata:///` URIs, however, so if you're working with downloaded or dynamically generated images, you'll need to identify a specific file yourself.

For XML payloads that refer to remote images, setting the `addImageQuery` option in the payload to `true` (as discussed in "Using Local and Web Images" in Chapter 16, "Alive with Activity") will append query strings to the remote URIs that indicate scale, contrast, and language:

```
?ms-scale=<scale>&ms-contrast=<contrast>&ms-lang=<language>
```

These details are described on Globalization and accessibility for tile and toast notifications, along with how to localize strings in the XML payload. We'll see these details for ourselves later on.

World Readiness and Localization

Over the years I've heard a number of words used to describe the process of making an app ready for different regional markets, and I imagine you have too: localization, localizability, internationalization, globalization, and world readiness. To be honest, the differences between these terms have confused me for some time, but I finally found a good explanation in an older book for desktop applications called *Developing International Software*, by Dr. International (Microsoft Press, 2003). The same ideas are also expressed on Understanding Internationalization. Let me begin this section then by offering a simple summary of that view.

The goal with Windows Store apps is to make them available in many markets around the world, as provided for so conveniently by the Store itself. To do this, an app must be written to adapt itself to just about any language and culture that it might encounter. In some situations you may need to produce specific builds of the app, but hopefully you can have one app with localized resources that works for most markets. Truly, the days of monolingual apps are over. Indeed, there are even aspects of localization, like branding, that apply to markets that otherwise share a common language, such as when you use a different company logo in England, Australia, New Zealand, Canada, and the USA.

To reach this goal you must first make your app *world-ready*. World readiness means that even though the app initially supports only one language and culture (most likely your own), it doesn't actually make any assumptions about those specifics anywhere within its HTML, CSS, and JavaScript (including tile and toast payloads and any WinRT components). That is, the core app is language-, culture-, and market-neutral, taking all these factors into account:

- Each and every text string that might be shown in the app's user interface, including element attributes like `aria-label` and `img.alt`, has been separated out into a resource file such that different resources can be loaded for different languages. Strings should include their punctuation because it is also subject to localization. (Using UTF-8 Unicode text is pretty much a given nowadays, so displaying text in many languages isn't an issue, but be sure to keep this in mind if you're migrating older software or using web services that might work otherwise.)
- Leave adequate space in your layout to allow strings to expand when localized.
- Each and every localized image (those that contain text or culture-specific content) has been organized into language-specific folders, appropriately named so that the resource loader can find them automatically.
- Localization isn't just for languages and doesn't have any inherent relationship with the user's region. A user living in one country might want to use a different language, and some resources for otherwise common languages might still need to be localized for different regions (such as specific word spellings, business logos and branding, etc.).
- Any formatting and manipulation of dates, times, time zones, and currencies use APIs that automatically apply regional settings.
- Use format strings to place dynamic content into strings rather than string concatenation. Similarly, do not make assumptions about where first names and surnames are placed, the structure of street addresses and phone numbers, and so on.
- Any sorting or collation of data takes the user's language into account, using APIs for this purpose.
- Make no assumptions about how strings are concatenated. Use format strings with appropriate placeholders instead, and include those in language-specific resource strings so that they can be localized.

- The web services an app uses might vary from location to location because of the need for local information or regional legal requirements.
- The app communicates language and region information to any services it uses to generate tile updates and toast notifications, and the XML payloads provided by those services set the appropriate `lang` attributes so that Windows can use the right fonts when rendering text.
- Text might be laid out left to right or right to left (including text within images). Vertical is also possible but might be implemented in a separate version of an app because of its unique layout needs.
- Text input just works for all languages, whether from a keyboard or an Input Method Editor (IME), which implies that you should avoid hard-coding font names that don't have full Unicode support. It's good to stick with the typography in the WinJS stylesheets—they have built-in support for at least 109 languages.
- The user might switch languages at run time, and the app responds accordingly.
- Test early and test often!

A world-ready app, in short, is both *globalized*—using APIs that isolate regional specifics—and is readily *localizable* such that adding support for another language requires no code changes, just the addition of new string and image resources. This is mostly a matter of how you structure those resources and how you reference them within the app's markup and source code.

The process of *localization*, then, is one of generating or acquiring those language-specific and culture-specific resources, for which some very helpful tools are available to streamline translation work. Adding a new set of resources does not require any changes to the app code, and even though you upload an app package with those new resources to the Windows Store, it will not trigger an app update to customers that don't need them.

Next we'll look at matters of globalization, we'll explore how to structure resources to be localizable, and we'll see how to go about obtaining localized resources. After that, we'll be ready to look at the last step in the long journey of an app's creation—uploading to the Store—which is the story of Chapter 20.

> **Alert!** One of the last steps in uploading your app will be providing localized information for your app's page in the Windows Store. Because you won't be asked for this until you thought you were all done and ready to release your app, the need for these resources will catch you off guard! To avoid that pain and avoid delays in releasing your app, we'll talk about what's needed here as part of the overall localization process. See "Part 3: Creating a Project for Nonpackaged Resources" later on.

Globalization

Besides language, the things that vary around the world are the representation of dates and times (including calendars and time zones); the representation of numbers, measures (units), phone numbers, and addresses; currencies; paper sizes (already discussed in Chapter 17, "Devices and Printing"); how text is sorted (collation); the direction of text; word breaking within text; and the fonts used for text along with the input method. To globalize an app means to make no assumptions about how any of this is accomplished, instead using the WinRT APIs that will do the right thing according to the current user's settings. Working with those APIs is what globalization is mostly about.

Globalization is also about analyzing an app's content, checking for words, phrases, or expressions that might be very difficult to translate (or potentially politically offensive), especially colloquialisms, vernacular, slang, metaphors, jargon, and the like. Use images that travel well and aren't likely to be misinterpreted elsewhere in the world (imagine wearing a T-shirt with such imagery in a country where you intend to market the app!). And exercise caution with maps because there is disagreement among different nations about where, exactly, their borders should be drawn. Be sure also to refer to "country/region" rather than just "country," because disputed territories might not be recognized specifically as a country.

Also be aware of your regional export laws regarding encryption algorithms, because you might not be allowed to make the app available in certain markets. See Staying within export restrictions on cryptography. In addition, if you're writing a game, be mindful of regional game rating requirements that might create more work for you than it's worth. See Preparing your Windows game for publishing.

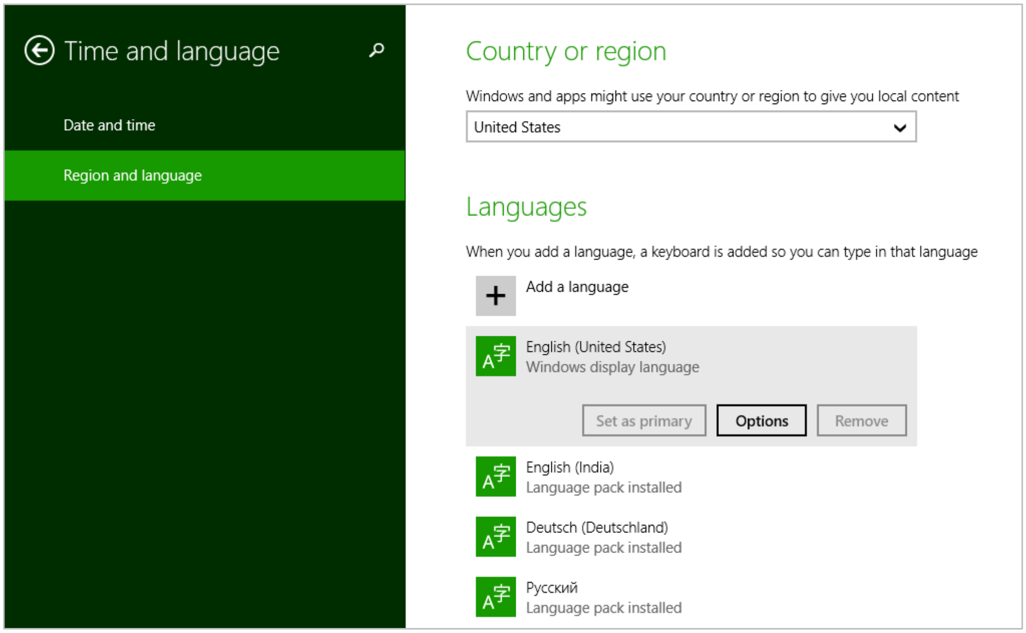
If you use web services, make sure they're appropriate to the user's locale. This might be required by law in some parts of the world (especially for financial transactions and maps) and often ensures that the user gets regionally relevant information from that service, unless they've specifically configured the app otherwise. Also communicate the user's locale and language to those services so that they can return content that's already localized. It's also helpful for the app's overall performance to use servers that are relatively close to the user rather than on the other side of the world!

The first step in any of this, however, is to know where your app is actually running and the user's language and cultural preferences, so let's see how that is accomplished.

User Language and Other Settings

When a user first acquires a Windows device or installs Windows on a machine, it will likely be configured for their country of residence. However, many users speak multiple languages irrespective of where they live and might want to work with Windows in a particular language that has nothing to do with their location. For this reason, always think about the user's preferences separately from the actual location of the device, applying the user's preferences to how your app displays information but using the physical location to control the services you use and other more functional aspects.

Languages, region, and other preferences are configured through PC Settings > Time and Language (or Control Panel > Clock, Language, and Region). Here you can add languages and select your primary one (see Figure 19-3), change input methods, specifically set your location (a country or territory), and set date and time formats (see Figure 19-4). The desktop control panel lets you also set number and currency formats (see Figure 19-5).



Time and language

Date and time

Region and language

Country or region

Windows and apps might use your country or region to give you local content

United States

Languages

When you add a language, a keyboard is added so you can type in that language

Add a language

English (United States)
Windows display language

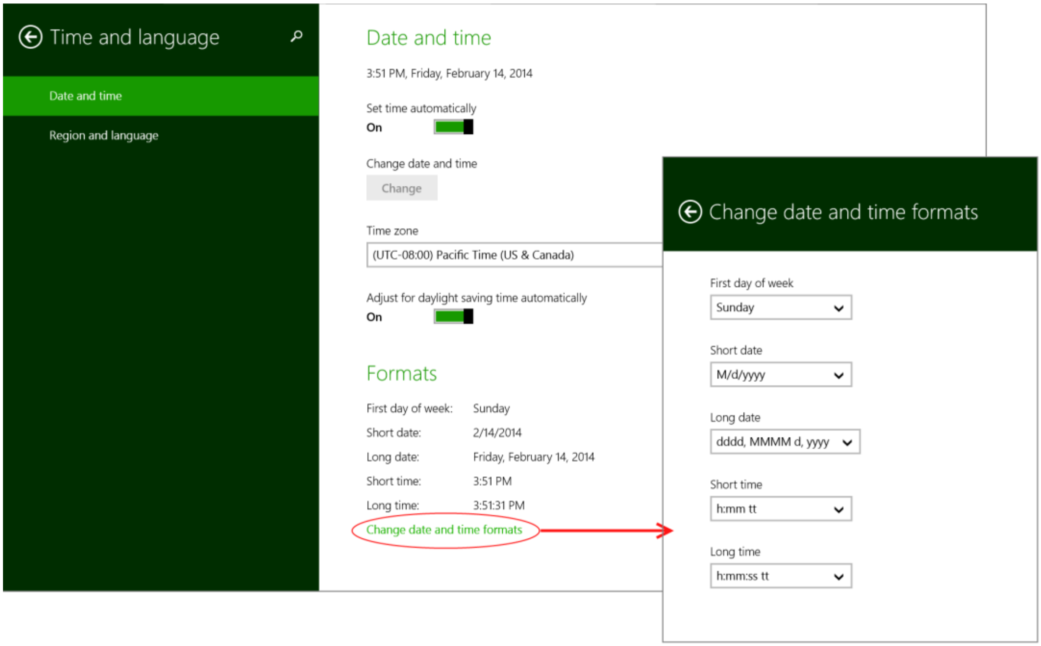
Set as primary | Options | Remove

English (India)
Language pack installed

Deutsch (Deutschland)
Language pack installed

Русский
Language pack installed

FIGURE 19-3 Managing and selecting a language in PC Settings.



Time and language

Date and time

Region and language

Date and time

3:51 PM, Friday, February 14, 2014

Set time automatically
On

Change date and time
Change

Time zone
(UTC-08:00) Pacific Time (US & Canada)

Adjust for daylight saving time automatically
On

Formats

First day of week:	Sunday
Short date:	2/14/2014
Long date:	Friday, February 14, 2014
Short time:	3:51 PM
Long time:	3:51:31 PM

Change date and time formats

Change date and time formats

First day of week
Sunday

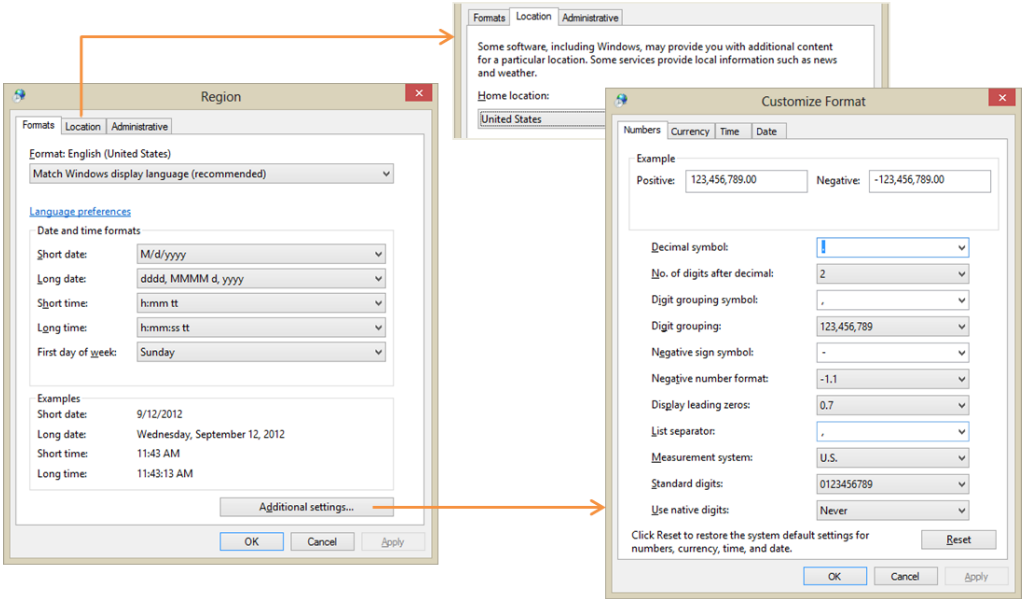
Short date
M/d/yyyy

Long date
dddd, MMMM d, yyyy

Short time
h:mm tt

Long time
h:mm:ss tt

FIGURE 19-4 PC Settings panes for formatting and region.



Region

Formats | Location | Administrative

Format: English (United States)

Match Windows display language (recommended)

Language preferences

Date and time formats

Short date: M/d/yyyy

Long date: dddd, MMMM d, yyyy

Short time: h:mm tt

Long time: h:mm:ss tt

First day of week: Sunday

Examples

Short date: 9/12/2012

Long date: Wednesday, September 12, 2012

Short time: 11:43 AM

Long time: 11:43:13 AM

Additional settings...

OK | Cancel | Apply

Formats | Location | Administrative

Some software, including Windows, may provide you with additional content for a particular location. Some services provide local information such as news and weather.

Home location: United States

Customize Format

Numbers | Currency | Time | Date

Example

Positive: 123,456,789.00 Negative: -123,456,789.00

Decimal symbol: .

No. of digits after decimal: 2

Digit grouping symbol: ,

Digit grouping: 123,456,789

Negative sign symbol: -

Negative number format: -1.1

Display leading zeros: 0.7

List separator: ,

Measurement system: U.S.

Standard digits: 0123456789

Use native digits: Never

Click Reset to restore the system default settings for numbers, currency, time, and date.

Reset

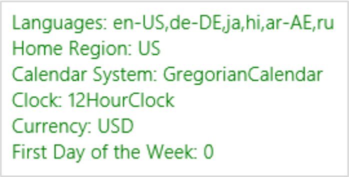
OK | Cancel | Apply

FIGURE 19-5 Control panel dialogs for formatting and region.

It's a good thing there are globalization APIs, because dealing with all the variations here would be quite a chore otherwise! (Note that changes to the formats in Figure 19-4 will affect only those Windows Store apps that are running in the language you're configuring; each set of custom formats is particular to a language.)

The basic details of the user's settings are available through the `Windows.System.UserProfile.GlobalizationPreferences` object and the classes in the `Windows.Globalization` namespace. `GlobalizationPreferences` just provides a handful of properties. Four of these—`calendars`, `clocks`, `currencies`, and `languages`—are each an array of strings (an `IVectorView` to be precise) with the user's preferred settings in order of preference. In the case of `languages`, it contains a list of BCP-47 language tags.

It also contains a string property called `homeGeographicRegion`, which is the abbreviation for the selected value in the Country or Region setting of Figure 19-3, and a property called `weekStartsOn`, which is a `DayOfWeek` value. Scenario 1 of the Globalization preferences sample will retrieve and display these values, except that you'll want to add a line for `currencies`, which is missing from the sample. Having made that change and added a number of languages to my system, I see this output:



```
Languages: en-US,de-DE,ja,hi,ar-AE,ru
Home Region: US
Calendar System: GregorianCalendar
Clock: 12HourClock
Currency: USD
First Day of the Week: 0
```

Generally speaking, these values are exactly what you'll typically need to communicate to a web service if it will be providing localized data to the app. However, the user's language preference is best obtained in a slightly different manner, as we'll see shortly.

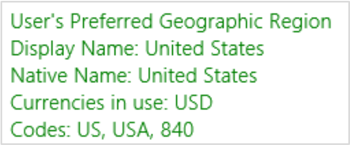
Oftentimes you'll need more detail for all of these settings, for which we can turn to the classes in `Windows.Globalization`. Some of these are static classes that are just there in the API to provide you with all the string identifiers that you would use to make comparisons in code without writing out the strings explicitly. `ClockIdentifiers`, for instance, just contains two string properties, `twelveHour` and `twentyFourHour`, whose values match those returned from `GlobalizationPreferences.clocks`. Similarly, `CalendarIdentifiers` contains string properties for `gregorian`, `hebrew`, `hijri`, `japanese`, `julian`, `korean`, `taiwan`, `thai`, and `umAlQura`. So, if you wanted to compare the user's preferred calendar to a specific one, you'd write code like this:

```
var userCalendar = Windows.System.UserProfile.GlobalizationPreferences.calendars[0];
if (userCalendar == Windows.Globalization.CalendarIdentifiers.julian) {
   // ...
}
```

This way you're fully honoring the key principle of globalization by not making any assumptions about what those calendar strings are.

The other globalization classes are somewhat richer in scope and function. The `Language` class, which you typically instantiate with a specific BCP-47 tag, provides details like the `displayName`, `nativeName`, `languageTag` (and subtags via the `getExtensionSubtags` method), and `script`. Scenario 2 of the aforementioned sample demonstrates this. The `Language` class also has two static members. One is the `isWellFormed` method, which will tell you if a string contains a valid BCP-47 tag. The other is the `currentInputMethodLanguageTag` property, which contains the BCP-47 tag for the user's preferred input. This can be customized in PC Settings to be something other than the language's default (this happens through the Options button shown for a language in Figure 19-3; this is also demonstrated in scenario 4 of the sample.)

Then we have the `GeographicRegion` class, which, if instantiated with no arguments, provides details on the user's home region. You can also instantiate it with a specific location string. Either way, it then provides you with and `isSupported` flag, a `displayName`, a `nativeName`, a variety of code formats for the region (`code`, `codeThreeDigit`, `codeThreeLetter`, and `codeTwoLetter`), and an array of ISO 4217 three-letter codes in `currenciesInUse`. Scenario 3 of the sample shows these values for your configuration, such as:



User's Preferred Geographic Region
Display Name: United States
Native Name: United States
Currencies in use: USD
Codes: US, USA, 840

The ApplicationLanguages class, for its part, contains just a few things. manifestLanguages is an array of languages as defined by the app's manifest; you'll set these when you localize an app. languages contains a combination of the GlobalizationPreferences.languages array and those from manifestLanguages. The first item in this list is the best value to use for the user's preferred language in your app, so this will be the one to send to web services for localization purposes.

Lastly, there's primaryLanguageOverride, a property (BCP-47 tag) that allows you to set an app-specific language preference (and add it to the mix of languages). Setting this (which is persistent across sessions) tells the system what language to use in its own UI that appears *in the context of the app* (such as system-provided flyouts). It's a relatively expensive operation, so avoid using primaryLanguageOverride for transient purposes, such as rendering a few elements in a different language. For that, create a new language context and use that explicitly; see ResourceContext.getForCurrentView and scenario 13 of the Application resources and localization sample.

The last class, Calendar, is quite extensive and contains too many members to list here, many of which work with formatting as well performing calendar math. Before stepping into that arena, however, let's look more broadly at the question of formatting data.

Formatting Culture-Specific Data and Calendar Math

If you clicked around within the formatting and region panes of PC Settings and Control Panel (refer back to Figure 19-4 and Figure 19-5), you'll find that the possible permutations for formatting something as simple as a number is quite mind boggling, let alone dates, times, and currencies! Fortunately, "formatter" classes in WinRT take care of all the details such that you can take a value from new Date(), for example, and get back a string that completely reflects the user's preferences. The APIs also provide parsing services that work in the opposite direction.

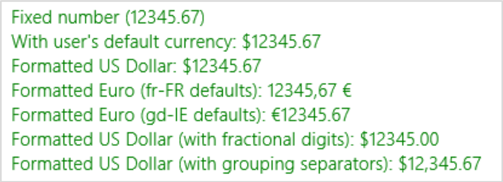
In Windows.Globalization.NumberFormatting we have CurrencyFormatter, DecimalFormatter, PercentFormatter, and PermilleFormatter, which you should always use when converting data values into UI display strings. All of these classes are demonstrated in the Number formatting and parsing sample, where the basic process is to instantiate the formatter with or without specific codes or languages, set any necessary properties for the formatter (such as the number of digits, separator usage, and the CurrencyFormatterMode), and then call its format method to obtain a string or, alternately, one of its parse* methods to turn a string into a number.

For example, to format a currency value (perhaps to present the cost of an in-app purchase in your UI), instantiate a CurrencyFormatter with a currency identifier (and an optional language list and geographic region), set up any options, and then call format (js/CurrencyFormatting.js; I've created a namespace variable for simplicity):

```
var nf = Windows.Globalization.NumberFormatting;
var currs = Windows.Globalization.CurrencyIdentifiers;
var userCurrency = Windows.System.UserProfile.GlobalizationPreferences.currencies;

var wholeNumber = 12345;
var fractionalNumber = 12345.67;
```

```
// Apply user defaults
var userCurrencyFormat = new nf.CurrencyFormatter(userCurrency);
var currencyDefault = userCurrencyFormat.format(fractionalNumber);

// Apply a specific currency
var currencyFormatUSD = new nf.CurrencyFormatter(curs.usd);
var currencyUSD = currencyFormatUSD.format(fractionalNumber);

// Apply a specific currency, language, and region (France, then Ireland)
var currencyFormatEuroFR = new nf.CurrencyFormatter(currs.eur, ["fr-FR"], "ZZ");
var currencyEuroFR = currencyFormatEuroFR.format(fractionalNumber);

var currencyFormatEuroIE = new nf.CurrencyFormatter(currs.eur, ["gd-IE"], "IE");
var currencyEuroIE = currencyFormatEuroIE.format(fractionalNumber);

// Include fractions with a whole number
var currencyFormatUSD1 = new nf.CurrencyFormatter(currs.usd);
currencyFormatUSD1.fractionDigits = 2;
var currencyUSD1 = currencyFormatUSD1.format(wholeNumber);

// Group integers
var currencyFormatUSD2 = new nf.CurrencyFormatter(currs.usd);
currencyFormatUSD2.isGrouped = 1;
var currencyUSD2 = currencyFormatUSD2.format(fractionalNumber);

// Formatted using currency code instead of currency symbol
var currencyFormatEuroModeSwitch = new nf.CurrencyFormatter(currs.eur);
currencyFormatEuroModeSwitch.mode = nf.CurrencyFormatterMode.useCurrencyCode;
var currencyEuroCode = currencyFormatEuroModeSwitch.format(fractionalNumber);

// Return back to currency symbol
currencyFormatEuroModeSwitch.mode = nf.CurrencyFormatterMode.useSymbol;
var currencyEuroSymbol = currencyFormatEuroModeSwitch.format(fractionalNumber);
```

The output of this code is as follows:



```
Fixed number (12345.67)
With user's default currency: $12345.67
Formatted US Dollar: $12345.67
Formatted Euro (fr-FR defaults): 12345,67 €
Formatted Euro (gd-IE defaults): €12345.67
Formatted US Dollar (with fractional digits): $12345.00
Formatted US Dollar (with grouping separators): $12,345.67
```

The other number formatters all work like this, so I'll leave it to you to check out the details in the documentation and the scenarios 1–4 of the sample. In addition, scenario 5 (js/RoundingAndPadding.js) demonstrates the use of rounding and padding capabilities of the `NumberFormatting` API, which involves the `IncrementNumberRounder` and `SignificantDigitNumberRounder` classes, along with the `RoundingAlgorithm` enumeration providing the options: `none`, `roundDown`, `roundUp`, `roundTowardZero`, `roundAwayFromZero`, `roundHalfDown`, `roundHalfUp`, `roundHalfTowardZero`, `roundHalfAwayFromZero`, `roundHalfToEven`, and `roundHalfToOdd`.

Scenario 6 (js/NumericalSystemTranslation.js) shows use of the NumeralSystemTranslator class that converts between the common Latin system and a wide variety of others used around the world. Refer to NumeralSystem values for the options. And scenario 7 shows how to do number formatting with Unicode extensions to language tags, such as a tag of ja-jp-u-nu-arab.

To format dates and times, we turn now to the Windows.Globalization.DateTimeFormatting namespace, where we find the DateTimeFormatter class along with many enumerations for the different ways to formats seconds, minutes, hours, days, months, and years. To use the API, you instantiate a formatter object specifying the desired formats and applicable languages. (There are no less than seven separate constructors here!) You then set options like the clock, geographicRegion, and so forth and call its format method with the desired Date value. You can even apply custom formats, if desired. Many such variations are demonstrated in the Date and time formatting sample, including tags with Unicode extensions in scenario 5 and time-zone handling in scenario 6. I trust a small code snippet will suffice here (from scenario 2, js/stringtemplate.js):

```
var mydatefmt1 = new Windows.Globalization.DateTimeFormatting.DateTimeFormatter(
    "month day");
var mytimefmt1 = new Windows.Globalization.DateTimeFormatting.DateTimeFormatter(
    "hour minute ");
var dateToFormat = new Date();
var mydate1 = mydatefmt1.format(dateToFormat);
var mytime1 = mytimefmt1.format(dateToFormat);
```

The other bit of relevant code from the SDK is the Calendar details and math sample. A world-ready app must not make assumptions about how time periods are computed or compared because this can vary with regional calendars. This is why the extensive Windows.Globalization.Calendar class contains ten distinct add* methods that range from addNanoseconds to addEras, along with its compare and compareDateTime methods (and a bunch to get all the little bits of calendar-related text). In other words, drill it into your mind now to never, ever use arithmetic operators on date and time values because they won't work properly in every locale. Even where you think you know what you're doing, you can get wrong answers when situations like daylight savings time get involved, where the number of hours in two days of every year will not actually be 24!

The bottom line is although we make many assumptions about calendar math in day to day life, where exact precision isn't needed, we must avoid writing code using those same assumptions. For example, if I say "I'll get back to you in a month" and today is February 12th, do I really mean March 12th or something else? What I don't want to do in code is just add 1 to the month (even if I do remember to check for December!). Instead, I should always use Calendar.addMonths method.

Sorting and Grouping

Just as a world-ready app cannot make assumptions about comparing date and time values, it cannot make assumptions about how strings are sorted. Every language has its own way of sorting that doesn't necessarily have anything to do with the values of the character codes involved. I was reminded of this just recently: my work on this second-edition chapter coincided with the opening of the 2014 Winter Olympics in Sochi, Russia (or, I should write, Сочи, Россия). During the opening ceremonies, some were surprised that the Zimbabwe team (Зимбабве in Russian) entered ahead of Chile (Чили), which happened because З comes before Ч in the Russian alphabet.

The bottom line here is that you should never sort strings by character code or based on any assumptions you might think would apply: always use a language-aware API instead.

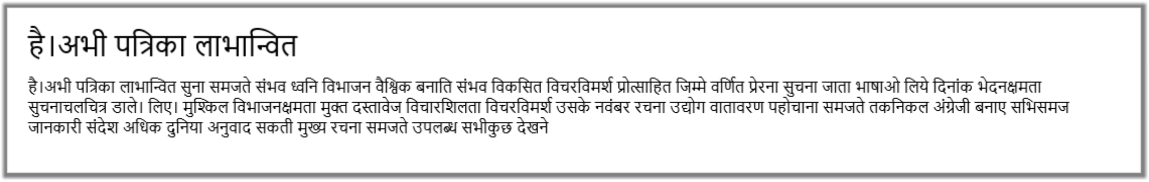
For Windows Store apps written in HTML and JavaScript, you can use the localeCompare method that's already built into strings (even for individual characters). This performs the comparison based on the user's current language. You can also use a string's tolocaleLowerCase and toLocaleUpperCase methods. In Chapter 7, "Collection Controls," specifically in "Quickstart #4: The ListView Grouping Sample," we also saw how to use the Windows.Globalization.Collation.CharacterGroupings API to create proper groupings by the first character of item titles.

Fonts and Text Handling

Thanks to Unicode and the ability of HTML to directly handle text in different languages, there's little you need to do to make text appear properly within your layout. For example, if you look at the Language font mapping sample, pages like html/scenario2.html that contain this markup:

```
<div id="scenario2Document">
    <h2 lang="hi" id="scenario2Heading" contenteditable="true">
        है।अभी पत्रिका लाभान्वित</h2>
    <p lang="hi" contenteditable="true">
        है।अभी पत्रिका लाभान्वित सुना समजते संभव ध्वनि विभाजन वैश्विक बनाति संभव विकसित
        विचरविमर्श प्रोत्साहित जिम्मे वर्णित प्रेरना सुचना जाता भाषाओ लिये दिनांक भेदनक्षमता
        सुचनाचलचित्र डाले। लिए। मुश्किल विभाजनक्षमता मुक्त दस्तावेज विचारशिलता विचरविमर्श
        उसके नवंबर रचना उद्योग वातावरण पहोचाना समजते तकनिकल अंग्रेजी बनाए सभिसमज जानकारी
        संदेश अधिक दुनिया अनुवाद सकती मुख्य रचना समजते उपलब्ध सभीकुछ देखने</p>
</div>
```

just show up like you expect they would (and if you read Hindi, you'll see that this is just jibberish):



है।अभी पत्रिका लाभान्वित

है।अभी पत्रिका लाभान्वित सुना समजते संभव ध्वनि विभाजन वैश्विक बनाति संभव विकसित विचरविमर्श प्रोत्साहित जिम्मे वर्णित प्रेरना सुचना जाता भाषाओ लिये दिनांक भेदनक्षमता सुचनाचलचित्र डाले। लिए। मुश्किल विभाजनक्षमता मुक्त दस्तावेज विचारशिलता विचरविमर्श उसके नवंबर रचना उद्योग वातावरण पहोचाना समजते तकनिकल अंग्रेजी बनाए सभिसमज जानकारी संदेश अधिक दुनिया अनुवाद सकती मुख्य रचना समजते उपलब्ध सभीकुछ देखने

What this particular sample is meant to demonstrate is the Windows.Globalization.Font.LanguageFontGroup object, which provides specific font recommendations for different parts of the UI. Once created using a specific BCP-47 tag, the object contains a number of properties, each of type LanguageFont, such as uITextFont and uIHeadingFont (notice the odd casing again). Each LanguageFont object then contains properties of fontFamily, fontStretch, fontStyle, fontWeight, and scaleFactor. Through a couple of helper functions in js/langfont.js, which are deceivingly added to the WinJS.UI namespace without being part of WinJS itself, these recommendations are applied to elements in the DOM simply by setting the appropriate styles for those elements.

Be clear that these font recommendations are really refinements and not necessary for basic functionality. As scenario 4 of the sample demonstrates, a basic English font (with Unicode characters, of course) applied to mixed English/Japanese text will still render the Japanese but perhaps not optimally. Applying the recommended font will make that refinement.

With text, let me remind you of a few APIs in the `Windows.Data.Text` namespace that we saw briefly in Chapter 15, "Contracts," and are worth mentioning again. These are the ones for semantic text querying (the `SemanticTextQuery` class), segmenting words (`WordsSegmenter`), segmenting selection ranges (`SelectableWordsSegmenter`), and tokenizing identifiers within strings (`UnicodeCharacters`). For demonstrations of these, refer to the Semantic text query sample, the Text segmentation API sample, and the Unicode string processing API sample.

The other aspect to working with different fonts and languages is how these affect your overall layout, something we didn't go into in Chapter 8. This is discussed in the documentation on How to adjust layout for various languages and support RTL layout, but let me summarize that material and add a bit more.

First of all, a world-ready app leaves extra space for various bits of content like headings and labels because the words and phrases will be longer in some languages and shorter in others. The general guidelines are to leave at least 30% more room over what's needed in English for typical strings and as much as 300% for really short strings or single words. As a simple example, the English word "wrench" translates into German as "Schraubenschlüssel"; the word "click" (if I'm to trust Bing Translator), translates into Greek as "Κάντε κλικ στο κουμπί" (literally "do click on the button"—the click word is just κλικ). You might need word wrapping in some cases too!

For all such purposes you can and should use the `:lang()/:-ms-lang()` pseudo-class selector in CSS to adjust styles like `width` for specific languages. Be sure to test your app with those languages or test thoroughly with the *pseudo-language* (see "Testing with the Pseudo Language" later).

Secondly, different languages flow text in directions other than the left to right (and top to bottom), like English and many Indo-European languages. Arabic and Hebrew, for instance, read right to left (RTL) instead of left to right; a few will flow top to bottom first, then right to left.

When making your app world-ready for RTL languages (considering that such markets are significant), you'll want to support what is called *mirroring* in your layout. It really means reversing your layout, including images, the direction of the back button, the direction of animations, panning directions, and so forth.

Fortunately, HTML and CSS layout automatically accommodate this, and the WinJS stylesheets, ui-light.css and ui-dark.css, set the CSS `direction` style appropriately as follows (something you should use on the element level for RTL languages rather than `align`):

```
html:-ms-lang(ar, dv, fa, he, ku-Arab, pa-Arab, prs, ps, sd-Arab, syr, ug, ur, qps-plocm) {
    direction: rtl;
}
```

In fact, look around in the WinJS stylesheets and you'll find many adjustments made for RTL languages with `:-ms-lang`, specifically with margins and padding. By using HTML, CSS, and WinJS—including built-in controls—much of the mirroring is taken care of automatically. Here My Am!, for instance, just works in Hebrew.

With images, you can reverse them when needed by applying a `transform: scaleX(-1)` style to the necessary elements. If, however, you have images that really need to be replaced (as when some parts would be mirrored but other parts would not), you can use *layoutdir-RTL* in the image filename in the same way we've seen for pixel densities and contrast. In fact, there are many qualifiers for use with resources that are described on How to name resources using qualifiers, something we'll be looking at more closely in a moment.

Sometimes you'll need to reverse a certain portion of text, as when mixing languages in the same paragraph. For this you can apply the `unicode-bidi` style in conjunction with `direction`. (Do note that numbers are generally direction-neutral so that they take on the directionality of their containing element, so you might need to set direction separately.) Along similar lines, you can also use the `-ms-writing-mode` style to flow text in just about any other direction, something you might use for an app that presents classical Chinese, Japanese, or Korean poetry.

Preparing for Localization

Once your app has been made world-ready and can handle just about any language and regional settings you want to throw at it, the next step is to make sure that language-specific resources in the app are cleanly separated from your HTML, CSS, and JavaScript and placed in your resources where the Windows resource loader (also referred to as the Resource Management System) can find them.

Before going further, there's an excellent topic in the documentation on this subject, How to prepare for localization, which provides suggestions for translation and other details. It's not productive to repeat all of that here, of course, so I'll instead to break that guidance down into a couple of steps that you can apply to an app and its default language before adding support for additional languages. As you'll see, one of these steps has nothing to do with the app package itself but has to do with localized resources you'll need when *submitting* the app to the Windows Store.

Tip Early in your development work, reserve localized names for your app in the App Name section of the Windows Store dashboard.

Note The resource loader supports *sparse localization* for dealing with slight variations between similar languages or regions. It means that with languages like American English (en-US) and British English (en-GB), most of the app's resources can be assigned to en-US with en-GB resources for only those bits that vary, like "color" vs. "colour" and changing "Contoso, Inc." to "Contoso, Ltd.," or vice-versa. You might also need to provide region-specific graphics, such as the logos under which you do business in those regions. In any case, because each resource is resolved individually according to the user's preferences, an app running in an en-GB context will find those specific bits first, if they exist, otherwise the loader will look in the en-US resources. There is also support for dealing with specific language exceptions through the use of resources marked with the undetermined tag *und*. See How to manage language and region, step 4 (toward the end), for details, along with Language Matching.

Part 1: Separating String Resources

The first step in preparing for localization is to move language-specific or region-specific strings from source files into a string resource file and then to insert resource references where necessary. Coming up, in “Part 2: Structuring Resources for the Default Language,” we’ll set up the folder structure for this file and image resources that will then accommodate localized versions.

To create your first string resource file, right-click your project in Visual Studio’s solution explorer, select Add > New Item, and select Resources File (.resjson). Although you can change the filename, just leave it set to the default resources.resjson for now. Press Add, and the file will be created in your project root, where we’ll also leave it until Part 2.

Omitting a comment at the top, the contents of this file appear as follows:

```
{
    “greeting”            : “Hello World!”,
    “_greeting.comment”     : “This is a comment to the localizer”
}
```

As you can see, the file is just plain JSON where each property has a string identifier and a string value; any resjson file can have as many properties as you want.

Clearly, too, there is a relationship between the two entries above. The first entry of the form *<identifier> : <value>*, is a real string resource that maps a valid JSON identifier (no whitespace) to a string value. This is what the resource loader will use to replace references to the identifier with the string value.

Any entry that begins with an underscore, as in *<_identifier.comment> : <value>* (a conventional form), is ignored by the resource loader. Such entries provide notes for a translator so that they can fully understand how the string is used and specific parts that shouldn’t be translated. A second optional entry, *<_identifier.source> : <value>*, provides the original string in the default language, which is very helpful for reference.

If you want to see some more extensive resjson files, open the Application resources and localization sample and look in the *strings* folder under a particular language. In the ja/resources.resjson file, for example, you’ll see the string resources along with both comment and source entries:

```
{
    “displayName”                    : “アプリケーション リソース JS SDK サンプル”,
    “_displayName.source”             : “Application Resources JS SDK Sample”,
    “_displayName.comment”            : “Don’t change ‘SDK’”,

    “description”                    : “アプリケーション リソース JS SDK サンプル”,
    “_description.source”             : “Application Resources JS SDK Sample”,
    “_description.comment”            : “Don’t change ‘SDK’”,
}
```

Turning back to your own app with the new resources.resjson file in hand, we're now ready to go on a search-and-replace mission throughout the app project, looking for localizable strings, extracting them into the resource file, and replacing them in the source files with an appropriate reference. The three primary places we need to look at are your HTML files, JavaScript files, and the app manifest. To demonstrate, I'll show what I did with the Here My Am! example (to which I've added a resources.resjson file).

> **Note** CSS files can contain string literals in the `content` and `quotes` styles; however, resource lookup from CSS is not supported for Windows Store apps. Localization must be done in CSS with the `:lang` and `:-ms-lang` pseudo-selectors.

JavaScript: Let's start with JavaScript, where you need to scrub your code for string literals, including any you are drawing to a `canvas`. In Here My Am! I found only a couple localizable strings, namely a folder name used in the Pictures library, text for placeholder images (six instances total), and the title and description used when formatting text for the Share contract and live tiles (pages/home/home.js):

```
var folderName = "HereMyAm";
data.properties.title = "Here My Am!";
setPlaceholderImage("photo", "Tap to capture photo");
setPlaceholderImage("mapDiv", "Could not create the map.", "Tap to try again.");
return "At latitude " + lat + ", longitude " + long;
return "At (" + lat + ", " + long + ")";
```

and the Settings commands in js/default.js:

```
e.detail.applicationcommands =
    {
        "about":   { title: "About", href: "/html/about.html" },
        "help":    { title: "Help", href: "/html/help.html" },
        "privacy": { title: "Privacy Statement", href: "/html/privacy.html" }
    };
```

Note that in home.js English strings are used for exceptions, but because they're only for debugging purposes they don't need to be localized.

Extracting the appropriate strings into resources.resjson, then, that file appears as below where I'm using regular comments to identify where the strings are used. Note that I've include the folder name and share title although they won't typically be translated, and note that the format strings are used in the `formatLocation` function in js/home.js:

```
{
    // pages/home/home.js
    "foldername" : "HereMyAm",
    "_foldername.comment" : "Not translated",
    "share_title" : "Here My Am!",
    "_share_title.comment" : "Not translated",
    "location_formatShort"         : "At (%s, %s)",
    "_location_formatShort_comment" : "Used to format a short location as in 'At (120, 45)",
    "location_formatLong"          : "At latitude %s, longitude %s",
    "_location_formatLong_comment"  :
                        "Used to format a long location, 'At latitude 120, longitude 45",

    // default.js
    // Settings panel commands
    "about_command"   : "About",
    "help_command"    : "Help",
    "privacy_command" : "Privacy Statement",
}
```

I highly recommend that you organize your entries by source file like this, and you can also use multiple resource files if you like, as explained in "Part 2" coming up. Also be careful about how you reuse the same string that occurs in multiple places. If it's for the same kind of UI with the same intent, that's fine, but if the usage context is different, make a separate copy of the string because it might translate differently for that context. In the resources above, notice how I also included a comment for *location_formatShort* because the word "At" by itself probably needs more context to translate properly.

With the strings separated as resources, we can now use the resource loader to obtain those strings at run time. This can be done in two ways. First is with the WinRT APIs directly, namely Windows.ApplicationModel.Resources.ResourceLoader.getString:

```
var loader = new Windows.ApplicationModel.Resources.ResourceLoader();
var text = loader.getString('location_formatShort');
```

or, more simply, with the WinJS.Resources.getString wrapper:

```
var text = WinJS.Resources.getString('location_formatShort').value;
```

that also happens to work in the web context where WinRT isn't defined (see scenario 12 of the Application resources and localization sample). Note that `getString` returns an object that has the string in its `value` property along with `lang` and an `empty` flag indicating if the resource wasn't found.

The concise WinJS method is clearly helpful in cases like our settings commands because we can call it inline. Thus, in our code we just replace the string literals with the WinJS call, such as the following in pages/home/home.js:

```
data.properties.title = WinJS.Resources.getString('share_title').value;
```

and the following inside the object for the Settings commands:

```
"about":   { title: WinJS.Resources.getString('about_command').value,
    href: "/html/about.html" },
```

Note that WinJS, being optimized for common scenarios, supports loading strings in the user's default language only. The WinRT `ResourceLoader` class, on the other hand, is much more flexible and can load a string for any specific language and apply a different language for each app view. You'll need to use that API when your requirements exceed what WinJS provides.

And that's really it for JavaScript. If you've made these changes to your app, now is a good time to use the Build > Build Solution command in Visual Studio. This will compile the resources.resjson file into a more efficient binary format called resources.pri, ignoring entries that begin with an underscore. Doing an occasional build (without starting the app) is a good practice when working with resources so that you can clean up any problems in your files, such as duplicate entries or syntax errors. Then you can run the app to see the resource loader in action—mostly by seeing no difference from the app as it was before! Be sure to test all the code paths that were affected, however, to ensure that all the strings are being loaded properly.

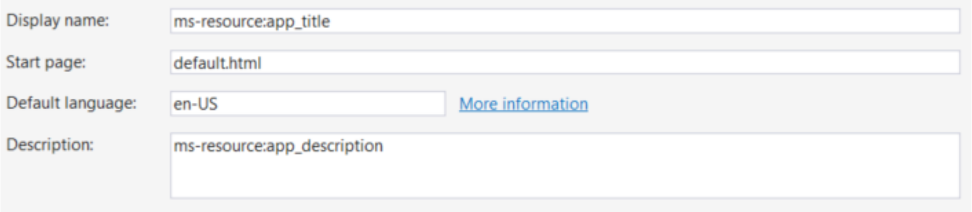
Manifest: Let's turn now to the manifest, where the story is even simpler because there's no code involved. The textual pieces here that might need localization are

- The Display Name, Description, and URI Template fields on the Application tab.
- The Short Name on the Visual Assets tab.
- Any text descriptions for specific entries on the Declarations tab, such as the Share Description for a Share Target declaration.
- The Package Display Name on the Packaging tab. (You would change the package name only if you were going to submit the app to the Store under a different name in certain markets, in which case it's a different app package entirely.)
- Possibly some URIs on the Content URIs tab.

While we're looking at the manifest, note the Default Language setting on the Application tab. This is what defines the app's default or *fallback* language if the user runs the app with a language that isn't provided for in your resources. We'll also come back to images in the manifest in "Part 2."

Looking at all the strings in the manifest, extract these to your resources.resjson file, giving them appropriate identifiers. Again, if you have some strings in the manifest that match those elsewhere in the app, carefully evaluate them to determine if they can all use the same resource. When in doubt, keep them separate (the overhead is quite small). In the case of Here My Am!, the app's display name and the string used for the header on the main page are the same and have the same usage, so they can refer to the same resource.

To refer to those resources in the manifest, use *ms-resource:<identifier>*. For example, I moved the Application > Display Name value into the resource file and called it *app_title*, so in that field of the manifest editor I simply write *ms-resource:app_title*. I did the same for the description and the package display name.



Display name: ms-resource:app_title

Start page: default.html

Default language: en-US

Description: ms-resource:app_description

Once you've made these changes, run the app and make sure that the text on your tile, if you're using it, shows up properly. You might temporarily check the Visual Assets > Show Name options so that you can see the name, but be sure to uncheck these boxes before you ship!

Sidebar: Localized Strings in Tile and Toast XML Payloads

As described on Globalization and accessibility for tile and toast notifications, XML payloads for tile and toast notifications use the `ms-resource:` syntax to identify string resources in `text` elements. This will trigger the resource loader's lookup mechanism when the tile or toast is rendered, and this works regardless of whether the notification was issued locally, obtained from a web service, or received as a push notification. The web services just need to be sure they use the app's particular resource identifiers.

A web service can also issue localized tile updates directly. In this case, an app will generally append query strings to the service URI to communicate the desired language, updating those parameters as necessary when the language changes (see "Localization Wrap-Up" for details). An app can also combine this with using regional web services that help localize the update content.

If you configure a periodic tile update in your manifest (on the Application tab), you won't be able to modify the URI from code, obviously. Instead, include the tokens *{language}* and *{region}* in your URI parameters, which Windows will replace at run time with the settings. For example, if you give the URI http://<*host*>/<*service*>?lang={languages}&region={region}, your service will receive a parameters string like *?lang=en-US&region=US*.

Finally, remember to set the `lang` attribute in various elements of your tile and toast XML payloads so that Windows can select the proper font for text content. Also, use the `addImageQuery` option in the payload to append query strings for scale, contrast, and language to any requests made for remote images. Refer to "Using Local and Web Images" in Chapter 16.

HTML: The final place we need to look for strings is our HTML, which I've saved for last because it's the most involved. In HTML, really scrub, scrub, scrub your markup for any hard-coded text that will become visible in the UI. Check the body content of elements like `p`, `h1`, `span`, `div`, `button`, `option`, and so on, as well as the value of attributes like `title`, `alt`, `aria-label`, etc. Also look inside WinJS controls like AppBar and Flyout, and look for any embedded URIs that you'll want to localize, including those of services you employ and content you show in a webview or `iframe`. Note, though, that `title` elements in a page `head` are not shown and do not need localization.

In Here My Am! I found lots of strings in pages/html/home.html, which I've highlighted below:

```
<header id=”header” aria-label=”Header content” role=”banner”>
<h1 class=”titlearea win-type-ellipsis”><span class=”pagetitle”>Here My Am! (16)</span></h1>
<section id=”section” aria-label=”Main content” role=”main”>
<div id=”photoSection” class=”subsection” aria-label=”Photo section”>
<h2 class=”group-title” role=”heading”>Photo</h2>
```

```
<img id="photoImg" src="#" alt="Tap to capture photo" role="img" />
<div id="cannotCapture" class="errorOverlay win-type-x-large">
    Widen the view to capture a photo</div>
<div id="locationSection" class="subsection" aria-label="Location section">
<h2 class="group-title" role="heading">Location</h2>
<img id="errorImg" src="#" alt="Could not create the map. Tap to try again." Role="img" />
<div id="noLocation" class="errorOverlay win-type-x-large">
    Unable to obtain geolocation;<br /> use the app bar to try again.</div>
<div id="retryFlyout" data-win-control="WinJS.UI.Flyout" aria-label="Trying geolocation"
    data-win-options="{anchor: 'mapDiv', placement: 'bottom', alignment: 'center'}">
    <div class="win-type-large">Attempting to obtain geolocation...</div>
</div>
```

In default.html, I also found labels and tooltips in the `data-win-options` attributes of the appbar commands (and I'm omitting some of the other markup for brevity):

```
<div id="appbar" data-win-control="WinJS.UI.AppBar" data-win-options="">
    <button data-win-options="{id:'cmdPickFile', label:'Load picture', icon:'browsephotos',
        section:'global', tooltip:'Load a picture through the file picker'}">
    </button>
    <button data-win-options="{id:'cmdRecentPictures', label:'Recent pictures',
        icon:'pictures',
        section:'global', tooltip:'Browse recent pictures taken in the app'}">
    </button>
    <button data-win-options="{id:'cmdRefreshLocation', label:'Refresh location',
        icon:'globe',
        section:'global', tooltip:'Refresh your location'}">
    </button>
</div>
```

Tip When preparing for localization, consider whether your appbar icons have a universal meaning. If not, they'll also need to be localized. Fortunately, the icon values are strings and can be localized as such, in which case you can treat them like the labels and tooltips.

Other affected files in this app include all of those in the HTML folder that are used for Settings commands. With such commands, be especially careful to note the short header labels that might be in a `div` amongst a bunch of other markup. Leave nothing behind!

Once you've located your strings, copy them as before to resources.resjson. This will likely be an extensive workout with copy and paste, so grab some refreshments and go for it. It's also fine to have HTML in these strings—they'll just be inserted into the markup and they'll render as such if you attach them to a property like `innerHTML`, but don't include the surrounding tag (we'll need it shortly). For example, in html/about.html I have a number of `p` elements with text, such as:

```
<p>Here My Am!<br />Version 2.0.0.0<br /></p>
```

for which I make the following string in the resources (no `p` tag):

```
"about1"   : "Here My Am!<br />Version 2.0.0.0<br />",
```

Using data-win-res attributes: Now comes the fun part. How do we reference the string resources in markup? If you think about it a little bit, we have to run some piece of code to go through and replace whatever references we make with the appropriate string from the resource file. Hmmm. Haven't we seen something like this before? Indeed we have. With controls, we added `data-win-control` attributes to the markup and used `WinJS.UI.processAll` or `WinJS.UI.process` to run the code to instantiate the control. We have a similar setup for resources: a `data-win-res` attribute and `WinJS.Resources.processAll`, the latter of which should be called in each page's `processed` or `ready` method (`processed` is a good place because it's called after `WinJS.UI.processAll`), or wherever else HTML content like the appbar is loaded, such as in the app's `activated` handler after `WinJS.UI.processAll` (so that the controls are instantiated).[130] Here's what you do in markup:

- Replace attributes and their string values with `data-win-res="{<attribute> : '<identifier>'}"` where <*attribute*> is the original attribute name and <*identifier*> matches the desired string in the resource file, contained in single quotes.
- Where there are multiple attributes in the same element, you can separate each <*attribute*> *:* *'<identifier>'* pair with a comma.
- When the string is directly inside a tag, we use `data-win-res` with the equivalent attribute name, such as `textContent` for a `div`, `p`, or `span`. If the string contains markup, use `innerHTML` instead, but use it only when necessary because `textContent` is much faster.
- For hyphenated attributes like `aria-label`, use the syntax *{attributes: {'<attribute>' : '<identifier>'}}* in the `data-win-res` value, using single quotes around <*attribute*>. This is how you combine localization and accessibility together.
- For properties of WinJS controls that would normally appear within `data-win-options`, place those in `data-win-res` with the syntax *{winControl: {<property> : '<identifier>'}}*. Multiple properties are again separated with a comma within the inner { }'s.

Here are some examples as modified from the earlier markup:

Original Markup	**Modified Markup**
`<img id="photoImg" src="#" alt="Tap to capture photo" role="img" />`	`<img id="photo" src="#" data-win-res = "{alt: 'photo_alt'}" role="img" />`
`<span class="pagetitle">Here My Am! (19)</span>`	`<span class="pagetitle" data-win-res = "{textContent : 'app_title'}"></span>`
`<div id="locationSection" class="subsection" aria-label="Location section">`	`<div id="locationSection" class="subsection" data-win-res="{attributes: {'aria-label' : 'aria_location'}}" >`
`<div id="noLocation" class="errorOverlay win-type-x-large">Unable to obtain geolocation;<br /> use the app bar to try again.`	`<div id="noLocation" class="errorOverlay win-type-x-large" data-win-res="{innerHTML : 'error_obtaingeoloc'}">`

[130] If you see a `null` reference exception within `WinJS.Resources.processAll`, it's probably because you're trying to map resources for a WinJS control that hasn't been instantiated.

<button data-win-control=”WinJS.UI.AppBarCommand” data-win-options=”{id:’cmdPickFile’, label:’Load picture’, icon:’browsephotos’, section:’global’, tooltip:’Load a picture through the file picker’}”>	<button data-win-control=”WinJS.UI.AppBarCommand” data-win-options=”{id:’cmdPickFile’, icon:’browsephotos’, section:’global’}” data-win-res=”{winControl: {label : ‘appbar_label1’, tooltip : ‘appbar_tooltip1’}}”>

Tip If you add a `data-win-res` attribute and it’s not working, make sure you are really calling `WinJS.Resources.processAll` in the page control’s `ready` or `processed` method, otherwise that page’s resource strings won’t be set.

When `WinJS.Resources.processAll` goes through the DOM, it doesn’t actually remove any of the `data-win-res` attributes; it just processes those values and adds other attributes to the element that contain the string resource. The advantage of this is that a later call to `processAll` will go through the DOM and refresh all those strings. That means that when handling the `WinJS.Resources.oncontextchanged` event, which tells you when the language changes, you can call `processAll` again and your UI appears in that new language! We’ll add this little piece of code later on once we’ve added a few more languages to Here My Am!.

It also means that if you want to perform WinJS data binding in conjunction with a string that originates in your resources, simply include `data-win-bind` attributes inside those strings, assign that string to an element’s `innerHTML` property within `data-win-res`, and then be sure to call `WinJS.Resources.processAll` *before* calling `WinJS.Binding.processAll`. This is demonstrated in scenario 8 of the Application resources and localization sample (html/scenario8.html and js/scenario8.js):

```
HTML: <p id=”messageCount” data-win-res=”{innerHTML: ‘scenario8MessageCount’}”>
Resources: “scenario8MessageCount” : “You have <span
         data-win-bind=\”innerText:count\”></span> message(s)”,
```

And with that (except for the following sidebar that I’ve cleverly inserted), we’re ready for the next step that will take care of our image resources and set us up to localize all this content we’ve extracted.

Sidebar: String Resources in Settings Flyouts

When localizing Here My Am!, one of the more challenging pieces of markup to work with were the HTML pages for settings flyouts, namely about.html, help.html, and privacy.html in the project’s HTML folder. These pages are not loaded until the settings command is invoked, and because that’s happening all within WinJS, we have to use the flyout’s `beforeshow` event to call `WinJS.Resources.processAll` for the flyout’s markup. To catch that event, I added `onbeforeshow : beforeShow` in each flyout’s `data-win-options` string and then this piece of code within a `script` tag at the end of the `body` element:

```
function beforeShow() {
   WinJS.Resources.processAll();
}
beforeShow.supportedForProcessing = true;
```

where the last line is necessary because WinJS will be calling `WinJS.UI.processAll` when the page is loaded. In any case, this works just great for patching up the string resources, including setting a webview's `src` attribute with a localized URI. Note, however, that this doesn't work for `iframe` attributes because that's blocked from `processAll` and will throw an exception that complains about something not being marked with `supportedForProcessing`. Bottom line is that you can't use `data-win-res` with an `iframe`.

Fortunately, the simple solution is to give the `iframe` an id (say *privacyFrame*), load the string manually in the `beforeshow` handler, and then set the `iframe.src` attribute:

```
document.getElementById("privacyFrame").src =
    WinJS.Resources.getString('privacy_URI').value;
```

This code is included in comments of the privacy.html file of Here My Am! for this chapter.

Part 2: Structuring Resources for the Default Language

In "Part 1: Separating String Resources," we created only a single resources.resjson file in the root folder of the project and we deferred any work on images. The next step is to introduce a little more structure into the project that will allow us to add localized resources for additional languages, including our images.

Starting with strings, do the following steps:

12. Create a *strings* folder in the root of your app project.
13. Within that folder, create a subfolder that matches the BCP-47 language tag specified as the default language in the manifest (for example, *en-US*, *fr-FR*, *ja-JP*, or just the base language like *en* or *ru*).
14. Move your resources.resjson file into that folder.

If you run your app again at this point, you should see that everything still works. If you go back to the How to name resources using qualifiers topic we mentioned earlier, you'll see that the resource loader is perfectly happy when you use qualifiers like a BCP-47 name as a folder name. It basically parses entire folder names looking for qualifiers, so you can create deep hierarchies to sort your resources however you like. That is, you can sort by contrast or scale first, if desired, and include language suffixes in the filename (where the format is *lang-<BCP-47 tag>*). What's more, you can create secondary .resjson files in these folders as well and play some other tricks. See "Sidebar: Secondary String Resource Files" for details.

Anyway, what you've just done by moving your resources into a folder for your default language is set your *fallback* language resources—this is what the resource loader will turn to if it cannot find a more specific match for the user's current language. Finding a match is actually a sophisticated process wherein the resource loader measures a kind of "distance" between the user's preferences and the available resources and chooses whichever is closest. This makes it possible to select en-GB as a closer match to en-AU than en-US, for example. Generally, though, it means that the resource loaded will search for a specific match like de-DE (German-Germany) first, then search for the next closest language using the base qualifier *de,* and then eventually fall back to your default language (if there are no resources for the user's other languages). The short of it is that you should always make sure the language identified in your manifest is fully populated with your full set of resources! Then even if you don't happen to localize some of those resources (say, for exact cultural alignment with images), one will still be found. For the complete story on this subject, refer to Language Matching in the documentation.

Sidebar: Secondary String Resource Files

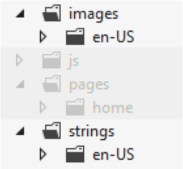
Both WinRT and WinJS are able to work with secondary string resource (.resjson) files, allowing you to organize your strings in multiple files, if desired. For example, it's common to separate error strings into a file called errors.resjson. When referencing a string identifier in one of these secondary files, all that's needed is that you use the syntax */<file>/<identifier>* instead of just *<identifier>*. This syntax works in HTML, JavaScript, and the app manifest. See scenario 5 of the Application resources and localization sample for an example.

Something else you can do with .resjson files is name them with other qualifiers for contrast, scale, home region, and so forth and even organize those files under any old folder. This is demonstrated in scenario 13 of the same sample, where it has many different .resjson files underneath *strings/scenario13*, each of which is named as *scenario13.<qualifiers>.resjson*. Because the folder name itself doesn't use a standard qualifier, you have to do a little more work to get at everything, using the `Windows.ApplicationModel.Resources.Core.ResourceManager` API, but it can be done if you're a serious resource junkie!

With images, we've already seen that if you have something like an *images* folder and you place files like logo.contrast-high_scale-140.png there, you can just refer to that file with a nonqualified relative URI like /images/logo.png and the resource loader will find them.

To prepare for localization, we only need to move those images into a folder for our default language, as we did with strings. Because you're already using relative URIs to refer to your images (with or without `ms-appx:///`), you can use whatever folder path you want as your image root. There, create a folder with the appropriate BCP-47 tag and move all your default language images into that one. In Here My Am!, for example, images live in the *images* folder, so all I need to do is create an *en-US* folder under there, move all the images, and all my references such as */images/tile.png* will still work. And because they now live in a folder that corresponds to the app's default language, they become the fallback images.

You can place images in as many folders as you like, provided that they each have language-specific folders therein. It's probably most convenient, however, to use a single root folder, or just a few; with Here My Am!, at the end of this step I had just two language folders in the project:



In your own app, then, look for app image references throughout your project. In HTML, look especially for `img` elements. In CSS, look especially for `background-image` styles. In the manifest, look at the Visual Assets tab for logos and badges, along with the Declarations tab for logos that are associated with file type/URI associations, for instance. In JavaScript, finally, check any URIs you might be assigning to element properties or CSS styles, as well as any you might be referring to in the XML for tiles, badges, and toasts.

After that, evaluate each graphic to determine whether it will require localization, including those that need to be reversed generally for right to left languages (for these you can use single copies for all RTL languages, named with the *layoutdir* qualifier; refer back to How to name resources using qualifiers). For images that don't require localization (perhaps your tile and other logos, along with plain graphic elements you use in your layout), keep them in your fallback language folder. These will be used if no other match to the current set of qualifiers is found (language, scale, contrast, etc.) Depend on the fallbacks only if you have no other variants.

With that, we're now ready to localize!

Sidebar: Managing Overall Package Size

The potential multiplicity of images with (scale variants * contrast variants * language variants) and potentially others (like direction) is an important consideration for your app package: more images will make the package size larger. Fortunately, this mainly affects your upload package to the Windows Store: provided that you've structured your resources as we've been discussing, the Store and the Windows deployment manager automatically minimizes the size of the download when a user installs the app. The system will download only those resources that the user needs based on their language, device characteristics, and other settings. (You can control this from Visual Studio, and we'll look at additional aspects of packaging in Chapter 20.)

This works even if you add new language resources at a later time (or update a set of resources of a language), *provided* that you don't update any other code files (.js, .html, and .css) and don't update the package's version number. That is, making a non-esource change will update the app for all users across languages, rather than just making a new set of resources available.

Irrespective of this automatic download optimization, it's worth carefully evaluating exactly which images really need to vary with these different factors, especially the larger ones, and to optimize the degree of compression for all of your images to minimize the package size. Ask especially whether your splash screen image—typically one of the largest, especially at the 180% scale—needs to be localized at all, and even test whether the 180% image will look good when scaled down to 140%, 100%. If your splash screen, in other words, is just imagery with a sufficient contrast ratio and you have used a universally acceptable app name, one file might work everywhere.

Note also that you don't want to provide an unqualified resource if you provide any other specific variants, because a scale variant will always be matched before the unqualified one. As a result, the unqualified resource will just take up space in the app package but will never be used.

Sidebar: The Application Resources and Localization Sample

The Application resources and localization sample shows many different scenarios for managing and referring to localized resources. It's worth spending some time with this sample because it will reinforce much of what we've discussed here: image resources (scenario 1); string resources in HTML, JavaScript, and the manifest (scenarios 2–4); using secondary resource files (scenario 5); sending language info to web services (scenario 7); combining resources and data binding (scenario 8); using resources with hyphenated attributes (scenario 9); triggering and detecting language changes (scenarios 10 and 6); overriding the default language context (scenario 11); using WinJS for resource lookup in the web context (scenario 12); and multidimensional fallback (scenario 13).

Part 3: Creating a Project for Nonpackaged Resources

Everything we've talked about so far has to do with the resources that are included in the app package that you'll eventually upload to the Windows Store. All of this is well and good, but I want to alert you now to another set of resources that you'll need as part of the Store onboarding process: localized text for your app's Store page, localized promotional graphics and screenshots, localized search keywords, and localized names for in-app purchases. All of these are essential for presenting your app well in local markets, especially the keywords.

The reason why I'm telling you about these now is that you won't be asked for them until very late in the onboarding process, namely *after* you've uploaded the app package. Early on, when you first set up an app, you can reserve multiple app names and also indicate the markets where you'd like to make the app available. These choices generally guide your localization efforts, of course. But unless you know that the Store is going to ask for a bunch of *other* resources later on, you'll find yourself desperate to create them at the last minute. If you're unprepared, then, it will mean going through another round of (rushed) translation and additional expense. It's much more efficient, then, to include your Store information with everything else you need translated.

As a preview to what we'll encounter in Chapter 20, here's what the Store dashboard will ask for on its Description page but only *after* you've uploaded your app package (in the order of appearance):

- The full text for your product page in the Windows Store (up to 10,000 characters).
- Short descriptions of your app's key features (up to twenty 200-character strings).
- At least one but as many as nine screenshots and representational graphics.
- Captions for each screenshot and graphic (200 characters each).
- Notes that are visible to customers, such as release notes.
- Short descriptions of required hardware configurations (up to eleven 200-character strings).
- Keywords (45-characters each) used to match customer searches in the Store. These are not visible to customers directly but very much affect app visibility.
- Copyright and trademark information (200 characters).
- Additional license terms (10,000 characters).
- Four promotional images that affect the chances of your app being featured.
- Your website URI.
- Your support contact information (email address or URI).
- URI for your privacy policy, which would ideally be available in multiple languages.
- Descriptions (100 characters each) for every one of your in-app purchases.

Clearly, there is quite a lot of localizable content here, none of which you want to leave until the very end of your project!

If nothing else, localize your search keywords Although it's entirely allowable to make a non-localized app available in any number of markets, the chances that consumers in those countries actually finding your app via search is next to *nil* because they'll be searching the Store using terms in their own language. If you submit an app with English keywords like "weather," it simply won't be found by users searching for *Wetter* in German, *Погода* in Russian, *Cuaca* in Indonesian. At the very least, then, make a definitive list of keywords and get them translated for each of your target markets.

Once you've collected all your localizable resources for the Store, you'll want to again structure them in such a way that makes it easy to involve translators. You could include everything as part of your app project in Visual Studio, but that would just make your package bigger with no benefit to your customers. Instead, *create a separate project* expressly for the sake of managing these Store resources. This way you can use all the tools that we'll be talking about next in "Creating Localized Resources" and have them all ready to go at the same time you build your app package for upload. Trust me: doing so will save you much anxiety when you're ready to release you app!

Creating Localized Resources: The Multilingual App Toolkit

Congratulations! With all the work you've done earlier in this chapter, you should have an app that's completely ready to be localized. The process here is really just one of acquiring translated strings for your various resjson files and translated copies of any necessary images—including all those in your separate project for Windows Store resources.

First tip If you have images that contain text, make sure that you have strings in your resources that match the image content, as you'd use for `img alt` attributes and `aria-label` attributes. In doing so, you'll obtain the necessary translations for the graphics in the process of localizing the strings.

Second tip When doing business with your app in different regions, be attentive to the region's privacy requirements and make sure your privacy statement contains region-specific sections or that you provide fully localized privacy statements.

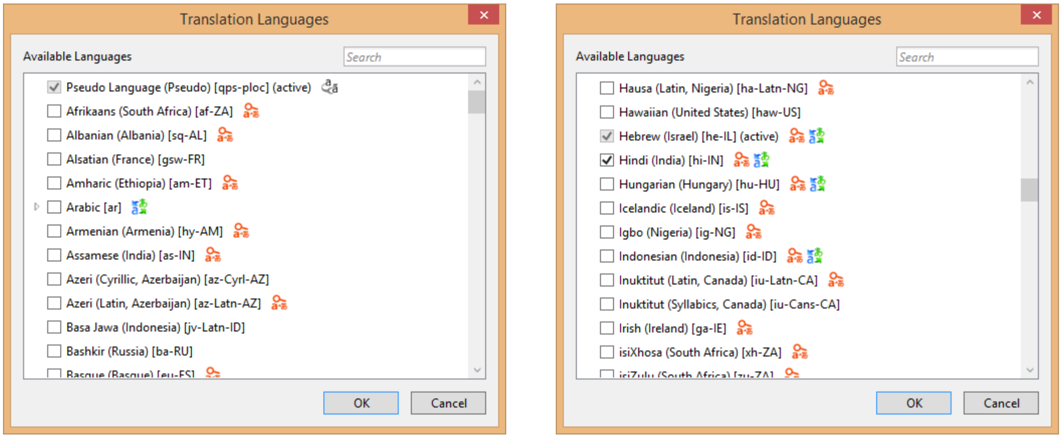
If you like, you can just send your resjson files, along with the text inside images, to an appropriate translator or translation agency and have them do the work. When you get them back, simply create additional BCP-47 folders in your *strings* and *images*, drop in those files, and away you go. You'll see such structures in the Application resources and localization sample, as we've referred to before.

Such manual translation can take a long time, however, and can become expensive. This is partly because professional translators don't necessarily have tools to work with resjson files other than a text editor. What they do have—namely sophisticated tools that help them track the status of translation jobs and much more—work with an industry-standard XML format known as XLIFF (XML Language Files). So it behooves us (and our cash flow!) to make life as easy as we can for translators, even reducing their job to reviewing suggested translations rather than generating everything from scratch.

Let me point to a couple of highly valuable resources. First is the Microsoft Language Portal, http://www.microsoft.com/Language, where you have access to very large translations dictionaries of terms as they've appeared in the UI of Microsoft products. These terms are also available as downloadable dictionaries on MSDN if you have a subscription. Various other resources (like www.bing.com/translator) also exist for general translation, and the Microsoft Manual of Style also has a chapter on localization that you might find helpful.

Beyond all these, a very helpful tool that provides automated assistance using these same dictionaries is the free Multilingual App Toolkit. Once you've downloaded and installed the Toolkit, load your project into Visual Studio and select the Tools > Enable Multilingual App Toolkit menu item. You have to do this for each project separately, because what happens from here on is that the Toolkit will be generating multilingual resources for your app—in the resources.pri file and separate .xlf (XLIFF) files—without you having to actually add any more .resjson files. Ultimately, then, you'll have only a single resjson file in your project but a number of other files generated by the Toolkit.

Once you've enabled the Toolkit, a command appears on the Project menu called Add Translation Languages. This brings up the Translation Languages dialog, as in Figure 19-6, in which you select desired target languages. At the top of the list, the Pseudo Language option (qps-ploc) will be automatically checked; we'll be using it later to test localization. This is something you typically want to do before doing specific localizations. Also note that many languages sport a "Microsoft Translator" logo, which means they can be mostly translated automatically, saving paid translators much time and saving you much money.



- ☑ Pseudo Language (Pseudo) [qps-ploc] (active)
- ☐ Afrikaans (South Africa) [af-ZA]
- ☐ Albanian (Albania) [sq-AL]
- ☐ Alsatian (France) [gsw-FR]
- ☐ Amharic (Ethiopia) [am-ET]
- ☐ Arabic [ar]
- ☐ Armenian (Armenia) [hy-AM]
- ☐ Assamese (India) [as-IN]
- ☐ Azeri (Cyrillic, Azerbaijan) [az-Cyrl-AZ]
- ☐ Azeri (Latin, Azerbaijan) [az-Latn-AZ]
- ☐ Basa Jawa (Indonesia) [jv-Latn-ID]
- ☐ Bashkir (Russia) [ba-RU]
- ☐ Basque (Basque) [eu-ES]

- ☐ Hausa (Latin, Nigeria) [ha-Latn-NG]
- ☐ Hawaiian (United States) [haw-US]
- ☑ Hebrew (Israel) [he-IL] (active)
- ☑ Hindi (India) [hi-IN]
- ☐ Hungarian (Hungary) [hu-HU]
- ☐ Icelandic (Iceland) [is-IS]
- ☐ Igbo (Nigeria) [ig-NG]
- ☐ Indonesian (Indonesia) [id-ID]
- ☐ Inuktitut (Latin, Canada) [iu-Latn-CA]
- ☐ Inuktitut (Syllabics, Canada) [iu-Cans-CA]
- ☐ Irish (Ireland) [ga-IE]
- ☐ isiXhosa (South Africa) [xh-ZA]
- ☐ isiZulu (South Africa) [zu-ZA]

FIGURE 19-6 The Multilingual App Toolkit's dialog for language selection. At left we see the option for Pseudo Language, an artificial language with lots of funky characters that represents the needs of most other languages.

Videos! For a video series on the Toolkit from the team who created it, see the following:

Introduction to the Multilingual App Toolkit (3m 50s)

Build Multi-language apps using the Multilingual App Toolkit (9m 01s), covers creating string resources as we've already discussed.

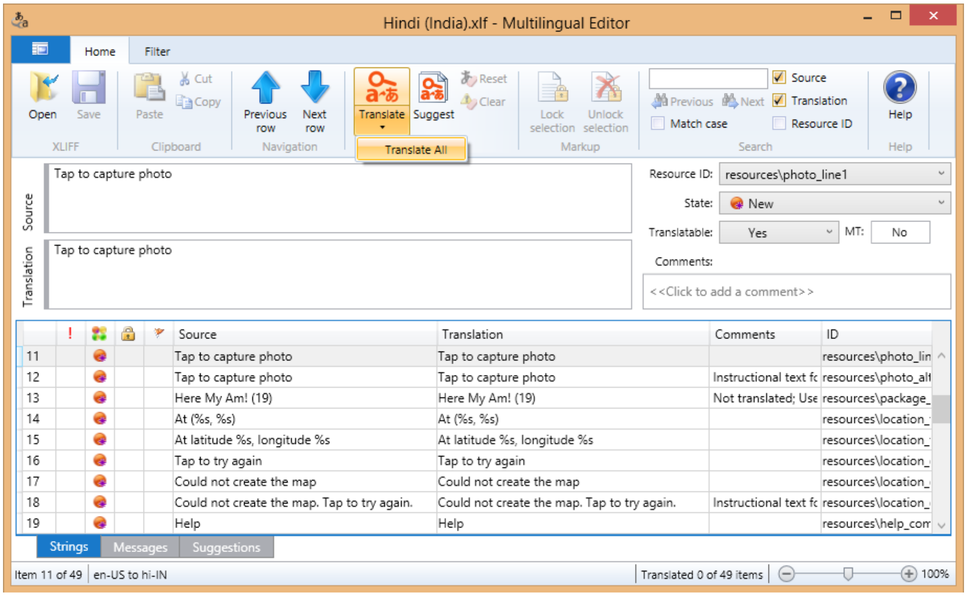
Test Multi-language apps using the Multilingual App Toolkit (5m 36s), covers what we'll talk about in "Testing with the Pseudo Language" later on.

Localize Multi-language apps using the Multilingual App Toolkit (6m 40s) demonstrates the Multilingual App Toolkit Editor as we'll see shortly.

Submitting your localized app to the Store (9m, 05s) highlights considerations for getting your app to the right markets.

Once you've made your selections (you can add more later), press OK and the Toolkit will create a folder in your project called MultilingualResources stocked with a bunch of XLF files (the ones the translators like). At first these will be mostly empty, but now here's why it was worth the effort to build up your default resources.resjson file. Right-click your project in Solution Explorer and select Build or Rebuild, or use the Build > [Re]build Solution menu item. This will go through your string resources (including any localized variants you might have created already) and populate all the XLF files with all your strings. The process will also draw in references to nonlogo images (that is, tiles and splash screens are omitted) that might also need translation.

Now for the real fun: double-click an XLF file to launch the Multilingual App Toolkit Editor, shown in Figure 19-7. Here you can manage which resources can or should be translated along with the state of the translation. If the language is also supported by Microsoft Translator, the Translate button at the top will be enabled to translate a single entry as well as to Translate All. Select the latter and sit back to enjoy the show. In a few moments you'll see that the tool has translated all your strings, marking each with the state of Review, as shown in Figure 19-8.



☑ Source
☑ Translation
□ Resource ID

Source: Tap to capture photo

Translation: Tap to capture photo

Resource ID: resources\photo_line1

State: New

Translatable: Yes

MT: No

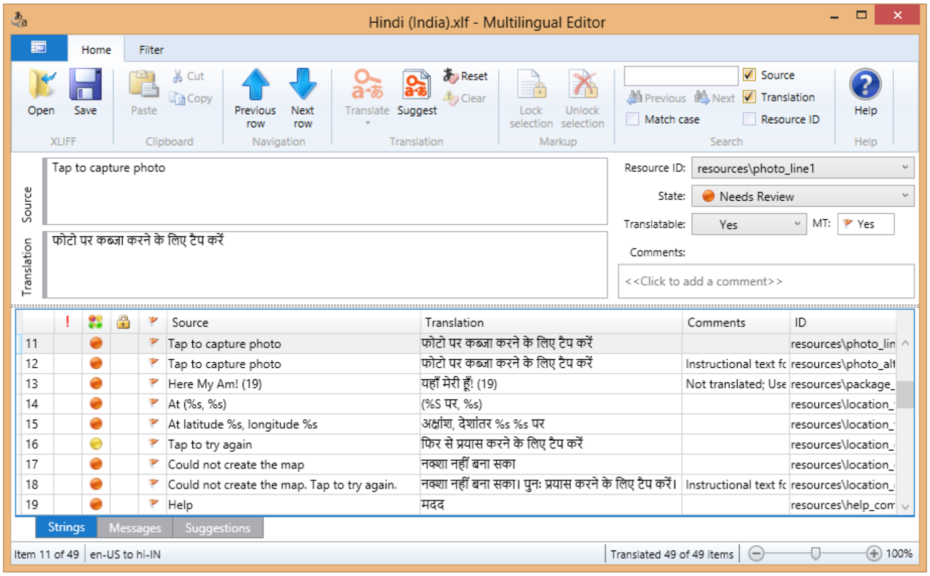
Comments:

<<Click to add a comment>>

	!				Source	Translation	Comments	ID
11		●			Tap to capture photo	Tap to capture photo		resources\photo_lin
12		●			Tap to capture photo	Tap to capture photo	Instructional text fc	resources\photo_alt
13		●			Here My Am! (19)	Here My Am! (19)	Not translated; Use	resources\package_
14		●			At (%s, %s)	At (%s, %s)		resources\location_
15		●			At latitude %s, longitude %s	At latitude %s, longitude %s		resources\location_
16		●			Tap to try again	Tap to try again		resources\location_
17		●			Could not create the map	Could not create the map		resources\location_
18		●			Could not create the map. Tap to try again.	Could not create the map. Tap to try again.	Instructional text fc	resources\location_
19		●			Help	Help		resources\help_com

Item 11 of 49 | en-US to hi-IN | Translated 0 of 49 items | 100%

FIGURE 19-7 The Multilingual App Toolkit Editor, an XLF editor with machine translation built in.



Search: ☑ Source ☑ Translation □ Resource ID □ Match case

Source: Tap to capture photo

Translation: फ़ोटो पर कब्ज़ा करने के लिए टैप करें

Resource ID: resources\photo_line1

State: Needs Review

Translatable: Yes MT: Yes

Comments: <<Click to add a comment>>

	!				Source	Translation	Comments	ID
11		●			Tap to capture photo	फ़ोटो पर कब्ज़ा करने के लिए टैप करें		resources\photo_lin
12		●			Tap to capture photo	फ़ोटो पर कब्ज़ा करने के लिए टैप करें	Instructional text fc	resources\photo_alt
13		●			Here My Am! (19)	यहाँ मेरी हूँ! (19)	Not translated; Use	resources\package_
14		●			At (%s, %s)	(%S पर, %s)		resources\location_
15		●			At latitude %s, longitude %s	अक्षांश, देशांतर %s %s पर		resources\location_
16		●			Tap to try again	फिर से प्रयास करने के लिए टैप करें		resources\location_
17		●			Could not create the map	नक्शा नहीं बना सका		resources\location_
18		●			Could not create the map. Tap to try again.	नक्शा नहीं बना सका। पुनः प्रयास करने के लिए टैप करें।	Instructional text fc	resources\location_
19		●			Help	मदद		resources\help_com

Item 11 of 49 | en-US to hi-IN | Translated 49 of 49 items | 100%

FIGURE 19-8 The string resources of Here My Am! after being machine-translated into Hindi.

Once the translation is complete, save the file and close the editor if you want. Go to PC Settings > Time and Language > Region and Language and make sure the target language is added and set as primary. Then return to Visual Studio and launch the app—and there is your first-pass localization, as shown in Figure 19-9 for Here My Am! (Notice that I elected to not translate the title, something that my translators suggested I keep in English because of its uniquely incorrect grammar.) And just to show how the app works with an RTL language, Figure 19-10 shows it running in Hebrew.

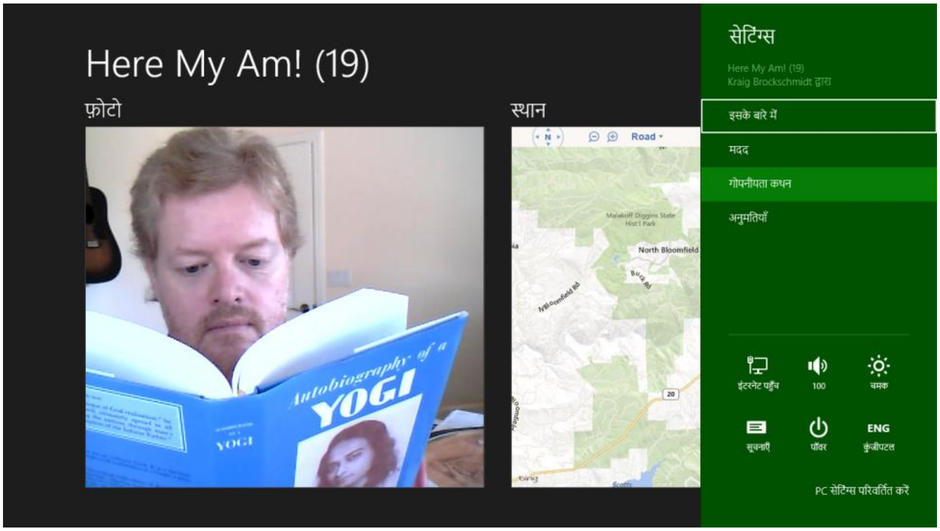


FIGURE 19-9 Here My Am! running in Hindi using machine translation output, which should still be checked by a native speaker, of course. As you can see, I'm checking if Yogananda has any advice apropos to the language.

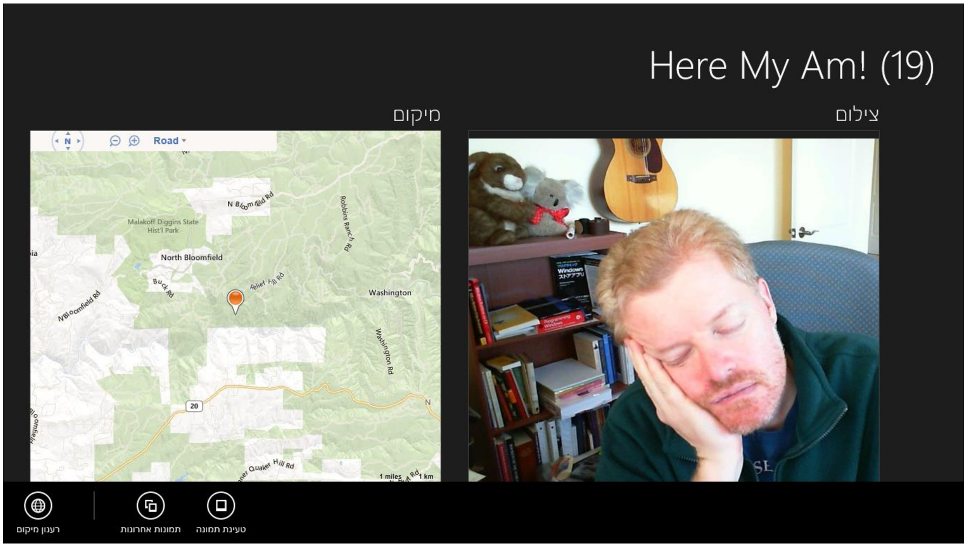


FIGURE 19-10 Here My Am! running in Hebrew (with human translation); notice how the direction of the whole experience changed, including the app bar, without having to write any code, and the nontranslated title still renders left to right. (Oh my, did I nod off while the camera timer was running?)

If you like living on the edge and don't mind shipping an app that people in other markets might laugh at or otherwise criticize for your carelessness, there's nothing stopping you from making your app available to those markets in the Windows Store with such machine translations. If you like positive ratings and reviews, on the other hand, it's a good idea to at least find some native speaker who can validate and correct what the automatic translation process suggested. You can have this helpful person use the Toolkit editor to review your XLF files, in fact. When those files are reviewed and returned to you, import them back into your project by right-clicking the existing XLF file in Visual Studio and selecting Import Translation. The new translations will then be included in your next build.

When working with professional translators, you can also select specific XLIFF Translation file formats by right-clicking the XLF file in Visual Studio and selecting Send For Translation. The dialog here gives you an option to create a ZIP file with multiple XLF files together.

Three other notes about this process. First, some strings or parts of strings won't require translation. In the Toolkit editor you can set the Translatable option to No for whole strings to prevent the machine translation from changing that string. For parts of a string, those will be translated but you can edit them back to their original and make a note in the Comments area for your translators.

Second, the Toolkit will detect if you've already made translations in an XLF file such that running a Build/Rebuild will not overwrite those strings. At the same time it will import any new strings you've added to your resource file in the meantime and remove any that have been deleted. A change in a resource identifier, however, is treated as a delete+add, meaning that the translation will be lost.

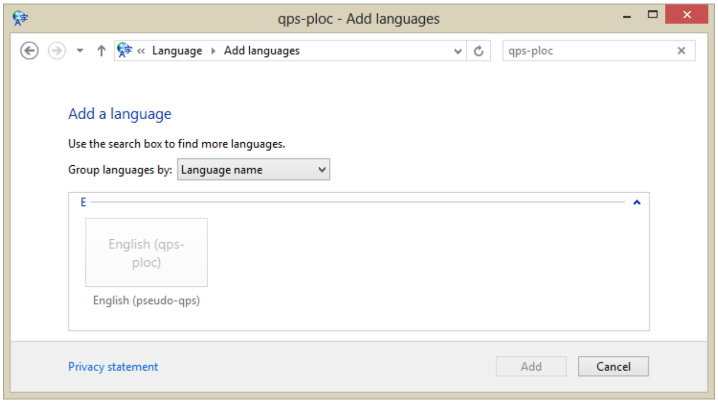
Lastly, if you want to remove a language, just right-click the XLF file and select Exclude From Project. This will keep the language out of the build while preserving the file (and its translations) in your project folder.

Testing with the Pseudo Language

As much fun as it is to produce many translations for your app, there is still the matter of testing it well, a task that is clearly overwhelming if you're targeting many languages! To reduce this burden, the best approach is to test your app using the *Pseudo Language*, a step that's ideally done before incurring the cost of specific translations. It helps you validate that your app can handle a variety of languages, because the fictitious Pseudo Language contains some of the most problematic characteristics of localized text.

As previously noted, this language is automatically added to your project through the Multilingual App Toolkit's language selection dialog. This creates a *Pseudo Language (pseudo).xlf* file in your MultilingualResources folder, alongside the real translations. Next, right-click that file and select the Generate Pseudo Translations command. This will populate the XLF file with translations of your default resources where basic characters are often converted to extended characters and strings are generally expanded with extra !!!'s tacked on. So, a string like "Recent pictures" gets translated to "[62BD8][!!_Ŗęćęйť þîćťµŗêš_!!!!]" where the hexadecimal stuff in the first set of square brackets is a resource identifier that helps testers identify the exact resource that's being used. (Note that this process will "translate" every string whether you will ultimately translate those strings for real, because it's helpful for testing.)

To run the app with this translation, you need to make Pseudo Language the system default. In Control Panel > Clock, Language, and Region > Language, click Add a Language, and enter *qps-ploc* in the search box. This is the only way to make the Pseudo Language option appear:



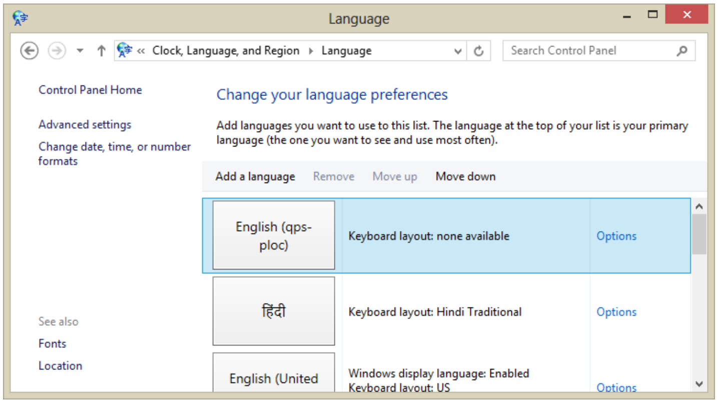
Add a language

Use the search box to find more languages.

Group languages by: Language name

English (pseudo-qps)

Select that language, click Add, and then move it to the top of the list:



Change your language preferences

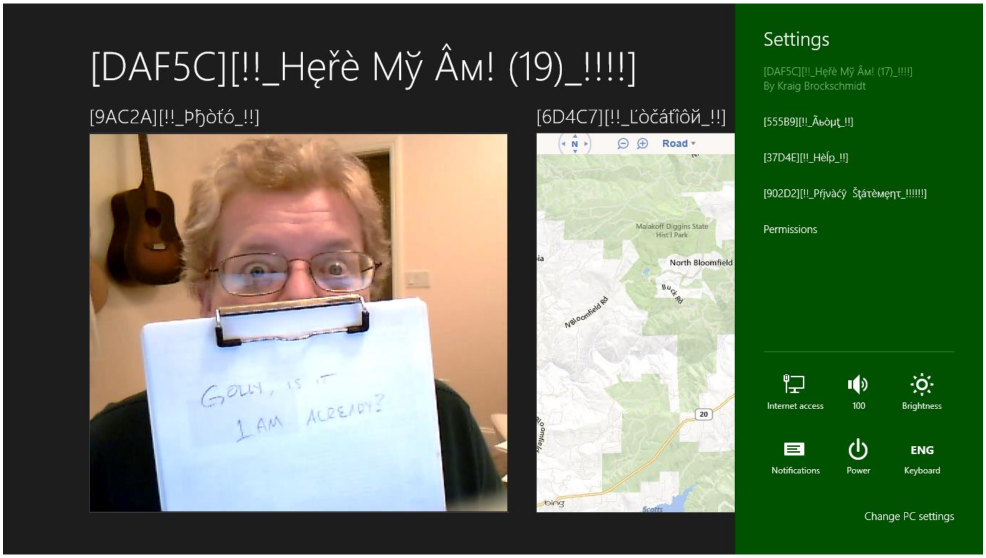
Add languages you want to use to this list. The language at the top of your list is your primary language (the one you want to see and use most often).

English (qps-ploc) — Keyboard layout: none available

हिंदी — Keyboard layout: Hindi Traditional

English (United — Windows display language: Enabled; Keyboard layout: US

When you run you app now, you should see it appear in Pseudo Language:



[DAF5C][!!_Hęřè Mÿ Âм! (19)_!!!!]

[9AC2A][!!_Þђòťó_!!]

[6D4C7][!!_Ľòčáťîôй_!!]

Settings

[DAF5C][!!_Hęřè Mÿ Âм! (17)_!!!!]
By Kraig Brockschmidt

[555B9][!!_Ãьòμţ_!!]

[37D4E][!!_Hèĺp_!!]

[902D2][!!_Pŕįvàćŷ Šţáτèмęητ_!!!!!!]

Permissions

With your app running in Pseudo Language, be sure to exercise every feature and option. Check every page in different views; check all your app bar commands; check all of your settings; check any error messages, flyouts, and message dialogs that might only appear under specific circumstances like changes in network connectivity; and test your app with all its activation paths according to the contracts you support. As you do so, look for any strings that don't appear in Pseudo Language, a clear indication that you missed pulling that string from your markup or code. Also check for truncated text, unintended word wrapping, and so forth, which reveals where your layout isn't accommodating longer translated strings.

This is the time to be as thorough as possible, because once you upload to the Store, issuing another update can take at least a few days, during which time your customers might find those problems and ding your ratings accordingly. It's always something to keep in mind, especially if you've been accustomed to instantly fixing bugs on websites: apps simply take longer, so you want to invest in testing ahead of time.

Localization Wrap-Up

Well, we're almost done with the app and ready to go to the Store! There are just a few more things to mention about localization:

- Testing with the Pseudo Language does not cover RTL language considerations; you'll need to run with those languages separately. I'm happy to say that when I ran the Hebrew translation of Here My Am!, the layout automatically mirrored, thanks to the `direction` style set in the WinJS stylesheets.

- Be sure to test any and all interactions with online services, including periodic tile/badge updates and those that arrive through push notifications.

- Test any toast notifications that you configure to appear with background transfers (see Chapter 4, "Web Content and Services," in "Completion and Error Notifications"), as well as any toasts or tile updates you issue from background tasks.

- To dynamically update your app when the user changes languages (which is a good idea instead of requiring a restart of the app), listen for the `WinJS.Resources.oncontextchanged` event and call `WinJS.Resources.processAll`. This code is in Here My Am! (default.js) as well as scenario 6 of the Application resources and localization sample:

```
WinJS.Resources.addEventListener("contextchanged", function () {
   WinJS.Resources.processAll();
});
```

- The above code will refresh string resources but not image resources or content obtained from online sources. You'll want to refresh those using additional code, such as giving Windows a new URI for periodic tile updates or indicating the new language to a service that issues push notifications. For the app's overall UI, try using `document.location = document.location + "?reload"`, picking up that URI parameter in your `activated` handler to take additional steps. This essentially mimics relaunching your app.

- If you like, you can allow the user to select the language for the app independent of the system settings. This is done by setting the `Windows.Globalization.ApplicationLanguages.primaryLanguageOverride` property, as demonstrated in scenario 10 the Application resources and localization sample. Scenario 11 also shows loading specific language resources rather than the default.

- In Visual Studio, open your manifest in the XML code editor (right-click and select View Code) and check if you see this line within the `Resources` element: `<Resource Language="x-generate" />`. If so, replace that line with individual entries like `<Resource Language="en-US" />`, where the first is your default language, and you must have localized resources for all the rest.

- For translation on the fly, you can use various web services such as Bing Translator.

What We've Just Learned

- Accessibility features are a concern for the majority of users, even those without disabilities who find those features useful at different times. Apps support accessibility by including ARIA attributes (for screen readers) in markup, implementing keyboard interaction, providing scaled raster graphics, and responding to high-contrast modes.
- Globalization is the process of removing language and cultural assumptions from an app, using globalization APIs to properly handle user language, calendars, formatting of numbers, dates, times, and currencies, sort orders, strings combining, string segmentation, varying text input methods, and configuring which web services are used from which regions.
- To prepare for localization, an app needs to be scrubbed for text and image content that will be subject to translation, separating strings into resources files and inserting references in their place, and then structuring those resources in folders and files that employ resource qualifiers.
- When onboarding an app to the Windows Store, you'll be asked for many localizable resources at the very end of the process. To have these in hand by that time, treat them like other in-app resources and include them in your translation efforts.
- For efficient localization, the Multilingual App Toolkit for Visual Studio generates and translates an app's default resources into any number of other languages, using the file formats employed by professional translators who can verify the results. It also produces a Pseudo Language translation for localization testing.

Chapter 20

Apps for Everyone, Part 2: The Windows Store

Understanding the Windows Store is really quite simple: the Windows Store is *the* point of distribution through which your apps—your feature-rich, accessible, and world-ready apps that we've been working on throughout this book—are made available to consumers around the world. Because of this, as I said at the beginning of Chapter 19, "Apps for Everyone, Part 1," to do business with apps is to do business with the Store, and your app's relationship to the Store very much reflects the nature of your business, or put more broadly, everyone's business.

That relationship affects all stages of the app lifecycle, from planning and development to distribution and servicing. Thinking about the Store is not something you want to do only when you've completed an app: you want to be thinking about it when you start first designing the apps you'd like to build—considerations like ad placement, and the fact that ad providers offer ads with specific sizes, will certainly affect your design. You also want to think about promoting the app early on too, because that can also affect design choices.

The Windows Store is like a pair of bookends to the whole app development process: you think about the Store when planning the business of your app, and, when all is said and done, you go to the Store's developer portal itself to make your app available to the customers with whom you're doing business.

You might in fact have come here directly from Chapter 1, "The Life Story of a Windows Store App," where I recommended reading "Your App, Your Business" (including its subsidiary topics) below, even before starting your first coding experiments. I suggest also reading "Releasing Your App to the World." This will help you understand what's needed when you reach the point of uploading your app, such as promotional graphics and localized text for your Store page, which you'll want to be working on well before you start the onboarding process.

It must be said, of course, that if you're working within an enterprise with line-of-business apps that will be side-loaded onto user's devices, you won't actually be working with the Store or making an effort to monetize your apps, and thus most of this chapter doesn't apply to you.

This chapter will explore a wide gamut of Store-related topics. We'll of course go through the technical side of the story—the capabilities of the Store for apps, working with the Store APIs, instrumenting your app to collect detailed telemetry, going through the process of onboarding and certification, and updating your app. We'll also talk about the business side of the story. Why? Because as soon as you publish an app to the Store, it means that you're running a business whether or not you want to admit it! You might not necessarily be in business to make money, but you're in business for some reason and so you might as well act like it. That's why we'll begin this chapter by looking at the relationship between your app and your business goals, exploring aspects of planning, monetization, and building a customer base. After a lengthy interlude of technical matters, we'll wrap up with a little more about marketing and discoverability and with some thoughts about what it means to run a business and think like a businessperson—supporting your customers, managing risk, and planning for future app releases.

For indeed, just as uploading an app to the Store is only the start of the app's real life in the world, so too is it just the beginning of your business. And although I cannot claim to be a business expert, my hope is to provide you with some inspiration and also the awareness of things you can do beyond the code to increase your success with your apps. It's a success that I certainly hope all of you, my readers, will realize more and more.

> **Before going further!** Submitting an app to the Windows Store requires a developer *account* in the Store (not a developer license in Visual Studio). It can take a day or two to set up an account (or longer if you're creating a company account), so if you haven't done this already, start the process on the Store dashboard. You must also fill out tax forms for your country of origin to receive any revenue. Once you have an account, make sure to reserve a name for your app (including localized names), which is the first step when you click Submit An App in the dashboard and can be done from Visual Studio through the Store > Reserve App Name menu.

Your App, Your Business

If you check in with your local psychologist of philosopher, they'd probably agree with the idea that just about all people, across all professions, cultures, and capabilities, are driven by a small number of fundamental motivations: fear, lust, power, love, service to others, and just plain ol' joy. Indeed, the wisest among them will even say that the last one—the quest for joy or happiness—is actually the root of all the others.

Leaving all that aside, and assuming that you're not programming under threat of death or working on apps that are going to be rejected from the Windows Store as a matter of policy, let's take a simpler view and identify the few basic motivations for writing apps:

- **Fortune** You want to make money.
- **Philanthropy** You want to contribute to and/or promote a cause.
- **Fame** You want social recognition, which also helps with Fortune and Philanthropy.**Fun** You just want to enjoy yourself through coding—an activity that, alas, nonprogrammers just don't understand!

I like to think of these motivations as another way to interpret the phrase "apps for everyone." Apps can serve the needs of many different customers around the word. So too can they serve the needs of many different developers and their business goals!

Your motivations—in whatever combination—essentially define your "business" as a developer. I use the term loosely here. In English, at least, there are about a dozen different definitions of this word, only a third of which relate to commercial activities, organizations, practices, and commerce. The other definitions have to do with what you consider to be important, as when we say "It's none of your business" or "I make it my business to know about such things." In short, apps can reflect the nature of your "business," whatever it is, and that nature is reflected in how you share apps with others.

Again, with the exception of side loading (which we'll talk about later), sharing your app means distributing it through the Windows Store. For that reason, your app's relationship to the Store effectively defines your business with that app, and that relationship spans the entire app lifecycle:

- **Planning and design** Determining whether the app can actually be a Store app, meet Store certification requirements, and be suitably monetized (if desired). Effective monetization is also a critical aspect of design—it's not something you can really do well at the end of the process!
- **Development** Implementing Store-related features and using the APIs for trial versions, in-app purchases, etc., along with incorporating other SDKs for ads and telemetry.
- **Testing** Using precertification tools prior to onboarding the app to the Store, and checking the app against certification requirements.
- **Availability** Making the app available in various markets through the Store developer portal, and indicating Accessibility support.
- **Marketing, sales, and support** Promoting your app, increasing its visibility, working with your customers (responding to ratings and reviews), linking it to your website (if applicable), and using Store analytics through the developer portal (as well as your own analytics).
- **Updates and growth** Improving your app over time, or removing it from the market.

If you look at this list—which merely reflects how the Store itself intersects with the business of creating and publishing an app—you'll notice that "development," which is where we've spent most of our time in this book, is a rather small part of the overall app lifecycle. It's essential, of course, because development is how you create the product around which you build your business. But the rest cannot be ignored because they affect how you run that business.

Indeed, by publishing an app you *are* running a business! We'll talk more about what this means at the end of this chapter, but let me summarize. To think like a businessperson means that you look for opportunities in the market where you can fulfill your business goals, whatever they are. You also see apps as a vehicle through which you explore and take advantage of those opportunities, and if one doesn't pan out like you hoped, you try again with something else. You also make sure to service the customers you do acquire, to support your apps over time, to provide new products. (Gathering telemetry from your apps is essential to this process, which we'll discuss in "Instrumenting Your App for Telemetry and Analytics.")

Truth be told, the "fortune" business is the one for which we find direct support in the Store and the WinRT APIs, because that's where the majority of developers will be focusing their energies. We'll be spending quite a bit of time in this chapter on those matters. If you're in business for "fame" or "philanthropy," on the other hand, there are some things you can consider doing to grow your customer base—all of which are also helpful if you're also trying to make money! As for "fun," well, just getting an app in the Store might be good enough for that, but if you're seeking to experience joy from the process, why not also produce apps that can bring joy to others as well?

Planning: Can the App Be a Windows Store App?

In a slight contradiction to this chapter's title, the idea of "apps for everyone" doesn't necessarily mean that every app can, in fact, be a Windows Store app. There are two sides to this: technical feasibility and meeting Store certification requirements.

Technically, as we covered in Chapter 3, "App Anatomy and Performance Fundamentals," and a few other places, Window Store apps run under certain conditions and restrictions. Here's a summary:

- Windows Store apps always run in the app container (at base trust) and have no access to APIs that can openly access the file system or any other sensitive resource (i.e., medium or full trust).
- Sharing data between Windows Store apps always goes through the Share contract, the clipboard, or web services. Local interprocess communication is not supported (except in side-loaded enterprise scenarios where the Store is not involved).
- Windows Store apps can use only the WinRT APIs and a subset of Win32 and .NET APIs; apps written in HTML and JavaScript can also use the intrinsic HTML and JavaScript APIs provided by the app host. Third-party libraries you use in the app must also use only these APIs.
- Apps cannot install custom device drivers or anything else that affects the system, nor can apps customize their install process. (Many devices, of course, do not need custom drivers and you can work with them through the APIs discussed in Chapter 17, "Devices and Printing.")
- Only certain apps can run in the background, and for specific purposes, as we've seen in Chapter 13, "Media," and Chapter 16, "Alive with Activity."
- Some UI interaction models aren't appropriate for touch input, such as high precision CAD. Because Windows Store apps must support all forms of input, high precision apps either need to be redesigned for touch or should be implemented as a desktop app.
- Windows Store apps run in variable-sized views alongside other apps and can utilize multiple views, but cannot utilize multiple overlapping windows.

If any of these technical aspects would prevent you from writing the kind of app you want to write, working with the Windows Store as your business location, if you will, is not really possible. For example, many development tools, network administration tools, file system utilities, antimalware utilities (that scan the whole hard drive), and database management systems must be implemented as desktop applications and distributed through the Internet or other retail channels.[131]

More generally, because Windows Store apps run full screen or side by side with one to three other apps, they are intended to be much more specifically focused on certain tasks. Apps that try to do too much—Swiss Army Knife apps, if you will—may end up feeling cumbersome or confusing. It's good to hone the purpose of the app, as discussed in Defining vision; otherwise you should probably implement a desktop app instead (for which there is still a very large market, mind you!).

When planning any app, be sure to review the App certification requirements for the Windows Store to understand whether the app you're thinking about will be summarily rejected during the onboarding process. For example, the very first requirement is that your app offers real value or utility for customers. So, if you're thinking to submit the next great bodily-functions-sound-effects app, you might think again. Other policies apply to advertising (section 2), predictable behavior (section 3), respecting privacy and putting the user in control (section 4), providing content that's appropriate for the global audience (section 5), and app identity (section 6).

Tip Windows Store policies change over time (they're on version 5 as of this writing), so be sure to review them before starting a new project. For the most part, policies are more often relaxed than hardened.

A final consideration is whether the Windows Store is itself available in a target market when you plan to release your app and whether there are locale-specific restrictions based on where you operate as a developer. The most up-to-date information is best found on Choosing your markets.

Planning for Monetization (or Not)

Just as there are a number of reasons why you're interested in creating apps in the first place, there are also a number of ways to fulfill your business goals. Will your app be completely free? Will it be free but supported by ads? Will it be paid, with or without a trial version? Will it involve in-app purchases? Each of these business models has its place, especially if you plan on releasing multiple apps, and many apps employ a combination of these models. Furthermore, it's likely that your business model or models will change over time as you improve your apps and creatively respond to competition. That's part of the fun of running a business!

In this section and the two that follow, we'll conceptually explore these different models and better understand how they relate to one another; the technical aspects will come later. Remember in this

[131] With developer tools, it's feasible that a Windows Store app could itself provide an interpreted runtime environment for developing apps that would always run inside that tool. A Windows Store app cannot, however, directly produce another Windows Store app because the necessary packaging and deployment APIs are only available to desktop apps.

whole context that licenses for apps and in-app purchases are granted to the user and will apply across their devices. This isn't typically a concern for apps because the details are automatically handled by the Store—in the unlikely case that a user goes over the limit, Windows will inform him or her appropriately.

Free Apps

You don't need to do anything special to create a free app so far as the Store is concerned. You write it, upload it to the Store, set the price to 0.00, have it certified, and then get the word out.

Free apps can serve several purposes:

- Earn you praise and glory from customers and possibly other developers.
- Grow a developer base who might be interested in later products.
- Give you experience producing apps (otherwise known as résumé items!).
- Provide a space for marketing your own products and/or services (as opposed to hosting third-party ads, as discussed below), so long as you do more than just show ads.

The first purpose here is self-explanatory and doesn't need any elaboration, I hope! If this is your motivation, I imagine you're already doing daily or hourly web searches on your own name and will be watching your app's ratings and reviews like a day trader watches stock tickers.

As for gaining experience, it's a great exercise, of course, but every app you make available through the Store, along with their ratings and reviews, becomes a permanent part of your developer reputation. Because of this, uploading apps to the Store before they're ready—or before you're really ready as a businessperson—could backfire over the long term. You don't want your reputation to be weighed down by early experiments when you finally have the idea that's really going to take off!

To manage this risk, you could start by sharing apps only with other developers who can side-load the packages you provide. Also consider creating a developer account just for your experimental work, keeping it separate from the account through which you'd post your real apps. This way, any negative reputation from your experiments doesn't accrue to your serious work; neither does positive reputation, of course, but that's a balance you have to find for yourself. Note that creating an extra account will require an additional annual fee, but that might be well worth it in the end.

As for marketing, what I mean here differs greatly from ad-supported apps. I'm specifically referring to promoting your own business or causes (such as a charity) through the functioning of the app where you have complete control over the content.

Be aware, however, that Section 2.1 of the present Store certification policy states, "Your app must not display only ads." This means that you can't just create an ad farm. But assuming that you do show meaningful content elsewhere, it's allowable to display ads just about anywhere else, including tiles, notifications, or on your app bar (which was a restriction until early 2014). It's also recognized that the very purpose of some apps is to provide offers, for example, in which case it's not really displaying ads, *per se*, but content for which the user has expressed interest through the act of installing the app.

The other point to these policies reflects Section 1.1 of the requirements: "Your app must offer customers unique, creative value or utility in all languages and markets that it supports." What this means is that if you want to promote a cause or business through the app, do it in a way that delivers value to consumers. For example, an app for a nonprofit organization should do more than just solicit donations (which you'd do through a consumable in-app purchase). It could help its users understand and be inspired by the organization's activities and keep them up to date on current projects, which helps inspire donations to that cause.

> **Tip** Collecting donations through an in-app purchase incurs 30% revenue sharing with the Store, so it's typically better to direct donations to a website or an alternate payment provider with less overhead.

More generally, free apps can provide some useful functions in themselves but otherwise be a demonstration of features of any number of other apps—something like a sampling tour of your paid offerings (so long as there's again real value in the app by itself). When related to only a single app, such a demo or "lite" version is usually quite different from a trial version of a full app. As we'll see shortly, a trial version should look and operate as if you had acquired a license to a full version, but it restricts its use through hobbled features or a time limit. A demo app, on the other hand, is meant more to showcase features rather than provide a complete experience.

For example, let's say you have a game with five levels in each of five distinct "worlds" through which a player would normally progress in the app's full version. A trial version would allow a player to start working through those worlds but would cease to operate completely after some short period of time, say 30 or 60 minutes. In that time, a player might not progress past the first few levels in the first world, so the experience of the overall game is incomplete. A free demo/lite version, on the other hand, could be played as much as one wished but would contain only one level from, say, three of the five worlds. This gives the user a broader taste of the app and, because it can be played many times, serves as a continual advertisement for the full experience without giving anything more away. A demo app is like a movie's teaser trailer: enough to create a hunger but not satisfy it. (And, yes, while there may be some people who only ever watch free trailers and never go see a full movie, those are a rare breed.)

Great free apps can also fit well into an overall business model without asking for anything: they can help build a positive reputation for your business, thereby supporting other paid offerings. Having multiple apps also gives you an easy way to cross-promote. Every app in the store also provides links to your website and support information, so each one is a doorway to the rest of your business. In this way, free apps are like the giveaways (or loss leaders) that many businesses offer to get you in the door so that you can explore and experience their full line of products.

Ad-Supported Apps

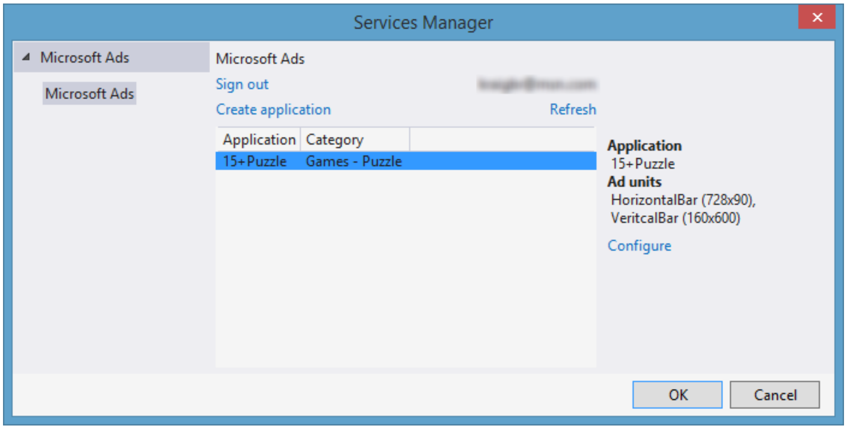
Ad-supported apps, which are typically free but can also be paid, are those that deliver some clear value in themselves and use that value to sell advertising space to others. Such advertising, although hopefully well-directed to the user's interests, typically isn't integral to the app's own function. Many free games, for instance, place *interstitial* (time-filling) ads between levels or boards, ostensibly to keep you entertained while the next level is loading but in truth to take full advantage of your captured attention in a moment of relaxation! The bottom line is that a user's attention has real value to advertisers, who are willing to pay you for a bit of that focus.

As a user of the web, you're undoubtedly familiar with how ads can appear in an app's overall layout: filling gaps in space rather than in time. Typically, an app places an ad control in its layout and lets the control acquire and display ads from its backend (known as *impressions*) and track *click-throughs* (which typically pay more than impressions).

Either way, many developers have found that selling ad space is a profitable means of monetizing an app and building a business, but of course you have to understand whether your target audience and will respond to advertising at all. There is also a common strategy of offering a free, ad-supported app with an in-app purchase to eliminate the ads, and the trick here is to make the ads *just* annoying enough to encourage customers to get rid of them but not so annoying that those customers will just abandon your app altogether! In other words, it's very important to consciously integrate ads into the overall user experience of your app, not as an afterthought. Ads are often beautifully designed in and of themselves, so they can be a pleasing and even delightful part of an app experience.

> **Note** At present sections 2.1, 2.2, 4.2, and 5 of the Store certification requirements pertain to ads, namely that your app does more than just display ads, that ads follow content policies (appropriate for the app's age rating), and that your app respects customer preferences.

The advertising control will come from the ad provider you choose to work with. Current providers that support Windows Store apps can be found on the Windows Partner Directory by filtering on "Advertising" on the left-hand side. For the most part you'll need to download the provider's SDK separately and add the appropriate references to your project. With Microsoft Ads, Visual Studio provides a little shortcut: right-click your project in the Solution Explorer, select Add > Connected Service, and select Microsoft Ads. If you haven't created an account with the Microsoft Ads, you can sign in here and then create an app configuration:



[redacted]

Application	Category
15+Puzzle	Games - Puzzle

Application
15+Puzzle
Ad units
HorizontalBar (728x90),
VeritcalBar (160x600)

Clicking OK will bring the Ads SDK into your project, which automatically supplies a Settings command through which the user can control preferences. Including the SDK also create a file called AdSamplePage.html.txt that includes sample markup according to the sizes you selected in the configuration:

```
<div id="MyAd_1" style="position: absolute; top: 50px; left: 0px; width: 728px; height: 90px; z-
index: 1"
    data-win-control="MicrosoftNSJS.Advertising.AdControl"
    data-win-options="{applicationId: 'b4d8cf08-d9b5-4148-a012-2b412bf1d70e',
        adUnitId: '163139'}">
</div>
<div id="MyAd_2" style="position: absolute; top: 140px; left: 0px; width: 160px; height: 600px;
z-index: 1"
    data-win-control="MicrosoftNSJS.Advertising.AdControl"
    data-win-options="{applicationId: 'b4d8cf08-d9b5-4148-a012-2b412bf1d70e',
        adUnitId: '163140'}">
</div>
```

Note that you'll have your own *applicationId* value—otherwise you'll be sending your revenue to my app instead! Not that I'd mind, but I'd rather you get the earnings you deserve.

Having now incorporated ad SDKs into your project and ad controls into your design, the most important thing for an ad-supported app is achieving a high *fill rate*, which is the overall percentage of time that the space to which you've dedicated ads is actually showing an ad. You see, for ads to appear in an ad control, they have to be served up by the provider's backend. Those ads get into that backend service because some advertiser has paid for that privilege. Ad providers, then, are constantly trying to stock their supply to meet demand. If, however, their salespeople aren't 100% successful in this, the ad control can come up blank, giving you a less than 100% fill rate.

Here are some strategies to avoid this:

- Always stay up to date with your provider's SDK in case there are backend changes.

- Use the recommended sizes proscribed by your provider, because using arbitrary sizes might not get any fill. For the Microsoft Ads SDK, the sizes are 250x250, 300x350, 160x600, 728x90, and 300x600. Your choices here clearly dictate how you use ads in your app's layout!
- Be mindful of your app's age rating because that information is communicated to the ad control's backend to determine what kinds of ads can be shown. In this case, a lower age rating means greater restriction. Indeed, a 3+ age rating will typically not allow any ads; some ads work at 7+, and most will work for 12+.
- Sign up with multiple ad providers and use an ad *rotator* to draw from those providers in a prioritized cascade. If your primary provider doesn't fill the ad space, the rotator attempts to retrieve an ad from the secondary provider. If that one is empty, it keeps going down the line until it gets an ad.
- In the hopefully rare case when you don't get ads from any provider, always have some default images to show in your ad space. This is why it's helpful to have other apps of your own in the Store that you can cross-promote!

Much more could be said on effective strategies for ad-supported apps, but we have other topics yet to explore. For more information, see the Monetization through Ads post on the Windows developer blog and continue to watch that blog for more monetization strategies in general.

Paid Apps and Trial Versions

Producing an app and charging for a license is certainly the one of the oldest means of monetizing, and it still works quite well, especially in certain Store categories such as Productivity and Reference. Value received for value delivered: that's the simple equation on which many successful products are built. Generally speaking, paid apps are free of advertising and are not advertisements themselves, hence customers' willingness to pay money for the apps in the first place.[132]

Tip For a broad discussion about price points, see You're Pricing It Wrong: Software Pricing Demystified (Smashing Magazine).

[132] We may eventually see creative ad insertion into paid apps: after all, you pay for issues of a magazine and yet that magazine contains ads (unless you pay for premium magazines that contain none). Think how we once balked at the idea of advertising on cable television or in movie theaters, but all that's just a matter-of-course now. The simple truth is that wherever there is a focus of customer attention, there is a value to advertisers and to the businesses that can sell them access to that attention. You just have to be careful not to abuse your customers!

An important consideration for paid apps especially (but really for all apps) is the need for marketing. The Windows Store is primarily a distribution mechanism and doesn't eliminate the need for finding your customers and making them aware of your product. Sure, customers can find your app through casual browsing, and there's a chance that your app (if it's really good) can be spotlighted, but you can't depend on that, nor can you depend on customers just finding the app through search. As a result, you have to generate interest and awareness through other means. This is again one of the functions of other free apps or demo versions that you might produce: if one of your free apps gets featured in some category, every user who downloads that free app at least has an opportunity to learn about your other products. And of course there are all the other means to market your product: the social web, your website and SEO, advertising in traditional media, and so forth.

You should also strongly consider offering a trial version of the paid app. Store data shows that customers are something like 12 times more likely to convert to a paid app if they can try it out first. Also, 70% of the most popular paid apps and 95% of the top grossing paid apps have trials. Trials give users a way to get to know the app and understand its value, both of which are important in making a purchase decision.

A trial is free and subject to a time period that is clearly shown to customers in the Windows Store. As noted before, a trial app looks, feels, and operates like the real thing but is simply time-limited or feature-hobbled. A picture editor might allow you to edit but not save your work, meaning that you get the full experience of using the product without the full benefits of owning a license. A video converter app might place a logo or watermark (that is, an advertisement) on the output video, so the functionality is all there, but the result isn't as useful. A trial version might also just disable in-app purchases, thereby limiting its extensibility until the full app is acquired.

Whether the trial is hobbled is your choice as a developer—if an app creates something and saves it in a particular format, such that you could not re-open those files without the same app (unlike pictures), there may be no reason to disable a save feature at all. In such cases, the strategy is to get the trial user heavily invested in continued use of the app, such that purchasing the full license is a better choice than letting go of that investment. Personal finance and contact management are good examples: in a 30-day trial period (or whatever period the app sets), users of such apps could amass quite a bit of useful data that they would not want to re-enter into another app. (Such a trial might also quietly disable any exporting features.)

If you implement a trial version, *be sure to remind the user of their trial status and make it super-easy for them to convert to a paid version*. That is, keep them well aware of the fact that they're just borrowing your app for the time being, make sure they know when, exactly, that loan period will run out, and make it easy to convert with a single tap or click. (Personally, I've been amazed at how many apps don't do any of this with their trials, which essentially defeats the purpose of a trial!)

As we'll see, the APIs for working with the Windows Store make it simple to check for trial status and its pending expiration. APIs also exist to initiate a streamlined purchase flow through which the user can acquire a full license with minimal disruption, all within the context of the app itself. In short, trial versions are an important monetization model that are, fortunately, quite easy to implement.

A technical stipulation of a trial version is that all the bits of the full version are actually already present on the user's machine: purchasing a full license from a trial version is simply an act of setting the license information in the Windows Store, and such a purchase will not initiate any new downloading. For a user, this means that to download and install a trial is to download and install the full version, with the Store simply indicating that the user doesn't have full rights. If such a full download would be an obstacle, however, such as when the app is large, it may be a better strategy to create a smaller demo version that invites the user to visit the Store to buy the full app.

All of this is really about creating a smooth and painless experience for users to try new software. One of the primary motivations behind the Store (and the associated packaging technology) is to eliminate nearly all of the past risk of software acquisition: unknown or untrusted sources, potential malware, inconsistent install/uninstall procedures, and so forth. Microsoft wants Windows users to feel confident that they can experiment and try out new apps—*your apps!*—without corrupting their system, compromising their data, or in other ways being exposed to those sorts of problems.

Sidebar: Piracy Protection

The existence of the Windows Store and the fact that users cannot install an app except in the context of the Store provides a certain inherent level of piracy protection. Users are blocked from accessing the folders that contain installed appx packages, and even if they managed to extract and install one elsewhere, the Store would report that the app is unlicensed for that user and would thus refuse to run it.

Beyond that, any additional levels of protection are up to the app. It's perfectly allowable for an app to ask the user to register with the publisher (because customer information isn't shared from the Store) and to obtain a secondary license key. Windows does not block such procedures but doesn't provide any such services itself. Do consider, however, that customers might be annoyed by such additional requirements. It's best to exercise caution in such a decision.

In-App Purchases

In-app purchases, commonly referred to as IAPs, are proving to be the most popular means of monetizing an application over time by selling incremental add-ons, options, in-app currency, periodicals, time-limited subscriptions/rentals, and so forth. In-app purchases are the basis for what are called *freemium* apps: the basic app is free, but extending its functionality, adding content, acquiring in-app features or game currency, and so on are all done through in-app purchases. Indeed, a Distmo report from February 2014 showed that 70% of revenue globally and over 90% in China and Japan come through freemium apps and that they generally produce more revenue per app than other monetization models. (It also showed that regions like Japan, South Korea, and Australia have the highest profit margins per download, suggesting that those are good regions in which to focus localization efforts. It's worthwhile to watch these kinds of reports!)

By definition, in-app purchases are *options*: the core operation of the app must not depend on any of them. Also, in-app purchases cannot be interdependent—that is, users cannot be required to purchase other options to use one they've already bought. Furthermore, an in-app purchase is made by a user and is thus licensed across all of their devices by default. If you need for whatever reason to place a limit on the applicability of a purchase (such as honoring contractual obligations of a backend service provider), you can use the Store's receipt feature to implement secondary validation as we'll cover later on.

There are two main classes of in-app purchases. *Durable* purchases are those that the user acquires once and that remain in effect permanently. This is true even if the user uninstalls the app on all of their devices and later reinstalls it: just as a user's purchase of a paid app is kept permanently with their Microsoft account, so too are in-app purchases. That said, a durable in-app purchase can have an expiration date (for example 30-day access to premium content), in which case the license gets reset after that time and the durable can be purchased again. Some apps, for example, use an expiring durable purchase to disable ads for a certain length of time.

The second type is a *consumable* purchase, which can be done many times because the Store doesn't record it as having been purchased (as it does with durables). Consumables are typically used for in-game currency, content rentals, or when an app separately maps a generic consumable purchase to a more specific one at run time. Because there is no Store-managed license involved, the app must track consumable purchases itself and must report fulfillment to the Store (such as a successful download from a service) before the item is made available again. This is essential for the user to trust that the purchase went through before their account is charged for the transaction.

With both types of purchases, the Store has a limit of 200 different listings, which seems like it would pose a problem for apps that offer access to large libraries or collections of things like books, movies, images, research reports, and so on. Fortunately, you can define generic purchases in the Store dashboard and then differentiate them with specific identifiers at run time through the Store API. For example, you can define a "3-day movie rental" item through the Store dashboard and fill in the specific title when the user chooses one in your UI. This also means you don't have to update your purchase definitions in the Store when you add new content to your catalog; you'd need to do that only if you introduce a new type of purchase altogether.

Whether in-app purchases are the right choice for your app involves a number of considerations:

- Implementing them well can be difficult because they introduce complexities into an app's architecture.
- The app has full responsibility for correct delivery of the purchased item or feature, as opposed to the Store handling all the details.
- In-app purchases effectively create multiple variations of an app, which can increase user support and interaction.

- Overuse or inappropriate use of in-app purchases can generate the perception that you're trying to get money from users at every possible opportunity. Users who don't or won't pay for in-app purchases can still leave bad reviews about their experience. Effective use of in-app purchases, in other words, is an act of conscious merchandising that takes user psychology into account.

- In-app purchases through the Windows Store do not trigger download of additional content; they only change the user's license for that product. If needed, an app can initiate its own downloads once the product license has been acquired.

- At present, the Windows Store does not provide support for subscription purchases apart from consumables; third-party providers do offer specific subscription services.

- An in-app purchase can activate or enable a feature in the app but cannot trigger download and installation of new code from the Store (or any other service, for that matter). That is, the code must already be present in the app package. In-app purchases of content, on the other hand, can be used to initiate a download from any service.

- At present, the lowest price tier that the Windows Store offers for in-app purchases is US$0.99, which means that whatever purchases you offer need to provide at least that much perceived value to your customers. You should group offers into a single purchase that you would otherwise think to offer separately at lower but unavailable price points. (Do note that this price tier is automatically adjusted up or down to reflect different standards of living across regions.)

On the flip side, offering a new full version of an app with new features might generate better sales than offering the same features as in-app purchases. A major app update is an event that can generate renewed interest in and energy around your product like the release of a new movie. In-app purchases, on the other hand, are by nature more prosaic, like the popcorn and drinks you buy in the theater—always there, and integral to the whole experience, but not particularly exciting outside that context. The best approach is probably to follow Hollywood's example and do both!

It's worthwhile at even the earliest stages of design to think about what kinds of in-app purchases make sense for your product and how and when to merchandise those options in your app. You might not even at this time have anything you plan to offer, but you might want to add them later on. In short, keep the door open for expansion and creativity without necessarily having to revise the app. It's equally important to also think about what makes sense for *your customers*. We emphasize this point because there have been stories of outright abuse in this area. Apps aimed at young children, for example, have been known to dangle in-app purchases like candy, enticing those children to press a "buy" button when they have no sense of the transaction. For this reason, Store policy (section 4.8) requires that transactions are authenticated (and the Windows Store does this automatically when you use its commerce engine).

The key thing is that if you try to be sleazy, you probably won't get far with your app. If you try to trick users out of their money, your app will certainly decline in ratings and reviews over time. And if you're found to be truly abusive, Microsoft does reserve the right to remove your app from the Store altogether, if it even passes certification at the outset.

Done well, though, in-app purchases are very effective for exchanging a little bit of money for a little bit of pain relief. This is a creative matter of user psychology where an app intentionally introduces small pain points alongside an in-app purchase to avoid that pain. For example, you can make users wait 15–30 seconds while the app displays an interstitial ad, or inject a visual distraction with ads in your layout, and then offer an in-app purchase to remove ads altogether. In many games, you can earn in-app currency through continued play or receive additional currency by simply waiting an hour or a day. This kind of thing becomes irritating for your more impatient users, so offering an in-app purchase that immediately relieves their anxiety can be very profitable. (In this context, I love the creative in-app purchase in Jetpack Joyride called the "counterfeit machine" that doubles all the coins you collect in the game. I've found this much more engaging that buying coins outright.)

All in all, these are considerations that will eventually affect how you set prices in the Store. You'll need to consider the tradeoffs involved between setting a higher price point with an initial-app purchase versus monetizing through multiple in-app purchases, and you'll need to be sensitive to how willing your target customers might be to making one purchase versus making multiple purchases. Apps that constantly nag their users to make additional purchases will be on par with pushy street vendors who just won't leave you alone. Apps that are sensitive to the user's engagement, on the other hand, present purchases (which is to say, upsell opportunities) at appropriate times for appropriate reasons and are thus much more likely to be appreciated than loathed.

Revenue Sharing and Custom Commerce for In-App Purchases

The subject of monetization is not complete without answering one of the most important questions: how much of the Store-related revenue stream do you, as the publisher of the app, get to keep? The basic answer is simple: 70% comes to you, 30% goes to the Store (you have to pay your rent). However, once an app achieves US$25,000 in sales (from both the app and in-app purchases), your share increases to 80%.

Revenue sharing is always in effect for paid apps. For in-app purchases, however, you have the option to bypass the Windows Store altogether and use a commerce platform of your own or a third-party provider (see the Windows Partner Directory and click Payments on the left side). Doing so potentially allows you to realize a much higher percentage of the revenue. This is an especially great option if you already have arrangements with a transaction provider through your existing websites. Be aware that Sections 4.7 through 4.9 of the present Store certification requirements apply here, where you need to identify the provider at the time of the transaction, ensure that the user enters credentials for each purchase, and ensure that each transaction meets the PCI Data Security Standards.

With this custom commerce option, you're pretty much on your own where all the details are concerned, including UI—the Windows Store API itself doesn't provide for extensibility of its own mechanisms. You might draw from The in-app purchase user experience for a customer topic in the documentation to understand the flow; third-party providers do typically supply their own UI.

Growing Your Customer Base and Other Value Exchanges

Whatever your business is with your app, it's unlikely to fulfill your goals if nobody is using it. When you're motivated primarily by fortune or philanthropy, expanding your customer base is likely your primary concern. And even if you're shooting for fortune, it might make more sense in the early stages of your app (or of your overall business) to focus first on growing your customer base and then after a time shift to monetizing that base. And by "a time" here it's generally best to think in terms of a couple of months, to catch the app's peak traffic levels.

Marketing your product to potential customers that are within reach of your campaigns is obviously a big part of this, and we'll talk more of this later in "Getting Known." What I want to discuss here is more about expanding your customer base *through* the app itself, because the strategies for this are often similar to those you use for monetization.

The general idea is that your customers have assets of value to you other than their money. The most important of these is their social network. While your general marketing efforts can reach random populations, reaching other potential customers through your existing ones is much more valuable because there's an existing relationship there that gives you some credibility.

I'm sure you've seen the effects of this on the receiving end, where friends in your social networks are posting or tweeting their status or activity in some game or another. The reason why this happens is that the game rewards your friend's social activity with some in-app value, such as extra in-game currency. Oftentimes—and this is an important point for app design—your friend can earn this currency through social sharing *instead* of spending real money.

What you're doing is sacrificing the income from a direct in-app purchase for a bit of marketing to a customer's social network, which is something you couldn't buy even if you had tons of money in your marketing budget! That is, instead of trying to make money from your existing customers that you might then spend on marketing, you just shortcut the process and reach new customers directly.

Another valuable asset is a customers' opinion of your app, meaning the time they might take to give you a rating and review in the Windows Store, as well as feedback through your own in-app mechanism. Leaving feedback, in short, is an important customer activity that you can reward with some in-app feature, rather than trying to monetize that feature. (Do note that it's not allowed to offer rewards for positive ratings and reviews, which you cannot determine at run time anyway.)

A third type of asset that your customers have is their usage data. Provided that you protect user identity, respect privacy, clearly inform your customers that you're collecting data, and give them the ability to opt out (see "Instrumenting Your App for Telemetry and Analytics" later), the data you collect over time can become highly valuable to other businesses. A number of companies earn their revenues by collecting, processing, packaging, and selling information (through reports, web services, etc.), which means that their contact points with customers—such as apps and websites—are merely collection portals rather than products they try to monetize directly.

For example, data collected from a music app can identify consumption trends that music producers might be very willing to pay for. In this case you'd want to get that app into the hands (and ears) of as many customers as possible, even to the point of giving away up-front value (such as free downloads) in an effort to increase your customer base, increase your data collection, and thereby increase the reliability and value of that data.

In short, when you're thinking like a businessperson, you're not thinking only about monetizing the app you happen to produce: you're thinking about the larger context in which your business operates and look for any and all opportunities to earn revenue. It makes you appreciate why "business" is indeed a creative profession!

Measuring and Experimenting with Revenue Performance

No matter what revenue model or models you choose to employ in your app, you'll want to know how well they're doing in relation to your business goals. The Windows Store itself will provide you with revenue tracking for paid apps as well as in-app purchases that go through the Store's commerce engine. If you're using a third-party commerce platform, it will provide you with similar information, as will your advertising providers.

Beyond such backend data, though, you'll want to instrument your app to collect its own telemetry where customer behavior is concerned (as we'll discuss later), especially if you're providing options like social sharing in lieu of an in-app purchase. You'll also want to know when and where, exactly, your customers are engaging with your monetization strategies, because your providers won't be able to make such determinations for you. For example, when and where are your customers most likely to make an in-app purchase? What ad placements within different pages of your app get the most impressions and click-throughs? Knowing such things can help you adjust the design of your app to produce more revenue and can give you valuable insights for other apps you'll produce.

Such telemetry, along with customer feedback, also enables you to experiment with your monetization and track the results, which you again use to optimize your apps *market* performance. For example, one publisher initially offered a $3.99 game with a seven-day trial but found that most users finished the game in five days. As a result, they weren't making much money. So they changed the model to a feature-differentiated trial—meaning you couldn't complete the game with just a trial license—and lowered the price to $2.49. In doing so their revenues rose. Another example is an app that started out with ads and in-app purchases to add content. Customers expressed interest through their Store reviews that they'd like to remove ads, so the app added three tiers of in-app purchases to remove ads for different periods of time.

So don't be afraid to try different things! After all, part of running a business with your app is creatively adjusting your business model to the needs of your customers and trends in the market.

The Windows Store APIs

Now that you've likely decided on a course for your app, let's see how you use the Windows Store APIs to accomplish those ends. These are found in the `Windows.ApplicationModel.Store` namespace; all objects referred to in this section are contained in this namespace unless noted.[133] Furthermore there's only one sample that we'll be drawing from: Trial app and in-app purchase sample.

For basic licensing and trial enforcement, the good news is that both are effortless: the app doesn't need to do anything at all! A user cannot acquire your app without going through the Store, and even if he did manage to, he'd have to have a developer license to install and run it. Furthermore, because the Store automatically tracks trial periods for apps, Windows will simply not launch an app once the trial is expired. Instead, Windows will redirect the user to the product's page in the Store where the user can purchase a full license. The same is true is Windows detects that a package has been tampered with: the Windows will direct the user to repair the app through the Store.

As noted before, apps can enforce a secondary licensing scheme if desired. Here it would ask the user for a separate registration or a separately acquired license key of some sort. Windows does not offer an API for this but will not block schemes of your own.

That said, WinRT provides for the following features:

- Retrieving app and product (in-app purchase) information from the Store, including price values formatted for the user's current locale.
- Retrieving license information for the app and in-app purchases, indicating trials, expirations, etc. The app can make any decisions it wants with these details.
- Prompting the user to purchase a full license during or after a trial period; this is especially useful when the app is running and the trial period expires.
- Handling in-app (product) purchases for durables and consumables, with large catalog support.
- Generating receipts, which are primarily used for secondary validation and purchase tracking.
- Testing all the app's Store interactions prior to uploading to the store.

Tip Although the Store API retrieves information from its backend service, an app is not required to declare the *Internet (Client)* capability itself unless it needs network access for other purposes.

When an app runs for real—that is, after it has been uploaded to the Store and has made its way into the hands of customers—interaction with the API happens through the static `CurrentApp` object:

[133] The `Windows.ApplicationModel.Package` class also provides a few details about the installed app package. Usage is simple, and you can refer to the App package information sample for more.

```
var currentApp = Windows.ApplicationModel.Store.CurrentApp;
```

whose methods and properties are as follows:[134]

Member	Description
appId	The GUID that uniquely identifies the app in the Store.
linkUri	The URI (Windows.Foundation.Uri) to the app's listing page in the Store. (If you recall from Chapter 15, "Contracts," this is the value you always want to place in the applicationListingUri property of a DataPackage used in the Share contract; doing so lets user who receive shared data easily find and acquire your app.)
licenseInformation	A LicenseInformation object reflecting the app's licensing state and active licenses for in-app purchases. The latter is represented by a collection of ProductLicense objects.
loadListingInformationAsync	Retrieves the ListingInformation object for the app through which you retrieve information about in-app products.
requestAppPurchaseAsync	Invokes the Store UI to invite the user to upgrade the app from a trial. This is used when the app is running and detects that a license has expired.
requestProductPurchaseAsync	Invokes the Store UI to invite the user to do an in-app purchase. A successful purchase results in a PurchaseResults object.
getAppReceiptAsync, getProductReceiptAsync	Requests an XML string that contains receipts for the app and any in-app purchases, or for a specific in-app purchase, respectively.
reportConsumableFulfillmentAsync	Reports that an in-app consumable purchase has been fulfilled (such as content being successfully downloaded from a service).
getUnfulfilledConsumabledAsync	Retrieves those consumables that are still outstanding.

A ListingInformation object contains a number of properties that come pre-localized as appropriate: ageRating (a number, currently one of 3, 7, 12, and 16), currentMarket (a BCP-47 string indicating the user's market that is used for transactions), description (a string containing the app's localized description as you provided to the Store dashboard), formattedPrice (a string containing the app's purchase price formatted for the user's current market and currency), name (a string with the app's name in the current language, as supplied to the dashboard), and productListings. The latter is an array of ProductListing objects, each of which represents an in-app purchase that you configured on the Store dashboard.

On Windows 8.1, a ProductListing contains four properties: productId (a string containing the app-defined product identifier), formattedPrice (a localized string containing the product price), a localized name (a string), and a productType value from the ProductType enumeration, which can be durable (1), consumable (2), or unknown (0). You can see, then, that the listing collection is exactly

[134] Excluding a few methods that are exclusive to Windows Phone, namely loadListingInformationBy[Keywords | ProductId]Async and reportProductFulfillment.

what you'll use to present the user with your localized list of in-app purchases, where the productId could be used to retrieve additional content like images from your package or a web service. You can also use productType to separate your lists of durable and consumable purchases.

The LicenseInformation object for its part contains simple properties of expirationDate (a Date), isActive (a Boolean), and isTrial (a Boolean). It has one event, licensechanged, which fires when these properties change. You can use this to prompt for purchase if a license expires while the app is running. The remaining property, productLicenses, is a collection of ProductLicense objects. Each contains the appropriate productId, expirationDate, isActive, and isConsumable properties. The LicenseInformation.licensechanged event also fires for changes in any ProductLicense.

Tip For globalization purposes, never compare two dates with simple arithmetic operators like <, >, and =. Instead. Use the [Windows.Globalization.Calendar.compareDateTime]() method, which will account for the specific needs of different calendar systems that might be in effect.

Roaming and offline access Because a user can make purchases on other devices, Windows makes sure to roam licenses to and cache them on all the user's devices. This allows apps to validate purchases and check license status when a device is offline. The upshot of this is that you should always use the API to check licenses rather than trying to manage such state yourself.

That's really the extent of the Store APIs in a nutshell, and we'll go into all the details in the discussions that follow. You may notice that the APIs don't concern themselves with ad-supported apps, because ads don't involve the Store itself and are implemented through ad-provider controls.

But you might be asking yourself some very significant questions: how on earth can this API return any meaningful information while the app is under development and has yet to be uploaded to the Store in the first place? How can you get product information and test all your purchase features when there's nothing yet available to purchase?

These are great questions, and the answers lie in the one other object in the Store namespace (and our next topic): the Windows Store app simulator.

Sidebar: Use the licensechanged Event for UI Updates

In general, a best practice is to use the licensechanged to update whatever UI is affected by changes to app licenses or in-app purchases. That is, this event is *the* central location where you pick up all changes, be it from purchases (which you are hoping for!) or licenses that expire (which you hope to convert before then). You have to use this event anyway to handle expirations while the app is running, and thus it's better to also use it to process the successful purchases, rather than doing it within your handlers for the request*PurchaseAsync methods.

The CurrentAppSimulator Object

To make it possible to test an app's interactions with the Store before the app is actually onboarded, WinRT provides the static CurrentAppSimulator object that is identical to CurrentApp with two exceptions. First, the simulator object works against data from a local XML file rather than live data from the Store, which won't exist until you've actually on-boarded the app. By definition, then, everything you do to test Store interactions before your app is published will use the simulator. The second difference is that the simulator object has an extra method, reloadSimulatorAsync, to reinitialize the simulator with such XML.

During development, you use this line of code to start your work with the API:

```
var currentApp = Windows.ApplicationModel.Store.CurrentAppSimulator;
```

and then delete the *Simulator* suffix when you're ready to send the app to the Store. (If you forget, you'll fail Store certification.) Alternately, you can use the method described in Chapter 2, "Quickstart," under "Sidebar: Debug or Release?" to select the right object based on your build target. Details are provided on my blog at A reliable way to differentiate Debug and Release builds for JavaScript apps and in the DebugRelease example in Chapter 2's companion content.

When your app accesses CurrentAppSimulator, WinRT looks for a file called WindowsStoreProxy.xml in your app data, specifically under *%userprofile%\AppData\local\packages\<package name>\LocalState\Microsoft\Windows Store\ApiData*. If it exists, the simulator is initialized from that data; otherwise the file is created with the following defaults (slightly formatted to fit the page):

```
<?xml version="1.0" encoding="utf-16" ?>
<CurrentApp>
  <ListingInformation>
    <App>
      <AppId>00000000-0000-0000-0000-000000000000</AppId>
      <LinkUri>
        http://apps.microsoft.com/webpdp/app/00000000-0000-0000-0000-000000000000</LinkUri>
      <CurrentMarket>en-US</CurrentMarket>
      <AgeRating>3</AgeRating>
      <MarketData xml:lang="en-us">
        <Name>AppName</Name>
        <Description>AppDescription</Description>
        <Price>1.00</Price>
        <CurrencySymbol>$</CurrencySymbol>
        <CurrencyCode>USD</CurrencyCode>
      </MarketData>
    </App>
    <Product ProductId="1" LicenseDuration="0">
      <MarketData xml:lang="en-us">
        <Name>Product1Name</Name>
        <Price>1.00</Price>
        <CurrencySymbol>$</CurrencySymbol>
        <CurrencyCode>USD</CurrencyCode>
      </MarketData>
    </Product>
  </ListingInformation>
  <LicenseInformation>
    <App>
      <IsActive>true</IsActive>
      <IsTrial>true</IsTrial>
    </App>
    <Product ProductId="1">
      <IsActive>true</IsActive>
    </Product>
  </LicenseInformation>
  <ConsumableInformation>
    <Product ProductId="2" TransactionId="00000000-0000-0000-0000-000000000000"
        Status="Active" />
  </ConsumableInformation>
</CurrentApp>
```

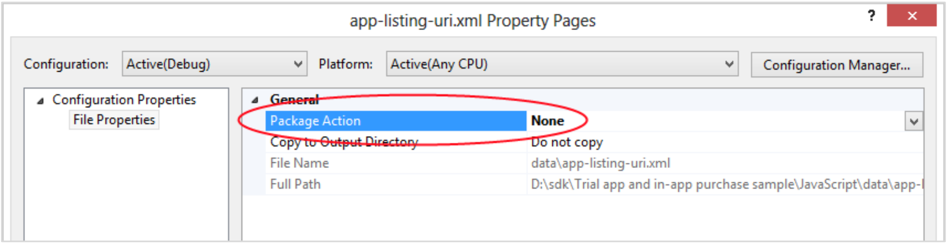
The full XML schema for this can be found on the CurrentAppSimulator page (except for the *ConsumableInformation* part, which I'll fill in below), and it's straightforward to see exactly where you'd modify the XML to test different scenarios:

- Create additional MarketData elements to specify app details for other locales. The CurrentMarket element indicates the default.
- Create additional Product elements (including their MarketData children) for each in-app purchase.
- In the App element under LicenseInformation, change the values of IsActive (that is, not expired) and IsTrial between true and false to test the variations: active/non-trial, active/trial, expired/non-trial, and expired/trial. You can also add an ExpirationDate element to indicate when the app expires (in UTC time), using the form of *yyyy-mm-ddThh:mm:ss.ssZ* (replacing yyyy:mm:dd with the date and mm:ss.ss with the time). For automated testing, additional elements let you hard-code result codes; details on the CurrentAppSimulator page.
- For each in-app purchase, add a Product element under LicenseInformation with the appropriate ProductId attribute and ProductType attribute (Consumable or Durable). Supported child elements are IsActive and ExpirationDate, with the same meaning as the app license.
- For each consumable in-app product, add a Product element under ConsumableInformation with the appropriate ProductId, an optional OfferId (for large catalogs), the TransactionId to report when you simulate a purchase, and a Status attribute. The latter can have values of Active (consumable is available), PurchasePending (waiting fulfillment), PurchaseRevoked (canceled on the backend), or ServerError. These let you test different scenarios when using the in-app purchase API for consumables.

Tip When you reach the point of publishing your app to the Store, you'll be configuring all of these options for real. By that point, your XML file that you use in development should be the exact representation of the capabilities that the app expects, so make sure that you accurately transfer all the details to the Store dashboard.

When using the methods in the simulator object that change license status, such as converting a trial app to a purchased app or acquiring in-app purchases, they will *not* alter the contents of the WindowsStoreProxy.xml file. This means you can just restart the app to reset the state of the simulator object, but it also means you'd need to edit the XML and launch the app again to test how different variations are handled on startup. (Note also that the Store simulator state is not persisted when the app is suspended and terminated.)

For this purpose, the simulator object's `reloadSimulatorAsync` method takes a `StorageFile` containing the XML initialization data. This can very much simplify your testing procedures, and often you'll have such files directly in your project folder such that you can refer to them with `ms-appx:///` URIs. However, make sure that these files don't end up in your app package when you upload to the Store. In Visual Studio, right-click the file in the Solution Explorer pane and select Properties. In the Property Pages dialog that appears, as shown in Figure 20-1, set Package Action to None.


app-listing-uri.xml Property Pages

Configuration: Active(Debug)

Platform: Active(Any CPU)

Package Action: None

Copy to Output Directory: Do not copy

File Name: data\app-listing-uri.xml

Full Path: D:\sdk\Trial app and in-app purchase sample\JavaScript\data\app-l

FIGURE 20-1 Make sure that XML configuration files for the simulator object don't end up in your Store packages.

The Trial app and in-app purchase sample, which we'll be drawing from in the explanations ahead, uses `reloadSimulatorAsync` to load a specific XML file for each of its scenarios (but note that it has not set the Package Action to None!). In scenario 7, for example (js/api-listing-uri.js), it loads data/app-listing-uri.xml as follows:

```
var currentApp = Windows.ApplicationModel.Store.CurrentAppSimulator;

var page = WinJS.UI.Pages.define("/html/app-listing-uri.html", {
    ready: function (element, options) {
        // ...
        loadAppListingUriProxyFile();  // Initialize the license proxy file
    },
    unload: function () {
        currentApp.licenseInformation.removeEventListener("licensechanged",
            appListingUriRefreshScenario);
    }
});

function loadAppListingUriProxyFile() {
    // We could also use folder.getFileFromPathAsync("ms-appx:///data/app-listing-uri.xml")
    // instead of the two-step process with getFileAsync as shown here.
    Windows.ApplicationModel.Package.current.installedLocation.getFolderAsync("data").done(
        function (folder) {
            folder.getFileAsync("app-listing-uri.xml").done(
                function (file) {
```

```
                currentApp.licenseInformation.addEventListener("licensechanged",
                    appListingUriRefreshScenario);
                Windows.ApplicationModel.Store.CurrentAppSimulator
                    .reloadSimulatorAsync(file).done();
            });
        });
}
```

Notice how this sample listens for the `licensechanged` event and makes sure to call `removeEventListener` when the page is unloaded. (See "WinRT Events and removeEventListener" in Chapter 3.)

This same scenario 7 shows the basic retrieval of app information from the Store. When you click the Show Uri button on that page, it goes to the handler below, which outputs the app's `linkUri` property:

```
function displayLink() {
    WinJS.log && WinJS.log(currentApp.linkUri.absoluteUri, "sample", "status");
}
```

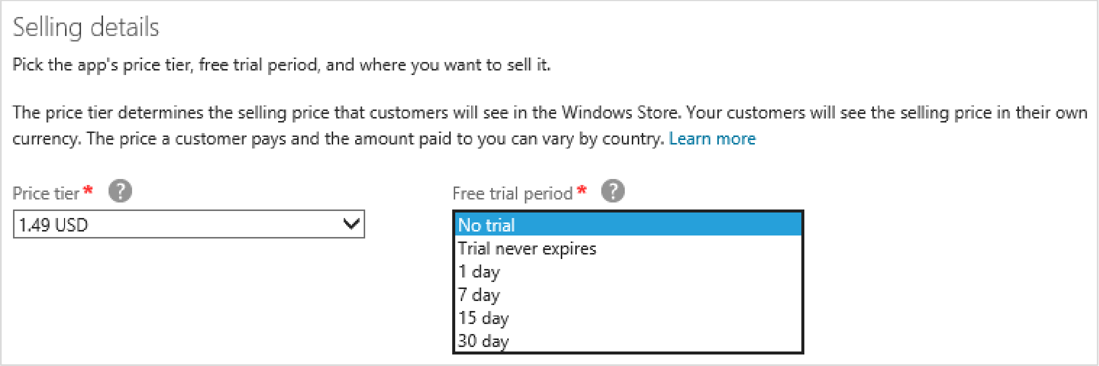
Getting at the app's other properties would look the same, just using `currentApp.loadListingInformationAsync` first to obtain that data. This is shown in scenario 1 (js/trial-mode.js):

```
function trialModeRefreshScenario() {
    currentApp.loadListingInformationAsync().done(
    function (listing) {
        document.getElementById("purchasePrice").innerText =
            "You can buy the full app for: " + listing.formattedPrice + ".";
    });

    displayCurrentLicenseMode();
}
```

And on that note, let's look at the rest of the sample more fully because it shows the other use scenarios of the Store API as a whole.

Trial Versions and App Purchase

When you configure your app on the Store dashboard, the first question under Selling Details is the Price Tier. If you set this to anything other than Free, you also set the Free Trial Period, as shown in Figure 20-2. As noted before, offering a free trial increases the likelihood of a customer making a full purchase by an order of magnitude, so it's definitely something to consider!



Selling details

Pick the app's price tier, free trial period, and where you want to sell it.

The price tier determines the selling price that customers will see in the Windows Store. Your customers will see the selling price in their own currency. The price a customer pays and the amount paid to you can vary by country. Learn more

Price tier * : 1.49 USD

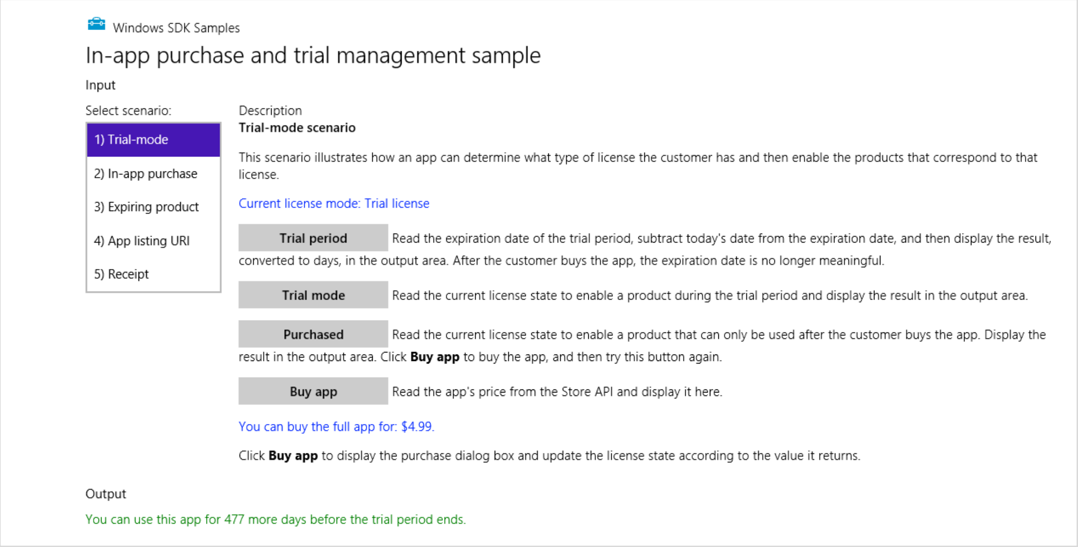
Free trial period * : No trial

FIGURE 20-2 Setting a price tier and trial period for an app on the Store dashboard.

In your XML for the `CurrentAppSimulator`, you set the app price in the ListingInformation > App > MarketData > Price element and configure a trial period under LicenseInformation > App > IsTrial and ExpirationDate. Handling a trial is demonstrated in scenario 1 of the Trial app and in-app purchase sample, and the relevant XML from data/trial-mode.xml is as follows (other parts omitted for brevity):

```
<CurrentApp>
  <ListingInformation>
    <App>
      <MarketData xml:lang="en-us">
        <Name>Trial management full license</Name>
        <Description>Sample app for demonstrating trial license management</Description>
        <Price>4.99</Price>
      </MarketData>
    </App>
  </ListingInformation>
  <LicenseInformation>
    <App>
      <IsActive>true</IsActive>
      <IsTrial>true</IsTrial>
      <ExpirationDate>2014-01-01T00:00:00.00Z</ExpirationDate>
    </App>
  </LicenseInformation>
</CurrentApp>
```

When you run this scenario in the sample, as shown in Figure 20-3, the simulator object is initialized with this XML. The app's `IsActive` and `IsTrial` elements are both set to `true`, meaning that the app can run and that it has a valid trial license. The `ExpirationDate` for this license is set to January 1, 2014, which is in the past as of this writing, so you'll need to update it to see expiration in the future rather than the past.



In-app purchase and trial management sample

Trial-mode scenario

This scenario illustrates how an app can determine what type of license the customer has and then enable the products that correspond to that license.

Current license mode: Trial license

Trial period — Read the expiration date of the trial period, subtract today's date from the expiration date, and then display the result, converted to days, in the output area. After the customer buys the app, the expiration date is no longer meaningful.

Trial mode — Read the current license state to enable a product during the trial period and display the result in the output area.

Purchased — Read the current license state to enable a product that can only be used after the customer buys the app. Display the result in the output area. Click **Buy app** to buy the app, and then try this button again.

Buy app — Read the app's price from the Store API and display it here.

You can buy the full app for: $4.99.

Click **Buy app** to display the purchase dialog box and update the license state according to the value it returns.

You can use this app for 477 more days before the trial period ends.

FIGURE 20-3 Scenario 1 of the Trial apps and in-app purchases sample (cropped slightly).

The Trial Period button calculates the number of days remaining in the trial period, using basic arithmetic and the `licenseInformation.expirationDate` property. Note that you should always use the `licenseInformation.isTrial` flag to check for trial validity instead of checking the date yourself, because it will properly handle regional variations in time/date handling. The sample's use of the `expirationDate` is just for UI purposes.

The Trial Mode and Purchased buttons just output different messages based on the state of the `isActive` and `isTrial` properties. Both button click handlers start like this:

```
var licenseInformation = currentApp.licenseInformation;
if (licenseInformation.isActive) {
    if (licenseInformation.isTrial) {
```

What can make the output from these buttons more interesting is modifying the data/trail-mode.xml file with different initial values for `IsActive` and `IsTrial`. Given that you'll be reading this book after the 1 January 2014 date in the sample, you'll see an expiration message on first run (`isActive` will be `false`, regardless of the flag's value in the XML). So try setting the `ExpirationDate` to a time in the future (remembering that its UTC time, not local time), rerun the sample, and you'll see that `IsActive` comes through as to `true`. Then set `ExpirationDate` about a minute in the future, set a breakpoint on the `trialModeRefreshScenario` function inside js/trial-mode.js, and restart the sample.

You won't hit your breakpoint immediately after `ExpirationDate` has passed, however. For performance reasons, the `licensechanged` event is not triggered instantly—there could be hundreds of expiration dates to track throughout the system. The event will instead fire reasonably soon, within about 20 minutes, so you might start such a test before going out for lunch.

This sample, of course, merely changes some output messages according to the validity of the license. In a real app you would either disable certain features for an active trial license or let the user do nothing more except purchase the app if the trial has expired. You'd want to make such checks both when the app is launched (for any reason) and in the resuming event.

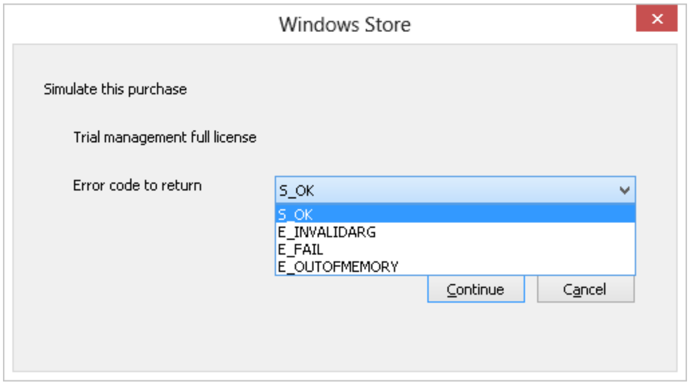
> **Tip** When an app license expires, Windows marks the app tile with an "X" glyph and prevents the app from being launched. Instead, the user is prompted to buy the app in the Store directly. Although this suggests that a newly launched app should never see licenseInformation.isActive set to false in the CurrentApp object, you always want to check for an active license on startup. This guards against side-loading hacks, because side-loaded apps will always have an inactive license.

Upgrading from an expired trial to a paid license is handled by the Buy App button in this scenario, an option that you should have to remind the user anytime they're running a trial, regardless of expiration status. This button calls a handler (named doTrialConversion) that makes use of the CurrentApp.requestAppPurchaseAsync method (js/trial-mode.js):

```
var licenseInformation = currentApp.licenseInformation;
if (!licenseInformation.isActive || licenseInformation.isTrial) {
    currentApp.requestAppPurchaseAsync(false).done(
    function () {
        if (licenseInformation.isActive && !licenseInformation.isTrial) {
            // Purchase was fulfilled
        } else {
            // Purchase UI was shown, but the user canceled. Trial is still in effect.
        }
    },
    function () {
        // There was an error in the transaction; purchase did not occur
    });
```

The one argument to requestAppPurchaseAsync indicates whether a receipt string is sent to your completed handler; see "Receipts" later on. In any case, if the user makes a purchase, the licensechanged event will fire as it does for trial expiration, so you can consolidate your license handling there.

With the CurrentAppSimulator, invoking requestAppPurchaseAsync won't show the actual Store UI. Instead you'll get an ultra-prosaic dialog in which you specify the exact return value (an HRESULT):


Windows Store

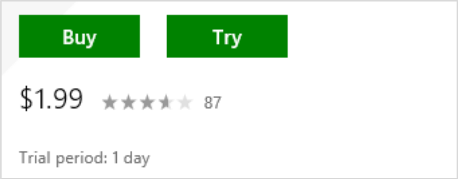
Simulate this purchase

Trial management full license

Error code to return: S_OK

Sending back S_OK indicates that the purchase was made. The `isTrial` flag should change to `false` and `isActive` set to `true`. Returning any of the other errors will invoke the error handler for `requestAppPurchaseAsync`. Pressing Cancel, on the other hand, will call your completed handler but the values of `isTrial` and `isActive` will remain unchanged.

In the real world, of course, consumers will not be fiddling around with simulated Store conditions. Instead, if your app is marked to offer a trial version (something you set while uploading to the Store), they'll see a Try button on the app's listing page like this:



$1.99 ★★★★☆ 87

Trial period: 1 day

Tapping Try will install the app and set both `isActive` and `isTrial` to `true`. At the point when the app calls `requestAppPurchaseAsync`, Windows will launch the Store and take the user to the app's listing page where they can tap the Buy button if they choose.

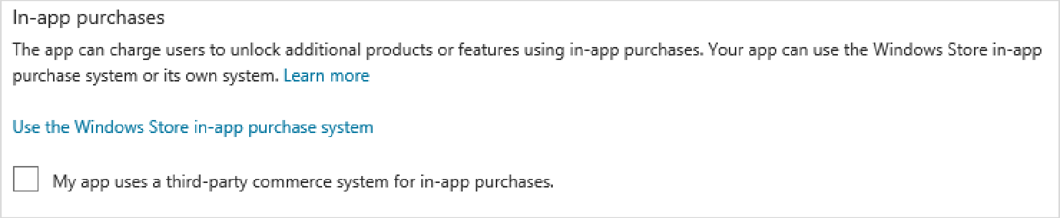
Tip When writing this book, I looked at a number of apps that were available in the Windows Store and found that although many offered trials, few of them gave me any indication about why and how to purchase a full version, nor told me that I was even running a trial and how long I had left in the trial period. I was presented with such an option only when the trial period had passed. If you want to convert trials into paid licenses, it's better, even as the sample demonstrates, to inform the user that she's running a trial and give her reminders and opportunities to convert!

Expiring apps? It's possible, when testing an app, to set an expiration time for the full app in the XML and not just for a trial. However, the Store dashboard doesn't provide a means to do this for a released app. Instead, you just stop making the app available.

Listing and Purchasing In-App Products

Working with your in-app purchases, or *products* as the API calls them, involves three aspects. The first is retrieving locale-specific product information from the Store for use in your own UI. The second is then completing the purchase and either activating the product's license for durables or reporting fulfillment for consumables. We'll cover these two steps in this section. The third aspect, covered in the section that follows, is handling large catalogs of purchase options that exceed the 200-item limit of the Store dashboard.

In-app purchases are configured on the Store Dashboard under Selling Details, as shown in Figure 20-4. If you check the box for using a third-party commerce system, you'll be entering details with that provider and not the Store and you'll be using that provider's API rather than what's discussed here. But if you're using the Store, click Use The Windows Store In-App Purchase System link and you'll go to the Services page where you add the details, as shown in Figure 20-5.



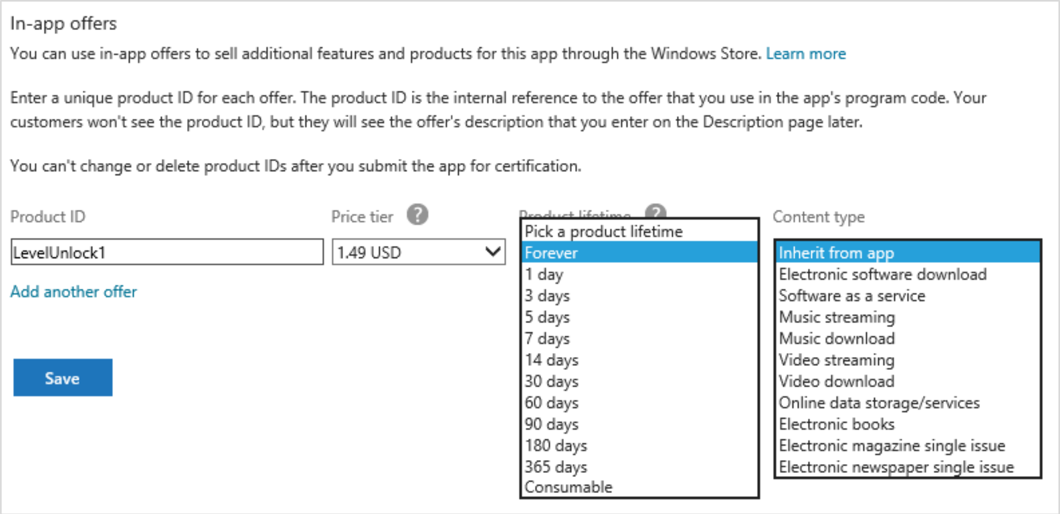
In-app purchases

The app can charge users to unlock additional products or features using in-app purchases. Your app can use the Windows Store in-app purchase system or its own system. Learn more

Use the Windows Store in-app purchase system

□ My app uses a third-party commerce system for in-app purchases.

FIGURE 20-4 The Store dashboard area for in-app purchases under Selling Details.



In-app offers

You can use in-app offers to sell additional features and products for this app through the Windows Store. Learn more

Enter a unique product ID for each offer. The product ID is the internal reference to the offer that you use in the app's program code. Your customers won't see the product ID, but they will see the offer's description that you enter on the Description page later.

You can't change or delete product IDs after you submit the app for certification.

Product ID: LevelUnlock1

Price tier: 1.49 USD

Product lifetime:
Pick a product lifetime
Forever
1 day
3 days
5 days
7 days
14 days
30 days
60 days
90 days
180 days
365 days
Consumable

Content type:
Inherit from app
Electronic software download
Software as a service
Music streaming
Music download
Video streaming
Video download
Online data storage/services
Electronic books
Electronic magazine single issue
Electronic newspaper single issue

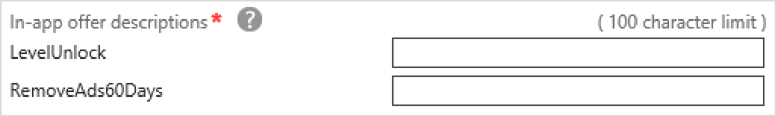
Add another offer

Save

FIGURE 20-5 Entering details for in-app purchases on the Store dashboard's Service page.

Each product has an ID that you assign, a price, a lifetime, and a specific content type. As you can see in Figure 20-5, the lifetime is where you differentiate durables and consumables, with every option in this drop-down being for durables except the very last one. As for the content type, this setting is needed for taxation purposes; it has no effect on the app and isn't exposed through the API. And note that when you're managing a large catalog, the products you describe here will be generic classes or types of offers for which you'll fill in specific details at run time (see "Handling Large Catalogs").

One bit of information you might be wondering about is the actual description of the in-app purchase that the Store will show to the user. It's not here because you'll enter it later after you upload your app package. At that time you'll then enter localized descriptions for each language you support at the same time you provide other descriptive text and graphics:



In-app offer descriptions * (100 character limit)

LevelUnlock

RemoveAds60Days

When using XML to configure the `CurrentAppSimulator`, you enter the same details (including the description) within the ListingInformation > Product nodes (for markets, pricing, duration, and purchase type) and the LicenseInformation > Product nodes (for license status and expiration dates). You can see some variations in scenarios 2–5 of the Trial app and in-app purchase sample.

Again, a consumable purchase can be repurchased as soon as a previous one is reported as fulfilled. A durable purchase is similar in that it can have an expiration date, meaning that the product license is in effect for a time, after which the user needs to repurchase it to continue its use. Either way, be sure that your app UI fully informs the user about the exact nature of the product at the time of purchase: don't surprise your users or they'll likely surprise you with less than favorable reviews in the Store!

Handling in-app purchases follows this general pattern:

- Retrieving a list of available products from the Store and displaying them in the app as needed.
- Initiating the purchase through the Store API and checking the result status, which reflects the user's actions in the Store's UI.
- Apply the purchase, which might initiate downloads or other async actions. At this point the particular consumable purchase is temporarily disabled while the transaction is in progress.
- If purchase-related actions for consumables are successful, the app reports fulfillment to complete the transaction. This is also necessary to re-enable the purchase.

Obtaining a list of in-app purchases is done through `CurrentApp.loadListingInformationAsync` to obtain the app's `ListingInformation`, whose `productListings` collection then contains *all* of the in-app purchases you've registered through the Store dashboard, regardless of type:

```
currentApp.loadListingInformationAsync().done(
    function (listingInfo) {
        // listingInfo.productListings contains in-app purchase details.
    });
```

The collection is a `MapView` object and not an array. To retrieve a specific item in the collection, use its `lookup` method:

```
var product1 = listing.productListings.lookup("product1");
```

or use a key-based lookup with the `[ ]` operator:

```
var product1 = listing.productListings["product1"];
```

You would do singular lookups like this when you know ahead of time which specific products you want to show in some part of your UI. That is, you might have certain set of purchases that you present in different parts of the app, or even individual purchases that you offer in specific contexts.

In other cases you'll want to iterate the collection and generate a list of applicable purchases in your UI. This takes a little more work because the MapView does not support index-based lookup nor the foreach method, as discussed in Chapter 6, "Data Binding, Templates, and Collections." You instead use an Iterator obtained through the first method, as shown here:

```
var iterator = listing.productListings.first()
var product;

while (iterator.hasCurrent) {
    product = iterator.current.value;
    // Use product.productId, name, formattedPrice, and productType to generate your UI.
    iterator.moveNext();
};
```

Here you'd use the product.productType, which will be durable or consumable (or unknown) to filter your list by type. For durable in-app purchases, you'll also likely filter out those products that have already been purchased. In that case, look up the product license within the CurrentApp.licenseInformation.productLicenses collection (a MapView of ProductLicense objects) using the product's ID as the key. Here's how we'd modify the code above to perform this additional step:

```
var iterator = listing.productListings.first()
var licenses = currentApp.licenseInformation.productLicenses;
var product;

while (iterator.hasCurrent) {
    product = iterator.current.value;

    if (licenses[product.productId].isActive) {
        // Product has already been purchased
    } else {
        // Product is available for purchase
    }

    iterator.moveNext();
};
```

With each consumable, you instead need to check if there is a pending fulfillment such that it cannot be repurchased at present. You do this through getUnfulfilledConsumablesAsync, using its results to omit or disable the appropriate items in your UI. The result of this method is a VectorView of UnfulfilledConsumable objects, each containing just a productId, a transactionId, and an offerId (for large catalogs, see the next section):

```
currentApp.getUnfulfilledConsumablesAsync().done(
    function (unfulfilledList) {
        unfulfilledList.forEach(function (product) {
            // Handle the unfulfilled consumable
        });
    });
```

To initiate an in-app purchase, call one of the three requestProductPurchaseAsync variants:

Arguments	Async result	Description
productId	PurchaseResults	Prompts the user to complete the purchase and provides a PurchaseResults object to the completed handler, which includes cancellation cases.
productId, offerId, displayProperties	PurchaseResults	Prompts the user to complete a purchase, using the information in the *displayProperties* object for the UI. The completed handler is called a PurchaseResults object in all cases.
productId, includeReceipt	string	Windows 8 only: prompts the user to complete the purchase and provides the XML receipt to the completed handler if *includeReceipt* is true. If the purchase is canceled, the error handler is called. Note that this is the in-app purchase API that's used in Windows 8 for durables only.

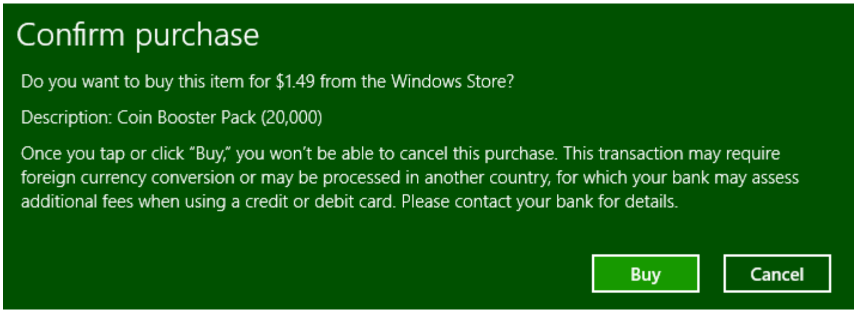
To be clear, the third variant above is the one that was available in Windows 8 for durables and exists for compatibility purposes. When targeting Windows 8.1, you'll use the first variant for both durables and consumables and the second when working with a large catalog.

Scenario 2 of the sample shows the most basic in-app durable purchase call, checking first that the user doesn't already have a license (js/in-app-purchase.js):

```
var licenseInformation = currentApp.licenseInformation;
if (!licenseInformation.productLicenses.lookup("product1").isActive) {
    currentApp.requestProductPurchaseAsync("product1").done(
       function () {
          if (licenseInformation.productLicenses.lookup("product1").isActive) {
             WinJS.log && WinJS.log("You bought Product 1.", "sample", "status");
          } else {
             WinJS.log && WinJS.log("Product 1 was not purchased.", "sample", "status");
          }
       },
       function () {
          WinJS.log && WinJS.log("Unable to buy Product 1.", "sample", "error");
       });
}
```

Note that the error handler for the async call is called only when there's an error in invoking the Store UI, such as when the device is offline and the Store cannot communicate with its backend services. Otherwise the completed handler will be called both when the purchase goes through and when the user cancels.

If the product already has an active license, requestProductPurchaseAsync will simply call your completed handler without showing any UI, as none is needed. Otherwise the user will see a series of prompts to confirm the purchase, including confirmation of their credentials. For the whole flow, see The in-app purchase user experience for a customer. A typical confirmation message is shown below (taken from Jetpack Joyride):



Confirm purchase

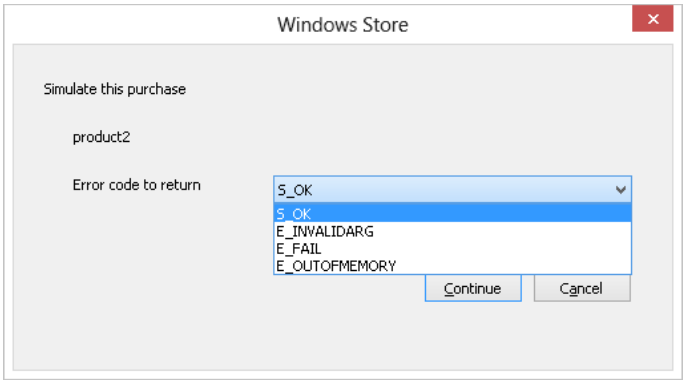
Do you want to buy this item for $1.49 from the Windows Store?

Description: Coin Booster Pack (20,000)

Once you tap or click "Buy," you won't be able to cancel this purchase. This transaction may require foreign currency conversion or may be processed in another country, for which your bank may assess additional fees when using a credit or debit card. Please contact your bank for details.

Note that the "Buy" warning in this dialog, which exists to meet regulations in some countries, is not entirely true: you can also cancel when entering your credentials in the next dialog unless you've specifically configured your account to not prompt for a password with each purchase.

When using the `CurrentAppSimulator`, of course, you won't see the Store prompts but only another simple dialog to control the result:



Windows Store

Simulate this purchase

product2

Error code to return: S_OK

In the code shown earlier, notice that it is checking the license status within the purchase request completed handler. It's better to keep any license-related code inside your `licensechanged` event handler instead. This is because the event is also fired if a time-limited product license expires, which is your signal to make the purchase available again (checking this expiration time for a durable in-app purchase is shown in scenario 3 of the sample[135]). It's likely that you'll want to alert the user to that status, perhaps with a toast notification or inline message when the user tries to access that feature.

In your completed handler, then, examine the `PurchaseResults` result to determine your next step. This object will contain a `transactionId`, an `offerId` (for large catalogs), a `receiptXml` string, and a `ProductPurchaseStatus` value in its `status` property. For durables the applicable status values are `succeeded`, `alreadyPurchased`, and `notPurchased` (user canceled). For consumables you can see `succeeded`, `notPurchased`, and `notFulfilled`, the latter of which means that there's already an

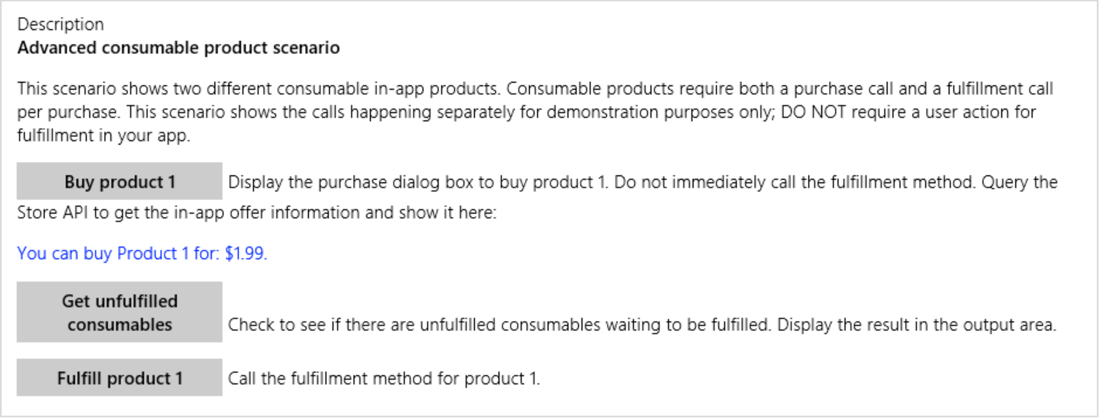
[135] There's a bug in the sample's js/expiring-product.js file that throws an exception when you switch to that scenario: a string `"scenario3Product1Message"` should be just `"product1Message"`, which is easily corrected.

outstanding purchase of this item that's awaiting fulfillment and that this attempted repeat purchase was blocked. For both durables and consumables alike, then, a status of `succeeded` indicates that you can go ahead and download content, activate a feature, or whatever else is necessary.

Checking the status is straightforward, as we can see code from scenario 4 of the sample below (js/in-app-purchase-consumables.js). Note that when using the `CurrentAppSimulator`, the purchase request will give you a simple dialog box in which you can control success and error conditions:

```
function purchaseProduct1() {
    currentApp.requestProductPurchaseAsync("product1").done(
        function (purchaseResults) {
            if (purchaseResults.status === ProductPurchaseStatus.succeeded) {
                // Grant the user their content here, and then call
                // currentApp.reportConsumableFulfillment when appropriate
            } else if (purchaseResults.status === ProductPurchaseStatus.notFulfilled) {
                // A previous purchase is waiting on fulfillment
                tempTransactionId["product1"] = purchaseResults.transactionId;
            } else if (purchaseResults.status === ProductPurchaseStatus.notPurchased) {
                // User canceled
            }
        },
        function () {
            // There was an error invoking the purchase UI; device could be offline.
        });
}
```

To see the relationship between purchase and fulfillment of consumables, scenario 5 of the sample, part of which is shown in Figure 20-6, lets you separately control purchase and fulfillment—and does so with the warning that you should *never* require a user action to fulfill a purchase! And in cases where you attempt to repurchase a consumable that's still unfulfilled, you'll see the `notFulfilled` status.



Description

Advanced consumable product scenario

This scenario shows two different consumable in-app products. Consumable products require both a purchase call and a fulfillment call per purchase. This scenario shows the calls happening separately for demonstration purposes only; DO NOT require a user action for fulfillment in your app.

Display the purchase dialog box to buy product 1. Do not immediately call the fulfillment method. Query the Store API to get the in-app offer information and show it here:

You can buy Product 1 for: $1.99.

Check to see if there are unfulfilled consumables waiting to be fulfilled. Display the result in the output area.

Call the fulfillment method for product 1.

FIGURE 20-6 Scenario 5 of the Trial apps and in-app purchases sample (cropped) allowing specific control of purchase and fulfillment of consumables.

Reporting fulfillment of a consumable purchase—which is appropriate when the app can confirm that it how has full access to whatever value the user believe he or she purchased—is done by calling reportConsumableFulfillmentAsync with the productId and the transactionId (the latter being the one reported in the PurchaseResults object). Scenario 4 of the sample does it as follows: (js/in-app-purchase-consumables.js):

```
currentApp.reportConsumableFulfillmentAsync(productId, transactionId).done(
    function (result) {
        switch (result) {
            case FulfillmentResult.succeeded:
                break;
            case FulfillmentResult.nothingToFulfill:
                break;
            case FulfillmentResult.purchasePending:
                break;
            case FulfillmentResult.purchaseReverted:
                // User's purchase was revoked, and they got their money back.
                // Revoke the user's access to the consumable content as necessary.
                break;
            case FulfillmentResult.serverError:
                break;
        }
    });
```

As you can see, this call will provide a FulfillmentResult value so you can determine if you've really completed the transaction. The possibilities here are as follows:

- succeeded The fulfillment is complete and the consumable can be offered again.
- nothingToFulfill The transactionId has already been fulfilled or is otherwise complete.
- purchasePending The purchase has not yet cleared and could still be revoked.
- purchaseReverted The transaction was canceled on the backend and the app should disable access to the feature or content.

Because it's the app's responsibility to fulfill each consumable to complete the transaction, there will be times when you need to determine which consumable purchases are still outstanding, for example, to disable those options in your UI. You do this through getUnfulfilledConsumablesAsync, which results in a VectorView of UnfulfilledConsumable objects, each of which contains just a productId, a transactionId, and an offerId (for large catalogs, see the next section). Scenario 5 of the sample uses this to keep a list of pending transactions (js/in-app-purchase-consumabled-advanced.js):

```
currentApp.getUnfulfilledConsumablesAsync().done(
    function (unfulfilledList) {
        unfulfilledList.forEach(function (product) {
            // Usually check fulfillment and call reportConsumableFulfillment if needed
            tempTransactionId[product.productId] = product.transactionId;
        });
    });
```

This brings up one last question with in-app purchases—how do you cancel an unfulfilled consumable? For example, if fulfillment depends on acquiring specific data from a backend, but for some reason that just isn't possible for that particular request, how do you cancel the transaction and thus re-enable the purchase for content that is available?

The present answer is that you don't—there's not an API to back out a pending purchase as it's a relatively rare scenario. Still, if you encounter such a situation, you'll want to inform the user that the content they originally selected is no longer available and give them a list of alternate choices. This encourages them to fulfill the purchase quickly in another way.

Handling Large Catalogs

By design, the Store dashboard limits the number of in-app purchases (durables and consumables combined) to 200. Although this is plenty for many apps, it is insufficient for apps that want to present users with a much larger array of choices, such as ebooks, movie libraries, music tracks, stock images, special reports, and so forth. In these cases, it doesn't make sense for each and every piece of content to have its own product ID in the Store. Instead, it makes sense for the products to represent *classes* of purchases, each with its own price tier, which can then be particularized further with another identifier. This way you can have up to 200 such classes, each backed by however many items your identification scheme can accommodate.

Supporting large catalogs is the purpose of the `requestProductPurchaseAsync` variant that takes three arguments: a *productId* (a string), an *offerId* (another string), and a `ProductPurchaseDisplayProperties` object. Here, the *productId* must match one of your entries in the Store dashboard, while *offerId* can be whatever you want because the Store does nothing more with it than pass it back to you in the `PurchaseResults.offerId` property. (This saves you the trouble of having to maintain your own associations between productId, offerId, and transactionId—isn't that considerate?)

The `ProductPurchaseDisplayProperties` object, and specifically its `name` property, is how you then provide an exact description of the *offerId*.[136] Thus, instead of having the Store UI just show the productId description, which might be "3-day movie rental," it can show "3-day rental: The Song of Bernadette" and so forth.

It almost goes without saying that how you map an *offerId* to names and images is completely up to you and—most likely—your app's backend. Indeed, almost every app that needs a large catalog will be retrieving catalog data from a backend in the first place, and will thus query some database through a service API, get back a list of applicable items (offerId, name, and image URI), and then show those results in a ListView or other gallery-style UI. When initiating a purchase, you'd just take the information from the item in that list, populate the `ProductPurchaseDisplayProperties`, and make the `requestProductPurchaseAsync` call.

Scenario 6 of the Trial app and in-app purchase sample provides a simple demonstration of making

[136] The other properties in this object apply only to Windows Phone.

the call, though of course it doesn't work with a large catalog of real data. Instead, it just lets you enter an offerId and a product name to use in the call (js/in-app-purchase-large-catalog.js):

```
var offerId = document.getElementById("offerId").value;
var displayPropertiesName = document.getElementById("displayPropertiesName").value;
var displayProperties = new ProductPurchaseDisplayProperties(displayPropertiesName);

currentApp.requestProductPurchaseAsync("product1", offerId, displayProperties).done(
    function (purchaseResults) {
        // Process results
    },
    function () {
        // Failure to show the Store UI
    });
}
```

As with all other purchases, the `CurrentAppSimulator` will display its return value dialog; with the live Store, you'll instead see the usual confirmation dialog with the specific product name from the display properties object.

Receipts

As we've discussed, an app validates purchases by checking license status and/or `PurchaseResults` values at run time. Many apps, however, present in-app purchases for capabilities or content that are managed on a server. For example, renting a video from a streaming service might have two parts: making a purchase in the app and then having your backend service validate itself on the app's behalf with its secondary providers. As a result, your services need a way to validate those purchases with some piece of trusted information, because those services don't have access to the Window Store APIs.

This is the purpose of receipts, which are XML strings that you can obtain from the `CurrentApp` `requestAppPurchaseAsync` and `requestProductPurchaseAsync` methods. The `getAppReceiptAsync` and `getProductReceiptAsync` methods also provide an all-up receipt (app and products) or an individual receipt at any time. You then send these XML documents to your services when needed.

Note Receipts are not cached locally for security purposes; the receipt APIs fail if the device is offline.

Receipts can also be used to protect your app against certain hacks that attempt to modify license information that's cached on a system. See Protecting your Windows Store app from unauthorized use for a discussion.

In all cases, the receipt XML contains the app or product id, the dates for both when the purchase was made and when the receipt was issued, and a digital signature. The details of the XML schema can be found on the reference pages linked above. As an example, here's the receipt string provided from `getAppReceiptAsync` in scenario 8 of the sample:

```
<?xml version="1.0" encoding="utf-8"?>
<Receipt Version="1.0" ReceiptDate="2014-02-21T23:20:22Z" CertificateId=""
    ReceiptDeviceId="94c007cb-12b4-47be-8a4a-d7e94d7ba6d6">
```

```
  <AppReceipt Id="917b34ef-738e-49bb-84ba-c783c2dafba6"
    AppId="Microsoft.SDKSamples.Store.JS_8wekyb3d8bbwe"
    PurchaseDate="2014-02-21T23:20:19Z" LicenseType="Full"  />
  <ProductReceipt Id="66505c93-fe55-4d77-a941-dbd61e7f1177"
    AppId="Microsoft.SDKSamples.Store.JS_8wekyb3d8bbwe"
    ProductId="product2" PurchaseDate="2014-02-21T23:20:19Z"
    ProductType="Durable" ExpirationDate="2014-01-01T00:00:00Z"  />
  <ProductReceipt Id="6974609e-5572-44ab-9444-1b24096f7d48"
    AppId="Microsoft.SDKSamples.Store.JS_8wekyb3d8bbwe"
    ProductId="product1" PurchaseDate="2014-02-21T23:20:19Z" ProductType="Durable" />
</Receipt>
```

An individual app or product receipt will look the same but have only one AppReceipt or ProductReceipt child node under Receipt.

Tip If you want to consume a receipt as an XML document instead of a string (for display, print, or programmatic traversal), it's a simple matter to create such an object like we did with tile and notification XML in Chapter 16:

```
var receiptDOM = new Windows.Data.Xml.Dom.XmlDocument();
receiptDOM.loadXml(receipt);
```

In the top-level Receipt node, the ReceiptDate is obviously when it was issued. CertificateId is a GUID when you get a receipt from the live Store (which is why it's empty in the sample), and in that case you'll also see a Signature node under Receipt. You use both of these on the server to validate the authenticity of the receipt. For details (using C# code for an ASP.NET service), refer to Using receipts to verify purchases.

The ReceiptDeviceId is then a unique identifier for the specific device where this receipt was issued. Remember that the Store maintains licenses for in-app purchases on a per-user basis, so it will report a valid license on every one of the user's devices. Each receipt, however, will have a unique ReceiptDeviceId, so a server can use this attribute to track how many unique requests are coming from the app under the same user account. This becomes important when your backend is using other services that have usage limits, such as perhaps five simultaneous sessions or no more than three requests for the same content. Thus, when your app requests content from a service, it uploads the receipt. The server validates the receipt and checks its necessary quotas. If access can be granted, the server responds accordingly; if the quota has been exceeded, the server can instead respond with an error code and the app can inform the user of that condition. The user might then choose to purchase additional access rights or will know that he needs to close a session on another device.

Having the purchase and expiration dates within the receipt allows the server to apply its own validation policies, including enforcing expiration dates and detecting old receipts ("replay attempts") that can no longer be used. In addition, the Id attribute of AppReceipt and ProductReceipt uniquely identifies a given purchase transaction (at a specific time on a specific device), which a server can use for further authentication and protection.

For further discussion about receipts, see //build 2013 session 3-126, Validating Windows Store Purchases for Your App.

Instrumenting Your App for Telemetry and Analytics

At this stage in your app's journey you're getting very near the point of uploading it to the Window Store and working with real customers. But before you do so, ask yourself this question: how will you understand what your customers are actually *doing* with the app?

First of all, the Windows Store provides you quite a bit of information, as described in the topics under Analyze and improve. This includes average app usage per day with comparisons to other apps in your category, quality reports (crash dumps, unresponsive events, and JavaScript exceptions, which I hope you never see!), ratings and reviews, adoption rates, download history, in-app purchases, and app sales. Furthermore, various third parties, such as Distimo and AppFeds, produce analytics for the Windows Store as a whole, which is very valuable for tracking overall market trends and activities.

However, none of this really tells you about users' *specific* activity within the app itself.

You might be thinking, "Hey, no problem—I've littered my code with all kinds of logging. Won't that work?" Well, logging is a good start, but logging is generally a tool that you use during development so that you can diagnose errors and code flows. Logging is typically oriented around the internal structure of your app rather than reflecting real-world customer usage. In short, logging is how you collect data about your app in the lab; instrumenting your app for *telemetry*, on the other hand, is how you collect data once the app is released into the wild.

Telemetry, or *tele-metering*, is automated remote measurement and data collection. It's used in all kinds of industries, from tracking spacecraft, tracking wildlife, medical monitoring, law enforcement, and so on. I especially like to think of it in the context of spacecraft, where it is *the* way in which mission controllers monitor the health and operation of a very expensive piece of hardware that is otherwise completely unreachable. Without telemetry, in short, you're flying blind.

Putting an app into the global market is, in many ways, similar to launching a spacecraft. Your app represents a significant investment, and once it gets out to more than a handful of customers the main issue is not so much distance, but scale. And what you need is a way to both collect the data and then condense it into useful *analytics*, or reports that human beings can read to make decisions.

These analytics are a gold mine. They enable you to peer over your user's shoulders (respecting privacy of course!) and know what they're really doing. This enables you to answer many questions, such as:

- How are customers really engaging with the app? Do people use the features you thought they wanted?
- Where are they spending (or not spending) their time? How long do users spend in each app session, between sessions, and between suspend/resume?
- What device configurations do your users prefer (view size, screen orientation, display types, input modalities)?

- How are customers using options you provide via Settings?
- What's going on when crashes happen? How often do users encounter noncrashing errors such as a failed HTTP request, a failed sync, timeouts, etc.?
- How successful and engaging are the various features of the app? Are social features being used? Are there usage patterns that you can reward or discourage in some way to drive behavior?
- How successful are your trial versions and/or in-app purchases? Where are trial conversion reminders and/or in-app purchase options most effectively displayed? How many users look at purchase options but don't buy anything?
- Are users clicking ads (or converting to a paid version to get rid of them)?
- How often do users run online vs. offline?
- Where should you concentrate future investments for quality, performance, and feature updates? What ideas could you test in your next update or in a separate app?
- Is the app truly serving your business goals?

I can't imagine that anyone who is serious about their apps business would *not* want answers to these kinds of questions, which is why many app developers consider telemetry a *must-do* for their apps, starting with any beta-testing or early previews they might put out.

Tip If you collect telemetry for beta or preview apps, be sure to separate that data from production versions, and also consider separating data from each major version of the app so that you can answer your questions clearly.

Early in this chapter I said that publishing an app means going into business, and telemetry tells you a lot about the success of that business. To return for a moment to the four "F's" as I call them—the four motivations for writing apps—here's how telemetry is typically used for each:

Business Type	**Use of Telemetry**
Fame	*Priority: Growing a customer base.* Are you acquiring new customers? How are you acquiring them? What strategies in the app lead to new customers?
Fortune	*Priority: Monetization.* Assuming you've done market analysis on the competition, are your differentiating features succeeding? Are your monetization strategies being employed? What use patterns can you take advantage of to further monetize?
Fun	*Priority: Sharing your joy.* Are your customers enjoying themselves and perhaps sharing that joy with their social network?
Philanthropy	*Priority: Promoting a message.* Are customers inspired by the message and the cause? Are customers willing to support the cause?

The questions you want to answer with your app will lead directly into how you instrument your app to gather telemetry. That is, your questions lead to telemetry design, and your telemetry design leads to what are called *events*. Events are broad, human-readable verbs or actions that you want to track—generally just strings that you use to categorize your telemetry data. This way they become the basic unit of organization for the analytics that you'll eventually get from your telemetry.

Typical apps log about 30 distinct events, which often include (but are not limited to) app start, app exist, registration, login/log off, settings changes, content sharing, recoverable errors, non-recoverable exceptions, view content, mark as favorite, comment on content, viewing and item or category in a catalog, search or filter, add to wishlist, begin/complete/abandon checkout, invite friend, accept a friend's invitation, game started, game completed, hint requested, and so on.

The relatively small number of events is driven by the fact that each one surfaces separately in the analytics that get generated from the telemetry—each event perhaps translates to a separate chart. Too many (or too few) events makes it difficult to harvest actionable information from the analytics, so when you think about events you really want to think about the insights you want to gain. For example, it's more important to track what content items are being tapped or clicked rather than the pointer events themselves, except perhaps to gather secondary data about the type of input (mouse, touch, or stylus).

Events help identify users' flow through the app and per-page feature usage. As such, they are usually static (not dynamically generated) and are usually somewhat generic, such as "Article read." Event attributes or properties (think adjectives or subject nouns) then provide specific details, such as the URI or title of the article being read. This way the top-level events answer one level of questions about app usage, with the attributes providing another level. This becomes very helpful when generating charts from all this data, where the overall chart is for an event with attributes and properties providing the individual data points, bars, lines, and so on.

If you think about charting—which is probably the most common way to consume analytics—it's also important that the properties and attributes you assign to events will chart well. For this reason, numerical attributes that can vary widely are best grouped into "buckets" or ranges, rather than reported as discrete values. For example, you could track the time spent on a given page or the time it took to complete a game level in terms of 0–5s, 6–10s, 11–20s, 21–60s, 61s–120s, and so on so that a pie chart or bar chart has a reasonable and meaningful number of elements.

It's also best to report events at the end of an action when you have all the data you might attach to it, thereby making each event as rich as you can and reducing clutter in your analytics.

I hope that I've convinced you that instrumenting your app for telemetry is a good idea! So how then do you go about it? Well, you have two choices:

- **The hard way** Implement some kind of tracing/logging API of your own, sprinkle calls to that API throughout your app, implement a backend service to which you regularly upload your collected data (and make sure it can scale to thousands or tens of thousands of users), and then implement an entire analytics engine to process that data into meaningful reports.

- **The easy way** Register with a third-party analytics provider who is in the business of doing most of the hard work for you. Incorporate their SDK into your app, make calls to their API as needed, and spend most of your time using the analytics to create better apps!

Unless you already have an infrastructure in place, I suspect that you'll choose the easy way! If you go to the Windows Partner Directory, where perhaps you've already looked for ads and commerce providers, you can click the Analytics filter and see a number of companies that offer great tools in this regard. Do note that the directory here intermixes partners that provide telemetry services, like we're talking about here, and those who provide services like error tracking, market performance tracking, or marketwide trend reports (which are also useful, but different). But just spend a little time looking through the list and you'll find the right ones—such as Localytics, MarkedUp, AppFireworks, mtiks, Adobe Omniture, and Parse. (Note that Flurry, the most popular analytics SDK for Windows Phone and other platforms, is not at present available for Windows Store apps, whereas many of the others in this list work on both. Furthermore, some providers might not offer an SDK for apps written in JavaScript, so check on the specifics.) Microsoft is also busy creating an analytics platform of its own called Application Insights.

What I very much appreciate is that these providers typically offer a free service tier so that you can get started without any up-front cost and start paying only when your app is successful enough to be generating significant data.

The typical flow with these providers is that you visit their portal to create an account, download the SDK and incorporate it into your project, and then register an app for which you receive a unique key. You provide this key when you initialize the SDK's main object, and then you call that object's methods to log your events. Depending on the SDK, you might need to request an upload of the data, or it might do it automatically. Either way, after using the app for some time, you visit the provider's portal and explore the results.

A few tips:

- **Use separate development, beta test, and production keys** The purpose of telemetry is to gather actionable data from real-world users, so you always want to keep any data gathered during your development phase separate from real customer data. It's likely that you'll also want to separate beta testing data (e.g., for side-loaded versions, which probably has correlations with distinct time periods) from final production data (for the app acquired from the Store). This means creating separate keys through the provider's portal and make sure to change them when you build each variant of the app.
- **Create a telemetry layer in your app** This allows you to easily switch providers at any time, if the need arises, and encapsulates any computation or bucketing logic behind the layer's interface so that it doesn't interfere with the rest of the app. If you define a simple enablement flag and check it within each method of your layer, it also makes it very easy to turn telemetry on and off (as when the user opts out) without touching any other part of your code.
- **Test your telemetry by visiting the analytics portal early and often** At the end of the day,

you're doing all of this to get actionable insights from the provider's analytics. So even when you first start doing your instrumentation, gather some data and then visit the provider's portal to see the results. You'll quickly find whether the events you defined are meaningful, whether the attributes and properties you include with the events have the right granularity, and whether you're correctly reporting that data.

- **Be sure to turn on the *Internet (Client)* capability in your manifest** Otherwise nothing will ever make it to the backend! Note that so long as you collect no personal information, email addresses, screenshots, or browsing history, gathering telemetry should not affect your app's required age rating.
- **Inform the user and allow opt out** Telemetry data is a form of user information, though not personally identifiable. For this reason, make sure that your app's page in the Store and your privacy policy, and provide an option in your settings to disable telemetry altogether if the user chooses to opt out.

As a small example, I instrumented the 15+Puzzle game I have in the Windows Store with telemetry so that I can determine usage patterns, especially patterns that affect different monetization strategies. I used the Localytics SDK, but anything specific to that SDK is isolated in my telemetry layer. That layer is implemented in my own `Telemetry` namespace that's defined as follows, where the `init` function gets everything going:

```
WinJS.Namespace.define("Telemetry", {
    inputType: { touch: 0, mouse: 1, keyboard: 2},

    session: null,
    enabled: true,
    _curPage: null,
    _lastNavTime: 0,
    _settingsTime: 0,

    _timeLastStarted: 0,
    _timeLastCompleted: 0,
    _inputTally: new Array(3),

    _now: function () {
        return new Date().getTime();
    },

    _convertToRange: function (value, granularity) {
        //For telemetry, we don't want to log every number, but should group them together
        // into ranges. This method returns a string based on the granularity. For 25 for
        // example, it'll create strings "0-24", "25-49", "50-74", etc.

        //Special case a true zero
        if (value === 0) {
            return "0";
        }

        var rangeBase = Math.floor(value / granularity);
```

```
         return (granularity * rangeBase) + "-" + ((granularity * (rangeBase + 1)) - 1);
      },

      init: function () {
         var keyDeveloper = "...";
         var keyBeta = "...";
         var keyProduction = "...";

         Telemetry.session = LocalyticsSession(/* key */);
         Telemetry.session.open();
         Telemetry.session.upload();
         } else {
            Telemetry.session = null;
         }

         Telemetry.enabled = (Telemetry.session !== null);

         //Automatically log app errors
         WinJS.Application.addEventListener("error", function I {
            Telemetry.error("WinJS.Application.onerror", { "Line": e.detail.errorLine,
            "Character": e.detail.errorCharacter, "File": e.detail.errorUrl,
            "Message": e.message });
         });
      },

      // Other event methods, such as error
   }
```

You can see here that I have a generic bucketing method, *_convertToRange*, which I use elsewhere in the layer, whose other methods reflect my event design. These include: *error, appStarted, syspending, resuming, visibilityChange, licenseChanged, appResized, newGrid, restarted, pageNav, scoredCleared, scoresViewChanged, gameStarted, gameRestarted, gameCompleted, tallyInput, optionsEnter, optionsExit, settingsEnter, settingsExit,* and *feedback*. (This last one, by the way, is how I log user comments in a Feedback panel that's part of the app's Settings. This way I don't need some other service to collect the data.)

To show what I mean by encapsulating telemetry logic in this layer, below is the *appResized* method that's called on `window.onresize` events; as you can see, I gather up information about the device characteristics, because I want to know if the customer is playing full screen in portrait or landscape, how large of a screen they're on, and the size of the app view itself. This data can inform future investments in layout and user experience. If I know, for example, that customers prefer portrait over landscape, I can focus on that layout first and landscape second:

```
appResized: function () {
   if (Telemetry.session === null || !Telemetry.enabled) { return; }

   var wgd = Windows.Graphics.Display;
   var displayInfo = wgd.DisplayInformation.getForCurrentView();
   var w = document.body.clientWidth;
   var h = document.body.clientHeight;
```

```
    //Convert orientation enum into a simple string value
    var orientation = "landscape";

    if (displayInfo.currentOrientation == wgd.DisplayOrientations.portrait
        || displayInfo.currentOrientation == wgd.DisplayOrientations.portraitFlipped) {
        orientation = "portrait";
    }

    //Here I group window sizes at a 200px granularity for better analytics.
    var data = {
        "Display Orientation": orientation,
        "Scale": displayInfo.resolutionScale,
        "View Orientation": (w < h) ? "portrait" : "landscape",
        "Width": Telemetry._convertToRange(w, 200),
        "Height": Telemetry._convertToRange(h, 200)
    };

    Telemetry.session.tagEvent("View resized", data);
},
```

With my *gameCompleted* event, I also report what kind of input was used to play the game, bucketing the specific counts, of course. This is data that I want to drive further investments in handling input, such as my implementation of keyboard support:

```
gameCompleted: function (data) {
    if (Telemetry.session === null || !Telemetry.enabled) { return; }

    data = data || {};
    Telemetry._timeLastCompleted = Telemetry._now();

    //Add a range for inputs used in this game.
    data["InputTally_Touch"] =
        Telemetry._convertToRange(Telemetry._inputTally[Telemetry.inputType.touch], 25);
    data["InputTally_Mouse"] =
        Telemetry._convertToRange(Telemetry._inputTally[Telemetry.inputType.mouse], 25);
    data["InputTally_Keyboard"] =
        Telemetry._convertToRange(Telemetry._inputTally[Telemetry.inputType.keyboard], 25);

    //Add an attribute for network connectivity: I don’t use this in the game at present,
    //but the data can help me understand whether to add network-related features.
    var connected = false;

    //Returns null if offline
    var profile = Windows.Networking.Connectivity.NetworkInformation
        .getInternetConnectionProfile();

    if (profile != null) {
        var level = profile.getNetworkConnectivityLevel();

    //internetAccess or constrainedInternetAccess is probably acceptable; none or
    //limitedAccess is not.
    connected =
        (level == Windows.Networking.Connectivity.NetworkConnectivityLevel.internetAccess) ||
        (level ==Windows.Networking.Connectivity.NetworkConnectivityLevel
            .constrainedInternetAccess);
    }

    data["Has Connectivity"] = connected;

    Telemetry.session.tagEvent("Game completed", data);
},
```

Where all of this telemetry and analytics becomes even more valuable is if I decide to produce multiple apps of the same nature. By collecting telemetry in one app, I learn a great deal that I can apply to the first versions of subsequent apps. Indeed, one reason why you might choose to release a free app and strive to build up a large user base is just to acquire telemetry that can then inform a more significant investment in an app you'd like to monetize. Like I said, people who are serious about their apps business consider telemetry as a must-do! I strongly encourage you to do the same.

Releasing Your App to the World

We have finally come full circle to the exact point where we started in Chapter 1: onboarding your world-ready app to the Windows Store and making it available to that world.

Because the onboarding process is well documented already in the Publish your app in the Windows Store topic, I'm not going to spend our time here giving you too many screenshots from the Store dashboard, where all of this action takes place. I'll point you to specific pages in those docs when appropriate, but it's definitely a section of the documentation where you should spend some time and not just assume that you know what's best. After all, the Windows Store is *the* retail channel for your app, so you want to understand that channel as best you can. The Store dashboard is also designed to lead you through the process directly.

For example, on the "Selling apps" topic there's a section on Write a great app description. I can't encourage you to think about this enough! Why? Because your app description, along with promotional graphics, are one of the primary ways that users in the Store make a decision about acquiring your app. Furthermore, if your app happens to be featured in the Store, the first line or two of that description is what users will see. Every word of that description counts and counts so much that it's one of the things that the Store team looks at when selecting apps to feature, because they want the Store's home page to look good too. Take time to do this well, and if you don't feel up to the challenge, hire a writer for this purpose.

But like I said, that section of the documentation is quite comprehensive, so I'm not going to repeat it. What I want to focus on here specifically are those aspects of the process that aren't always so obvious, based on the real-world experience that I and my teammates in the Windows Ecosystem Team have gained through working with the partners to submit apps to the Store. Through this I hope to raise your awareness of issues that you'll likely face so that you're more prepared to address them.

Sidebar: Build Targets and Maintaining Windows 8 and Windows 8.1 Versions

When you create your app package to upload to the Store, be mindful to set your project's configuration to Release instead of Debug, otherwise it will fail certification. When choosing the target platform, set this to "Any CPU" unless you specifically have WinRT components written in C++. That is, JavaScript and .NET languages (C#/VisualBasic) are architecture-neutral; anything written in C++, on the other hand, must target x86, x64, and ARM specifically and therefore requires uploading three builds to the Store.

Similarly, your customer base might require that you support both Windows 8 and Windows 8.1 versions of your app for a time. The Store supports this, and you can find more information on the blog post Managing your app on Windows 8 and Windows 8.1. Also note that your Windows 8.1 version must have a higher version number than your Windows 8 version, but be sure to leave space between the version numbers so that you can update the Windows 8 app and increment the version number.

Promotional Screenshots, Store Graphics, and Text Copy

Before you starting thinking of the obvious steps of uploading a package to the Store, , review the topic How your app appears in the Windows Store and a couple of its associated topics: What to name your app, Your app's description, Choosing your app images, and Make your app easy to promote.

The reason why I specifically call out these topics is because you've invested or you're going to invest a lot of time and energy developing your app (and testing it, as we'll discuss soon), and so you should make a comparable effort to make it look great in the Store. All of the content described by the links above—your app's name and description, its details, keywords (search terms, that is), in-app purchase descriptions, and its promotional images, *localized for each market you want to target*—constitute your customers' first experience of your app and play a definite role in the possibility of your app getting featured in the Store. (Go back to Chapter 19 and review "Part 3: Creating a Project for Nonpackaged Resources" for a complete list of what you'll need when onboarding your app.)

Let me say that again: all of this information is what potential customers will use to evaluate your app before they tap any button to acquire it. It's what the Store team also uses to evaluate the marketability of your app. In short, it is all marketing material, plain and simple, so make it shine! Spend time writing really good copy for your app description—even to the point of having it professionally edited or hiring a professional writer. If you feel your app is fun and engaging, *communicate* that experience through your description and imagery. Share everything you love about your own work! Truly, you want customers' first impression of your app—just from a quick glance at your app's page in the Store—to be WOW! And this content is all that determines that response.

The other reason I emphasize this so strongly is that the Store won't ask for any of this information until you're very near the end of the onboarding process, which is exactly when you'll be most anxious to complete the process! If you haven't prepared those materials already—including localized versions—and you're trying to get the app out as quickly as possible, you'll end up cutting some serious corners. As a result, your app's first impression will be nowhere near as good as it could be.

> **Tip** You can enable access to the Description area of the dashboard if you upload *any* package to the Store for your app, regardless of its completeness. I recommend that you do this early on with any old test package just so that you can see for yourself what information will be needed later.

I also encourage you to watch Pete Brown's //build 2013 session entitled The Wow Factor: Making Your Windows Store App Promotable, where he discusses the many criteria that the Store team uses to select featured apps and those that are used in demos and conference keynotes. To summarize:

- Build a great app, because that's what gets attention first and foremost, but then your app also has to be promotable.
- Have a great Start screen presence with a beautiful tile and/or live tiles. Whenever you see a Start screen in advertisements, a keynote session, or anywhere else, every tile has been consciously chosen. If you don't support live tiles, invest the most in your 150x150 square tile and be sure your branding is clear. Also provide scaled versions of your tile for maximum sharpness.
- An app should have a "best-at" statement that's reflected in the first line of its description. Apps that have a clear and valuable purpose make a great featured item in the Store, because users will see it and say "I want this."
- In your app description, say what the user experiences in your app in support of your best-at statement. Grab attention with the first sentences (which can show in the Store feature page), use lists and short paragraphs, avoid dry or pedantic language, and be clear about trials, in app purchases, and so forth. Get inspiration by reviewing write-ups for other apps, and don't forget spelling and grammar.
- Be honest in your app description, and make sure to disclose geographical restrictions, hardware requirements, partial localization, and why you've declared any questionable capabilities in your manifest.
- If you're using ads, consciously incorporate them into the app design so that they integrate naturally and beautifully and contribute, rather than detract, from the user experience.
- Handle view sizing really well so that the app appears beautifully alongside other apps, and support multiple views if appropriate. Views and view handling are a key differentiator for the Windows platform as a whole and thus something that's highlighted in marketing. If your app works really well in those scenarios, it might be included in that marketing.
- Use the app bar and nav bar intelligently and creatively.
- Offer a trial for paid apps.

- Be part of the bigger picture with apps as a whole by supporting contracts like Share.
- Submit awesome age-appropriate screenshots and promotional graphics (the first one being the most important), and think outside the box a little. Graphics don't have to be straight-on screenshots but can show the app in other contexts, including human interaction with the product. In other words, think like you're designing a box for your app that would sit on a retail store shelf—you'd show what the app does but not limit yourself to screenshots.
- Match exact sizes for promotional graphics: 414x180 (the only one if you do one), 414x468 (used more in Windows 8.1), 558x756, and 846x468.
- Support every architecture—x86, x64, and ARM—because otherwise a customer might see your app in advertising and then be unable to find it on their device.

In short, as much as you plan to build a great app, also plan for its promotion rather than just leaving it all to chance.

The last bit I'll mention here is that you also want to make it easy for people outside of the Store and Microsoft to promote your app. The information on your Store page helps with this, but you should also offer a "press kit" on your website that includes various forms of promotional text, press releases, a variety of marketing images, app and/or company logos, links to your Facebook page, your Twitter handle, links to other web resources (like YouTube videos), and the like. By doing this work ahead of time, you make it all the easier for others to promote you.

Testing and Pre-Certification Tools

Unless you're a born tester, app testing is an activity that has little glory and thrill compared to development, yet it can make a huge difference in the success of your app. Indeed, for many developers—especially those who have been primarily focused on the web, as I expect many readers are—rigorous testing is not one of their skill sets. I think this is because the nature of web development, where you can upload a fix to a site and have it take effect immediately, has not demanded much testing discipline. How often have you seen one of your favorite websites just blow up one day, hobble around for a few hours, and then come back to life? (I've seen this even with Facebook.) It's probably because some developer introduced a nasty bug that was discovered and purged during those hours of awkwardness. For some sites, that downtime can be disastrous, but for many others the impact is small to negligible.

The costliness of bugs in web apps is generally quite small because the update time is also very small. But this is *not* a reality with apps. The time from when you submit an app to the Windows Store to when it's made available averages two days, but it can be longer. This means that each submission is far more significant.

Just look at it in terms of turnaround time. Let's say it takes five minutes to upload a fix to a web app. Compare that to the number of minutes in three days, which is 4320. The ratio? 1 to 864. In other words, it's *nearly hundreds of times more expensive in terms of time and effort to update an app in the Store.* Practically speaking, this means that you might need to spend orders of magnitude more effort testing apps than testing websites. That's significant! (And don't make the argument that because you spend zero time testing web apps the multiple still comes out zero.)

If you don't have some testing methodology in place, start building one, even from the basics. For example, be sure to always test your app on a clean install of Windows on a machine without a developer license, as well as on low-end machines whose performance is similar to many ARM devices. One developer I worked with had an app rejected by the Store because it came up blank on first run—he never saw this happen because of all the cached data on his development machine!

You also want to develop a solid checklist of how to poke and prod your app to exercise all its code paths. This should include subjecting it to all the conditions that come from outside your app: changing view sizes and device orientations; invocation of the different charms; changes in network connectivity; running on slow networks; varying screen sizes and pixel densities; input from different sources; having your temp files cleaned out with the Disk Cleanup tool; signing on with different credentials; suspending, resuming, and restarting after termination; running with high contrast modes and other accessibility features; and running under different languages. The better your app behaves under all these circumstances—which are all those things that real-world users will subject the app to!—the more solid it will look and feel to those same customers who will be writing ratings and reviews.

Having great performance, of course, is also essential, but we already talked about that back in Chapter 3. Be sure to review the Debugging and Testing Windows Store apps and Analyzing the performance of Windows Store apps in the documentation.

The other very important part of testing is running your app through the Windows App Certification Kit, otherwise known as the WACK. This tool—which you can find through the Start screen by searching on "Cert Kit"—subjects your app to all the automated tests that will happen when you onboard to the Store, thereby letting you correct any problems it finds beforehand. Passing the tests in the WACK is no guarantee that your app will be fully certified, but it will certainly save you a great deal of time waiting for onboarding results and having to resubmit over and over.

You should, in fact, run the WACK just about every day during development. You won't necessarily fix everything it brings up immediately, but the ongoing data will be very valuable.

For complete details on the tool and what it does, see Using the Windows App Certification Kit and Windows App Certification Kit tests.

Tip If you find the WACK coming up blank (showing no apps to test), try uninstalling SDK samples that you might have run from Visual Studio. It seems the tool can get overloaded sometimes.

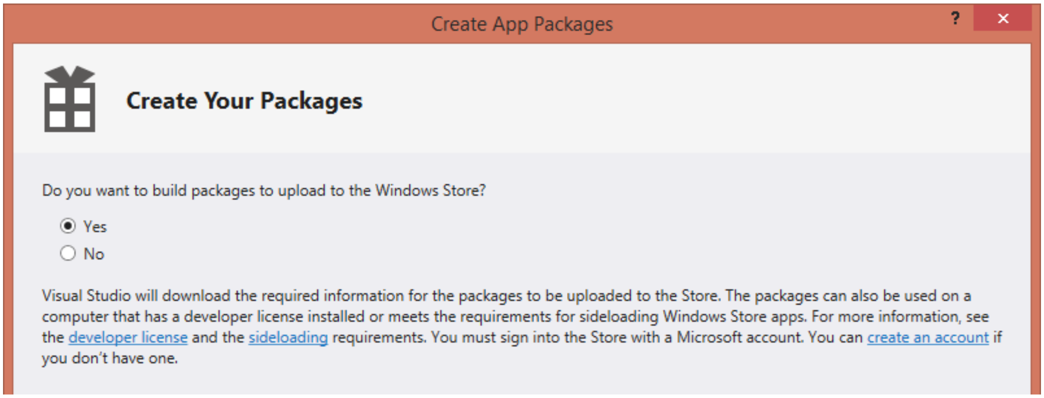
Creating the App Package

When you're confident that your app and its presence in the Store is ready for the world, it's time to finally create an app package for upload! To do this, start the submission process for your app on the

Store dashboard to create an identity with the Store and reserve names for it (including localized names). Then, to make sure the information in your app manifest matches what's in the Store, which is absolutely necessary if you're using push notification and similar services, use the Visual Studio menu command Store > Associate App with Store.

> **Tip** Before building the package, make one last check of the Store Logo image in your manifest's Visual Assets section. This graphic is never used at run time and easily escapes your attention, and yet it's very visible on your app's page in the Windows Store.

Now you can create the app package. For starters, *remember to set your build target to Release* in Visual Studio, and then select the menu command Store > Create App Package. The first dialog that appears (see Figure 20-7) asks whether the package you create will be uploaded. (Note that you can also create a package outside of Visual Studio; see Create an app package at the command prompt.)



Create App Packages

Create Your Packages

Do you want to build packages to upload to the Windows Store?

● Yes

○ No

Visual Studio will download the required information for the packages to be uploaded to the Store. The packages can also be used on a computer that has a developer license installed or meets the requirements for sideloading Windows Store apps. For more information, see the developer license and the sideloading requirements. You must sign into the Store with a Microsoft account. You can create an account if you don't have one.

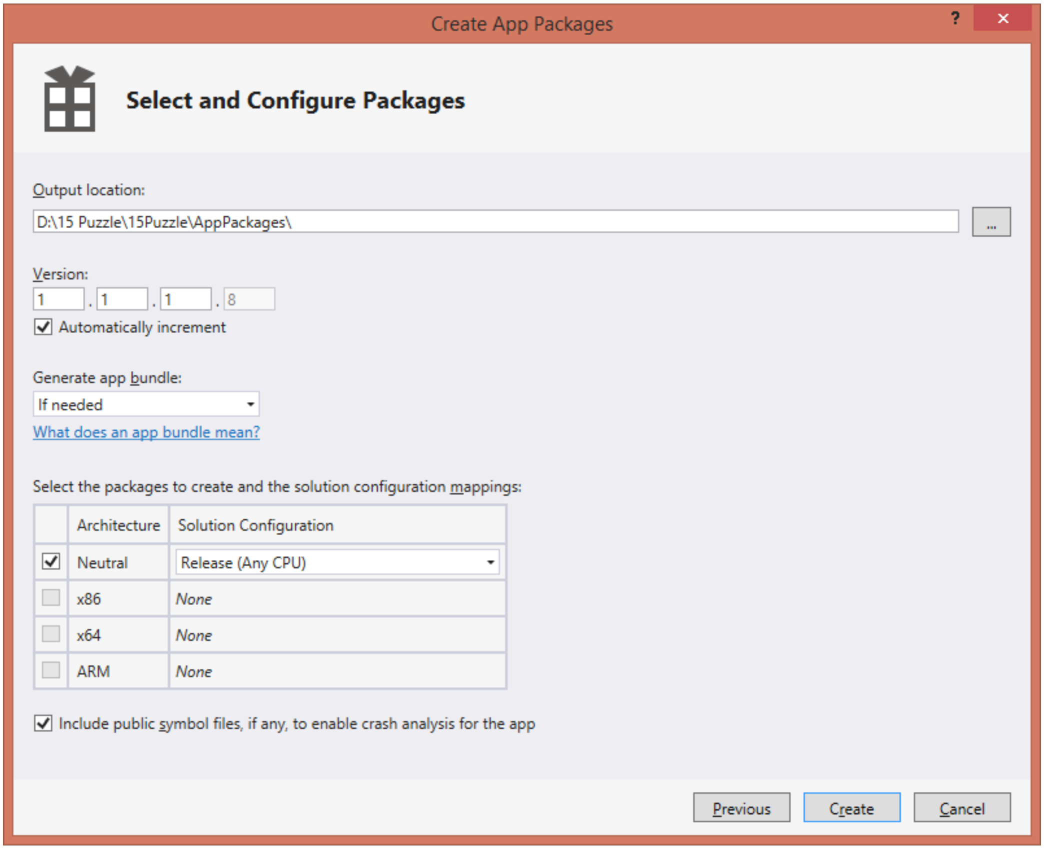
FIGURE 20-7 The first dialog box shown when you create an app package in Visual Studio.

If you want to create packages for side-loading (see "Sidebar: Side-Loading" coming up), select No. If you're planning to submit now, on the other hand, press Yes and click Next, after which you'll be asked to sign in again and then you'll double-check your app's association with those you've created in the Store.

Next you'll specify an output directory for your package and indicate target architectures, if needed, as shown in Figure 20-8. Here's where I want to explain a little about the Generate App Bundle option you see there, which is important if you have localized resources, scaled resources (that is, 100%, 140%, and 180% raster graphics), and resources for different versions of DirectX (which won't happen with JavaScript apps, but I wanted to mention it). As I've noted elsewhere, such as in "Sidebar: Managing Overall Package Size" in Chapter 19, having lots of resources to cover variations in scale, language, and contrast can make your upload package somewhat large (the allowable limit if 8GB). Fortunately, this mostly affects your upload to the Store rather than customer downloads, *if* you select If Needed or Always for Generate App Bundle drop down.

A bundle, in other words, is a way to structure your package so that the Store can selectively download only those resources that a user needs for their particular device configuration. For most apps, selecting If Needed is sufficient because it instructs Visual Studio to handle everything automatically. Choosing between Always and Never applies mostly when you are using C++ code that must be specifically compiled for x86, x64, and ARM (including any WinRT components). Choosing Never means that you'll upload separate architecture-specific packages yourself; selecting Always (or If Needed) will include builds for each architecture in the bundle, saving you some trouble during upload.

> **Note** You can set the default bundle option in your app manifest in the Packaging > Generate App Bundle control.



Select and Configure Packages

Output location:

D:\15 Puzzle\15Puzzle\AppPackages\

Version:

1 . 1 . 1 . 8

☑ Automatically increment

Generate app bundle:

If needed

What does an app bundle mean?

Select the packages to create and the solution configuration mappings:

	Architecture	Solution Configuration
☑	Neutral	Release (Any CPU)
☐	x86	*None*
☐	x64	*None*
☐	ARM	*None*

☑ Include public symbol files, if any, to enable crash analysis for the app

FIGURE 20-8 The second step in creating a package for the Store, where you indicate an output folder, a package version, specify bundle options, and identify architecture targets.

Speaking of compiled components, another aspect of packaging affects the user experience of app acquisition: when a user acquires any new app, the Store works with Windows to automatically check whether the user already any of matching library components installed on their device (checking for exact version matches, file sizes, and so forth). If so—and this applies to JavaScript libraries as much as compiled .dll and .winmd files—Windows shares those components between multiple apps, thereby reducing download time and saving storage space.[137] If an app update requires a newer or different version of a component, of course the Store will bring down the one it needs.

For the most part, this is completely transparent to app developers with one exception: if you're using any kind of third-party library or component that might be used by other apps, *avoid making modifications or recompiling that library*. If you do, you effectively create another variant of the library and Windows will not be able to share it with other apps. To be honest, this doesn't matter much with small JavaScript libraries: it matters most with large frameworks like game engines that can be hundreds of megabytes. Still, keep this in mind for any frameworks that you employ.

In any case, once you press Create in the dialog box shown in Figure 20-7, Visual Studio will build the package in the specified location. When it's done, if you indicated that you wanted a package to upload to the Store, the last dialog box (not shown here) gives you the option to run the WACK as a final check before you upload.

If you go now to the folder where Visual Studio created your package, you'll see an *.appxupload* file (along with a folder ending in *_Test* that contains what you need for side-loading; see "Sidebar: Side-Loading"). As explained in Chapter 1, a package is just a ZIP file, so you can append .zip to the full filename and then explore its contents. What you'll then find is an *.appxbundle* file (if you created a bundle) or an *.appx* file (if you did not bundle). If you copy these out of the ZIP file and rename them to .zip as well, you can see the full package contents.

In the case of .appx, you'll see the structure described in Chapter 1, with your source files, certificates, and the block map. In the .appxbundle, on the other hand, you'll see that it contains a number of individual .appx files organized by scale, language, and DirectX versions, each of which is somewhat smaller than the full upload package. These smaller .appx packages are the ones that the Store will download as needed according to each user's configuration.

Anyway, when you're ready to upload a package, just make sure that you remove the .zip extension from your .appxupload file!

Protecting your code When inspecting the package contents for an app written in JavaScript, you'll see that all your source code is sitting right there, which means that your customers will get all that source code on their machines. Naturally, you'll be asking how you can protect this code from hacks and/or individuals who would unscrupulously "borrow" from you. For the answers, see Dan Reagan's post Protecting your Windows Store app from unauthorized use, which describes some of the steps that Windows does to validate package integrity, along with my blog post Protecting Your Code.

[137] During the production of Windows 8, we investigated whether it made sense to enable separate onboarding of third-party frameworks and library components to the Store, such that the Store would manage dependencies and updates. In the end, we concluded that the risk-to-reward ratio was too high and we enabled this sort of thing only for a few libraries like WinJS. The solution presented here for Windows 8.1 took the previous investigation into account, avoiding the risks while making faster downloads and saving end-user storage space, which were the primary goals.

Sidebar: Side-Loading

Side-loading means installing an app without going through the Store. This is most common for developers (who have a developer license on their machine) to share their apps without sharing their projects, perhaps to get early testing. It's also used in enterprise environments and line-of-business apps where the Store isn't involved. Either way, the process is the same: you create an app package through Visual Studio as already described, just choosing No in Figure 20-7. This will take you through the same configuration step as in Figure 20-8 but skips the step to associate the app with the Store.

Going to the package folder you'll see a folder whose name ends with *_Test*. In that folder you'll see the following:

- The app package (.appx or .appxbundle file).
- A temporary certificate (.cer file).
- A Dependencies folder that contains any libraries that would normally be provided by the Store; WinJS is one such library.
- Most importantly, a PowerShell script named *Add-AppDevPackage.ps1* (and a folder with associated resources) that will install the app on a side-loading-capable machine.

Running the PowerShell script installs the app very much like it would be from the Store, so if testers side-load on a machine where your app has not been installed before, it closely approximates a typical user's environment. This way you can truly test the first-run experience of your app on a variety of devices.

For more information about side-loading, see Deploying enterprise apps and Try It Out: Sideload Windows Store Apps.

Onboarding and Working through Rejection

With your app package in hand, you're now ready to send it out from home for the first time. Uploading your package is done through the Store dashboard in the Packages section for your app. Visual Studio's Store > Upload App Package command, for its part, just takes you to the dashboard.

In the Store, you might find that the Packages section is disabled. This means that you probably still have a few bits to fill in first. If you haven't done so already, be sure to go into your Selling Details to set your price tier, set a trial period (if applicable), select the markets in which you want your app to be available, specify a release date, assign a category and subcategory (if applicable), and check the box for Accessibility if you've honestly done the work we discussed in Chapter 19. See Declaring your app as accessible.

Limiting your app's initial reach? One feature that you'll find missing from the present Windows Store is a way to do beta or preview testing with a limited audience. You can do some of this through side-loading, but that requires each user to have a developer license, which generally means they need to install Visual Studio or at least the Remote Debugging tools. Hardly an attractive requirement! (That said, there are third parties that can help with these arrangements.)

What you can do instead is target only a single market for your initial release—Trinidad and Tobago seems to be the favorite as it's the smallest country in the list! Users in that country will be able to see the app in the Store, but users in other regions will not unless you send them a direct link as part of your preview invitation.

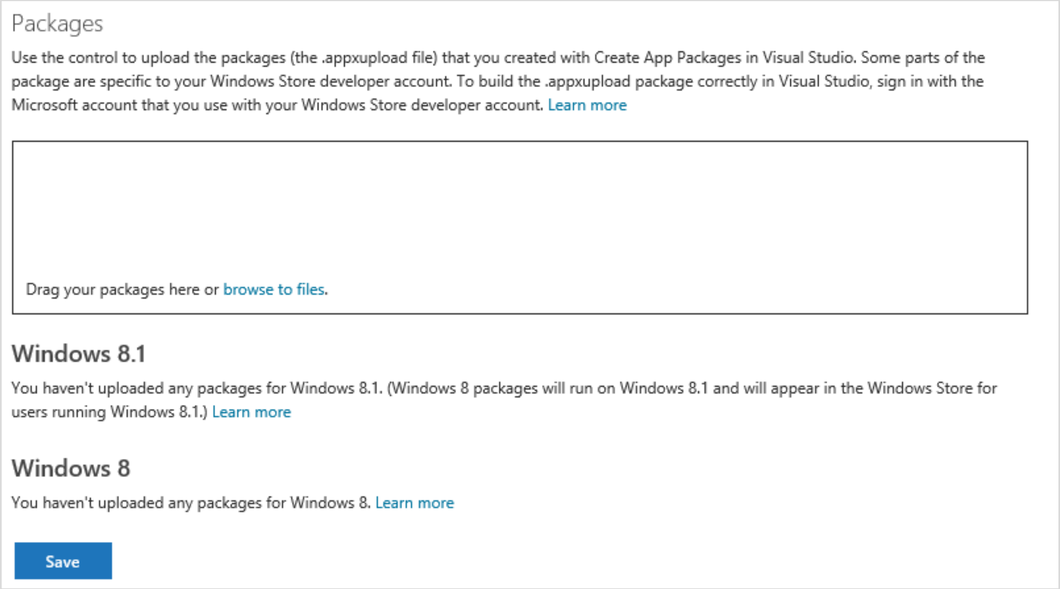
Next go into Services and, if you have nothing to do here, just click Save. Then visit Age Rating and Rating Certificates, *and carefully read the restrictions for each rating*. I emphasis this because the certification testers take age rating very seriously and will reject your app if you don't meet those restrictions. You should also double-check your privacy policy to make sure that you're being clear and forthright about how you collect and/or use personal information—such information is typically the key factor in age ratings. Remember that you *must* have a privacy policy if the app is network-enabled in any way (such as declaring the *Internet (Client)* capability) or uses devices that require consent.

Tip Store requirement 4.1 describes what must be in your privacy statement. Also be sure that your privacy policy is accessible through your apps Settings commands—the lack of this will cause the app to fail certification.

Game rating certificates When uploading an app into the Games category, you'll be required to provide game rating certificates for different markets in the form of a GDF file. For details, see Prepare your Windows game for publishing. You upload these certificates on the Age Rating and Rating Certificates page.

The last piece before you enable the Packages area in the Store dashboard is Cryptography, where you must declare whether your app calls, supports, contains, or uses cryptography or encryption, because your country of origin might have export restrictions.

Finally, you'll get to the Packages section, which might seem anticlimactic after all this! As shown in Figure 20-9, you'll just drag and drop your .appxupload file onto this page. This is also where you can upload both Windows 8 and Windows 8.1 packages. Note that uploading begins as soon as you drag and drop—wait for the upload to complete before clicking Save.


Packages

Use the control to upload the packages (the .appxupload file) that you created with Create App Packages in Visual Studio. Some parts of the package are specific to your Windows Store developer account. To build the .appxupload package correctly in Visual Studio, sign in with the Microsoft account that you use with your Windows Store developer account. Learn more

Drag your packages here or browse to files.

Windows 8.1

You haven't uploaded any packages for Windows 8.1. (Windows 8 packages will run on Windows 8.1 and will appear in the Windows Store for users running Windows 8.1.) Learn more

Windows 8

You haven't uploaded any packages for Windows 8. Learn more

FIGURE 20-9 The Packages page in the Store dashboard.

Next you'll go to the Description page, where you enter all the information that will show up on your app's page in the Store. It's ironic that this page comes so near the end of the onboarding process (and isn't even enabled until you upload a package) because most developers, I imagine, are getting quite anxious by this point to get the app certified! But, please, take a deep breath, relax, and convince yourself that you should invest all the time and energy you need here to make your app shine in the Store. This is exactly why talked about promotional graphics and text copy much earlier—you shouldn't leave these very critical elements of your app to the last minute!

Remember too that you'll have the opportunity to provide versions of all these details for each language that you support. This again includes descriptive text (including any necessary disclosure about keyboard/mouse support for requirement 6.13.4), graphics, descriptions of in-app purchases, website URIs, a link to your privacy policy, and so forth. A full list can be found on the App submission checklist and in Chapter 19 in "Part 3: Creating a Project for Nonpackaged Resources." As I wrote then, pay special attention to your app's keywords and prioritize them for localization—even if you do no other translation work—because customers will be searching the Store in their regional languages.

The very last step in the process is writing notes to testers. Include here any details that are necessary for a real person to exercise your app as part of the certification process, such as credentials for a test account, finding nonobvious features, using background tasks, and so on. And yes, a real human being will look at your app (and read your notes, so be courteous)! Automated tests can only accomplish so much—in the end, someone needs to run the app and make sure it does what it says.

With all the information and packages in place for your app, you're ready to click the Submit for Certification button on the app's page in the dashboard. After that, you'll get an acknowledgment email followed by the joy of waiting for the results! I'm told that typical certification turnaround is about two days. It can be shorter if you happen to submit during a relative lull—when I submitted Here My Am! as part of writing this chapter it went through in less than two hours! Certification can take longer, too, if you're submitting during a busy season or if special circumsta nces will take more time (such as declaring special use capabilities with a company account that require written justifications). Whatever the case, the Store dashboard tries to keep the process transparent.

Once your app has completed the testing process, it will either be accepted or rejected. Acceptance is really a nonissue—that's what you're looking for! The app will then become available either immediately or on the release date that you specified. So congratulations! You're in business!

> **Tip** When your app is certified, you'll get an email with the link to its page in the Store, which looks like *http://apps.microsoft.com/webpdp/app/<app_id>*. You can also retrieve this from the Store dashboard. Click Apps In The Store, and then click Details for that app.

If your app is rejected, on the other hand, the Store will tell you why, specifically citing violations of the certification requirements. Indeed, these policies contain the *only* reasons that an app can be rejected, so any rejection must necessarily indicate the particular requirement that isn't being met. The Store also provides some information about failures, such as where an app crashed (a violation of requirement 1.2).

By and large, most of the policies are straightforward such that if you fail on them, it's pretty clear why. A few, however, seem to be more confusing or subjective, and in the early days of Windows 8 previews they were downright mysterious. Now, fortunately, there is an extensive list of reasons why an app might fail a number of requirements on Resolving certification errors, many of which come from our experience with real apps submitted to the Store. The Store testers can also provide direct feedback regarding specific failures, like where and when an app might have crashed, which certainly caused them to reject it.

Hopefully you only need to correct the problems that are reported and then resubmit your app. If you get rejected multiple times and just can't figure out what to do, I suggest asking questions on the Windows Store forum where you'll find assistance.

App Updates

One thing that apps and books share in common is that the moment you release them, you'll find errors, bugs, typos, and a hundred other things you wish you could change. Fortunately, updating apps is easier than updating books, which I guess makes up for the fact that fixing a typo in a book is much easier than fixing a typical app bug!

There are many reasons to issue an update besides fixing obvious problems and adding features that didn't make it into your current version. As described on Releasing improved versions, you might want to respond to user reviews and requests, add more in-app purchase options, add features to pursue new opportunities, add new language resources, update your promotional information, or update your telemetry layer to improve your analytics. In fact, many insights for your next updates will come from your telemetry.

For nearly all of these purposes, the data that the Store itself gathers will be highly valuable. There is a series of topics in the documentation on this, starting with Understanding app quality and performance. This gets you to the index for applicable subtopics, including Viewing app usage, Reviewing app ratings and feedback, and Tracking app sales. You'll want to start reviewing the data for your app as soon as it's in the Store, and make a plan for monitoring those reports that are most interesting to you. Monitoring ratings and reviews, for example, is an important part of ongoing customer support. In fact, schedule some time in the future right now to check in with these reports, lest you forget they exist. Truly, these reports are your best link to real customers![138] Remember that you published an app, so you're now running a business!

All such data will surely feed back into your planning and development processes, ultimately bringing you to the point of once again uploading another app package and perhaps updating other app information in the Store. A great feature of this in Windows 8.1 is that your update is automatically deployed to all of your users except those that have specifically opted out. To cover the latter, you could have your app check its version number, available from `Windows.ApplicationModel.Package.current.id.version`, against your most recent update as reported by a service, so that you can inform opted-out users that an update is available. You can then direct them to the Store's update page by launching the `ms-windows-store:Updates` URI. If you want to give details about the updates to your users, you can also have your service provide that data.[139]

> **Note** Certification requirement 3.4 states that an app update "must not decrease your app's functionality in a way that would be unexpected to a reasonable consumer." This suggests that you can prune off seldom-used features (as revealed by your telemetry), but you'll fail certification if you take a popular feature of your free app, for example, and start charging for it as an in-app purchase.

When creating the new package, be sure to increment the version number in the manifest. (There's an option to do that automatically in the Visual Studio dialog; refer back to Figure 20-8.) Onboarding is then the same as for any other app—no matter how little you might have changed, the app goes through the whole certification process again (something the Store team is working to improve, by the way). For this reason, don't think to make whimsical updates—make each one count! Also, be aware that the certification requirements change over time, so make it a point to periodically review the requirements.

In your updated app, be prepared to migrate any state that might already exist on the machine, if

[138] These reports will not give you any personal information about your customers, of course. If you want to collect that, you'll need to implement an opt-in registration system in the app that complies with requirement 4.1.2.

[139] It's very easy to create a service endpoint through Windows Azure Mobile Services, rather than having to set up a web host separately. Refer to "Windows Azure and Azure Mobile Services" in Chapter 16.

you've changed state versions. We talked about this in Chapter 10, "The Story of State, Part 1," in "State Versioning," where we distinguished between the version of an app and the version of its state; many app versions can use the same state version. However, if the app now uses a new state version, the old state must be migrated. Remember too that you can use the `servicingComplete` background task for this purpose, as mentioned in Chapter 16 in "Tasks for System Triggers (Non-Lock Screen)." Finally, once you introduce new versions of your state, roaming data will roam between apps of the same version only—you'll be able to migrate old state when the new app is run, but once the state version is increased, that data will no longer roam to devices with apps that use older state.

> **Tip** To test an update, load both new and old app projects into separate instances of Visual Studio. Edit each one's *.user* file to add a `<LayoutDir>[folder]</LayoutDir>` element that points to the same folder in both. Run your old version, exercise the app to establish your state, and then stop it. Then set breakpoints on your activation and migration code in your new version and run that one.

Another key point about updates is that although your new app package might be fairly large, existing customers will *not* have to download the whole thing again. If you go way, way back in this book to Figure 1-1 in Chapter 1, we talked about the package's *blockmap*, which describes how the app package is segmented into 64K blocks. For an update, *only those blocks that have actually changed between versions are necessary to download*. This means that you shouldn't worry about making a critical update to your app: if it affects only a small part of the code, the download impact is as little as 64K. (To help this along, try to have more small files in your project than a few large ones, and it's better to make changes at the ends of files than at the beginning or the middle.)

Other kinds of updates don't involve code but do require recertification. Fortunately, recertification typically goes much quicker, especially if no code changes are involve. One such update is adding a new set of localized resources to your app. Although this means uploading your app package to the Store and recertifying, it won't affect any of your customers except those who are operating in your newly supported language. Another type of update is when you want to change the pricing of the app or in-app purchases or change the app's name. To do this, you just go into your app's page on the Store dashboard, make the necessary changes, and resubmit.

Finally, a kind of nonupdate is if you need to delist the app either temporarily or permanently. You do this through the Store dashboard on the Details page for your app by clicking Manage Availability > Remove This App's Listing. This hides the app in the Store, although existing customers will be able to reinstall it (but cannot make in-app purchases). So long as you don't completely delete the app from your dashboard, you can later click Manage Availability > Restore This App's Listing to make it available once again.

Getting Known: Marketing, Discoverability, and the Web

Long ago, when the Internet and the notion of "search engines" first became popular, many people believed that you could just put up a website to sell whatever trinkets and gazingus pins you happened to have, and customers would come flocking on their own. We might laugh at such folly today, and yet oftentimes we make the same mistake with apps. We mistakenly believe that just getting an app into the Store is enough. Truth is, *the Windows Store is fundamentally a distribution mechanism*, which just so happens to handle transactions and highlight a few apps here and there. And although this is a big improvement over distributing software via the Web, and despite whatever praises we might sing for app stores in general, they haven't really changed what's been true about software for decades: to be successful, you have to write great software *and* you have to invest in marketing.

If you were to set up a small brick-and-mortar shop in your home town, you'd immediately engage in promotional activities of some kind because the business will quickly go under otherwise. And you'd never question that need especially because you've invested so much in getting the business started, and the cost of failure is losing that investment. With apps, you can get something in the Store with a much smaller up-front investment, and so the perceived risk of failure is much lower. As a result, developers aren't as invested in the success of that app as they might be with a real business. But if you want that app to succeed and not just exist, you have to treat it like a business investment.

What's also true in business is that when you start to generate revenue from your initial promotions, part of that stream should go into continued marketing as well as expansion of the business. In other words, some portion of every dollar you earn with an app should be earmarked for marketing activities, because that's how you'll begin to build a sustainable flow. By also earmarking another portion for further development, you're investing in the longer-term sustainability of the business when revenue from the first product begins to wane.

The bottom line is that although apps stores have brought a new distribution method with built-in commerce, running a successful software business requires the same dedication and investment as it always has: have a great product, and invest in the effort to let people know about it.

Toward this end, the Promote section of the documentation has some helpful guidance, some of which we've already covered. It doesn't talk about promotion outside the Store, however, and for that I cannot claim much insight for myself. My best recommendation is to find a marketing specialist who can work for you on an hourly basis initially to get things moving. You can then establish a relationship into which you can direct later investments when you have the funds. In the meantime, there's plenty of material on the web about app marketing, such as The Art of Launching an App: A Case Study (Smashing Magazine).

One important part that you can do that is outside of the Store is making sure you link your app to your website, especially if your app is presenting the same or similar content that can show up in search results. Let's see those details next.

OEM preinstalls One other promotional possibility is that you might be approached by an OEM to include the app preinstalled on their devices. If this happens—it's quite a prize!—the OEM will give you special instructions about how to onboard and maintain an app specifically for their customers. For general details, see Working with OEMs.

Connecting Your Website and Web-Mapped Search Results

If you have an app, you'll almost certainly have a site that provides additional information and support. (Requirement 6.3 deals with support specifically.) What, then, if potential customers come to your site (or some other) first? Surely you'll want to provide easy ways for them to acquire your app if they're running on Windows. The same is also true if they search the web—whether in Internet Explorer or through the Search charm—and find results from your site that could be displayed in your app.

To make a connection between your site and your app, there are first some URIs that get to your app's page in the Store. For browsers, the form is *http://apps.microsoft.com/webpdp/app/<app_id>*. To see your specific URI, go to the Apps in the Store section of the Store dashboard and click Details for the app. You can also get it at run time from `CurrentApp.linkUri`, which is the same link you include with data packages through the Share contract, as covered in Chapter 15.

Another URI scheme, `ms-windows-store`, opens the Store itself on a Windows device, where the specific URIs are described on Linking to your app:

- `ms-windows-store:PDP?PFN=<package_family_name>` Links directly to the app's page; the package family name is what comes back from `Windows.ApplicationModel.Package.current.id.familyName`.
- `ms-windows-store:REVIEW?PFN=<package_family_name>` Goes to the Ratings and Reviews section of the Store for your app, which is what the Rate And Review command in your Settings pane does. You can launch this URI from your app directly when inviting a review.
- `ms-windows-store:Publisher?name=<publisher_display_name>` Links to a page that shows all your apps, which you might do from an About pane in your Settings. The publisher name can be from `Package.current.id.publisher`.
- `ms-windows-store:Updates` Opens the Store's updates page (no arguments). As described earlier in "App Updates," you can use this if you detect that the user is running an older version of your app, which suggest they've opted out of auto-updates.
- `ms-windows-store:Search?query=<search_string>` Executes a search in the Store.

With Internet Explorer, a little bit of metadata in your web page's `<head>` makes it easy for a customer to acquire your app and even run it if the app is already installed. This is fully explained on Connect your website to your Windows Store app, which shows how Internet Explorer lets users know that an app is available. From your point of view, all you need is a bit of markup, such as:

```
<meta name="msApplication-ID" content="14750NuthatchProductions.HereMyAm"/>
<meta name="msApplication-PackageFamilyName" content="14750NuthatchProductions.HereMyAm"/>
```

where the two `content` values come from the Package Name and Package Family Name fields in your manifest's Packaging tab. Again, if the user doesn't have your app, this makes the acquisition process easy. If the user does have your app, he'll have the opportunity to launch it, in which case the app will be activated with the launch kind of `protocol`. If you want to customize the string passed with activation, you can add a `<meta name="msAapplication-Arguments" content="[string]">`. You can similarly specify an `msApplication-MinVersion` for the app and also `msApplication-OptOut` to prevent the behavior if desired.

> **Note** For a working example, visit the site of Inrix Traffic, one of the earliest app partners who implemented these features.

The other step you can take to more deeply connect your app and your website is to support *app linking* through the Search charm. This means that web-based results that show up in Windows search can go straight to your app rather than going to the website first only to redirect the user to the app anyway. To do this, you link your web domain to your app through the Bing webmaster portal, add markup to your web pages to create mappings to deep links in the app, and then enable deep linking by processing the arguments in your site's markup through the app's activation handler, just as you would to support secondary tiles. (See "Secondary Tiles" in Chapter 16.)

I'd say more but full information was not available at the time of writing, so refer to Introducing App Linking for Windows Search.

Face It: You're Running a Business!

In different parts of this chapter I've repeated the sentiment that to publish an app in the Windows Store is to go into business. You might not think of yourself as a businessperson, but publishing an app makes you one by default. Thus, whether you're in business for fame, fortune, fun, or philanthropy, why not acknowledge that reality, make a business plan, and execute on it to give yourself a better chance of fulfilling your goals? Here I want to outline some of the aspects of running a business, primarily for your awareness and inspiration because I cannot personally guarantee your success!

Let's start with a basic truth: customers acquire apps to solve problems and have an experience. They do not acquire apps to pay you money, to support you or your cause, or to make you famous—those are just side effects of their receiving value through your apps. If you do well at solving customer problems and giving them great experiences, the rewards should follow naturally.

For the sake of simplicity, let me focus now on some principles as they apply to the business of fortune, because that's the one that most developers are involved with and one for which there are the most resources to draw from. (Many of the principles can apply to other business models, of course.[140])

[140] Two resources that I found helpful are Thinking Like a Businessperson and Stop Thinking Like an Employee.

Look for Opportunities

In simple terms, business is commerce, meaning a trade or exchange within some given market. A key business skill is to look for opportunities in whatever market you might enter. Look for areas where the market can support another player, or areas that no one has yet explored—analytics services like Distmo, AppFeds, and AppAnnie are super-valuable here. Watch reports on the global app economy too, especially to see regions where profit margins are higher and where you might concentrate localization efforts. Once you identify the opportunity, invest in fulfilling a current market need as soon as you can, before the opportunity passes by.

A good piece of advice is to not enter a market *solely* on personal interest: that can help and serve as a source of inspiration, but focus primarily on picking and solving problems because they are business opportunities, not because you're going to change the world through your efforts. This is not to say that there's no place for your idealism, but it's good to balance that idealism with a strong dose of practicality and reality.

When looking for opportunities, *competitive analysis is a must*. You have to know what others are doing in the market that's similar to you, which helps you identify gaps you can fill, problems you can solve, and the strengths that your solution can uniquely offer. And remember that your competition doesn't sit still, so such analysis is an ongoing process.

Invest in Your Business

Going into business always requires some up-front investment. Ask yourself: what are the core assets you need for your business to sustain itself? It's one thing to get started, but another thing altogether to keep moving and not end up as a "one-shot wonder." This is one reason why you don't need to implement every great idea in the first version of your first app!

Investing also means understanding risk. Look to manage multiple investments to spread your risk, using safe investments to support riskier ones. Investment is also a long-term concern: it's a relationship with your customers, most of all, and also an ongoing relationship with the market in which you're operating. Indeed, with app publishing it's very important to invest in real human relationships. It's fascinating that we've created a marketplace with app stores where you can write and publish an app to solve customer problems without ever having contact with another human being. And yet human beings are behind each and every sale!

Fear Not the Marketing

We've talked a bit about this already, but let me say again that you should plan to constantly steer some of your revenue into marketing to help build and sustain momentum. Otherwise you risk making a splash and then going silent! Social media channels are essential here too, because they're one of the primary ways to keep energy around your apps, but sustaining that energy is always the key.

Cultivating relationships is also important for marketing. Support your customers now and commit to doing so in the future, because they are your foundation. Also make relationships with other app publishers, who can provide mutual support if not cross-promotion opportunities. And get involved in the industry more widely than just your areas of involvement—that is, see yourself as a participant in a larger reality to which you can contribute and from which you can draw energy and inspiration.

Plan also keep your products fresh and lively, just like a merchant does when stocking shelves. In your app, this means bringing in new content, making changes in how you present or highlight in-app purchases, and otherwise helping the user become more away of everything you have to offer. Outside of the app, consider periodically refreshing your screenshots, promotional graphics, description, and anything else on your Store page, if for no other reason than keeping energy around it all. I worked for a while in a retail bookstore where my wife was the gift buyer, and we regularly "fluffed" the shelves which meant going through the entire store to move things around, tidy up areas that have been neglected, and move products that were gathering momentum to more prominent positions. We also made sure to cut losses and discontinue products that weren't working, thereby releasing energy to invest in new areas. In short, don't get attached to products, features, or how you describe and promote them: if something isn't serving the business, an astute businessperson will let it go.

Another marketing trend is to share your expertise to assist buyers, without trying to sell yourself or your products directly. This follows a principle of providing value first before trying to sell, in order to earn trust. For example, you can do reviews of apps in your category, being honest about their strengths, even those of your competitors. By positioning yourself as an expert in that area, customers are more likely to trust what you produce yourself.

Support Your Customers

This almost goes without saying, but many app publishers probably haven't given this any thought to customer support at all. As soon as you publish an app, commit some time every day to watch your ratings, reviews, and other feedback, and be immediately responsive to your customer's needs. Demonstrate that you're listening to their problems, and issue app updates in a timely manner to fix bugs and provide other critical updates. Along these lines, watch your telemetry and crash reports and respond to them quickly as well.

Also think about how you're *thanking* your customers rather than just trying to exploit them or ignoring them in your quest to acquire new customers. In other words, your existing customers are the ones who have already made at least some commitment to you and have given you some of their life energy. How can you give them some energy in return? Truly, it's the *flow* of energy that creates positive magnetism, so keep that flow strong.

Plan for the Future

This can be put simply: it's not necessary to implement every idea in the first version of your app, nor is it necessary to implement every idea in one app. Plan ahead for two major updates, which helps keep up a flow of ideas and will give your customers reasons to re-engage with your product. Remember that apps on Windows 8.1 are updated by default, which makes it much easier to put new features and capabilities in front of your user base, including new in-app purchases. Also remember to educate your customers on what's new in your updates, because they won't be going to the Store to see what you've written in your release notes.

Plan also for having multiple apps. For one, this builds up a presence in the Store for your business, not just for one app. The Store encourages this by listing other apps from the same publisher on any given app's product page, but if you're a one-app shop, you're sacrificing that space to others.

Allowing yourself to spread ideas across multiple apps keeps each app more focused to its primary purpose. Through contracts including URI scheme activations, you can build a family of apps that work great together as well as with apps from other publishers. You can also think about repurposing an existing app, especially games, with new themes that require nothing more than a new set of graphical assets (as with a "holiday" variant that can sell separately). Doing this reduces per-app costs while increasing per-app visibility.

In all of this, I think of how movie studios generally have a multiyear pipeline for their releases. This is necessary in part because it takes several years to produce any given film, but it means those studios are always looking at their business over a five-year or even ten-year timespan. With apps it might be more appropriate to think in terms of 6–18 months, but the principle is the same.

Selling Your App When It's Not Running

User experience is not just what's on the screen when the app is running: there are a number of places where you app has a presence when it's not in the foreground. This include tiles, notifications, raw push notifications, background tasks, your lock screen presence, your roaming experience, and your description page in the Store. Think also about your presence in the media, both for your app and business. Are you blogging? Are you sharing the experience of the app on YouTube (even cheat videos for your game)? Do you provide press kits on your website so that others understand how to promote you and have access to the high-quality assets they'd need for that purpose?

Think also about the overall customer experience lifecycle of your app. When we talk about user experience, we typically refer to the experience inside the app itself, but this is only a small part of the story. How you attract users in the first place is the first experience users will have of your app and your business. How are you presenting yourself in social media channels? And are you giving your existing customers an incentive to share their experience to their social network?

When users first express interest in your app, which means visiting your website and/or your product page in the Store, how are you orienting them to the experience you provide? You want to make sure they understand what the app does and why they should care. Again, you want to invest good time and energy into your app's description and promotional graphics because these make a huge difference to potential customers!

I mentioned before the idea of thanking your customers every now and then. This is important for customer retention; it gives them a reason to come back. That is, an important part of the overall user experience is the long-term value of the relationship, which is expressed through your updates, fresh content, re-engagement with your customers, and maybe even the occasional giveaway.

Indeed, thanking your customers is a way to involve them in an emotionally positive way. And once customers are invested like this, they are more likely to become your advocates or even your champions. Indeed, be sure to enable and empower your most passionate users to put their own credibility on the line. When they do, thank them profusely. Give them special previews of new work you're doing. Send them chocolate. Do anything you can for them, because their passion is worth far more than any bits of money you might get from them as a single user.

You're Not Alone

Publishing apps is something you can do in complete isolation, but that won't serve you or your business in the long run. Given that your primary instincts are those of a developer and probably not those of a businessperson, it's a good idea to find a business mentor to guide and counsel you, or perhaps a support group. Your mentor or associates don't need to be in the software business specifically—what you're looking for is experience and magnetism in *doing business*, which has many universal principles and best practices. Along these same lines, find opportunities like business lunches or other events where you can just mix with businesspeople and draw from their magnetism. More often than not, people at such events are more than happy to discuss your challenges and offer insights that could make a huge difference for you.

Final Thoughts: Qualities of a Rock Star App

As Gandalf the White says to Frodo, Sam, Merry, and Pippin at the conclusion of the *Lord of the Rings* movies, "Here at last, on the shores of the sea, comes the end of our fellowship." And, my friends, it has been a delight to share the journey with you! I'd like to leave you with just a few final thoughts.

After Windows 8 was first released, it didn't take long for the Windows Store to become crowded, so differentiating your app and yourself as a developer is paramount. There's plenty to do with marketing and gaining awareness for your app, of course, as well as being responsive to customers. But beyond that, what does it really mean to make an app that's truly special?

Early on, long before Microsoft landed on the rather pedestrian term "Windows Store apps," we referred to them as "tailored apps." To play with that older term, think of what tailoring means in the context of clothes: well-tailored clothes are very distinctive. They make you look really good. They make you feel *great*. That's how you want the users of your app to feel when they're immersed in your experience. Indeed, just as joy and happiness are the undercurrent behind your own app-building efforts, so also do they live in the hearts of your customers. If you can deliver joy to them through your app, I think you'll have a winner!

Another meaning of "tailored" implies that the kinds of apps we've been building in this book—apps that run full screen (usually) and deeply immerse a user in an experience—lend themselves well to being very specific to both the device and the user's context. As we saw in Chapter 12, "Input and Sensors," sensors give you the ability to know the device's relationship to the physical world, which is an extension of the user who is holding that device. Ask then, "What can I do with that information? How can the app really light up when it has a deeper understanding of where the user is and how the user is moving about in this world of ours? Is there something more the app can do to say, 'Aren't you glad you brought me along?'"

To differentiate your app, think through how a consumer might use various form factors in different situations and have the app present itself differently in those contexts. This kind of tailoring means that the app surfaces the most relevant features or content for the most likely or appropriate use cases. As shown in the last figure of Chapter 1 within "Sidebar: The Opportunity of Per-User Licensing and Data Roaming," I like to think of there being one app across many devices and that the user has a much stronger relationship to the app than to the devices it's running on. The app and its underlying state becomes the consistent element across the whole experience, with the devices just being the vehicles. The more you can deliver an app that understands and supports this (and obviously roaming data is important here!), the more I think the app will stand out from others that, sure, run on Windows but otherwise offer the same experience as we've had for many years.

So, what about being a rock star? Let's be honest here. You're in this game for name and fame, right? And for the big money that could come with it? What kind of app will get you there?

In what is now the very last paragraph of the main body of this book, I can't really give you a bunch of specific ideas. (Otherwise I'd be writing those apps instead of writing books, but someone has to do this dirty work….) But ponder this: what makes a rock star in the music industry? Well, it's not typically about the philosophical depth of the lyrics or the virtuosity of the musicians, it's about performance, personality, and sheer entertainment value. It's about delivering a joyful experience that turns everyday customers into raving lunatic fans who can't wait to be your greatest champions. In a very real way, the experience is one that truly lets people escape their everyday realities and become part of something larger for a time, or even just part of a fantasy. And like great music or movies, the app experience is one that people want to repeat many times over and not just check the box as another "Been there, done that." Although there are certainly aspects of timing and sheer luck, all rock stars—along with great athletes, Oscar-winning movies like *Lord of the Rings*, and so on—strive for and achieve one thing above all: *excellence*. Commit yourself to that. Commit yourself to excellence in everything you do—not just in your apps but in all parts of your life. Such striving, certainly, will bring many rewards!

What We've Just Learned

- An app's relationship to the Windows Store is closely related to your business as a developer, because it supports a range of options from free apps, ad-supported apps, limited-time trials, paid apps, and in-app purchases (using a custom commerce engine for the latter if desired).
- Side-loading of app packages is supported for developers (on a machine with a developer license) and for enterprises. Otherwise all apps come from the Windows Store.
- The Windows Store APIs provide for managing app licenses, licenses for in-app purchases, and

receipts. During development, the app uses a simulator object where data is obtained from a local XML file instead of the live Store, which allows for testing different types of transactions and license conditions.

- To track customer activity within your app directly, instrument it to collect telemetry data according to your business goals for the app. For this you typically use a third-party provider who also turns telemetry data into actionable analytics.
- Getting an app in the Store starts with testing the app both manually and through the Windows App Certification Kit and being prepared for possible rejection during the onboarding process.
- As part of onboarding, you're asked for an app description, promotional graphics, search keywords, feature lists, and other information that should be localized for each supported language. Be sure to give yourself enough time before submission to gather this information.
- App updates can be submitted to the Store at a later time, with improvements based on feedback and telemetry, and the updated code needs to be ready to migrate state.
- Being in the Windows Store does not reduce the need for marketing. Cross-linking your app and website and supporting Windows search can very much help discoverability.
- Publishing an app means that you're running a business, so it's essential that you start thinking (at least a little) like a businessperson to help build your business.

Appendix A

Demystifying Promises

In Chapter 3, "App Anatomy and Performance Fundamentals," we looked at promises that an app typically encounters in the course of working with WinJS and the WinRT APIs. This included working with the then/done methods (and their differences), joining parallel promises, chaining and nesting sequential promises, error handling, and few other features of WinJS promises like the various timeout methods.

Because promises pop up as often as dandelions in a lawn (without being a noxious weed, of course!), it helps to study them more deeply. Otherwise, they and certain code patterns that use them can seem quite mysterious. In this Appendix, we'll first look at the whole backstory, if you will, about what promises are and what they really accomplish, going so far as to implement some promise classes from scratch. We'll then look at WinJS promises specifically and some of the features we didn't see in Chapter 3, such as creating a new instance of WinJS.Promise to encapsulate an async operation of your own. Together, all of this should give you enough knowledge to understand some interesting promises code, as we'll see at the end of this appendix. This will also enable you to understand item-rendering optimizations with the ListView control, as explained in Chapter 7, "Collection Controls."

Demonstrations of what we'll cover here can be found in the WinJS Promise sample of the Windows SDK, which we won't draw from directly, along with the Promises example in the appendices' companion content, which we'll use as our source for code snippets. If you want the fuller backstory on *async* APIs, read Keeping apps fast and fluid with asynchrony in the Windows Runtime on the Windows developer blog. You can also find a combined and condensed version of this material and that from Chapter 3 in my post All about promises (for Windows Store apps written in JavaScript) on that same blog.

Finally, as a bonus, this appendix also deconstructs the batching function used to optimize ListView item rendering, as described in Chapter 7 in the section "Template Functions (Part 2)." It's a bit of promise-heavy code, but it's fascinating to take it apart.

What Is a Promise, Exactly? The Promise Relationships

As noted in the "Using Promises" section of Chapter 3, a promise is just a code construct or a calling convention with no inherent relationship to async operations. Always keep that in mind, because it's easy to think that promises in and of themselves create async behavior. They do not: that's still something you have to do yourself, as we'll see. In other words, as a code construct, a promise is just a combination of functions, statements, and variables that define a specific way to accomplish a task. A *for* loop, for instance, is a programming construct whose purpose is to iterate over a collection. It's a way of saying, "For each item in this collection, perform these actions" (hence the creation of `forEach`!). You use such a construct anytime you need to accomplish this particular purpose, and you know it well because you've practiced it so often!

A promise is really nothing different. It's a particular code structure for a specific purpose: namely, the delivery of some value that might not yet be available. This is why a promise as we see it in code is essentially the same as we find in human relationships: an agreement, in a sense, between the originator of the promise and the consumer or recipient.

In this relationship between originator and consumer there are actually two distinct stages. I call these *creation* and *fulfillment*, which are illustrated in Figure A-1.

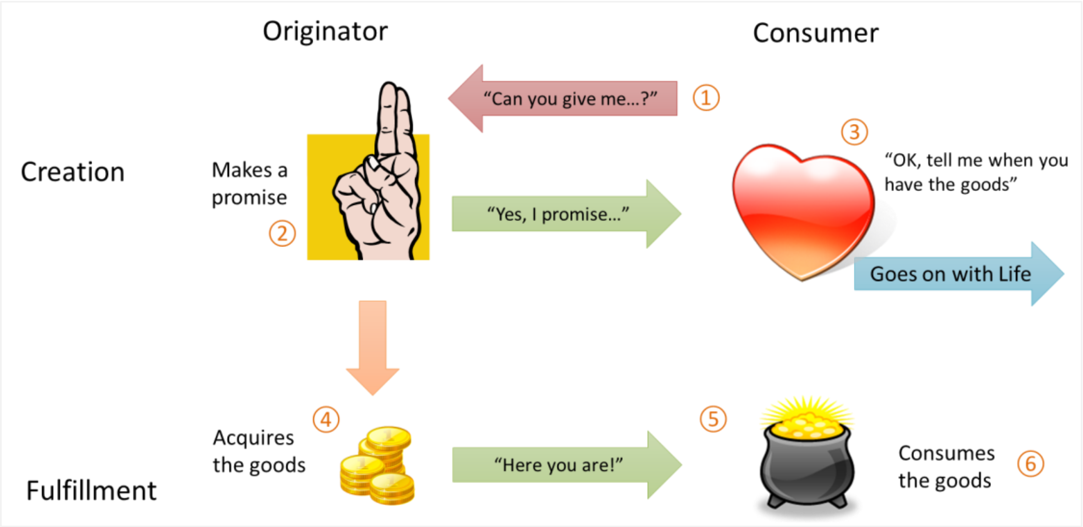


FIGURE A-1 The core relationship encapsulated in a promise.

Having two stages of the relationship is what bring up the asynchronous business. Let's see how by following the flow of the numbers in Figure A-1:

1. The relationship begins when the consumer asks an originator for something: "Can you give me...?" This is what happens when an app calls some API that provides a promise rather than an immediate value.

2. The originator creates a promise for the goods in question and delivers that promise to the consumer.

3. The consumer acknowledges receipt of the promise, telling the originator how the promise should let the consumer know when the goods are ready. It's like saying, "OK, just call this number when you've got them," after which the consumer simply goes on with its life (asynchronously) instead of waiting (synchronously).

4. Meanwhile, the originator works to acquire the promised goods. Perhaps it has to manufacture the goods or acquire them from elsewhere; the relationship here assumes that those goods aren't necessarily sitting around (even though they could be). This is the other place where asynchronous behavior arises, because acquisition can take an indeterminate amount of time.

5. Once the originator has the goods, it brings them to the consumer.

6. The consumer now has what it originally asked for and can consume the goods as desired.

Now, if you're clever enough, you might have noticed that by eliminating a part of the diagram—the stuff around (3) and the arrow that says "Yes, I promise…"—you are left with a simple synchronous delivery model. Which brings us to this point: receiving a promise gives the consumer a chance to do something with its time (like being responsive to other requests), while it waits for the originator to get its act together and deliver the promised goods.

And that, of course, is also the whole point of asynchronous APIs, which is why we use promises with them. It's like the difference (to repeat my example from Chapter 3) between waiting in line at a restaurant's drive-through for a potentially very long time (the synchronous model) and calling out for pizza delivery (the asynchronous model): the latter gives you the freedom to do other things while you're waiting for the delivery of your munchies.

Of course, there's a bit more to the relationship that we have to consider. You've certainly made promises in your life, and you've had promises made to you. Although many of those promises have been fulfilled, the reality is that many promises are broken—it is possible for the pizza delivery person to have an accident on the way to your home! Broken promises are just a fact of life, one that we have to accept, both in our personal lives and in asynchronous programming.

Within the promise relationship, then, this means that originator of a promise first needs a way to say, "Well, I'm sorry, but I can't make good on this promise." Likewise, the recipient needs a way to know that this is the case. Secondly, as consumers, we can sometimes be rather impatient about promises made to us. When a shipping company makes a promise to deliver a package by a certain date, we want to be able to look up the tracking number and see where that package is! So, if the originator can track its progress in fulfilling its promise, the consumer also needs a way to receive that information. And third, the consumer can also tell the originator that it no longer needs whatever it asked for earlier. That is, the consumer needs the ability to cancel the order or request.

This complete relationship is illustrated in Figure A-2. Here we've added the following:

7. While the originator is attempting to acquire the goods, it can let the consumer know what's happening with periodic updates. The consumer can also let the originator know that it no longer needs the promise fulfilled (cancellation).

8. If the originator fails to acquire the goods, it has to apologize with the understanding that there's nothing more it could have done. ("The Internet is down, you know?")

9. If the promise is broken, the consumer has to deal with it as best it can!

With all this in mind, let's see in the next section how these relationships manifest in code.

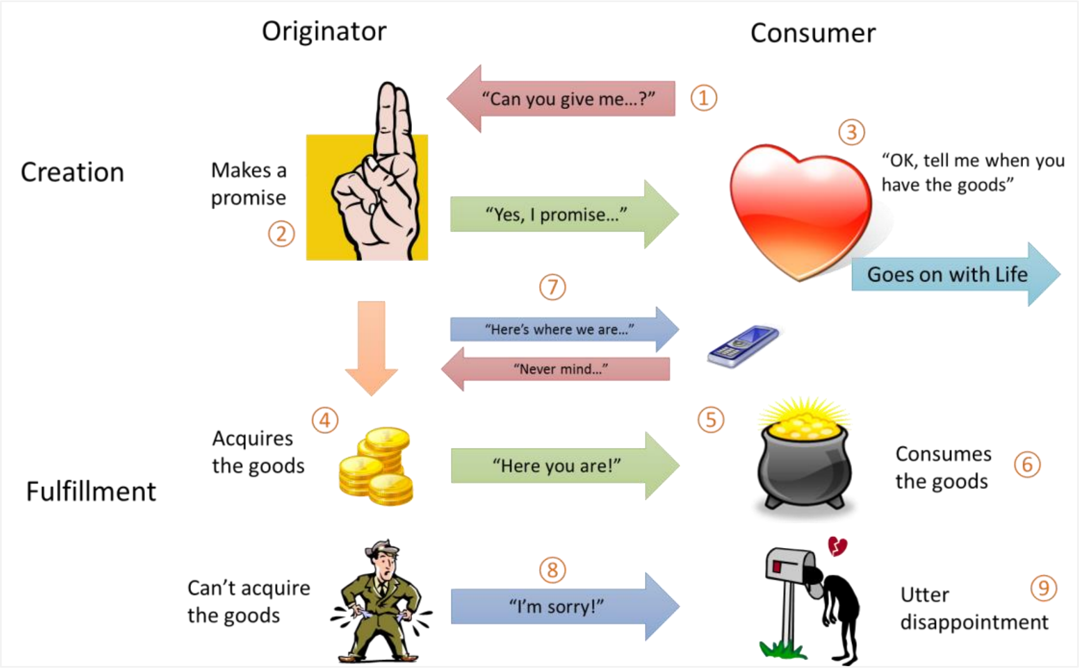


FIGURE A-2 The full promise relationship.

The Promise Construct (Core Relationship)

To fulfill the core relationship of a promise between originator and consumer, we need the following:

- A means to create a promise and attach it to whatever results are involved.
- A means to tell the consumer when the goods are available, which means some kind of callback function into the consumer.

The first requirement suggests that the originator defines an object class of some kind that internally wraps whatever process is needed to obtain the result. An instance of such a class would be created by an asynchronous API and returned to the caller.

For the second requirement, we can take two approaches. One way is to have a simple property on the promise object to which the consumer assigns the callback function. The other is to have a method on the promise to which the consumer passes its callback. Of the two, the latter (using a method) gives the originator more flexibility in how it fulfills that promise, because until a consumer assigns a callback—which is also called *subscribing* to the promise—the originator can hold off on starting the underlying work. You know how it is—there's work you know you need to do, but you just don't get around to it until someone actually gives you a deadline! Using a method call thus tells the originator that the consumer is now truly wanting the results.[141] Until that time, the promise object can simply wait in stasis.

In the definition of a promise that's evolved within the JavaScript community known as Common JS/Promises A (the specification that WinJS and WinRT follow), the method for this second requireme nt is called `then`. In fact, this is the very definition of a promise: *an object that has a property named 'then' whose value is a function.*

That's it. In fact, the static WinJS function `WinJS.Promise.is`, which tests whether a given object is a promise, is implemented as follows:

```
is: function Promise_is(value) {
    return value && typeof value === "object" && typeof value.then === "function";
}
```

Note In Chapter 3 we also saw a similar function called `done` that WinJS and WinRT promises use for error handler purposes. This is not part of the Promises A specification, but it's employed within Windows Store apps.

Within the core relationship, `then` takes one argument: a consumer-implemented callback function known as the *completed handler*. (This is also called a *fulfilled handler*, but I prefer the first term.) Here's how it fits into the core relationship diagram shown earlier (using the same number labels):

1. The consumer calls some API that returns a promise. The specific API in question typically defines the type of object being asked for. In WinRT, for example, the `Geolocator.getGeolocationAsync` method returns a promise whose result is a `Geoposition` object.
2. The originator creates a promise by instantiating an instance of whatever class it employs for its work. So long as that object has a `then` method, it can contain whatever other methods and properties it wants. Again, by definition a promise must have a method called `then`, and this neither requires nor prohibits any other methods and properties.
3. Once the consumer receives the promise and wants to know about fulfillment, it calls `then` to subscribe to the promise, passing a completed handler as the first argument. The promise must internally retain this function (unless the value is already available—see below). Note again that `then` can be called multiple times, by any number of consumers, and the promise must

[141] The method could be a *property setter*, of course; the point here is that a method of some kind is necessary for the object to have a trigger for additional action, something that a passive (non-setter) property lacks.

In this context I'll also share a trick that Chris Sells, who was my manager at Microsoft for a short time, used on me repeatedly. For some given deliverable I owed him, he'd ask, "When will you have it done?" If I said I didn't know, he'd ask, "When will you know when you'll have it done?" If I still couldn't answer that, he'd ask, "When will you know when you'll know when you'll have it done?" *ad infinitum* until he extracted some kind of solid commitment from me!

maintain a list of all those completed handlers.

4. Meanwhile, the originator works to fulfill the promise. For example, the WinRT `getGeolocationAsync` API will be busy retrieving information from the device's sensors or using an IP address–based method to approximate the user's location.

5. When the originator has the result, it has the promise object call all its completed handlers (received through `then`) with the result.

6. Inside its completed handler, the consumer works with the data however it wants.

As you can see, a promise is again *just a programming construct* that manages the relationship between consumer and originator. Nothing more. In fact, it's not necessary that any asynchronous work is involved: a promise can be used with results that are already known. In such cases, the promise just adds the layer of the completed handler, which typically gets called as soon as it's provided to the promise through `then` in step 3 rather than in step 5. While this adds overhead for known values, it allows both synchronous and asynchronous results to be treated identically, which is very beneficial with async programming in general.

To make the core promise construct clear and also to illustrate an asynchronous operation, let's look at a few examples by using a simple promise class of our own as found in the Promises example in the companion content.

Don't do what I'm about to show you Implementing a fully functional promise class on your own gets rather complex when you start addressing all the details, such as the need for `then` to return another promise of its own. For this reason, always use the `WinJS.Promise` class or one from another library that fully implements Promises A and allows you to easily create robust promises for your own asynchronous operations. The examples I'm showing here are strictly for education purposes; they do not implement the full specification.

Example #1: An Empty Promise!

Let's say we have a function, `doSomethingForNothing`, whose results are an empty object, `{ }`, delivered through a promise:

```
//Originator code
function doSomethingForNothing() {
    return new EmptyPromise();
}
```

We'll get to the `EmptyPromise` class in a moment. First, assume that `EmptyPromise` follows the definition and has a `then` method that accepts a completed handler. Here's how we would use the API in the consumer:

```
//Consumer code
var promise = doSomethingForNothing();
```

```
promise.then(function (results) {
    console.log(JSON.stringify(results));
});
```

The output of this would be as follows:

```
{}
```

App.log in the example Although I'm showing console.log in the code snippets here, the Promises sample uses a function `App.log` that ties into the `WinJS.log` mechanism and directs output to the app canvas directly. This is just done so that there's meaningful output in the running example app.

The consumer code could be shortened to just `doSomethingForNothing().then(function (results) { ... });` but we'll keep the promise explicitly visible for clarity. Also note that we can name the argument passed to the completed handler (*results* above) whatever we want; it's just a variable.

Stepping through the consumer code above, we call `promise.then`, and sometime later the anonymous completed handler is called. How long that "sometime later" is, exactly, depends on the nature of the operation in question.

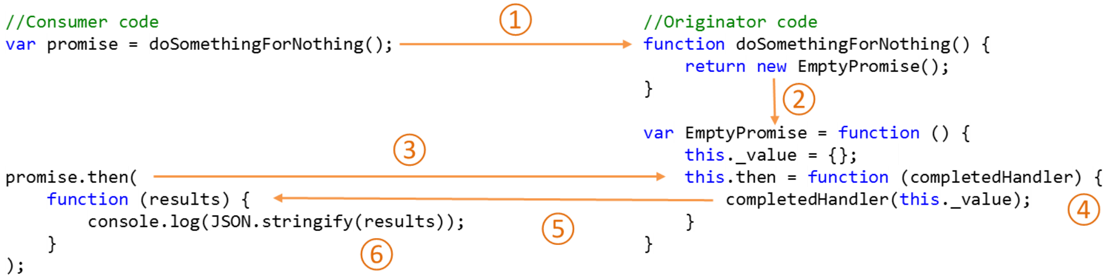
Let's say that `doSomethingForNothing` already knows that it's going to return an empty object for results. In that case, `EmptyPromise` can be implemented as follows (and please, no comments about the best way to do object-oriented JavaScript):

```
//Originator code
var EmptyPromise = function () {
    this._value = {};
    this.then = function (completedHandler) {
        completedHandler(this._value);
    }
}
```

When the originator does a `new` on this constructor, we get back an object that has a `then` method that accepts a completed handler. Because this promise already knows its result, its implementation of `then` just turns around and calls the given completed handler. This works no matter how many times `then` is called, even if the consumer passes that promise to another consumer.[142]

Here's how the code executes, identifying the steps in the core relationship:

[142] The specification for `then`, again, stipulates that its return value is a promise that's fulfilled when the completed handler returns, which is one of the bits that makes the implementation of a promise quite complicated. Clearly, we're ignoring that part of the definition here; we'll return to it in "The Full Promise Construct."



```
//Consumer code
var promise = doSomethingForNothing();

promise.then(
    function (results) {
        console.log(JSON.stringify(results));
    }
);
```

```
//Originator code
function doSomethingForNothing() {
    return new EmptyPromise();
}

var EmptyPromise = function () {
    this._value = {};
    this.then = function (completedHandler) {
        completedHandler(this._value);
    }
}
```

Again, when a promise already knows its results, it can synchronously pass them to whatever completed handler it received through `then`. Here a promise is nothing more than an extra layer that delivers results through a callback rather than directly from a function. For pre-existing results, in other words, a promise is pure overhead. You can see this by placing another `console.log` call after `promise.then`, and you'll see that the `{}` result is logged before `promise.then` returns.

All this is implemented in scenario 1 of the Promises example.

Example #2: An Empty Async Promise

Using promises with known values seems like a way to waste memory and CPU cycles for the sheer joy of it, but promises become much more interesting when those values are obtained asynchronously.

Let's change the earlier `EmptyPromise` class to do this. Instead of calling the completed handler right away, we'll do that after a timeout:

```
var EmptyPromise = function () {
    this._value = { };
    this.then = function (completedHandler) {
        //Simulate async work with a timeout so that we return before calling completedHandler
        setTimeout(completedHandler, 100, this._value);
    }
}
```

With the same consumer code as before and suitable `console.log` calls, we'll see that `promise.then` returns before the completed handler is called. Here's the output:

```
promise created
returned from promise.then
{}
```

Indeed, *all* the synchronous code that follows `promise.then` will execute before the completed handler is called. This is because `setTimeout` has to wait for the app to yield the UI thread, even if that's much longer than the timeout period itself. So, if I do something synchronously obnoxious in the consumer code like the following, as in scenario 2 of the Promises example:

```
var promise = doSomethingForNothing();
console.log("promise created");

promise.then(function (results) {
    console.log(JSON.stringify(results));
});

console.log("returned from promise.then");

//Block UI thread for a longer period than the timeout
var sum = 0;
for (var i = 0; i < 500000; i++) {
    sum += i;
}

console.log("calculated sum = " + sum);
```

the output will be:

```
promise created
returned from promise.then
calculated sum = 124999750000
{}
```

This tells us that for async operation we don't have to worry about completed handlers being called before the current function is done executing. At the same time, if we have multiple completed handlers for different async operations, there's no guarantee about the order in which they'll complete. This is where you need to either nest or chain the operations, as we saw in Chapter 3. There is also a way to use `WinJS.Promise.join` to execute parallel promises but have their results delivered sequentially, as we'll see later.

> **Note** Always keep in mind that while async operations typically spawn additional threads apart from the UI thread, all those results must eventually make their way back to the UI thread. If you have a large number of async operations running in parallel, the callbacks to the completed handlers on the UI thread can cause the app to become somewhat unresponsive. If this is the case, implement strategies to stagger those operations in time (e.g., using `setTimeout` or `setInterval` to separate them into batches).

Example #3: Retrieving Data from a URI

For a more realistic example, let's do some asynchronous work with meaningful results from `XMLHttpRequest`, as demonstrated in scenario 3 of the Promises example:

```
//Originator code
function doXhrGet(uri) {
    return new XhrPromise(uri);
}

var XhrPromise = function (uri) {
    this.then = function (completedHandler) {
        var req = new XMLHttpRequest();
        req.onreadystatechange = function () {
            if (req.readyState === 4) {
                if (req.status >= 200 && req.status < 300) {
                    completedHandler(req);
                }

                req.onreadystatechange = function () { };
            }
        };

        req.open("GET", uri, true);
        req.responseType = "";
        req.send();
    }
}


//Consumer code (note that the promise isn't explicit)
doXhrGet("http://kraigbrockschmidt.com/blog/?feed=rss2").then(function (results) {
    console.log(results.responseText);
});

console.log("returned from promise.then");
```

The key feature in this code is that the asynchronous API we're using within the promise does not itself involve promises, just like our use of `setTimeout` in the second example. `XMLHttpRequest.send` does its work asynchronously but reports status through its `readystatechange` event. So what we're really doing here is wrapping that API-specific async structure inside the more generalized structure of a promise.

It should be fairly obvious that this `XhrPromise` as I've defined it has some limitations—a real wrapper would be much more flexible for HTTP requests. Another problem is that if `then` is called more than once, this implementation will kick off additional HTTP requests rather than sharing the results among multiple consumers. So don't use a class like this—use `WinJS.xhr` instead, from which I shamelessly plagiarized this code in the first place, or the API in `Windows.Web.Http.HttpClient` (see Chapter 4, "Web Content and Services").

Benefits of Promises

Why wrap async operations within promises, as we did in the previous section? Why not just use functions like `XMLHttpRequest` straight up without all the added complexity? And why would we ever want to wrap known values into a promise that ultimately acts synchronously?

First, by wrapping async operations within promises, the consuming code no longer has to know the specific callback structure for each API. Just in the examples we've written so far, methods like `setImmediate`, `setTimeout`, and `setInterval` take a callback function as an argument. `XMLHttpRequest` raises an event instead, so you have to add a callback separately through its `onreadystatechange` property or `addEventListener`. Web workers, similarly, raise a generic `message` event, and other async APIs can pretty much do what they want. By wrapping every such operation with the promise structure, the consuming code becomes much more consistent.

A second reason is that when all async operations are represented by promises—and thus match async operations in the Windows Runtime API and WinJS—we can start combining and composing them in interesting ways. These scenarios are covered in Chapter 3: chaining dependent operations sequentially, joining promises (`WinJS.Promise.join`) to create a single promise that's fulfilled when all the others are fulfilled, or wrapping promises together into a promise that's fulfilled when the first one is fulfilled (`WinJS.Promise.any`).

This is where using `WinJS.Promise.as` to wrap potentially known or existing values within promises becomes useful: they can then be combined with other asynchronous promises. In other words, promises provide a unified way to deal with values whether they're delivered synchronously or asynchronously.

Promises also keep everything much simpler when we start working with the full promise relationship, as described earlier. For one, promises provide a structure wherein multiple consumers can each have their own completed handlers attached to the same promise. A real promise class—unlike the simple ones we've been working with—internally maintains a list of completed handlers and calls all of them when the value is available.

Furthermore, when error and progress handlers enter into the picture, as well as the ability to cancel an async operation through its promise, managing the differences between various async APIs becomes exceptionally cumbersome. Promises standardize all that, including the ability to manage multiple completed/error/progress callbacks along with a consistent `cancel` method.

Let's now look at a more complete promise construct, which will help us understand and appreciate what WinJS provides in its `WinJS.Promise` class!

The Full Promise Construct

Supporting the full promise relationship means that whatever class we use to implement a promise must provide additional capabilities beyond what we've seen so far:

- Support for error and (if appropriate) progress handlers.
- Support for multiple calls to `then`—that is, the promise must maintain an arbitrary number of handlers and share results between multiple consumers.
- Support for cancellation.
- Support for the ability to chain promises.

As an example, let's expand the `XhrPromise` class from Example #3 to support error and progress handlers through its then method, as well as multiple calls to then. This implementation is also a little more robust (allowing null handlers) and supports a cancel method. It can be found in scenario 4 of the Promises example:

```
var XhrPromise = function (uri) {
    this._req = null;

    //Handler lists
    this._cList = [];
    this._eList = [];
    this._pList = [];

    this.then = function (completedHandler, errorHandler, progressHandler) {
        var firstTime = false;
        var that = this;

        //Only create one operation for this promise
        if (!this._req) {
            this._req = new XMLHttpRequest();
            firstTime = true;
        }

        //Save handlers in their respective arrays
        completedHandler && this._cList.push(completedHandler);
        errorHandler && this._eList.push(errorHandler);
        progressHandler && this._pList.push(progressHandler);

        this._req.onreadystatechange = function () {
            var req = that._req;
            if (req._canceled) { return; }

            if (req.readyState === 4) {  //Complete
                if (req.status >= 200 && req.status < 300) {
                    that._cList.forEach(function (handler) {
                        handler(req);
                    });
                } else {
                    that._eList.forEach(function (handler) {
                        handler(req);
                    });
                }

                req.onreadystatechange = function () { };
            } else {
                if (req.readyState === 3) {  //Some data received
                    that._pList.forEach(function (handler) {
                        handler(req);
                    });
                }
            }
        };

        //Only start the operation on the first call to then
        if (firstTime) {
            this._req.open("GET", uri, true);
            this._req.responseType = "";
            this._req.send();
        }
    };

    this.cancel = function () {
        if (this._req != null) {
            this._req._canceled = true;
            this._req.abort;
        }
    }
}
```

The consumer of this promise can now attach multiple handlers, including error and progress as desired:

```
//Consumer code
var promise = doXhrGet("http://kraigbrockschmidt.com/blog/?feed=rss2");
console.log("promise created");

//Listen to promise with all the handlers (as separate functions for clarity)
promise.then(completedHandler, errorHandler, progressHandler);
console.log("returned from first promise.then call");

//Listen again with a second anonymous completed handler, the same error
//handler, and a null progress handler to test then's reentrancy.
promise.then(function (results) {
    console.log("second completed handler called, response length = " + results.response.length);
}, errorHandler, null);

console.log("returned from second promise.then call");


function completedHandler (results) {
    console.log("operation complete, response length = " + results.response.length);
}

function errorHandler (err) {
    console.log("error in request");
}

function progressHandler (partialResult) {
    console.log("progress, response length = " + partialResult.response.length);
}
```

As you can see, the first call to `then` uses distinct functions; the second call just uses an inline anonymous complete handler and passes `null` as the progress handler.

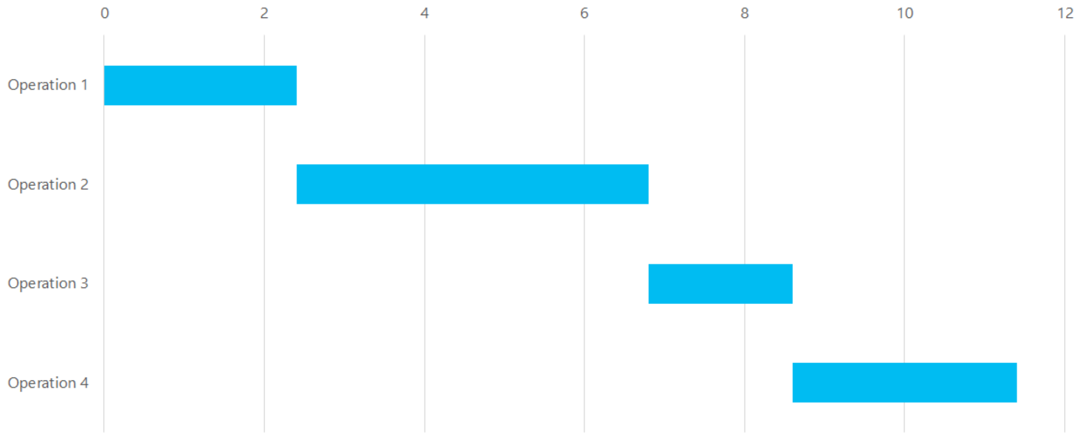
Running this code, you'll see that we pass through all the consumer code first and the first call to `then` starts the operation. The progress handler will be called a number of times and then the completed handlers. The resulting output is as follows (the numbers might change depending on the current blog posts):

```
promise created
returned from first promise.then call
returned from second promise.then call
progress, response length = 4092
progress, response length = 8188
progress, response length = 12284
progress, response length = 16380
progress, response length = 20476
progress, response length = 24572
progress, response length = 28668
progress, response length = 32764
progress, response length = 36860
progress, response length = 40956
progress, response length = 45052
progress, response length = 49148
progress, response length = 53244
progress, response length = 57340
progress, response length = 61436
progress, response length = 65532
progress, response length = 69628
progress, response length = 73724
progress, response length = 73763
operation complete, response length = 73763
second completed handler called, response length = 73763
```

In the promise, you can see that we're using three arrays to track all the handlers sent to `then` and iterate through those lists when making the necessary callbacks. Note that because there can be multiple consumers of the same promise and the same results must be delivered to each, results are considered *immutable*. That is, consumers cannot change those results.

As you can imagine, the code to handle lists of handlers is going to look pretty much the same in just about every promise class, so it makes sense to have some kind of standard implementation into which we can plug the specifics of the operation. As you probably expect by now, `WinJS.Promise` provides exactly this, as well as cancellation, as we'll see later.

Our last concern is supporting the ability to chain promises. Although any number of asynchronous operations can run simultaneously, it's a common need that results from one operation must be obtained before the next operation can begin—namely, when those results are the inputs to the next operation. As we saw in Chapter 2, "QuickStart," this is frequently encountered when doing file I/O in WinRT, where you need obtain a `StorageFile` object, open it, act on the resulting stream, and then flush and close the stream, all of which are async operations. The flow of such operations can be illustrated as follows:



The nesting and chaining constructs that we saw in Chapter 3 can accommodate this need equally, but let's take a closer look at both.

Nesting Promises

One way to perform sequential async operations is to nest the calls that start each operation within the completed handler of the previous one. This actually doesn't require anything special where the promises are concerned.

To see this, let's expand upon that bit of UI-thread-blocking code from Example #2 that did a bunch of counting and turn it into an async operation—see scenario 5 of the Promises example:

```
function calculateIntegerSum(max, step) {
    return new IntegerSummationPromise(max, step);
}

var IntegerSummationPromise = function (max, step) {
    this._sum = null;  //null means we haven't started the operation
    this._cancel = false;

    //Error conditions
    if (max < 1 || step < 1) {
        return null;
    }

    //Handler lists
    this._cList = [];
    this._eList = [];
    this._pList = [];

    this.then = function (completedHandler, errorHandler, progressHandler) {
        //Save handlers in their respective arrays
        completedHandler && this._cList.push(completedHandler);
        errorHandler && this._eList.push(errorHandler);
        progressHandler && this._pList.push(progressHandler);
        var that = this;

        function iterate(args) {
            for (var i = args.start; i < args.end; i++) {
                that._sum += i;
            };

            if (i >= max) {
                //Complete--dispatch results to completed handlers
                that._cList.forEach(function (handler) {
                    handler(that._sum);
                });
            } else {
                //Dispatch intermediate results to progress handlers
                that._pList.forEach(function (handler) {
                    handler(that._sum);
                });

                //Do the next cycle
                setImmediate(iterate, { start: args.end, end: Math.min(args.end + step, max) });
            }
        }

        //Only start the operation on the first call to then
        if (this._sum === null) {
            this._sum = 0;
            setImmediate(iterate, { start: 0, end: Math.min(step, max) });
        }
    };

    this.cancel = function () {
        this._cancel = true;
    }
}
```

The `IntegerSummationPromise` class here is structurally similar to the `XhrPromise` class in scenario 4 to support multiple handlers and cancellation. Its asynchronous nature comes from using `setImmediate` to break up its computational cycles (meaning that it's still running on the UI thread; we'd have to use a web worker to run on a separate thread).

To make sequential async operations interesting, let's say we get our inputs for `calculateIntegerSum` from the following function (completely contrived, of course, with a promise that doesn't support multiple handlers):

```
function getDesiredCount() {
    return new NumberPromise();
}

var NumberPromise = function () {
    this._value = 5000;
    this.then = function (completedHandler) {
        setTimeout(completedHandler, 100, this._value);
    }
}
```

The calling (consumer) code looks like this, where I've eliminated any intermediate variables and named functions:

```
getDesiredCount().then(function (count) {
    console.log("getDesiredCount produced " + count);

    calculateIntegerSum(count, 500).then(function (sum) {
        console.log("calculated sum = " + sum);
    },

    null,  //No error handler

    //Progress handler
    function (partialSum) {
        console.log("partial sum = " + partialSum);
    });
});

console.log("getDesiredCount.then returned");
```

The output of all this appears below, where we can see that the consumer code first executes straight through. Then the completed handler for the first promise is called, in which we start the second operation. That computation reports progress before delivering its final results:

```
getDesiredCount.then returned
getDesiredCount produced 5000
partial sum = 124750
partial sum = 499500
partial sum = 1124250
partial sum = 1999000
partial sum = 3123750
partial sum = 4498500
partial sum = 6123250
partial sum = 7998000
partial sum = 10122750
calculated sum = 12497500
```

Although the consumer code above is manageable with one nested operation, nesting gets messy when more operations are involved:

```
operation1().then(function (result1) {
    operation2(result1).then(function (result2) {
        operation3(result2).then(function (result3) {
            operation4(result3).then(function (result4) {
                operation5(result4).then(function (result5) {
                    //And so on
                });
            });
        });
    });
});
```

Imagine what this would look like if all the completed handlers did other work between each call! It's very easy to get lost amongst all the braces and indents.

For this reason, real promises can also be chained in a way that makes sequential operations cleaner and easier to manage. When more than two operations are involved, chaining is typically the preferred approach.

Chaining Promises

Chaining is made possible by a couple of requirements that a part of the Promises/A spec places on the `then` function:

> *This function [then] should return a new promise that is fulfilled when the given [completedHandler] or errorHandler callback is finished. The value returned from the callback handler is the fulfillment value for the returned promise.*

Parsing this, it means that any implementation of `then` must return a promise whose result is the return value of the completed handler given to `then` (which is particular for each call to `then`). With this characteristic, we can write consumer code in a chained manner that avoids indentation nightmares:

```
operation1().then(function (result1) {
    return operation2(result1)
}).then(function (result2) {
    return operation3(result2);
}).then(function (result3) {
    return operation4(result3);
}).then(function (result4) {
    return operation5(result4)
}).then(function (result5) {
    //And so on
});
```

This structure makes it easier to read the sequence and is generally easier to work with. There's a somewhat obvious flow from one operation to the next, where the `return` for each promise in the chain is essential. Applying this to the nesting example in the previous section (dropping all but the completed hander), we have the following:

```
getDesiredCount().then(function (count) {
    return calculateIntegerSum(count, 500);
}).then(function (sum) {
    console.log("calculated sum = " + sum);
});
```

Conceptually, when we write chained promises like this, we can conveniently think of the return value from one completed handler as the promise that's involved with the next `then` in the chain: the result from `calculateIntegerSum` shows up as the argument `sum` in the next completed handler. We went over this concept in detail in Chapter 2.

However, at the point that getDesiredCount.then returns, we haven't even started calculateIntegerSum yet. This means that whatever promise is returned from getDesiredCount.then is *some other intermediary promise*. This intermediary has its *own* then method and can receive its *own* completed, error, and progress handlers. But instead of waiting directly for some arbitrary asynchronous operation to finish, this intermediate promise is instead waiting on the results of the completed handler given to getDesiredCount.then. That is, the intermediate promise is a child or subsidiary of the promise that created it, such that it can manage its relationship on the parent's completed handler.

Looking back at the code from scenario 5 in the last section, you'll see that none of our then implementations return anything (and are thus incomplete according to the spec). What would it take to make it right?

Simplifying the matter for the moment by not supporting multiple calls to then, a promise class such as NumberPromise would look something like this:

```
var NumberPromise = function () {
    this._value = 1000;
    this.then = function (completedHandler) {
        setTimeout(valueAvailable, 100, this._value);
        var that = this;

        function valueAvailable(value) {
            var retVal = completedHandler(value);
            that._innerPromise.complete(retVal);
        }

        var retVal = new InnerPromise();
        this._innerPromise = retVal;
        return retVal;
    }
}
```

Here, then creates an instance of a promise to which we pass whatever our completed handler gives us. That extra InnerPromise.complete method is the private communication channel through which we tell that inner promise that it can fulfill itself now, which means it calls whatever completed handlers *it* received in its own then.

So InnerPromise might start to look something like this (this is *not* complete):

```
var InnerPromise = function (value) {
    this._value = value;
    this._completedHandler = null;
    var that = this;

    //Internal helper
    this._callComplete = function (value) {
        that._completedHandler && that._completedHandler(value);
    }

    this.then = function (completedHandler) {
        if (that._value) {
            that._callComplete(that._value);
        } else {
            that._completedHandler = completedHandler;
        }
    };

    //This tells us we have our fulfillment value
    this.complete = function (value) {
        that._value = value;
        that._callComplete(value);
    }
}
```

That is, we provide a `then` of our own (still incomplete), which will call its given handler if the value is already known; otherwise it saves the completed handler away (supporting only one such handler). We then assume that our parent promise calls the `complete` method when it gets a return value from whatever completed handler it's holding. When that happens, this `InnerPromise` object can then fulfill itself.

So far so good. However, what happens when the parameter given to `complete` is itself a promise? That means this `InnerPromise` must wait for that other promise to finish, using yet another completed handler. Only then can it fulfill itself. Thus, we need to do something like this within `InnerPromise`:

```
    //Test if a value is a promise
    function isPromise(p) {
        return (p && typeof p === "object" && typeof p.then === "function");
    }

    //This tells us we have our fulfillment value
    this.complete = function (value) {
        that._value = value;

        if (isPromise(value)) {
            value.then(function (finalValue) {
                that._callComplete(value);
            })
        } else {
            that._callComplete(value);
        }
    }
```

We're on a roll now. With this implementation, the consumer code that chains `getDesiredCount` and `calculateIntegerSum` works just fine, where the value of `sum` passed to the second completed handler is exactly what comes back from the computation.

But we still have a problem: `InnerPromise.then` does not itself return a promise, as it should, meaning that the chain dies right here. As such, we cannot chain another operation onto the sequence.

What should `InnerPromise.then` return? Well, in the case where we already have the fulfillment value, we can just return ourselves (which is in the variable that). Otherwise we need to create yet another InnerPromise that's wired up just as we did inside NumberPromise.

```
    this._callComplete = function (value) {
        if (that._completedHandler) {
            var retVal = that._completedHandler(value);
            that._innerPromise.complete(retVal);
        }
    }

    this.then = function (completedHandler) {
        if (that._value) {
            var retVal = that._callComplete(that._value);
            that._innerPromise.complete(retVal);
            return that;
        } else {
            that._completedHandler = completedHandler;

            //Create yet another inner promise for our return value
            var retVal = new InnerPromise();
            this._innerPromise = retVal;
            return retVal;
        }
    };
```

With this in place, InnerPromise supports the kind of chaining we're looking for. You can see this in scenario 6 of the Promises example. In this scenario you'll find two buttons on the page. The first runs this condensed consumer code:

```
getDesiredCount().then(function (count) {
    return calculateIntegerSum(count, 500);
}).then(function (sum1) {
    console.log("calculated first sum = " + sum1);
    return calculateIntegerSum(sum1, 500);
}).then(function (sum2) {
    console.log("calculated second sum = " + sum2);
});
```

where the output is:

```
calculated first sum = 499500
calculated second sum = 124749875250
```

The second button runs the same consumer code but with explicit variables for the promises. It also turns on noisy logging from within the promise classes so that we can see everything that's going on. For this purpose, each promise class is tagged with an identifier so that we can keep track of which is which. I won't show the code, but the output is as follows:

```
p1 obtained, type = NumberPromise
InnerPromise1 created
p1.then returned, p2 obtained, type = InnerPromise1
InnerPromise1.then called
InnerPromise1.then creating new promise
```

```
InnerPromise2 created
p2.then returned, p3 obtained, type = InnerPromise2
InnerPromise2.then called
InnerPromise2.then creating new promise
InnerPromise3 created
p3.then returned (end of chain), returned promise type = InnerPromise3
NumberPromise completed.
p1 fulfilled, count = 1000
InnerPromise1.complete method called
InnerPromise1 calling IntegerSummationPromise.then
IntegerSummationPromise started
IntegerSummationPromise completed
IntegerSummationPromise fulfilled
InnerPromise1 calling completed handler
p2 fulfilled, sum1 = 499500
InnerPromise2.complete method called
InnerPromise2 calling IntegerSummationPromise.then
IntegerSummationPromise started
IntegerSummationPromise completed
IntegerSummationPromise fulfilled
InnerPromise2 calling completed handler
p3 fulfilled, sum2 = 124749875250
InnerPromise3.complete method called
InnerPromise3 calling completed handler
```

This log reveals what's going on in the chain. Because each operation in the sequence is asynchronous, we don't have any solid values to pass to any of the completed handlers yet. But to execute the chain of `then` calls—*which all happens a synchronous sequence*—there has to be some promise in there to do the wiring. That's the purpose of each `InnerPromise` instance. So, in the first part of this log you can see that we're basically creating a stack of these `InnerPromise` instances, each of which is waiting on another.

Once all the `then` methods return and we yield the UI thread, the async operations can start to fulfill themselves. You can see that the `NumberPromise` gets fulfilled, which means that `InnerPromise1` can be fulfilled with the return value from our first completed handler. That happens to be an `IntegerSummationPromise`, so `InnerPromise1` attaches its own completed handler. When that handler is called, `InnerPromise1` can call the second completed handler in the consumer code. The same thing then happens again with `InnerPromise2`, and so on, until the stack of inner promises are all fulfilled. It's at this point that we run out of completed handlers to call and the chain comes to an end.

In short, having `then` methods return another promise to allow chaining basically means that a chain of async operations builds a stack of intermediate promises to manage the connections between as-yet-unfulfilled promises and their completed handlers. As results start to come in, that stack is unwound such that the intermediate results are passed to the appropriate handler so that the next async operation can begin.

Let me be very clear about what we've done so far: the code above shows how chaining really works in the guts of promises, and yet we still have a number of unsolved problems:

- `InnerPromise.then` can manage only a single completed handler and doesn't provide for error and progress handlers.
- There's no provision for handling exceptions in a completed handler, as specified by Promises A.
- There's no provision for cancellation of the chain—namely, that canceling the promise produced by the chain as a whole should also cancel all the other promises involved.
- Some code structures are repeated, so some kind of consolidation seems appropriate.
- This code hasn't been fully tested.

I will openly admit that I'm not right kind of developer to solve such problems—I'm primarily a writer! A number of subtle issues start to arise when you put this kind of thing into practice, but fortunately some software engineers *adore* this kind of a challenge, and fortunately a number of them work in the WinJS team. As a result, they've done all the hard work for us within the `WinJS.Promise` class. And we're now ready to see—and fully appreciate!—what that library provides.

Promises in WinJS (Thank You, Microsoft!)

When writing Windows Store apps in JavaScript, promises pop up anytime an asynchronous API is involved and even at other times. Those promises all meet the necessary specifications, because their underlying classes are supplied by the operating system, which is to say, WinJS. From the consuming code's point of view, then, these promises can be used to their fullest extent possible: nested, chained, joined, and so forth. These promises can also be trusted to manage any number of handlers, to correctly process errors, and basically to handle any other subtleties that might arise.

I say all this because the authors of WinJS have gone to great effort to provide highly robust and complete promise implementations, and this means there is really no need to implement custom promise classes of your own. WinJS provides an extensible means to wrap any kind of async operation within a standardized and well-tested promise structure, so you can focus on the operation and not on the surrounding construct.

We've already covered most of the consumer-side WinJS surface area for promises in Chapter 3, including all the static methods of the `WinJS.Promise` object: `is`, `theneach`, `as`, `timeout`, `join`, and `any`. The latter of these are basically shortcuts to create promises for common scenarios. Scenario 7 of the Promises example gives a short demonstration of many of these.

In Chapter 4 we also saw the `WinJS.xhr` function that wraps `XMLHttpRequest` operations in a promise, which is a much better choice than the wrapper we have in scenario 4 of the Promises example (though we might go all the way and use `Windows.Web.Http.HttpClient` instead). Here, in fact, is the equivalent (and condensed) consumer code from scenario 7 that matches that of scenario 4:

```
var promise = WinJS.xhr("http://kraigbrockschmidt.com/blog/?feed=rss2");
promise.then(function (results) {
    console.log("complete, response length = " + results.response.length);
},
function (err) {
    console.log("error in request: " + JSON.stringify(err));
},
function (partialResult) {
    console.log("progress, response length = " + partialResult.response.length);
});
```

What's left for us to discuss is the instantiable `WinJS.Promise` class itself, with which you can, as an *originator*, easily wrap any async operation of your own in a full promise construct.

> **Note** The entire source code for WinJS promises can be found in its base.js file, accessible through any app project that has a reference to WinJS. (In Visual Studio's Solution Explorer, expand References > Windows Library For JavaScript > js under a project, and you'll see base.js.)

The WinJS.Promise Class

Simply said, `WinJS.Promise` is a generalized promise class that allows you to focus on the nature of a custom async operation, leaving `WinJS.Promise` to deal with the promise construct itself, including implementations of `then` and `cancel` methods, management of handlers, and handling complex cancellation processes involved with promise chains.

As a comparison, in scenario 6 of the Promises example we created distinct promise classes that the `getDesiredCount` and `calculateIntegerSum` functions use to implement their async behavior. All that code got to be rather intricate, which means it will be hard to debug and maintain! With `WinJS.Promise`, we can dispense with those separate promise classes altogether. Instead, we just implement the operations directly within a function like `calculateIntegerSum`. This is how it now looks in scenario 8 (omitting bits of code to handle errors and cancellation, and pulling in the code from default.js where the implementation is shared with other scenarios):

```
function calculateIntegerSum(max, step) {
    //The WinJS.Promise constructor's argument is a function that receives
    //dispatchers for completed, error, and progress cases.
    return new WinJS.Promise(function (completeDispatch, errorDispatch, progressDispatch) {
        var sum = 0;

        function iterate(args) {
            for (var i = args.start; i < args.end; i++) {
                sum += i;
            };

            if (i >= max) {
                //Complete--dispatch results to completed handlers
                completeDispatch(sum);
            } else {
                //Dispatch intermediate results to progress handlers
                progressDispatch(sum);
                setImmediate(iterate, { start: args.end, end: Math.min(args.end + step, max) });
            }
        }

        setImmediate(iterate, { start: 0, end: Math.min(step, max) });
    });
}
```

Clearly, this function still returns a promise, but it's an instance of `WinJS.Promise` that's essentially been configured to perform a specific operation. That "configuration," if you will, is supplied by the function we passed to the `WinJS.Promise` constructor, referred to as the *initializer*. The core of this initializer function does exactly what we did with `IntegerSummationPromise.then` in scenario 6. The great thing is that we don't need to manage all the handlers nor the details of returning another promise from `then`. That's all taken care of for us. Whew!

All we need is a way to tell `WinJS.Promise` when to invoke the completed, error, and progress handlers it's managing on our behalf. That's exactly what's provided by the three dispatcher arguments given to the initializer function. Calling these dispatchers will invoke whatever handlers the promise has received through `then` or `done`, just like we did manually in scenario 6. And again, we no longer need to worry about the structure details of creating a proper promise—we can simply concentrate on the core functionality that's unique to our app.

By the way, a helper function like `WinJS.Promise.timeout` also lets us eliminate a custom promise class like the `NumberPromise` we used in scenario 6 to implement the `getDesiredCount`. We can now just do the following (taken from scenario 8, which matches the quiet output of scenario 6 with a lot less code!):

```
function getDesiredCount() {
    return WinJS.Promise.timeout(100).then(function () { return 1000; });
}
```

To wrap up, a couple of other notes on `WinJS.Promise`:

- Doing `new WinJS.Promise()` (with no parameters) will throw an exception: an initializer function is required.
- If you don't need the `errorDispatcher` and `progressDispatcher` methods, you don't need to declare them as arguments in your function. JavaScript is nice that way!
- Any promise you get from WinJS (or WinRT for that matter) has a standard `cancel` method that cancels any pending async operation within the promise, if cancellation is supported. It has no effect on promises that contain already-known values.
- To support cancellation, the `WinJS.Promise` constructor also takes an optional second argument: a function to call if the promise's `cancel` method is called. Here you halt whatever operation is underway. The full `calculateIntegerSum` function of scenario 8, for example (again, it's in default.js), has a simple function to set a `_cancel` variable that the iteration loop checks before calling its next `setImmediate`.

Originating Errors with WinJS.Promise.WrapError

In Chapter 3 we learned about handling async errors as a consumer of promises and the role of the done method vs. then. When originating a promise, we need to make sure that we cooperate with how all that works.

When implementing an async function, you must handle two different error conditions. One is obvious: you encounter something within the operation that causes it to fail, such as a network timeout, a buffer overrun, the inability to access a resource, and so on. In these cases you call the error dispatcher (the second argument given to your initialization function by the WinJS.Promise constructor), passing an error object that describes the problem. That error object is typically created with WinJS.ErrorFromName (using new), using an error name and a message, but this is not a strict requirement. WinJS.xhr, for example, passes the request object directly to the error handler because that object contains much richer information already.

To contrive an example, if calculateIntegerSum (from default.js) encountered some error while processing its operation, it would do the following:

```
if (false /* replace with any necessary error check -- we don't have any here*/) {
    errorDispatch(new WinJS.ErrorFromName("calculateIntegerSum", "error occurred"));
}
```

The other error condition is more interesting. What happens when a function that normally returns a promise encounters a problem such that it cannot create its usual promise? It can't just return null, because that would make it very difficult to chain promises together. What it needs to do instead is return a promise that *already* contains an error, meaning that it will immediately call any error handlers given to its then.

For this purpose, WinJS has a special function WinJS.Promise.wrapError whose argument is an error object (again typically a WinJS.ErrorFromName). wrapError creates a promise that has no fulfillment value and will never call a completed handler. It will pass its error to any error handler only if you call then or done. Still, its then function must yet return a promise itself; in this case it returns a promise whose fulfillment value is the error object.

For example, if calculateIntegerSum receives max or step arguments that are less than 1, it has to fail and can just return a promise from wrapError (see default.js):

```
if (max < 1 || step < 1) {
    var err = new WinJS.ErrorFromName("calculateIntegerSum (scenario 7)"
        , "max and step must be 1 or greater");
    return WinJS.Promise.wrapError(err);
}
```

The consumer code looks like this, as found in scenario 8:

```
calculateIntegerSum(0, 1).then(function (sum) {
    console.log("calculateIntegerSum(0, 1) fulfilled with " + sum);
}, function (err) {
    console.log("calculateIntegerSum(0, 1) failed with error: '" + err.message +"'");
    return "value returned from error handler";
}).then(function (value) {
    console.log("calculateIntegerSum(0, 1).then fulfilled with: '" + value + "'");
});
```

Some tests in scenario 8 show this output:

```
calculateintegersum(0, 1) failed with error: 'max and step must be 1 or greater'
calculateIntegerSum(0, 1).then fulfilled with: 'value returned from error handler'
```

Another way that an asynchronous operation can fail is by throwing an exception rather than calling the error dispatcher directly. This is important with async WinRT APIs, as those exceptions can occur deep down in the operating system. WinJS accommodates this by wrapping the exception itself into a promise that can then be involved in chaining. The exception just shows up in the consumer's error handler.

Speaking of chaining, WinJS makes sure that errors are propagated through the chain to the error handler given to the last `then`/`done` in the chain, allowing you to consolidate your handling there. This is why promises from `wrapError` are themselves fulfilled with the error value, which they send to their completed handlers instead of the error handlers.

However, because of some subtleties in the JavaScript projection layer for the WinRT APIs, exceptions thrown from async operations within a promise chain will get swallowed and will not surface in that last error handler. Mind you, this doesn't happen with a single promise and a single call to `then`, nor with nested promises, but most of the time the consumer is chaining multiple operations. Such is why we have the `done` method alongside `then`. By using this in the consumer at the end of a promise chain, you ensure that any error in the chain is propagated to the error handler given to `done`.

Some Interesting Promise Code

Finally, now that we've thoroughly explored promises both in and out of WinJS, we're ready to dissect various pieces of code involving promises and understand exactly what they do, beyond the basics of chaining that we've seen.

Delivering a Value in the Future: WinJS.Promise.timeout

To start with a bit of review, the simple `WinJS.Promise.timeout(<n>).then(function () { <value> });` pattern delivers a known value at some time in the future:

```
var p = WinJS.Promise.timeout(1000).then(function () { return 12345; });
```

Of course, you can return another promise inside the first completed handler and chain more `then` calls, which is just an example of standard chaining.

Internals of WinJS.Promise.timeout

The first two cases of `WinJS.Promise.timeout`, `timeout()` and `timeout(n)`, are implemented as follows, using a new instance of `WinJS.Promise` where the initializer calls either `setImmediate` or `setTimeout(n)`:

```
// Used for WinJS.Promise.timeout() and timeout(n)
function timeout(timeoutMS) {
    var id;
    return new WinJS.Promise(
        function (c) {
            if (timeoutMS) {
                id = setTimeout(c, timeoutMS);
            } else {
                setImmediate(c);
            }
        },
        function () {
            if (id) {
                clearTimeout(id);
            }
        }
    );
}
```

The `WinJS.Promise.timeout(n, p)` form is more interesting. As before, it fulfills `p` if it happens within `n` milliseconds; otherwise `p` is canceled. Here's the core of its implementation:

```
function timeoutWithPromise(timeout, promise) {
    var cancelPromise = function () { promise.cancel(); }
    var cancelTimeout = function () { timeout.cancel(); }
    timeout.then(cancelPromise);
    promise.then(cancelTimeout, cancelTimeout);
    return promise;
}
```

The `timeout` argument comes from calling `WinJS.Promise.timeout(n)`, and `promise` is the `p` from `WinJS.Promise.timeout(n, p)`. As you can see, `promise` is just returned directly. However, see how the promise and the timeout are wired together. If the `timeout` promise is fulfilled first, it calls `cancelPromise` to cancel `promise`. On the flipside, if `promise` is fulfilled first or encounters an error, it calls `cancelTimeout` to cancel the timer.

Parallel Requests to a List of URIs

If you need to retrieve information from multiple URIs in parallel, here's a little snippet that gets a `WinJS.xhr` promise for each and joins them together:

```
// uris is an array of URI strings
WinJS.Promise.join(
    uris.map(function (uri) { return WinJS.xhr({ url: uri }); })
).then(function (results) {
    results.forEach(function (result, i) {
        console.log("uri: " + uris[i] + ", " + result);
    });
});
```

The array `map` method simply generates a new array with the results of the function you give it applied to each item in the original array. This new array becomes the argument to `join`, which is fulfilled with an array of results.

You can of course replace `WinJS.xhr` with `Windows.Web.Http.HttpClient.get*` methods (see Chapter 4 in the section "Using Windows.Web.Http.HttpClient") or use any other array of input values for `uris` and any other async method for that matter.

Parallel Promises with Sequential Results

`WinJS.Promise.join` and `WinJS.Promise.any` work with parallel promises—that is, with parallel async operations. The promise returned by `join` will be fulfilled when all the promises in an array are fulfilled. However, those individual promises can themselves be fulfilled in any given order. What if you have a set of operations that can execute in parallel but you want to process their results in a well-defined order—namely, the order that their promises appear in an array?

The trick is to basically join each subsequent promise to all of those that come before it, and the following bit of code does exactly that. Here, `list` is an array of values of some sort that are used as arguments for some promise-producing async call that I call `doOperation`:

```
list.reduce(function callback (prev, item, i) {
    var result = doOperation(item);
    return WinJS.Promise.join({ prev: prev, result: result}).then(function (v) {
        console.log(i + ", item: " + item+ ", " + v.result);
    });
})
```

To understand this code, we have to first understand how the array's reduce method works, because it's slightly tricky. For each item in the array, `reduce` calls the function you provide as its argument, which I've named `callback` for clarity. This `callback` receives four arguments (only three of which are used in the code):

- `prev` The value that's returned from the *previous* call to `callback`. For the first item, `prev` is `null`.
- `item` The current value from the array.
- `i` The index of item in the list.
- `source` The original array.

It's also important to remember that `WinJS.Promise.join` can accept a list in the form of an object, as shown here, as well as an array (it uses Object.keys(list).forEach to iterate).

To make this code clearer, it helps to write it out with explicit promises:

```
list.reduce(function callback (prev, item, i) {
    var opPromise = doOperation(item);
    var join = WinJS.Promise.join({ prev: prev, result: opPromise});

    return join.then(function completed (v) {
        console.log(i + ", item: " + item+ ", " + v.result);
    });
})
```

By tracing through this code for a few items in list, we'll see how we build the sequential dependencies.

For the first item in the list, we get its opPromise and then join it with whatever is contained in prev. For this first item prev is null, so we're essentially joining, to express it in terms of an array, [WinJS.Promise.as(null), opPromise]. But notice that we're not returning join itself. Instead, we're attaching a completed handler (which I've called completed) to that join and returning the promise from its then.

Remember that the promise returned from then will be fulfilled when the completed handler returns. This means that what we're returning from callback is a promise that's not completed until the first item's completed handler has processed the results from opPromise. And if you look back at the result of a join, it's fulfilled with an object that contains the results from the promises in the original list. That means that the fulfillment value v will contain both a prev property and a result property, the values of which will be the values from prev (which is null) and opPromise. Therefore v.result is the result of opPromise.

Now see what happens for the next item in list. When callback is invoked this time, prev contains the promise from the previous join.then. So, in the second pass through callback we create a new join of opPromise$_2$ and opPromise$_1$.then. As a result, this join will not complete until both opPromise$_2$ is fullfilled *and* the completed handler for opPromise$_1$ returns. Voila! The completed$_2$ handler we now attach to this join will not be called until after completed$_1$ has returned.

In short, the same dependencies continue to be built up for each item in list—the promise from join.then for item *n* will not be fulfilled until completed$_n$ returns, and we've guaranteed that the completed handlers will be called in the same sequence as list.

A working example of this construct using calculateIntegerSum and an array of numbers can be found in scenario 9 of the Promises example. The numbers are intentionally set so that some of the calculations will finish before others, but you can see that the results are delivered in order.

Constructing a Sequential Promise Chain from an Array

A similar construct to the one in the previous section is to use the array's `reduce` method to build up a promise chain from an array of input arguments such that each async operation doesn't *start* before the previous one is complete. Scenario 10 of the Promises example demonstrates this:

```
//This just avoids having the prefix everywhere
var calculateIntegerSum = App.calculateIntegerSum;

//This op function attached other arguments to each call
var op = function (arg) { return calculateIntegerSum(arg, 100); };

//The arguments we want to process
var args = [1000000, 500000, 300000, 150000, 50000, 10000];

//This code creates a parameterized promise chain from the array of args and the async call
//in op. By using WinJS.Promise.as for the initializer we can just call p.then inside
//the callback.
var endPromise = args.reduce(function (p, arg) {
    return p.then(function (r) {
        //The first result from WinJS.Promise.as will be undefined, so skip logging
        if (r !== undefined) { App.log("operation completed with results = " + r) };

        return op(arg);
    });
}, WinJS.Promise.as());

//endPromise is fulfilled with the last operation's results when the whole chain is complete.
endPromise.done(function (r) {
    App.log("Final promise complete with results = " + r);
});
```

PageControlNavigator._navigating (Page Control Rendering)

The final piece of code we'll look at in this appendix comes from the navigator.js file that's included with the Visual Studio templates that employ WinJS page controls. This is an event handler for the `WinJS.Navigation.onnavigating` event, and it performs the actual loading of the target page (using `WinJS.UI.Pages.render` to load it into a newly created `div`, which is then appended to the DOM) and unloading of the current page (by removing it from the DOM):

```
_navigating: function (args) {
    var newElement = this._createPageElement();
    var parentedComplete;
    var parented = new WinJS.Promise(function (c) { parentedComplete = c; });

    this._lastNavigationPromise.cancel();

    this._lastNavigationPromise = WinJS.Promise.timeout().then(function () {
        return WinJS.UI.Pages.render(args.detail.location, newElement,
           args.detail.state, parented);
    }).then(function parentElement(control) {
            var oldElement = this.pageElement;
            if (oldElement.winControl && oldElement.winControl.unload) {
                oldElement.winControl.unload();
            }
            WinJS.Utilities.disposeSubTree(this._element);
            this._element.appendChild(newElement);
            this._element.removeChild(oldElement);
            oldElement.innerText = "";
            parentedComplete();
        }.bind(this));

        args.detail.setPromise(this._lastNavigationPromise);
    },
```

First of all, this code cancels any previous navigation that might be happening, then creates a new one for the current navigation. The `args.detail.setPromise` call at the end is the WinJS deferral mechanism that's used in a number of places. It tells `WinJS.Navigation.onnavigating` to defer its default process until the given promise is fulfilled. In this case, WinJS waits for this promise to be fulfilled before raising the subsequent `navigated` event.

Anyway, the promise in question here is what's produced by a `WinJS.Promise.timeout().then().then()` sequence. Starting with a `timeout()` promise means that the process of rendering a page control first yields the UI thread via `setImmediate`, allowing other work to complete before we start the rendering process.

After yielding, we then enter into the first completed handler that starts rendering the new page control into `newElement` with `WinJS.UI.Pages.render`. Rendering is an async operation itself (it involves a file loading operation, for one thing), so `render` returns a promise. At this point, `newElement` is an orphan—it's not yet part of the DOM, just an object in memory—so all this rendering is just a matter of loading up the page control's contents and building that standalone chunk of DOM.

When `render` completes, the next completed handler in the chain, which is actually named `parentElement` ("parent" in this case being a verb), receives the newly loaded page `control` object. This code doesn't make use of this argument, however, because it knows that it's the contents of `newElement` (`newElement.winControl`, to be precise). So we now unload any page control that's currently loaded (`that.pageElement.winControl`), calling its `unload` method, if available, and also making sure to free up event listeners and whatnot with `WinJS.Utilities.disposeSubtree`. Then we

can attach the new page's contents to the DOM and remove the previous page's contents. This means that the new page contents will appear in place of the old the next time the rendering engine gets a chance to do its thing.

Finally, we call this function `parentedComplete`. This last bit is really a wiring job so that WinJS will not invoke the new page's `ready` method until it's been actually added to the DOM. This means that we need a way for WinJS to hold off making that call until parenting has finished.

Earlier in _navigating, we created a parentedPromise variable, which was then given as the fourth parameter to WinJS.UI.Pages.render. This parentedPromise is very simple: we're just calling new WinJS.Promise and doing nothing more than saving its completed dispatcher in the parented-Complete variable, which is what we call at the end of the process.

For this to serve any purpose, of course, someone needs to call parentedPromise.then and attach a completed handler. A WinJS page control does this, and all its completed handler does is call ready. Here's how it looks in base.js:

```
this.renderComplete.then(function () {
    return parentedPromise;
}).then(function Pages_ready() {
    that.ready(element, options);
})
```

In the end, this whole _navigating code is just saying, "After yielding the UI thread, asynchronously load up the new page's HTML, add it to the DOM, clean out and remove the old page from the DOM, and tell WinJS that it can call the new page's ready method, because we're not calling it directly ourselves."

Bonus: Deconstructing the ListView Batching Renderer

The last section of Chapter 7, titled "Template Functions (Part 2): Optimizing Item Rendering," talks about the *multistage batching renderer* found in scenario 1 of the HTML ListView optimizing performance sample. The core of the renderer is a function named thumbnailBatch, whose purpose is to return a completed handler for an async promise chain in the renderer:

```
renderComplete: itemPromise.then(function (i) {
    item = i;
    // ...
    return item.ready;
}).then(function () {
    return item.loadImage(item.data.thumbnail);
}).then(thumbnailBatch()
).then(function (newimg) {
    img = newimg;
    element.insertBefore(img, element.firstElementChild);
    return item.isOnScreen();
}).then(function (onscreen) {
    //...
})
```

To understand how this works, we first need a little more background as to what we're trying to accomplish. If we just had a ListView with a single item, various loading optimizations wouldn't be noticeable. But ListViews typically have many items, and the rendering function is called for each one. In the *multistage renderer* described in Chapter 7, the rendering of each item kicks off an async item.loadImage operation to download its thumbnail from an arbitrary URI, and each operation can take an arbitrary amount of time. For the list as a whole, we might have a bunch of simultaneous loadImage calls going on, with the rendering of each item waiting on the completion of its particular thumbnail. So far so good.

An important characteristic that's not at all visible in the multistage renderer, however, is that the img element for the thumbnail is *already* in the DOM, and the loadImage function will set that image's src attribute as soon as the download has finished. This in turn triggers an update in the rendering engine as soon as we return from the rest of the promise chain.

It's possible, then, that a bunch of thumbnails could come back to the UI thread within a short amount of time. This will cause excess churn in the rendering engine and poor visual performance. To avoid this churn, we need to create and initialize the img elements *before* they're in the DOM, and then we need to add them in batches such that they're all handled in a single rendering pass.

This is the purpose of the function in the sample called createBatch, which is called just once to produce the oft-used thumbnailBatch function:

```
var thumbnailBatch;
thumbnailBatch = createBatch();
```

As shown above, a call to this thumbnailBatch function, as I'll refer to it from here on, is inserted into the promise chain of the renderer. This purpose of this insertion, given the nature of the batchi ng code that we'll see shortly, is to group together a set of loaded img elements, releasing them for further processing at suitable intervals. Again, just looking at the promise chain in the renderer, a call to thumbnailBatch() *must* return a completed handler function that accepts, as a result, whatever loadImage produces (an img element). In this case, loadImage is creating that img element for us, unlike the previous multistage renderer that uses one that's already in the DOM.

Because the call to thumbnailBatch is in a chain, the completed handler it produces must also return a promise whose the fulfillment value (looking at the next step in the chain) must be an img element that can *then* be added to the DOM as part of a batch.

Now let's see how that batching works. Here's the createBatch function in its entirety:

```
function createBatch(waitPeriod) {
    var batchTimeout = WinJS.Promise.as();
    var batchedItems = [];

    function completeBatch() {
        var callbacks = batchedItems;
        batchedItems = [];
        for (var i = 0; i < callbacks.length; i++) {
            callbacks[i]();
        }
    }

    return function () {
        batchTimeout.cancel();
        batchTimeout = WinJS.Promise.timeout(waitPeriod || 64).then(completeBatch);
        var delayedPromise = new WinJS.Promise(function (c) {
            batchedItems.push(c);
        });

        return function (v) {
            return delayedPromise.then(function () {
                return v;
            });
        };
    };
}
```

Again, createBatch is called just *once* and its thumbnailBatch result is called for *every item in the list*. The completed handler that thumbnailBatch generates is then called whenever a loadImage operation completes.

Such a completed handler might just as easily have been inserted directly into the rendering function, but what we're trying to do here is coordinate activities *across multiple items* rather than just on a per-item basis. This coordination is achieved through the two variables created and initialized at the beginning of createBatch: batchedTimeout, initialized as an empty promise, and batchedItems, initialized an array of functions that's initially empty. createBatch also declares a function, completeBatch, that simply empties batchedItems, calling each function in the array:

```
    function completeBatch() {
        //Copy and clear the array so that the next batch can start to accumulate
        //while we're processing the previous one.
        var callbacks = batchedItems;
        batchedItems = [];
        for (var i = 0; i < callbacks.length; i++) {
            callbacks[i]();
        }
    }
```

Now let's see what happens within thumbnailBatch (the function returned from createBatch), which is again called for each item being rendered. First, we cancel any existing batchedTimeout and immediately re-create it:

```
        batchTimeout.cancel();
        batchTimeout = WinJS.Promise.timeout(waitPeriod || 64).then(completeBatch);
```

The second line shows the future delivery/fulfillment pattern discussed earlier: it says to call completeBatch after a delay of *waitPeriod* milliseconds (with a default of 64ms). This means that so long as thumbnailBatch is being called again within *waitPeriod* of a previous call, batchTimeout will be reset to another *waitPeriod*. And because thumbnailBatch is called only *after* an item.loadImage call completes, we're effectively saying that any loadImage operations that complete within *waitPeriod* of the previous one will be included in the same batch. When there's a gap longer than *waitPeriod*, the batch is processed (images are added to the DOM) and the next batch begins.

After handling this timeout business, thumbnailBatch creates a new promise that simply pushes the complete dispatcher function into the batchedItems array:

```
var delayedPromise = new WinJS.Promise(function (c) {
   batchedItems.push(c);
});
```

Remember that a promise is just a code construct, and that's all we have here. The newly created promise has no async behavior in and of itself: we're just adding the complete dispatcher function, `c`, to `batchedItems`. But, of course, we don't do anything with the dispatcher until `batchedTimeout` completes (asynchronously), so there is in fact an async relationship here. When the timeout happens and we clear the batch (inside `completeBatch`), we'll invoke any completed handlers given elsewhere to `delayedPromise.then`.

This brings us to the last line of code in `createBatch`, which is the function that `thumbnailBatch` actually returns. This function is exactly the completed handler that gets inserted into the renderer's whole promise chain:

```
return function (v) {
   return delayedPromise.then(function () {
      return v;
   });
};
```

In fact, let's put this piece of code directly into the promise chain so that we can see the resulting relationships:

```
      return item.loadImage(item.data.thumbnail);
   }).then(function (v) {
      return delayedPromise.then(function () {
         return v;
      });
   ).then(function (newimg) {
```

Now we can see that the argument `v` is the result of `item.loadImage`, which is the `img` element created for us. If we didn't want to do batching, we could just say `return WinJS.Promise.as(v)` and the whole chain would still work: `v` would then be passed on synchronously and show up as `newimg` in the next step.

Instead, however, we're returning a promise from `delayedPromise.then` that won't be fulfilled—with `v`—until the current `batchedTimeout` is fulfilled. At that time—when again there's a gap of *waitPeriod* between `loadImage` completions—those `img` elements are delivered to the next step in the chain where they're added to the DOM.

And that's it. Now you know!

Appendix B

WinJS Extras

In this appendix:

- Exploring WinJS.Class Patterns
- Obscure WinJS Features
- Extended Splash Screens
- Custom Layouts for the ListView Control

Exploring WinJS.Class Patterns

Much has been written in the community about object-oriented JavaScript, and various patterns have emerged to accomplish with JavaScript's flexible nature the kinds of behaviors that are built directly into languages like C++. Having done more C++ than functional programming in the past, I've found the object-oriented approach to be more familiar and comfortable. If you are in the same camp, you'll find that the WinJS.Class API encapsulates much of what you need.

WinJS.Class.define

One place object-oriented programmers quickly hit a bump in JavaScript is the lack of a clear "class" construct—that is, the definition of a type of object that can be instantiated. JavaScript, in short, has no *class* keyword. Instead, it has functions, and only functions. Conveniently, however, if you create a function that populates members of its this variable, the name of that function works exactly like the name of a class such that you can create an instance with the new operator.

To borrow a snippet of code from *Eloquent JavaScript* (a book I like for its depth and brevity, and which is also available for free in electronic form—thank you Marijn!), take the following function named *Rabbit*:

```
function Rabbit(adjective) {
    this.adjective = adjective;
    this.speak = function (line) {
        print("The ", this.adjective, " rabbit says '", line, "'");
    };
}
```

By itself, this function doesn't do anything meaningful. You can call it from wherever and this will end up being your global context. Maybe handy if you like to pollute the global namespace!

When used with the new operator, on the other hand, the function becomes an object constructor and effectively defines a class as we know in object-oriented programming. For example:

```
var fuzzyBunny = new Rabbit("cutie");
fuzzyBunny.speak("nibble, nibble!");
```

As Marijn points out, using new with a constructor function like this has a nice side effect that assigns the object's constructor property with the constructor function itself. This is not true if you just create a function that returns a newly instantiated object.

To make our class even more object-oriented, the other thing that we typically do is assign default values to properties and assign methods within the class, rather than on individual instances. In the first example above, each instance gets a new copy of the anonymous function assigned to speak. It would be better to define that function such that a single copy is used by the different instances of the class. This is accomplished by assigning the function to the class's prototype:

```
function Rabbit(adjective) {
    this.adjective = adjective;
}
Rabbit.prototype.speak = function (line) {
    print("The ", this.adjective, " rabbit says '", line, "'");
};
```

Of course, having to write this syntax out for each and every member of the class that's shared between instances is both cumbersome and prone to errors. Personally, I also like to avoid messing with prototype because you can really hurt yourself if you're not careful.

WinJS provides a helper that provides a cleaner syntax as well as clear separation between the constructor function, instance members, and static members: WinJS.Class.define:

```
var ClassName = WinJS.Class.define(constructor, instanceMembers, staticMembers);
```

where *constructor* is a function and *instanceMembers* and *staticMembers* are both objects. The general structure you see in code looks like this (you can find many examples in the WinJS source code itself):

```
var ClassName = WinJS.Class.define(
    function ClassName_ctor() { //adding _ctor is conventional
    },
    {
        //Instance members
    },
    {
        //Static members
    }
);
```

Because many classes don't have static members, the third parameter is often omitted. Also, if you pass null as the constructor, WinJS will substitute an empty function in its place. You can see this in the WinJS source code for define (base.js, comments added):

```
function define(constructor, instanceMembers, staticMembers) {
    constructor = constructor || function () { };
    //Adds a supportedForProcessing property set to true. This is needed by
    //WinJS.UI.process[All], WinJS.Binding.process, and WinJS.Resources.process.
    WinJS.Utilities.markSupportedForProcessing(constructor);
    if (instanceMembers) {
        initializeProperties(constructor.prototype, instanceMembers);
    }
    if (staticMembers) {
        initializeProperties(constructor, staticMembers);
    }
    return constructor;
}
```

You can also see how `define` treats static and instance members (`initializeProperties` is a helper that basically iterates the object in the second parameter and copies its members to the object in the first). Static members are specifically added as properties to the class function itself, `constructor`. This means they exist singularly on that object—if you change them anywhere, those changes apply to all instances. I consider that a rather dangerous practice, so I like to consider static members as read-only.

Instance members, on the other hand, are specifically added to `constructor.prototype`, so they are defined just once (especially in the case of methods) while still giving each individual instance a copy of the properties that can be changed without affecting other instances.

You can see, then, that `WinJS.Class.define` is just a helper: you can accomplish everything it does with straight JavaScript, but you end up with code that's generally harder to maintain. Indeed, the team that wrote WinJS needed these structures for themselves to avoid making lots of small mistakes. But otherwise there is nothing magical about `define`, and you can use it in your own app code or not.

Along these lines, people have asked how `define` relates to the class structures of TypeScript. When all is said and done, they accomplish the same things. In fact, you can derive WinJS classes from TypeScript classes and vice versa because they all end up working on the prototype level.

The one exception is that call to `WinJS.Utilities.markSupportedForProcessing` in WinJS. This is a requirement for functions that are used from other parts of WinJS (see Chapter 5, "Controls and Control Styling," in the section "Strict Processing and processAll Functions") and is the only "hidden" benefit in WinJS. If you use TypeScript or other libraries to create classes, you'll need to call that function directly.

Sidebar: Asynchronous Constructors?

The constructors created through `WinJS.Class.define` are inherently synchronous, and there's no simple pattern to create an asynchronous constructor. Instead, use the usual synchronous constructor and then include an `initializeAsync` method in the class that must be called to make the rest of the instance usable. Your async construction work then happens within `initializeAsync`, which should return a promise that's completed when initialization is done.

WinJS.Class.derive

The next part of object-oriented programming that we typically need is the ability to create derived classes—that is, to add instance and static members to an existing class. This is the purpose of `WinJS.Class.derive`. The syntax is:

```
WinJS.Class.derive(baseClass, constructor, instanceMembers, staticMembers);
```

where *baseClass* is the name of a class previously defined with `WinJS.Class.define` or `WinJS.Class.derive`, or, for that matter, any other object. The other three parameters are the same as those of `define`, with the same meaning.

Peeking into the WinJS sources (base.js), we can see how `derive` works:

```
if (baseClass) {
    constructor = constructor || function () { };
    var basePrototype = baseClass.prototype;
    constructor.prototype = Object.create(basePrototype);
    WinJS.Utilities.markSupportedForProcessing(constructor);
    Object.defineProperty(constructor.prototype, "constructor", { value: constructor, writable:
true, configurable: true, enumerable: true });
    if (instanceMembers) {
        initializeProperties(constructor.prototype, instanceMembers);
    }
    if (staticMembers) {
        initializeProperties(constructor, staticMembers);
    }
    return constructor;
} else {
    return define(constructor, instanceMembers, staticMembers);
}
```

You can see that if `baseClass` is `null`, `derive` is just an alias for `define`, and if you indicate `null` for the constructor, an empty function is provided. Otherwise you can see that the derived class is given a copy of the base class's prototype so that the two won't interfere with each other. After that, `derive` adds the static and instance properties as did `define`.

Because `baseClass` already has its own instance and static members, they're present in its prototype, so those members carry over into the derived class, as you'd expect. But again, if you make later changes to members of that original `baseClass`, those changes affect only the derivation and not the original.

Looking around the rest of WinJS, you'll see that `derive` is used in a variety of places to centralize implementation that's shared between similar controls and data sources, for example.

Mixins

Having covered the basics of `WinJS.Class.define` and `WinJS.Class.derive`, we come to the third method in `WinJS.Class`: the `mix` method.

Let's first see what differentiates a mixin from a derived class. When you derive one class from another, the new class is the combination of the base class's members and those additional ones you specify for the derivation. In `WinJS.Class.derive`, you can specify only one base class.

Object-oriented programming includes the concept of *multiple inheritance* whereby you can derive a class from multiple base classes. This is frequently used to attach multiple independent interfaces on the new class. (It's used all the time in Win32/COM programming, where an interface is typically defined as a virtual base class with no implementation and supplies the necessary function signatures to the derived class.)

JavaScript doesn't have object-oriented concepts baked into the language—thus necessitating helpers like `WinJS.Class.define` to play the necessary tricks with prototypes and all that—so there isn't a built-in method to express multiple inheritance.

Hence the idea of a *mix* or *mixin*, which isn't unique to WinJS, as evidenced by this mixin article on Wikipedia. `WinJS.Class.mix` is a way to do something like multiple inheritance by simply combining all the *instance* members of any number of other objects. The description in the documentation for `WinJS.Class.mix` puts it this way: "Defines a class using the given constructor and the union of the set of instance members specified by all the mixin objects. The mixin parameter list is of variable length."

Here we see two other differences between a mix and a derivation: a mix does not operate on static members, and it does not concern itself with any constructors other than the one given directly to `mix`. (This is also why it's not multiple inheritance in the strict object-oriented sense.)

WinJS itself uses the mixin concept for its own implementation. If you look in the docs, you'll see several mixins in WinJS that you can use yourself:

- `WinJS.UI.DOMEventMixin` contains standard implementations of `addEventListener`, `removeEventListener`, `dispatchEvent`, and `setOptions`, which are commonly used for controls. All the WinJS controls bring this into their mix.
- `WinJS.Utilities.eventMixin` is the same as `DOMEventMixin` but without `setOptions`. It's meant for objects that don't have associated UI elements and thus can't use the listener methods on an element in the DOM. The mixin supplies its own implementation of these methods.
- `WinJS.Binding.observableMixin` adds functionality to an object that makes it "observable," meaning that it can participate in data binding. This consists of methods `bind`, `unbind`, and `notify`. This is used with the `WinJS.Binding.List` class (see Chapter 6, "Data Binding, Template, and Collections").

- WinJS.Binding.dynamicObservableMixin builds on the observableMixin with methods called setProperty, getProperty, updateProperty, addProperty, and removeProperty. This is helpful for wiring up two-way data binding, something that WinJS doesn't do but that isn't too difficult to pull together. The Declarative binding sample in the Windows SDK shows how.

With events (in the first two mixins) you commonly use WinJS.Utilities.createEventProperties to also create all the stuff a class needs to support named events. createEventProperties returns a mixin object that you can then use with WinJS.Class.mix. For example, if you pass this method an array with just ["statusChanged"], you'll get a function property in the mixin named onstatuschanged and the ability to create a listener for that event.

Mixins, again, are a way to add pre-canned functionality to a class and a convenient way to modularize code that you'll use in multiple classes. It's also good to know that you can call WinJS.Class.mix multiple times with additional mixins and the results simply accumulate (if there are duplicates, the last member mixed is the one retained).

Obscure WinJS Features

In the main chapters of this book we generally encounter the most common features of WinJS, but if you dig around in the documentation you'll find that there are a bunch of other little features hanging off the WinJS tree like pieces of fruit. To enjoy that harvest a bit, let's explore those features, if for no other reason than to make you aware that they exist!

Wrappers for Common DOM Operations

The first set of features are simple wrappers around a few common DOM operations.

WinJS.Utilities.QueryCollection is a class that wraps an element.querySelectorAll or document.querySelectorAll with a number of useful methods. An instance of this class is created by calling WinJS.Utilities.query(<query>[,]) where <query> is a usual DOM query string and the optional <element> scopes the query. That is, if you provide <element>, the instance wraps element.querySelectorAll(<query>); if you omit <element>, the instance uses document.querySelectorAll(<query>). Similarly, WinJS.Utilities.QueryCollection(<id>) does a document.getElementById(<id>) and then passes the result to new WinJS.Utilities.QueryCollection. WinJS.Utilities.QueryCollection creates a QueryCollection that contains children of a specified element.

Anyway, once you have a QueryCollection instance, it provides methods to work with its collection—that is, with the results of the DOM query that in and of itself is just an array. As such, you'd normally be writing plenty of loops and iterators to work with the items in the array, and that's exactly what QueryCollection provides as a convenience. It follows along with other parts of WinJS, which are utilities that most developers end up writing anyway, so it might as well be in a library!

We can see this in what the individual methods do:

- forEach calls Array.prototype.forEach.apply on the collection, using the given callback function and this argument.
- get returns [<index>] from the array.
- setAttribute iterates the collection and calls setAttribute for each item.
- getAttribute gets an attribute for the first item in the collection.
- addClass, hasClass, removeClass, toggleClass each iterates the collection and calls WinJS.Utilities.addClass, hasClass, removeClass, or toggleClass for each item, respectively.
- listen, removeEventListener iterates the collection, calling addEventListener or removeEventListener for each item.
- setStyle and clearStyle iterate the collection, setting a given style to a value or "", respectively.
- include adds other items to this collection. The items can be in an array, a document fragment (DOM node), or a single item.
- query executes a querySelectorAll on each item in the collection and calls include for each set of results. This could be used, for instance, to extract specific children of the items in the collection and add them to the collection.
- control, given the name of a control constructor (a function) and an options object (as WinJS controls typically use), creates an instance of that control and attaches it to each item in the collection. It's allowable to call this method with just an options object as the first argument, in which case the method calls WinJS.UI.process on each item in the collection followed by WinJS.UI.setOptions for each control therein. This allows the collection to contain elements that have WinJS control declarations (data-win-control attributes) that have not yet been instantiated.
- template renders a template element that is bound to the given data and parented to the elements included in the collection. If the collection contains multiple elements, the template is rendered multiple times, once at each element per item of data passed.

As for WinJS.Utilities.addClass, removeClass, hasClass, and toggleClass, as used above, these are helper functions that simply add one or more classes (space delimited) to an element, remove a single class from an element, check whether an element has a class, and add or remove a class from an element depending on whether it's already applied. These operations sound simple, but their implementation is nontrivial. Take a look at the WinJS source code (in base.js) to see what I mean! Good code that you don't have to write.

WinJS.Utilities.data, convertToPixels, and Other Positional Methods

The next few methods in WinJS to look at are `WinJS.Utilities.data` and `WinJS.Utilities.convertToPixels`.

`WinJS.Utilities.data` is documented as "gets a data value associated with the specific element." The data value is always an object and is attached to the element by using a property name of `_msDataKey`. So `WinJS.Utilities.data(<element>)` always just gives you back that object, or it creates one if one doesn't yet exist. You can then add properties to that object or retrieve them. Basically, this is a tidy way to attach extra data to an arbitrary element, knowing that you won't interfere with the element otherwise. WinJS uses this internally in various places.

`WinJS.Utilities.convertToPixels` sounds fancier than it is. It's just a helper to convert a CSS positioning string for an element to a real number. In CSS you often use values suffixes like "px" and "em" that aren't meaningful values in any computations. This function converts those values to a meaningful number of pixels. With "px" values, or something without suffixes, it's easy—just pass it to `parseInt`, which will strip "px" automatically if it's there. For other CSS values—basically anything that starts with a number—this function assigns the value to the element's `left` property (saving the prior value so that nothing gets altered) and then reads back the element's `pixelLeft` property. In other words, it lets the DOM engine handle the conversion, which will produce 0 if the value isn't convertible.

Along these same lines are the following `WinJS.Utilities` methods:

- `getRelativeLeft` gets the left coordinate of an element relative to a specified parent. Note that the parent doesn't have to be the immediate parent but can be any other node in the element's tree. This function then takes the `offsetLeft` property of the element and keeps subtracting off the `offsetLeft` of the next element up until the designated ancestor is reached.
- `getRelativeTop` does the same thing as `getRelativeLeft` except with `offsetTop`.
- `getContentWidth` returns the `offsetWidth` of an element minus the values of `borderLeftWidth`, `borderRightWidth`, `paddingLeft`, and `paddingRight`, which results in the actual width of the area where content is shown.
- `getTotalWidth` returns the `offsetWidth` of the element plus the `marginLeft` and `marginRight` values.
- getContentHeight returns the `offsetHeight` of an element minus the values of `borderTopWidth`, `borderBottomWidth`, `paddingTop`, and `paddingBottom`, which results in the actual height of the area where content is shown.
- `getTotalHeight` returns the `offsetHeight` of the element plus the `marginTop` and `marginBottom` values.
- `getPosition` returns an object with the `left`, `top`, `width`, and `height` properties of an element

relative to the topmost element in the tree (up to document or body), taking scroll positions into account.

Again, take a look in base.js for the implementation of these functions so that you can see what they're doing and appreciate the work they'll save you!

WinJS.Utilities.empty, eventWithinElement, and getMember

Before leaving the WinJS.Utilities namespace, let's finish off the last few obscure methods found therein. First, the empty method removes all child nodes from a specified element. This is basically a simple iteration over the element's childNodes property, calling removeNode for each in turn (actually in reverse order). A simple bit of code but one that you don't need to write yourself.

Next is eventWithinElement. To this you provide an element and an eventArgs object as you received it from some event. The method then checks to see if the eventArgs.relatedTarget element is contained within the element you provide. This basically says that an event occurred *somewhere* within that element, even if it's not directly on that element. This is useful for working with events on controls that contain some number of child elements (like a ListView).

Finally there's getMember, to which you pass a string name of a "member" and a root object (this defaults to the global context). The documentation says that this "Gets the leaf-level type or namespace specified by the name parameter." What this means is that if you give it a name like "navigate", it will look within the namespace of the root you give for that member and return it. In the case of "navigate", it will find WinJS.Navigation.navigate. Personally, I don't know when I'd use this, but it's an interesting function nonetheless!

WinJS.UI.scopedSelect and getItemsFromRanges

Wrapping up our discussion of obscure features are two members of the WinJS.UI namespace that certainly match others in WinJS.Utilites for obscurity!

The first is WinJS.UI.scopedSelect, to which you provide a CSS selector and an element. This function is documented like this: "Walks the DOM tree from the given element to the root of the document. Whenever a selector scope is encountered, this method performs a lookup within that scope for the specified selector string. The first matching element is returned." What's referred to here as a "selector scope" is a property called msParentSelectorScope, which WinJS sets on child elements of a fragment, page control, or binding template. In this way, you can do a querySelector within the scope of a page control, fragment, or template without having to start at the document level. The fact that it keeps going up toward the document root means that it will work with nested page controls or templates.

The other is WinJS.UI.getItemsFromRanges, which takes a WinJS.Binding.List and an array of ISelectionRange objects (with firstIndex and lastIndex properties). It then returns a promise whose results are an array of items in that data source for those ranges. This exists to translate multiple selections in something like a ListView control into a flat array of selected items—and, in fact, is what's used to implement the getItems method of a ListView's selection property. So if you implement a control of your own around a WinJS.Binding.List, you can use getItemsFromRanges to do the same. The method is provided, in other words, to work with the data source because that is a separate concern from the ListView control itself.

Extended Splash Screens

In most cases, it's best for an app to start up as quickly as it can and bring the user to a point of interactivity with the app's home page, as discussed in "Optimizing Startup Time" in Chapter 3, "App Anatomy and Performance Fundamentals." In some scenarios, though, an app might have a great deal of unavoidable processing to complete before it can bring up that first page. For example, games must often decompress image assets to optimize them for the device's specific hardware—the time spent at startup makes the whole game run much smoother from then on. Other apps might need to process an in-package data file and populate a local database for similar reasons or might need to make a number of HTTP requests. In short, some scenarios demand a tradeoff between startup time and the performance of the app's main workflows.

It's also possible for the user to launch your app shortly after rebooting the system, in which case there might be lots of disk activity going on. As a result, any disk I/O in your activation path could take much longer than usual.

In all these cases, it's good to show the user that something is actually happening so that she doesn't think to switch away from the default splash screen and risk terminating the app. You might also just want to create a more engaging startup experience than the default splash screen provides.

An *extended splash screen* allows you to fully customize the splash screen experience. In truth, an extended splash screen is not a system construct—it's just an implementation strategy for your app's initial page behind which you'll do your startup work before displaying your real home page. In fact, a typical approach is to just overlay a full-sized div on top of your home page for this purpose and then remove that div from the DOM (or animate it out of view) when initialization is complete.

The trick (as described on Guidelines for splash screens) is to make this first app page initially look exactly like the default splash screen so that there's no visible transition between the two. At this point many apps simply add a progress indicator with some kind of a "Please go grab a drink, do some jumping jacks, or enjoy a few minutes of meditation while we load everything" message. Matching the system splash screen, however, doesn't mean that the extended splash screen has to *stay* that way. Because the entire display area is now under your control, you can create whatever experience you want: you can kick off animations, perform content transitions, play videos, and so on. And because your first page is up, meaning that you've returned from your activated handler, you're no longer subject to the 15-second timeout. In fact, you can hang out on this page however long you need, even waiting for user input (as when you require a login to use the app).

I recommend installing and running various apps from the Store to see different effects in action. Some apps gracefully slide the default splash screen logo up to make space for a simple progress ring. Others animate the default logo off the screen and pull in other interesting content or play a vido. Now compare the experience such extended splash screens to the static default experience that other apps provide. Which do you prefer? And which do you think users of your app will prefer, if you must make them wait?

Making a seamless transition from the default splash screen is the purpose of the `args.detail.splashScreen` object included with the `activated` event. This object—see [`Windows.ApplicationModel.Activation.SplashScreen`]—contains an `imageLocation` property (a `Windows.Foundation.Rect`) indicating the placement and size of the splash screen image on the current display. (These values depend on the screen resolution and pixel density.) On your extended splash screen page, then, initially position your default image at this exact location and entertain your users by animating it elsewhere. You also use `imageLocation` to determine where to place other messages, progress indicators, and other content relative to that image.

The `splashScreen` object also provides an `ondismissed` event so that you can perform specific actions when the system-provided splash screen is dismissed and your extended splash screen comes up. This is typically used to trigger the start of on-page animations, starting video playback, and so on.

Important Because an extended splash screen is just a page in your app, it can be placed into any view at any time. As with every other page in your app, make sure your extended splash screen can handle different sizes and orientations, as discussed in Chapter 8, "Layout and Views." See also "Adjustments for View Sizes" later in this appendix.

For one example of an extended splash screen, refer to the [Splash screen sample] in the Windows SDK. Although it shows the basic principles in action, all it does it add a message and a button that dismisses the splash screen (plus the SDK sample structure muddies the story somewhat). So let's see something more straightforward and visually interesting, which you can find in the ExtendedSplashScreen1 example in the companion content for the appendices. The four stages of this splash screen (for a full landscape view) are shown in Figure B-1, and you can find a video demonstration of all this in Video B-1. (The caveat is that this example doesn't handle other views, but we'll come back to that in the next section.)

The first stage, in the upper left of Figure B-1, is the default splash screen that uses only the logo image. The pie graphic in the middle is 300x300 pixels, with the rest of the PNG file transparent so that the background color shows through. Now let's see what happens when the app gets control.

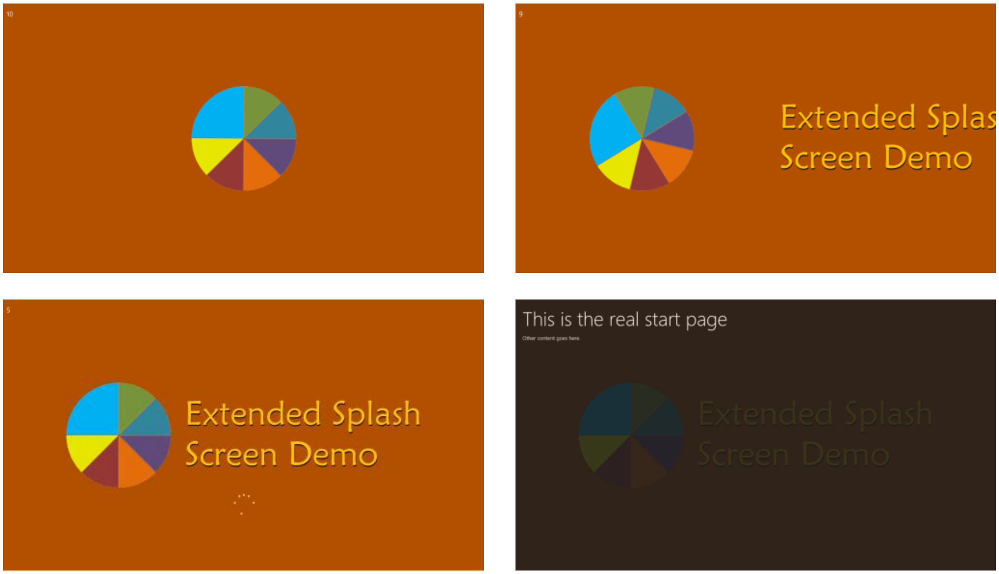


FIGURE B-1 Four stages of the ExtendedSplashScreen1 example: (1) the default splash screen, upper left, (2) animating the logo and the title, upper right, (3) showing the progress indicator, lower left, and (4) fading out the extended splash screen to reveal the main page, lower right. A 10-second countdown in the upper left corner of the screen simulates initialization work.

The home page for the app, default.html, contains two `div` elements: one with the final (and thoroughly unexciting) page contents and another with the contents of the extended splash screen:

```
<div id="mainContent">
    <h1>This is the real start page</h1>
    <p>Other content goes here.</p>
</div>

<div id="splashScreen">
    <p><span id="counter"></span></p>
    <img id="logo" src="/images/splashscreen.png" />
    <img id="title" src="/images/splashscreentitle.png" />
    <progress id="progress" class="win-ring win-large"></progress>
</div>
```

In the second `div`, which overlays the first because it's declared last, the `counter` element shows a countdown for debug purposes, but you can imagine such a counter turning into a determinate progress bar or a similar control. The rest of the elements provide the images and a progress ring. But we can't position any of these elements in CSS until we know more about the size of the screen. The best we can do is set the `splashScreen` element to fill the screen with the background color and set the `position` style of the other elements to `absolute` so that we can set their exact location from code. This is done in default.css:

```
#splashScreen {
    background: #B25000;  /* Matches the splash screen color in the manifest */
    width: 100%;          /* Cover the whole display area */
    height: 100%;
}

    #splashScreen #counter {
        margin: 10px;
        font-size: 20px;
    }

    #splashScreen #logo {
        position: absolute;
    }

    #splashScreen #title {
        position: absolute;
    }

    #splashScreen #progress {
        position: absolute;
        color: #fc2; /* Use a gold ring instead of default purple */
    }
```

In default.js now, we declare some modulewide variables for the splash screen elements, plus two values, one to control how long the extended splash screen is displayed (simulating initialization work) and one that indicates when to show the progress ring:

```
var app = WinJS.Application;
var activation = Windows.ApplicationModel.Activation;

var ssDiv = null;          //Splash screen overlay div
var logo = null;           //Child elements
var title = null;
var progress = null;

var initSeconds = 10;      //Length in seconds to simulate loading
var showProgressAfter = 4; //When to show the progress control in the countdown
var splashScreen = null;
```

In the `activated` event handler, we can now position everything based on the `args.detail.splashScreen.imageLocation` property (note the comment regarding `WinJS.UI.processAll` and `setPromise`, which we're not using here):

```
app.onactivated = function (args) {
    if (args.detail.kind === activation.ActivationKind.launch) {
        //WinJS.UI.processAll is needed ONLY if you have WinJS controls on the extended
        //splash screen, otherwise you can skip the call to setPromise, as we're doing here.
        //args.setPromise(WinJS.UI.processAll());

        ssDiv = document.getElementById("splashScreen");
        splashScreen = args.detail.splashScreen;    //Need this for later
        var loc = splashScreen.imageLocation;
        //Set initial placement of the logo to match the default start screen
        logo = ssDiv.querySelector("#logo");
        logo.style.left = loc.x + "px";
        logo.style.top = loc.y + "px";

        //Place the title graphic offscreen to the right so we can animate it in
        title = ssDiv.querySelector("#title");
        title.style.left = ssDiv.clientWidth + "px";  //Just off to the right
        title.style.top = loc.y + "px";               //Same height as the logo

        //Center the progress indicator below the graphic and initially hide it
        progress = ssDiv.querySelector("#progress");
        progress.style.left = (loc.x + loc.width / 2 - progress.clientWidth / 2) + "px";
        progress.style.top = (loc.y + loc.height + progress.clientHeight / 4) + "px";
        progress.style.display = "none";
```

At this stage, the display still appears exactly like the upper left of Figure B-1, only it's our extended splash screen page and not the default one. Thus, we can return from the activated handler at this point (or complete the deferral) and the user won't see any change, but now we can do something more visually interesting and informational while the app is loading.

To simulate initialization work and make some time for animating the logo and title, we have a simple countdown timer using one-second setTimeout calls:

```
        //Start countdown to simulate initialization
        countDown();
    }
};

function countDown() {
    if (initSeconds == 0) {
        showMainPage();
    } else {
        document.getElementById("counter").innerText = initSeconds;

        if (--showProgressAfter) {
            progress.style.display = "";
        }

        initSeconds--;
        setTimeout(countDown, 1000);
    }
}
```

Notice how we show our main page when (our faked) initialization is complete and how the previously positioned (but hidden) progress ring is shown after a specified number of seconds. You can see the progress ring on the lower left of Figure B-1.

To fade from our extended splash screen to the main page (a partial fade is shown on the lower right of Figure B-1), the showMainPage function employs the WinJS Animations Library as below, where WinJS.UI.Animation.fadeOut takes an array of the affected elements. fadeOut returns a promise, so we can attach a completed handler to know when to hide the now-invisible overlay div, which we can remove from the DOM to free memory:

```
function showMainPage() {
    //Hide the progress control, fade out the rest, and remove the overlay
    //div from the DOM when it's all done.
    progress.style.display = "none";
    var promise = WinJS.UI.Animation.fadeOut([ssDiv, logo, title]);

    promise.done(function () {
        ssDiv.removeNode(true);
        splashScreen.ondismissed = null; //Clean up any closures for this WinRT event
    });
}
```

Refer to Chapter 14, "Purposeful Animations," for details on the animations library. Keep in mind is that animations run in the GPU (graphics processing unit) when they affect only *transform* and *opacity* properties; animating anything else runs on the CPU and generally performs poorly.

To complete the experience, we now want to add some animations to translate and spin the logo to the left and to slide in the title graphic from its initial position off the right side of the screen. The proper time to start these animations is when the args.detail.splashScreen.ondismissed event is fired, as I do within activated just before calling my countDown function. This dismissed event handler simply calculates the translation amounts for the logo and title and sets up a CSS transition for both using the helper function WinJS.UI.executeTransition:

```
        //Start our animations when the default splash screen is dismissed
        splashScreen.ondismissed = function () {
            var logoImageWidth = 300;  //Logo is 620px wide, but image is only 300 in the middle
            var logoBlankSide = 160;   //That leaves 160px to either side

            //Calculate the width of logo image + separator + title. This is what we want to end
            //up being centered on the screen.
            var separator = 40;
            var combinedWidth = logoImageWidth + separator + title.clientWidth;

            //Final X position of the logo is screen center - half the combined width - blank
            //area. The (negative) translation is this position minus the starting point (loc.x)
            var logoFinalX = ((ssDiv.clientWidth - combinedWidth) / 2) - logoBlankSide;
            var logoXTranslate = logoFinalX - loc.x;

            //Final X position of the title is screen center + half combined width - title width.
            //The (negative) translation is this position minus the starting point (screen width)
            var titleFinalX = ((ssDiv.clientWidth + combinedWidth) / 2) - title.clientWidth;
            var titleXTranslate = titleFinalX - ssDiv.clientWidth;

            //Spin the logo at the same time we translate it
            WinJS.UI.executeTransition(logo, {
                property: "transform", delay: 0, duration: 2000, timing: "ease",
                to: "translateX(" + logoXTranslate + "px) rotate(360deg)"
            });
            //Ease in the title after the logo is already moving (750ms delay)
            WinJS.UI.executeTransition(title, {
                property: "transform", delay: 750, duration: 1250, timing: "ease",
                to: "translateX(" + titleXTranslate + "px)"
            });
        }
```

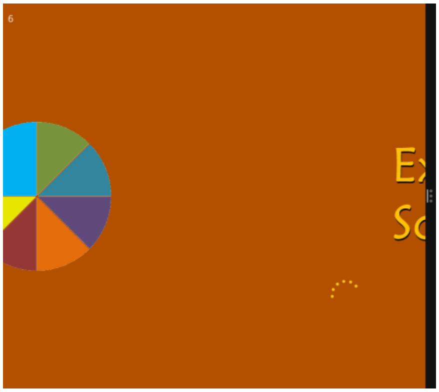
This takes us from the upper left of Figure B-1 through the upper right stage, to the lower left stage. To really appreciate the effect, of course, run the example!

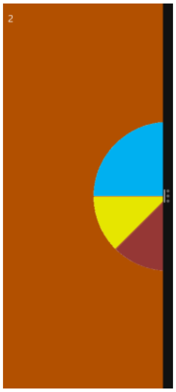
This code structure will likely be similar to what you need in your own apps, only use a single `setTimeout` call to delay showing a `progress` control, replace the `countDown` routine with your real async initialization work, and set up whatever elements and animations are specific to your splash screen design. Take special care that the majority of your initialization work happens either asynchronously or is started within the `dismissed` handler so that the default splash screen is dismissed quickly. Never underestimate a user's impatience!

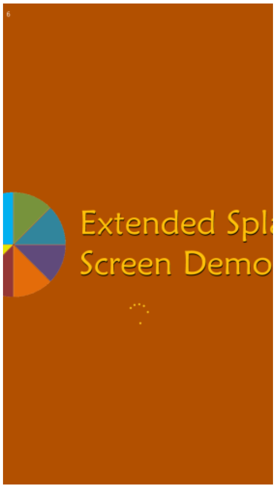
As for handling different views, this is a matter of handling the `window.onresize` event and adjusting element positions accordingly.

Adjustments for View Sizes

The ExtendedSplashScreen1 example we saw in the previous section works great for full screen activation or when the app is launched into a smaller width or portrait mode to begin with. If the view is changed while the extended splash screen is active, however—to a narrower landscape view, a very narrow portrait view, or full screen portrait, for instance—we get dreadful results:





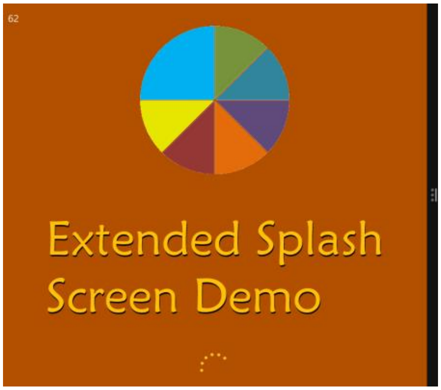


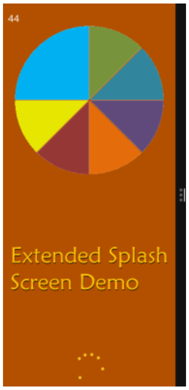
If we used only static content on the extended splash screen, we could take care of these situations with CSS. Our example, however, uses absolute positioning so that we can place elements relative to the default splash screen image and animate them to their final positions. What we need to pay attention to now is the fact that the default position might be different on startup and can then change while the extended splash screen is visible. (While testing all this, by the way, I set the countdown timer to 100 seconds instead of 10, giving me much more time to change the view's size and position.)

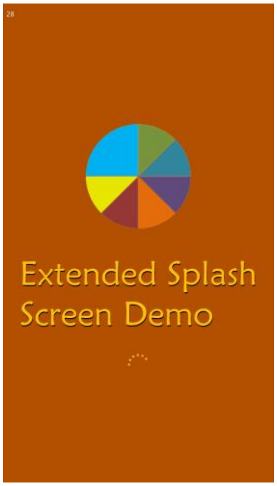
Here's how we can handle the important events:

- Within the `activated` event, where we receive the default splash screen image location, we initially move the logo, title, and progress ring to positions relative to that location, scale them if needed, and set up the splash screen `dismissed` handler.
- While the default splash screen is still visible, we use `window.onresize` to reposition and rescale those elements as needed. That is, the user can modify the view in the short time the default splash screen is visible.
- Once the splash screen's `dismissed` event fires, we check if the view has a landscape aspect ratio that's 1024px or wider. If so, we can place the logo and title side by side as we did originally, using the same animations as before. Otherwise we're in a view that's too narrow for that design, so we stack the logo and title vertically and instead of animating the logo to the left we animate it upwards.
- Once we get the extended splash screen going, we change the `window.onresize` handler to reposition the elements in their final (post-animation) locations. If a resize happens while animations are in progress, those animations are canceled (by canceling their promises). A resize can switch the view between the wide landscape layout and the vertical layout, so we just change element positions accordingly.

These changes are demonstrated in the ExtendedSplashScreen2 example, which now adapts well to different views:







Not quite perfect The one case in this example that still doesn't work is when you resize the *default* splash screen in the short time before the extended splash screen appears. The problem is that Windows does not (in my tests) update the splashScreen.imageLocation coordinates, so the placement of your extended splash screen elements is inaccurate. I have not found a workaround for this particular issue, unfortunately.

Custom Layouts for the ListView Control

For everything that the GridLayout, ListLayout, and CellSpanningLayout classes provide for a ListView, they certainly don't support every layout your designers might dream up for a collection, such as a cell-spanning ListLayout, a vertically panning cell-spanning layout, or nonlinear layouts.

What's very important to understand about this model is how much the ListView is still doing on your behalf when a custom layout is involved. HTML and CSS let you create any kind of layout you want without a ListView, but then you'd have to implement your own keyboarding support, accessibility, data binding, selection and invocation behaviors, grouping, and so forth. By using a custom layout, you take advantage of all those features, so you need concern yourself with only the positional relationships of the items within the ListView and a few layout-specific details like virtualization and animation, if needed.

Fortunately, it's straightforward to create a custom layout by using some CSS and a class with a few methods through which the ListView talks to the layout. The simplest custom layout, in fact, doesn't require much of anything: just a class with one method called initialize, which can itself be empty. But to understand this better, let's first look at the general structure of the layout object and how the ListView uses it.

Did you know? Custom layouts were possible with WinJS 1.0 but were exceeding difficult to implement because of the ListView's use of absolute positioning and the fact that it wasn't designed for layout extensibility. ListView was revamped for WinJS 2.0 to use a straight-up CSS layout (grid or flexbox), a change that enables you to write custom layouts without a WinJS Ph.D.! This change is also responsible for the removal of item recycling within template functions, if you care to know that detail.

A layout class selectively implements the methods and properties of the WinJS.UI.ILayout2 interface.[143] The one read-write property is orientation, which can have a value from the WinJS.UI.Orientation interface (horizontal or vertical); the methods are as follows:

[143] There are ILayout and ILayoutSite interfaces in WinJS 1.0 that are used with the WinJS 1.0 ListView. These are much more complicated and should not be confused with the WinJS 2.0 interfaces.

ILayout2 method	Description
`initialize`	Called to provide the layout with the rendering *site* object and a flag indicating whether grouping is enabled in the ListView. The site object implements the `WinJS.UI.ILayoutSite2` interface and is how the layout gets information from the ListView.
`uninitialize`	Called to release resources obtained during `initialize`, typically when the ListView changes layouts.
`layout`	Performs a layout pass given information about the most recently affected items.
`itemsFromRange`	Returns the indices of items within a pixel range (`start` and `end` arguments).
`getAdjacent`	Called when arrow keys, Page Up, or Page Down are used in the ListView to navigate items. This method receives the current item and the pressed key as arguments, and returns the next item to receive keyboard focus.
`dragLeave`	Called when a drag and drop item leaves the ListView. This tells the layout to remove any drop indicators or to readjust any layout changes made for potential reordering (see `dragOver` below). If the ListView has `itemsDraggable: true`, the layout can use `dragLeave` to visually indicate that the item has been taken out of the list. If the ListView has `itemsReorderable: true`, and the layout adjusts itself in `dragOver` to make visual space for a drop, `dragLeave` is used to readjust the layout back to its original state.
`hitTest`, `dragOver`	If the ListView's `itemsReorderable` is `true`, these are called when an item is dragged over the ListView. `hitTest` determines which item is at a particular (x,y) coordinate; `dragOver` receives the coordinates and a `dragInfo` object and signals the layout to open up potential drop space for the item being dragged.
`setupAnimations`, `executeAnimations`	Called when it's the right time to configure and execute animations, typically in response to movement of items in the ListView. Neither method has arguments.

Fortunately, the ListView magnanimously allows you to implement only those parts of a layout class that you really need and ignore the rest (the ListView will provide no-op stubs for methods you don't implement). We call this a "pay for play" model. In the following table, the required method, `initialize`, is shaded in orange, the recommended methods in green (to support the most flexibility depending on your scenario), and truly optional methods in blue:

Layout capability	Applicable methods
minimal	`initialize` (plus `uninitialize` if there's any cleanup work)
non-vertical layout	Minimal plus `layout`
virtualization support	Minimal plus `itemsFromRange`
keyboard support	Minimal plus `getAdjacent`
drag and drop visual indicators (general)	Add `dragLeave`
reordering and positional drag and drop indicators	Add `hitTest`, `dragOver` (both are required together)
animations	Add `setupAnimations`, `executeAnimations`

All this means again that the most basic custom layout object need only support `initialize` and possibly `uninitialized`, which will produce a vertical layout by default. To do a horizontal or nonlinear layout, you'll need the layout method, and then you can add others as you like to support additional features. Let's look at a number of examples.

Minimal Vertical Layout

Our first example is scenario 1 of the HTML ListView custom layout sample, whose data source (of ice cream flavors once again!) is a `WinJS.Binding.List` defined in js/data.js and accessed through the `Data.list` and `Data.groupedList` variables.

The custom layout object in scenario 1 is named `SDKSample.Scenario1.StatusLayout` and is created with `WinJS.Class.define` (js/scenario1.js):

```
WinJS.Namespace.define("SDKSample.Scenario1", {
    StatusLayout: WinJS.Class.define(function (options) {
        this._site = null;
        this._surface = null;
    },
    {
        // This sets up any state and CSS layout on the surface of the custom layout
        initialize: function (site) {
            this._site = site;
            this._surface = this._site.surface;

            // Add a CSS class to control the surface level layout
            WinJS.Utilities.addClass(this._surface, "statusLayout");

            // This isn't really used, create an orientation property instead
            return WinJS.UI.Orientation.vertical;
        },

        // Reset the layout to its initial state
        uninitialize: function () {
            WinJS.Utilities.removeClass(this._surface, "statusLayout");
            this._site = null;
            this._surface = null;
        },
    })
});
```

And if you look in css/default.css you'll see that the styles assigned to the *statusLayout* class are also quite minimal (everything else in css/scenario1.css is for the item templates, not the layout):

```
.listView .statusLayout {
    position: static;   /* This isn't explicitly needed as it's the default anyway */
}

    .listView .statusLayout .win-container {
        width: 250px;
        margin-top: 10px;
    }

    .listView .statusLayout .win-item {
        width: 250px;
        height: 100%;
        background-color: #eee;
    }
```

Together, then, you can see that this custom layout doesn't do much in terms of customization: it merely defines a set width and background color but doesn't say anything specific about where items are placed. It simply relies on the default positioning provided by the app host's layout engine, just as it provides for all other HTML in your app.

The rest of the code in js/scenario1.js and css/scenario1.css is all about setting up and styling item templates for the ListView and doesn't affect the layout portion that we're concerned with here. The same is true for most of html/scenario1.html, the bulk of which are the declarative item templates. The one line we care about with our layout is the ListView declaration at the end:

```
<div class="listView" data-win-control="WinJS.UI.ListView"
    data-win-options="{itemDataSource: Data.list.dataSource,
        layout: {type: SDKSample.Scenario1.StatusLayout}}"></div>
```

Here you can see that we specify a custom layout just as we would a built-in one, using the `ListView.layout` option. Remember from Chapter 7, "Collection Controls," that `layout` is an object whose `type` property contains the name of the layout object constructor. Any other properties you provide will then be passed to the constructor as the `options` argument. That is, the following markup:

```
layout: { type: <layout_class>, <option1>: <value1>, <option2>: <value2>, ...}
```

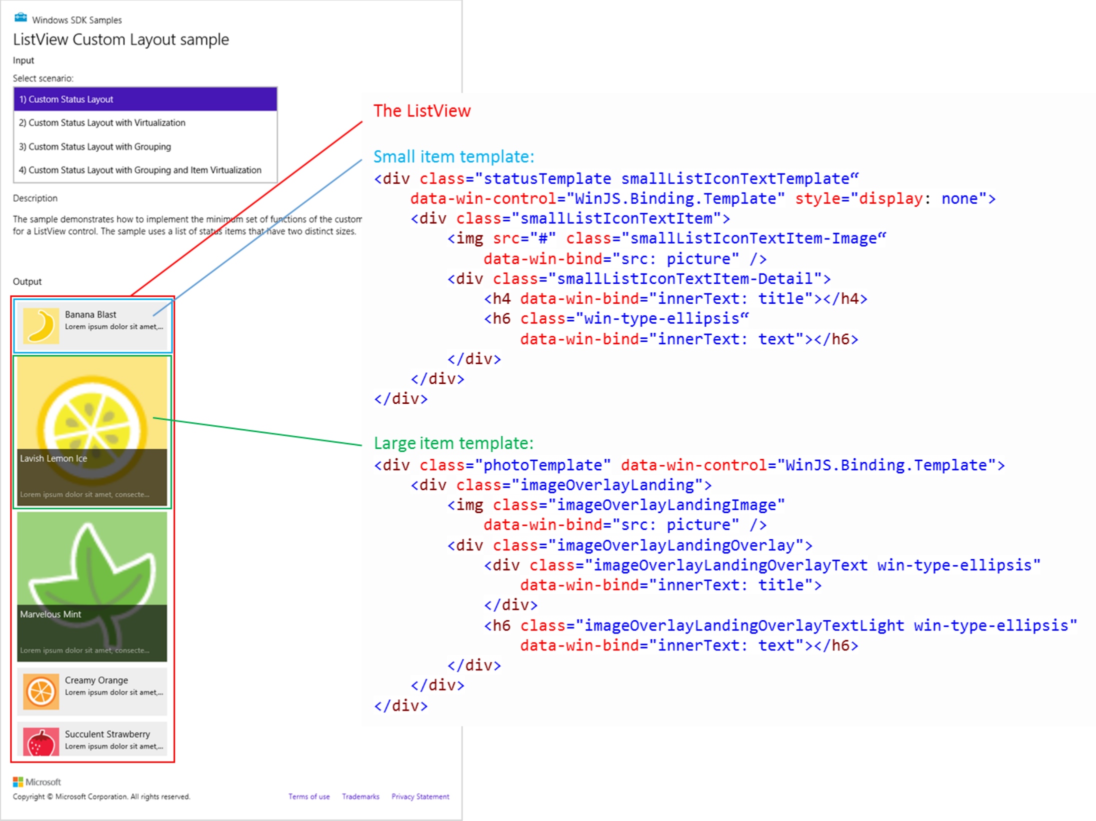
is the same thing as using the following code in JavaScript:

```
listView.layout = new <layout_class>({ <option1>: <value1>, <option2>, <value2>, ... });
```

And that's it! Running the sample with this layout now in portrait mode, as shown in Figure B-2, we see that the result is a simple vertically oriented ListView, with each item's height determined naturally by its item template. What's impressive about this is that it took almost no code to effectively create the equivalent of a variable-size vertical `ListLayout`, something that was amazingly difficult to do in WinJS 1.0 (and was one of the most requested features for WinJS 2.0!).

The reason for this is that the ListView builds up a DOM tree of `div` elements for the items in the list, but otherwise it leaves it to the layout to define their positioning. Because our layout doesn't do anything special, we end up with the default layout behavior of the app host. Because `div` elements use block layout by default (`position: static`), they are rendered vertically like any other markup.

In running the sample, also notice that all the usual ListView item interactions are fully present within the custom layout. The layout, in other words, has no effect on *item* behavior: all that is still controlled by the ListView.



Windows SDK Samples
ListView Custom Layout sample
Input
Select scenario:
1) Custom Status Layout
2) Custom Status Layout with Virtualization
3) Custom Status Layout with Grouping
4) Custom Status Layout with Grouping and Item Virtualization
Description

The sample demonstrates how to implement the minimum set of functions of the custom for a ListView control. The sample uses a list of status items that have two distinct sizes.

Output
Banana Blast
Lorem ipsum dolor sit amet,...
Lavish Lemon Ice
Lorem ipsum dolor sit amet, consecte...
Marvelous Mint
Lorem ipsum dolor sit amet, consecte...
Creamy Orange
Lorem ipsum dolor sit amet,...
Succulent Strawberry
Lorem ipsum dolor sit amet,...
Microsoft
Copyright © Microsoft Corporation. All rights reserved.
Terms of use
Trademarks
Privacy Statement
The ListView

Small item template:

```
<div class="statusTemplate smallListIconTextTemplate“
    data-win-control="WinJS.Binding.Template" style="display: none">
    <div class="smallListIconTextItem">
        <img src="#" class="smallListIconTextItem-Image“
            data-win-bind="src: picture" />
        <div class="smallListIconTextItem-Detail">
            <h4 data-win-bind="innerText: title"></h4>
            <h6 class="win-type-ellipsis“
                data-win-bind="innerText: text"></h6>
        </div>
    </div>
</div>
```

Large item template:

```
<div class="photoTemplate" data-win-control="WinJS.Binding.Template">
    <div class="imageOverlayLanding">
        <img class="imageOverlayLandingImage"
            data-win-bind="src: picture" />
        <div class="imageOverlayLandingOverlay">
            <div class="imageOverlayLandingOverlayText win-type-ellipsis"
                data-win-bind="innerText: title">
            </div>
            <h6 class="imageOverlayLandingOverlayTextLight win-type-ellipsis"
                data-win-bind="innerText: text"></h6>
        </div>
    </div>
</div>
```

FIGURE B-2 Scenario 1 of the HTML ListView custom layout sample, shown in portrait view so that we can see more of the control. The templates as defined in html/scenario1.html are also shown here. Because each item in the list is rendered in a `div` and a `div` uses block layout in the rendering engine, the items render vertically by default.

Minimal Horizontal Layout

Given the default vertical behavior, how would we change scenario 1 of the sample to do a horizontal layout? It's mostly just a matter of styling:

- Style the `win-itemscontainer` class to use a horizontal layout, such as a flexbox with a `row` direction. This is the most essential piece because it tells the rendering engine to do something other than the default vertical stacking. You could also use a CSS grid.

- Define an `orientation` property in the layout class with the value of `“horizontal”`. This tells the ListView to use the `win-horizontal` style on itself rather than `win-vertical` (the default). Note that the return value from the layout's `initialize` method doesn't have an effect here, though for good measure it should return `WinJS.UI.Orientation.horizontal` as well.

- Change the app's styles for the ListView to be appropriate for horizontal, such as using `width: 100%` instead of `height: 100%`.

- Implement the layout class's `layout` method as needed.

These changes are implemented in scenario 1 of the Custom Layout Extras example in the companion content for the appendices. We first change the *statusLayout* class set within the layout's `initialize` method to *statusLayoutHorizontal* and return the horizontal orientation. Let's also add the orientation property (other code omitted):

```
WinJS.Namespace.define("SDKSample.Scenario1", {
    StatusLayoutHorizontal: WinJS.Class.define(function (options) {
        // ...
        this.orientation = "horizontal";
    },
    {
        initialize: function (site) {
            // ...
            WinJS.Utilities.addClass(this._surface, "statusLayoutHorizontal");
            return WinJS.UI.Orientation.horizontal;
        },
        // ...
    })
});
```

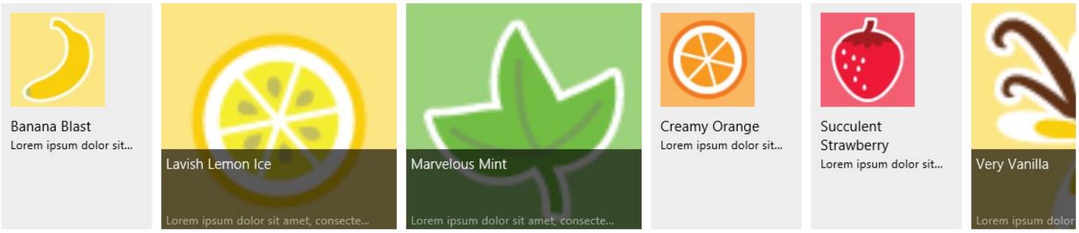
Hint Make sure that *orientation* is spelled correctly in your property name or else the ListView won't find it. I made the mistake of leaving out the last *i* and was scratching my head trying to understand why the ListView wasn't responding properly!

Note The sample uses a *listView* style class in css/default.css across all its scenarios, so making changes in default.css will apply everywhere. It's best to change the class in html/scenario1.html on the ListView to something like *listview_s1* and keep scenario-specific styles in css/scenario1.css.

Next, taking the note above into account, let's style the ListView as follows in css/scenario1.css:

```
.listView_s1 {
    width: 100%;    /* Stretch horizontally */
    height: 270px;  /* Fixed height */
}

/* Change the layout model for the ListView */
.listView_s1 .statusLayoutHorizontal .win-itemscontainer {
    height: 250px;
    display: -ms-flexbox;
    -ms-flex-direction: row;
}

/* These are for the items, same as the vertical orientation */
.listView_s1 .statusLayoutHorizontal .win-item {
    width: 100%;
    height: 100%;
    background-color: #eee;
}
```

I also figured that we'll want variable widths on the items rather than heights. Making some CSS changes for the small items, we get the following result:



The one problem we have, however, is that the ListView doesn't pan horizontally, nor do scrollbars appear. This is because the `win-itemscontainer` element with the flexbox gets a computed width equal to the width of the `win-surface` element that contains it. To correct this, we need to make sure that this element has a width appropriate to all the items it contains. Of course, unless we know we have a fixed number of items in the underlying data source, we won't know this ahead of time to specify directly in CSS.

This is where the `layout` method comes into play, because it's where the layout object receives the tree of elements that it's working with:

```
layout: function (tree, changedRange, modifiedElements, modifiedGroups) {
    // ...
    return WinJS.Promise.as();
}
```

The first argument is clearly the element tree. The other three signal when changes happen because of items being added and moved, allowing you to minimize the modifications you need to make to other items (and thus to reduce rendering overhead). The return value of `layout` is a promise that's fulfilled when the layout is complete. Or, for more optimization, you can return an object with two promises, one that's fulfilled when the layout of the modified range is complete and the other when all layout is complete.

Anyway, in this particular layout we need to add up the item sizes and set the width of the `win-itemscontainer` element. To do this, we iterate the item list and do a lookup of each item's size by using an *itemInfo* method and a size map:

```
layout: function (tree, changedRange, modifiedElements, modifiedGroups) {
    var container = tree[0].itemsContainer;
    var items = container.items;
    var realWidth = 0;
    var itemsLength = items.length;
    var type;

    for (var i = 0; i < itemsLength; i++) {
        type = this._itemInfo(i).type;
        realWidth += this._itemSizeMap[type].width;
    }
    //Set the true width of the itemscontainer now.
    container.element.style.width = realWidth + "px";

    // Return a Promise or {realizedRangeComplete: Promise, layoutComplete: Promise};
    return WinJS.Promise.as();
},
```

Tip Remember that a `style.width` property must include units. In my first tests I left off the "px" on the second-to-last line of code above, so nothing worked, much to my confusion!

If you've played around with the `CellSpanningLayout` in WinJS, the idea of an `itemInfo` function will be familiar. To generalize for a moment, the `ILayout2` interface is what the *ListView* uses to communicate with the layout object, but that same layout object can provide other methods and properties through which it can communicate with the *app*. After all, the app must create the layout object and supply it to the ListView, so the layout can supply added members.

In this case, the *StatusLayoutHorizontal* class that we're implementing here supports two additional properties, `itemInfo` and `itemSizeMap`, with internal defaults, of course:

```
/*
 * These members are not part of ILayout2
 */

// Default implementation of the itemInfo function
_itemInfo: function (i) {
    return "status";
},

//Default size map
_itemSizeMap: {
    status: { width: 100 },
},

// getters and setters for properties
itemInfo: {
    get: function () {
        return this._itemInfo;
    },
    set: function (itemInfoFunction) {
        this._itemInfo = itemInfoFunction;
    }
},

itemSizeMap: {
    get: function () {
        return this._itemSizeMap;
    },
    set: function (value) {
        this._itemSizeMap = value;
    }
}
```

You can see that if the app doesn't supply its own itemInfo and itemSizeMap properties, each item will be set to a width of 100. In the example, however, the layout is created this way:

```
this._listView.layout = new CustomLayouts.StatusLayoutHorizontal({itemInfo: this._itemInfo,
    itemSizeMap: itemSizeMap });
```

where the app's _itemInfo and itemSizeMap are as follows:

```
var itemSizeMap = {
    status: { width: 160 }, // plus 10 pixels to incorporate the margin
    photo:  { width: 260 }   // plus 10 pixels to incorporate the margin
};

// Member of the page control
_itemInfo: function (i) {
    var item = Data.list.getItem(i);
    return { type: item.data.type };
}
```

When the layout processes the items within its layout method, it will get back widths of either 160 or 260, allowing the layout to compute the exact width and style the container accordingly.

The key here is that we have a clear interface between the layout and the app that's using it. We could easily write the layout to draw from app variables directly, which could be more efficient if you're really sensitive to performance. For good reusability of your layout, however, using a scheme like the one shown here is preferable.

Two-Dimensional and Nonlinear Layouts

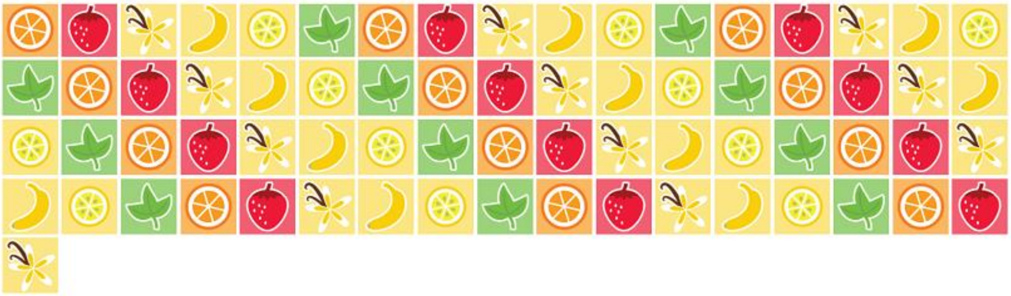
Having seen basic vertical and horizontal layouts, let's see what other creative ideas we can implement. Let's start with a grid layout like WinJS.UI.GridLayout but one that pans vertically and populates items across and then down (instead of down and across). The easiest way to do this is with a CSS flexbox with row wrapping. (It's also possible to do something similar with a CSS grid, but it gets more complicated because you need to figure out how many rows and columns to use in styling.)

The flexbox approach can be done completely in CSS, as demonstrated in scenario 2 of the Custom Layout Extras example, where CustomLayouts.VerticalGrid_Flex is implemented in js/scenario2.js with only the skeletal methods that adds the class *verticalGrid_Flex* to the ListView's win-surface element. The declared ListView in html/scenario2.html references this layout and uses a simple item template that just shows a single image. What makes it all work are just these bits in css/scenario2.css:

```
.listView_s2 .verticalGrid_Flex .win-itemscontainer {
    width: 100%;
    height: 100%;
    display: -ms-flexbox;
    -ms-flex-direction: row;
    -ms-flex-wrap: wrap;
}

/* Tighten the default margins on the ListView's per-item container */
```

```
.listView_s2 .verticalGrid_Flex .win-container {
    margin: 2px;
}
```

We can clearly see that the items are being laid out left to right and then top to bottom (this is panned down a little so we can see the wrapping effect):



Now let's go way outside the box and implement a *circular* layout, as shown in Figure B-3 and implemented in scenario 3 of the example (thanks to Mike Mastrangelo of the WinJS team for this one, although I added the spiral option). Note that you may need to run on a larger monitor to see a good effect; on 1366x768 it gets rather cramped. Video B-2 also shows the animation effect.

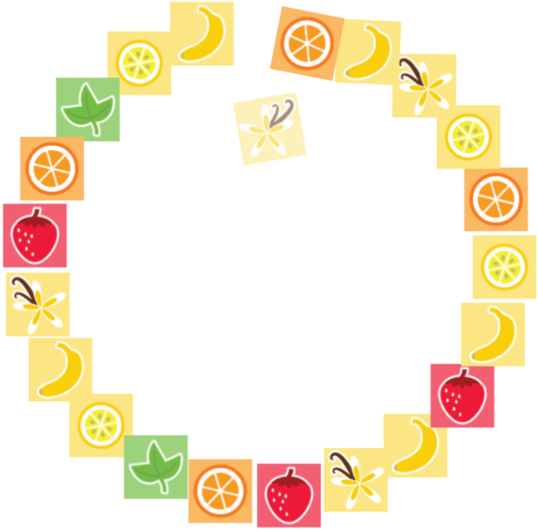


FIGURE B-3 Scenario 3 of the Custom Layout Extras example, showing a circular layout in the middle of animating new items into the list.

With this layout, all that's needed is a little CSS and a `layout` function. First, here's the extent of the markup in html/scenario3.html:

```
<div class="itemTemplate_s3" data-win-control="WinJS.Binding.Template">
    <img data-win-bind="src: picture; title: title" height="72" width="72" draggable="false" />
</div>
<div class="listView_s3" data-win-control="WinJS.UI.ListView" data-win-options="{
    itemDataSource: Data.smallList.dataSource, itemTemplate: select('.itemTemplate_s4'),
    layout: { type: CustomLayouts.CircleLayout }}">
</div>
```

and the full extent of the CSS, where you can see we'll use absolute positioning for the layout (css/scenario3.css):

```
.listView_s3 .win-container {
    position: absolute;
    top: 0;
    left: 0;
    transform: rotate(180deg);
    transition: transform 167ms linear, opacity 167ms linear;
}

.listView_s3 img {
    display: block;
}
```

Note that the `win-container` selector here targets the individual *items*, not the whole layout surface. Also note the small transition set up on the `transform` property, which is initially set to 180 degrees. This, along with some of the JavaScript code in `layout`, creates the fade-in spinning effect seen in Video B-2.

I'll leave it to you to look at the layout code in detail; it's mostly just calculating a position for each item along the circle. The spiral option that I've added plays a little with a variable radius as well.

What's very powerful with this example is that even in the circular layout you still get all the other ListView behaviors, such as selection, invocation, keyboarding, and so forth. Again, this is the main advantage of custom layouts over implementing your own collection control from scratch!

Virtualization

The next bit you can add to a custom layout is the `itemsFromRange` method to support virtualization. Virtualization means that the layout has a conversation with the ListView about what items are visible, because the ListView is creating and destroying elements as panning happens (clearly this isn't important for something like the circular layout where all the items are always in view). The layout, for its part, needs to have an understanding about instructing the app host how and where to render those visible items. The essence of that conversation is as follows:

- When the ListView is panned, it asks the layout about which items are visible for a particular pixel range through `itemsFromRange`, because the layout understands how big its items are and where they're placed.

- Once the ListView knows what items are visible, it calls layout with only those items in the *tree* argument. The layout, accordingly, sets the necessary styles and positioning on those items.

Scenario 2 of the HTML ListView custom layout sample now (the one from the Windows SDK, not the example in the companion content) shows the addition of virtualization support. The ListView's options, including the layout and its options, are also set from JavaScript in js/scenario2.js:

```
this._listView.layout = new SDKSample.Scenario2.StatusLayout(
    { itemInfo: this._itemInfo, cssClassSizeMap: cssClassSizeMap });
this._listView.itemTemplate = this._statusRenderer.bind(this);
this._listView.itemDataSource = Data.list.dataSource;
```

The custom layout here has two options: an itemInfo function and a cssClassSizeMap (very much like we did with the horizontal layout earlier). It uses these to perform size lookup on a per-item basis:

```
_itemInfo: function (itemIndex) {
    var item = Data.list.getItem(itemIndex);
    var cssClass = "statusItemSize";
    if (item.data.type === "photo") {
        cssClass = "photoItemSize";
    }

    return cssClass;
}

var cssClassSizeMap = {
    statusItemSize: { height: 90 }, // plus 10 pixels to incorporate the margin-top
    photoItemSize: { height: 260 }  // plus 10 pixels to incorporate the margin-top
};
```

Again, be very, very clear that such constructs are entirely specific to the custom layout and have no impact whatsoever on the ListView itself except as the layout implements methods like layout and itemsFromRange. In the sample, layout just uses this to cache various information about the items to improve performance. itemsFromRange, for its part, uses that cached size information to translate pixels to indexes for items. Here's how it's implemented in js/scenario2.js:

```
itemsFromRange: function (firstPixel, lastPixel) {
    var totalLength = 0;

    // Initialize firstIndex and lastIndex to be an empty range
    var firstIndex = 0;
    var lastIndex = -1;

    var firstItemFound = false;

    var itemCacheLength = this._itemCache.length;
    for (var i = 0; i < itemCacheLength; i++) {
        var item = this._itemCache[i];
        totalLength += item.height;

        // Find the firstIndex
        if (!firstItemFound && totalLength >= firstPixel) {
            firstIndex = item.index;
            lastIndex = firstIndex;
            firstItemFound = true;
        } else if (totalLength >= lastPixel) {
            // Find the lastIndex
            lastIndex = item.index;
            break;
        } else if (firstItemFound && i === itemCacheLength - 1) {
            // If we are at the end of the cache and we have found the firstItem,
            // the lastItem is in the range
            lastIndex = item.index;
        }
    }

    return { firstIndex: firstIndex, lastIndex: lastIndex };
},
```

The most important thing to keep in mind with `itemsFromRange` is to make it *fast*. This is why the sample employs a caching strategy, which would become essential if you had a large data source.

Along these lines, be careful about the properties you access on HTML elements—some of them can trigger layout passes especially if other layout-related properties have been changed. That is, the accuracy of dimensional properties requires that the layout is up to date before values can be returned, and clearly you don't want this to happen inside a method like `itemsFromRange`.

Grouping

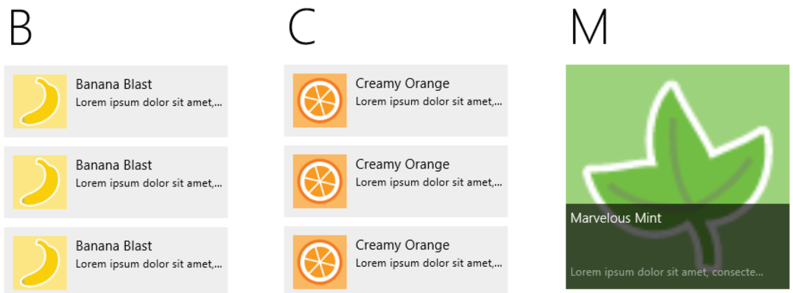
When a ListView has a group data source in addition to its item data source, what changes for a custom layout is that the *tree* argument to the `layout` method will contain more than one item. Each item in *tree* is a single group that contains whatever items belong to that group. That is, we've been using code like this to look at the items:

```
var container = tree[0].itemsContainer;
```

but now we need to process the whole tree array instead. Scenarios 3 and 4 of the HTML ListView custom layout sample (again, the SDK sample) do this, as shown here from js/scenario3.js:

```
layout: function (tree, changedRange, modifiedElements, modifiedGroups) {
    var offset = 0;

    var treeLength = tree.length;
    for (var i = 0; i < treeLength; i++) {
        this._layoutGroup(tree[i], offset);
        offset += tree[i].itemsContainer.items.length;
    }

    return WinJS.Promise.wrap();
},

// Private function that is responsible for laying out just one group
_layoutGroup: function (tree, offset) {
    var items = tree.itemsContainer.items;
    var itemsLength = items.length;
    for (var i = 0; i < itemsLength; i++) {
        // Get CSS class by item index
        var cssClass = this._itemInfo(i + offset);
        WinJS.Utilities.addClass(items[i], cssClass);
    }
}
```

All this is again a matter of adding CSS styles to various elements; the layout of the group headers themselves are defined by the *headerTemplate* element in html/scenario3.html and applicable styles in css/scenario3.css. As shown below, this is just a large character for each group (the groups are arranged vertically; I'm showing portions of each horizontally here):



B

Banana Blast
Lorem ipsum dolor sit amet,...

Banana Blast
Lorem ipsum dolor sit amet,...

Banana Blast
Lorem ipsum dolor sit amet,...

C

Creamy Orange
Lorem ipsum dolor sit amet,...

Creamy Orange
Lorem ipsum dolor sit amet,...

Creamy Orange
Lorem ipsum dolor sit amet,...

M

Marvelous Mint
Lorem ipsum dolor sit amet, consecte...

In short, support for grouping isn't tied to the implementation of specific `ILayout2` methods—it's primarily a matter of how you implement the `layout` method itself.

You can, of course, implement both grouping and virtualization together by including `itemsFromRange`, as we saw in the previous section. Scenario 4 of the sample demonstrates this by basically combining the code from scenarios 2 and 3.

The Other Stuff

Let's now look at the remaining methods of `ILayout2`, most of which are demonstrated in scenario 4 of the Custom Layout Extras example, which extends the minimum horizontal layout in scenario 2.

Some of these methods must return an object that contains an item `index` and a `type` (a value from `WinJS.UI.ObjectType`). This object doesn't have an assigned class but looks like these examples:

```
return { type: WinJS.UI.ObjectType.item, index: itemIndex };
return { type: WinJS.UI.ObjectType.groupHeader, index: headerIndex };
```

The `hitTest` method should also include an `insertAfterIndex` in the object it returns so as to indicate the actual insertion point of a moved item.

Keyboard Support: getAdjacent

The ListView calls getAdjacent in response to the arrow keys as well as page up and page down, because the layout object is what knows how items are arranged in relation to one another. Without this method, the default behavior (as you can see in scenario 2) is that the right arrow, down arrow, and page down move the focus to the next item in the list, and left arrow, up arrow, and page up go to the previous item. (Home and End always go to the first and last item.)

In a top-to-bottom grid layout, however, we want the up, down, page up, and page down to work by rows (one row or three rows at a time). We implement getAdjacent, then, which accepts an *item* object (containing index and type properties as in the return object) and a WinJS.UtilitiesKey and which returns an object describing the next item to get the focus. (If you simply return an object with index set to item.index+1 or item.index-1, for forward and backward keys, respectively, and type: WinJS.UI.ObjectType.item, you'll duplicate the default behavior.)

In our case we need to determine how many items are in a row. This is fairly straightforward. Remember the site object we get through initialize? It contains a viewportSize property whose width tells us the width of the ListView. If we divide this by the item width—and it's best to have a property on the layout object through which the app can specify this as a fixed size—we get the items per row. Because this value will change only when the layout is recomputed, we can do this in the layout method:

```
layout: function() {
    this._itemsPerRow = Math.floor(this._site.viewportSize.width / this._itemWidth);
    this._totalRows = Math.floor(this._site.itemCount._value / this._itemsPerRow);

    //We don't need to do anything else.
    return WinJS.Promise.as();
},
```

Assuming that we have an itemWidth property on the layout that's been set appropriately for the real item size (content size + borders, etc.), we can implement getAdjacent as follows. Here, page up and page down move three rows at a time, and up/down arrow one row:

```
getAdjacent: function (item, key) {
    var Key = WinJS.Utilities.Key;
    var index = item.index;
    var curRow = Math.floor(index / this._itemsPerRow);
    var curCol = index % this._itemsPerRow;
    var newRow;

    //The ListView is gracious enough to ignore our return index if it's out of bounds,
    //so we don't have to check for that here.

    switch (key) {
        case Key.rightArrow:
            index = index + 1;
            break;

        case Key.downArrow:
            index = index + this._itemsPerRow;
            break;

        case Key.pageDown:
            //If we page down past the last item, this will go to the last item
            newRow = Math.min(curRow + 3, this._totalRows);
            index = curCol + (newRow * this._itemsPerRow);
            break;

        case Key.leftArrow:
            index = index - 1;
            break;

        case Key.upArrow:
            index = index - this._itemsPerRow;
            break;

        case Key.pageUp:
            newRow = Math.max(curRow - 3, 0);
            index = curCol + (newRow * this._itemsPerRow);
            break;
    }

    return { type: WinJS.UI.ObjectType.item, index: index };
},
```

With this, we get good keyboard navigation within the layout. Note the comment that the ListView will ignore invalid index values; we don't have to sweat over validating what we return.

Drag and Drop: dragLeave, dragOver, and hitTest

One drag-and-drop scenario for a ListView is that it can serve as a general drop source by setting its `itemsDraggable` property to `true`. If so, the ListView will check for and call the layout's `dragLeave` method whenever an item is dragged out of the ListView control, as well as when an item is dragged and dropped back in the list.

This method has no arguments and no return value, so it's simply a notification for the layout to take any desired action. To be honest, there's not too much it can do in this case. It's more interesting when items are reorderable as well—that is, when the ListView's `itemsReorderable` property is also `true` (html/scenario4.html):

```
<div class="listView_s4" data-win-control="WinJS.UI.ListView"
       data-win-options="{
       itemDataSource: Data.list.dataSource,
       itemTemplate: select('.itemTemplate_s4'),
       itemsDraggable: true,
       itemsReorderable: true,
       layout: {type: CustomLayouts.VerticalGrid_Flex4, itemWidth: 65, itemHeight: 65}}">
</div>
```

Here you must also implement hitTest and dragOver together. hitTest is pretty simple to think about: given x/y coordinates within the layout surface, return an item object for whatever is at those coordinates where the object contains type, index, and insertAfterIndex properties, as described earlier. In scenario 4 of the example, we can use the same itemWidth/itemHeight properties that help with getAdjacent (js/scenario4.js):

```
_indexFromCoordinates: function (x, y) {
    var row = Math.floor(y / this._itemHeight);
    var col = Math.floor(x / this._itemWidth);
    return (row * this._itemsPerRow) + col;
},

hitTest: function (x, y) {
    var index = this._indexFromCoordinates(x, y);

    //Only log the output if the index changes.
    if (this._lastIndex != index) {
        console.log("hitTest on (" + x + ", " + y + "), index = " + index);
        this._lastIndex = index;
    }

    return { type: WinJS.UI.ObjectType.item, index: index, insertAfterIndex: index - 1 };
},
```

Note If you support variable item sizes, you can't rely on fixed values here and need to have an itemInfo function to retrieve these dimensions individually. You'd want to cache these values within the layout method as well so that hitTest can return more quickly.

The insertAfterIndex is important for reordering the ListView, because the control automatically moves an item dragged within the list to this index. Note that insertAfterIndex should be set to -1 to insert at the beginning of the list.

With hitTest in place, the dragOver method is then called whenever the index from hitTest changes. It receives the x/y coordinates of the drag operation (along with an object called *dragInfo* that's for the ListView's internal use). The method doesn't have a return value, so its whole purpose is to give your layout an opportunity to visually indicate the result of a drop. For example, you can highlight the insertion point, show a little wiggle room in the layout (the built-in layouts move hit-tested items by 12px), or whatever else you want. You then use dragLeave, of course, to reset that indicator. (If you want to look at what the built-in layouts do for dragOver, look in the ui.js file of WinJS for the *_LayoutCommon_dragOver* function.)

In the example (still in scenario 4) I just rotate the item at the drop point a little (see Video B-3) and make sure to clear that transform on dragLeave:

```
dragOver: function (x, y, dragInfo) {
    //Get the index of the item on which we're dropping
    var index = this._indexFromCoordinates(x, y);

    console.log("dragOver on index = " + index);
    //Get the element and scale it a little (like a button press)
    var element = this._site.tree[0].itemsContainer.items[index];
    element && this._addAnimateDropPoint(element);
    this._lastRotatedElement && this._clearAnimateDropPoint(this._lastRotatedElement);
    this._lastRotatedElement = element;
},

dragLeave: function () {
    console.log("dragLeave");
    if (this._lastRotatedElement) {
        this._clearAnimateDropPoint(this._lastRotatedElement);
        this._lastRotatedElement = null;
    }
},

_addAnimateDropPoint: function (element) {
    element.style.transition = "transform ease 167ms";
    element.style.transform = "rotate(-20deg)";
},

_clearAnimateDropPoint: function (element) {
    element.style.transition = "";
    element.style.transform = "";
}
```

With this, the ListView will automatically take care of moving items around in the data source without any other special handling. In the example, I've added identification numbers to each of the items in the list so that you can see the effects.

Animations: setupAnimations and executeAnimations

The last two members of `ILayout2` are `setupAnimations` and `executeAnimations`, whose purpose is to animate the layout in response to a change in the data source, as when an item is dragged within the ListView.

These are somewhat involved, however, and the best place to learn about these is in the WinJS source code itself. Search for "Animation cycle," and you'll find a page-long comment on the subject!

Appendix C

Additional Networking Topics

In this appendix:

- XMLHttpRequest and WinJS.xhr
- Breaking up large files (background transfer API)
- Multipart uploads (background transfer API)
- Notes on Encryption, Decryption, Data Protection, and Certificates
- Syndication: RSS and AtomPub APIs in WinRT
- Sockets
- The Credential Picker UI
- Other Networking SDK Samples

XMLHttpRequest and WinJS.xhr

Transferring data to and from web services through HTTP requests is a common activity for Windows Store apps, especially those written in JavaScript for which handling XML and/or JSON is straightforward. For this purpose there is the Windows.Web.Http.HttpClient API, but apps can also use the XMLHttpRequest object as well as the WinJS.xhr wrapper that turns the XMLHttpRequest structure into a simple promise. For the purposes of this section I'll refer to both of these together as just XHR.

To build on what we already covered in the "HTTP Requests" section in Chapter 4, "Web Content and Services," I need to make a few other points where XHR is concerned, most of which come from the section in the documentation entitled Connecting to a web service.

First, Downloading different types of content provides the details of the different content types supported by XHR for Windows Store apps, summarized here:

Type	Use	responseText	responseXML
arraybuffer	Binary content as an array of Int8 or Int64, or another integer or float type.	undefined	undefined
Blob	Binary content represented as a single entity.	undefined	undefined
document	An XML DOM object representing XML content (MIME type of text/XML).	undefined	The XML content
json	JSON strings.	The JSON string	undefined
ms-stream	Streaming data; see XMLHttpRequest enhancements.	undefined	undefined
Text	Text (the default).	The text string	undefined

Second, know that XHR responses can be automatically cached, meaning that later requests to the same URI might return old data. To resend the request despite the cache, add an *If-Modified-Since* HTTP header, as shown on How to ensure that WinJS.xhr resends requests.

Along similar lines, you can wrap a `WinJS.xhr` operation in another promise to encapsulate automatic retries if there is an error in any given request. That is, build your retry logic around the core XHR operation, with the result stored in some variable. Then place that whole block of code within `WinJS.Promise.as` (or a new `WinJS.Promise`) and use that elsewhere in the app.

In each XHR attempt, remember that you can also use `WinJS.Promise.timeout` in conjunction with `WinJS.Xhr`, as described on Setting timeout values with WinJS.xhr., because `WinJS.xhr` doesn't have a timeout notion directly. You can, of course, set a timeout in the raw `XMLHttpRequest` object, but that would mean rebuilding everything that `WinJS.xhr` already does or copying it from the WinJS source code and making modifications.

Generally speaking, XHR headers are accessible to the app with the exception of cookies (the *set-cookie* and *set-cookie2* headers)—these are filtered out by design for XHR done from a local context. They are not filtered for XHR from the web context. Of course, access to cookies is one of the benefits of `Windows.Web.Http.HttpClient`.

Finally, avoid using XHR for large file transfers because such operations will be suspended when the app is suspended. Use the Background Transfer API instead (see Chapter 4), which uses HTTP requests under the covers, so your web services won't know the difference anyway!

Tips and Tricks for WinJS.xhr

Without opening the whole can of worms that is `XMLHttpRequest`, it's useful to look at just a couple of additional points around `WinJS.xhr`. This section is primarily provided for developers who might still be targeting Windows 8.0 where the preferred WinRT `HttpClient` API is not available.

First, notice that the single argument to `WinJS.xhr` is an object that can contain a number of properties. The `url` property is the most common, of course, but you can also set the `type` (defaults to "GET") and the `responseType` for other sorts of transactions, supply `user` and `password` credentials, set `headers` (such as *If-Modified-Since* with a date to control caching), and provide whatever other additional `data` is needed for the request (such as query parameters for XHR to a database). You can also supply a `customRequestInitializer` function that will be called with the `XMLHttpRequest` object just before it's sent, allowing you to perform anything else you need at that moment.

The second tip is setting a timeout on the request. You can use the `customRequestInitializer` for this purpose, setting the `XMLHttpRequest.timeout` property and possibly handling the `ontimeout` event. Alternately, use the `WinJS.Promise.timeout` function to set a timeout period after which the `WinJS.xhr` promise (and the async operation connected to it) will be canceled. Canceling is accomplished by simply calling a promise's `cancel` method. Refer to "The WinJS.Promise Class" in Appendix A, "Demystifying Promises," for details on `timeout`.

You might have need to wrap WinJS.xhr in another promise, perhaps to encapsulate other intermediate processing with the request while the rest of your code just uses the returned promise as usual. In conjunction with a timeout, this can also be used to implement a multiple retry mechanism.

Next, if you need to coordinate multiple requests together, you can use WinJS.Promise.join, which is again covered in Chapter 3 in the section "Joining Parallel Promises."

Finally, for Windows Store apps, using XHR with localhost: URIs (local loopback) is blocked by design. During development, however, this is very useful for debugging a service without deploying it. You can enable local loopback in Visual Studio by opening the project properties dialog (Project menu > <project> Properties…), selecting Debugging on the left side, and setting Allow Local Network Loopback to Yes. Using the localhost is discussed also in Chapter 4.

Breaking Up Large Files (Background Transfer API)

Because the outbound (upload) transfer rates of most broadband connections are significantly slower than the inbound (download) rates and might have other limitations, uploading a large file to a server (generally using the background transfer API) is typically a riskier operation than a large download. If an error occurs during the upload, it can invalidate the entire transfer—a frustrating occurrence if you've already been waiting an hour for that upload to complete!

For this reason, a cloud service might allow a large file to be transferred in discrete chunks, each of which is sent as a separate HTTP request, with the server reassembling the single file from those requests. This minimizes, or at least reduces, the overall impact of connectivity hiccups.

From the client's point of view, each piece would be transferred with an individual UploadOperation; that much is obvious. The tricky part is breaking up a large file in the first place. With a lot of elbow grease—and what would likely end up being a complex chain of nested async operations—it is possible to create a bunch of temporary files from the single source. If you're up to a challenge, I invite to you write such a routine and post it somewhere for the rest of us to see!

But there is an easier path: BackgroundUploader.createUploadFromStreamAsync, through which you can create separate UploadOperation objects for different segments of the stream. Given a StorageFile for the source, start by calling its openReadAsync method, the result of which is an IRandomAccessStreamWithContentType object. Through its getInputStreamAt method you then obtain an IInputStream for each starting point in the stream (that is, at each offset depending on your segment size). You then create an UploadOperation with each input stream by using createUploadFromStreamAsync. The last requirement is to tell that operation to consume only some portion of that stream. You do this by calling its setRequestHeader("content-length", <length>) where <length> is the size of the segment plus the size of other data in the request; you'll also want to add a header to identify the segment for that particular upload. After all this, call each operation's startAsync method to begin its transfer.

Multipart Uploads (Background Transfer API)

In addition to the `createUpload` and `createUploadFromStreamAsync` methods, the [`BackgroundUploader`]() provides another method, called `createUploadAsync` (with three variants), which handles what are called *multipart uploads*.

From the server's point of view, a multipart upload is a *single* HTTP request that contains various pieces of information (the parts), such as app identifiers, authorization tokens, and so forth, along with file content, where each part is possibly separated by a specific boundary string. Such uploads are used by online services like Flickr and YouTube, each of which accepts a request with a multipart Content-Type. (See [Content-type: multipart]() for a reference.) For example, as shown on [Uploading Photos – POST Example](), Flickr wants a request with the content type of `multipart/form-data`, followed by parts for `api_key`, `auth_token`, `api_sig`, `photo`, and finally the file contents. With YouTube, as described on [YouTube API v2.0 – Direct Uploading](), it wants a content type of `multipart/related` with parts containing the XML request data, the video content type, and then the binary file data.

The background uploader supports all this through the [`BackgroundUploader.createUploadAsync`]() method. (Note the `Async` suffix that separates this from the synchronous `createUpload`.) There are three variants of this method. The first takes the server URI to receive the upload and an array of [`BackgroundTransferContentPart`]() objects, each of which represents one part of the upload. The resulting operation will send a request with a content type of `multipart/form-data` with a random GUID for a boundary string. The second variation of `createUploadAsync` allows you to specify the content type directly (through the sub-type, such as `related`), and the third variation then adds the boundary string. That is, assuming `parts` is the array of parts, the methods look like this:

```
var uploadOpPromise1 = uploader.createUploadAsync(uri, parts);
var uploadOpPromise2 = uploader.createUploadAsync(uri, parts, "related");
var uploadOpPromise3 = uploader.createUploadAsync(uri, parts, "form-data", "-------123456");
```

To create each part, first create a `BackgroundTransferContentPart` by using one of its [three constructors]():

- `new BackgroundContentPart()` Creates a default part.
- `new BackgroundContentPart(<name>)` Creates a part with a given name.
- `new BackgroundContentPart(<name>, <file>)` Creates a part with a given name and a local filename.

In each case you further initialize the part with a call to its `setText`, `setHeader`, and `setFile` methods. The first, `setText`, assigns a value to that part. The second, `setHeader`, can be called multiple times to supply header values for the part. The third, `setFile`, is how you provide the `StorageFile` to a part created with the third variant above.

Scenario 2 of the Background transfer sample shows the latter using an array of random files that you choose from the file picker, but probably few services would accept a request of this nature. Let's instead look at how we'd create the multipart request for Flickr shown on Uploading Photos – POST Example. For this purpose I've created the MultipartUpload example in the appendices' companion content. Here's the code from js/uploadMultipart.js that creates all the necessary parts for the tinyimage.jpg file in the app package:

```
// The file and uri variables are already set by this time. bt is a namespace shortcut
var bt = Windows.Networking.BackgroundTransfer;
var uploader = new bt.BackgroundUploader();
var contentParts = [];

// Instead of sending multiple files (as in the original sample), we'll create those parts that
// match the POST example for Flickr on http://www.flickr.com/services/api/upload.example.html
var part;

part = new bt.BackgroundTransferContentPart();
part.setHeader("Content-Disposition", "form-data; name=\"api_key\"");
part.setText("3632623532453245");
contentParts.push(part);

part = new bt.BackgroundTransferContentPart();
part.setHeader("Content-Disposition", "form-data; name=\"auth_token\"");
part.setText("436436545");
contentParts.push(part);

part = new bt.BackgroundTransferContentPart();
part.setHeader("Content-Disposition", "form-data; name=\"api_sig\"");
part.setText("43732850932746573245");
contentParts.push(part);

part = new bt.BackgroundTransferContentPart();
part.setHeader("Content-Disposition", "form-data; name=\"photo\"; filename=\"" + file.name +
"\"");
part.setHeader("Content-Type", "image/jpeg");
part.setFile(file);
contentParts.push(part);

// Create a new upload operation specifying a boundary string.
uploader.createUploadAsync(uri, contentParts,
    "form-data", "-----------------------------7d44e178b0434")
   .then(function (uploadOperation) {
       // Start the upload and persist the promise
       upload = uploadOperation;
       promise = uploadOperation.startAsync().then(complete, error, progress);
   }
);
```

The resulting request will look like this, very similar to what's shown on the Flickr page (just with some extra headers):

```
POST /website/multipartupload.aspx HTTP/1.1
Cache-Control=no-cache
Connection=Keep-Alive
Content-Length=1328
Content-Type=multipart/form-data; boundary="-----------------------------7d44e178b0434"
Accept=*/*
Accept-Encoding=gzip, deflate
Host=localhost:60355
User-Agent=Mozilla/5.0 (compatible; MSIE 10.0; Windows NT 6.2; Win64; x64; Trident/6.0; Touch)
UA-CPU=AMD64
-------------------------------7d44e178b0434
Content-Disposition: form-data; name="api_key"

3632623532453245
-------------------------------7d44e178b0434
Content-Disposition: form-data; name="auth_token"

436436545
-------------------------------7d44e178b0434
Content-Disposition: form-data; name="api_sig"

43732850932746573245
-------------------------------7d44e178b0434
Content-Disposition: form-data; name="photo"; filename="tinysquare.jpg"
Content-Type: image/jpeg

{RAW JFIF DATA}
-------------------------------7d44e178b0434--
```

To run the sample and also see how this request is received, you'll need two things. First, set up your localhost server as described in "Sidebar: Using the Localhost" in Chapter 4. Then install Visual Studio Express *for Web* (which is free) through the Web Platform Installer. Now go to the MultipartUpload-Server folder in appendices' companion content, load website.sln into Visual Studio Express for Web, open MultipartUploadServer.aspx, and set a breakpoint on the first `if` statement inside the `Page_Load` method. Start the site in the debugger (which runs it in Internet Explorer), which opens that page on a localhost debugging port (and click Continue in Visual Studio when you hit the breakpoint). Copy that page's URI from Internet Explorer for the next step.

Switch to the MultipartUpload example running in Visual Studio for Windows, paste that URI into the URI field, and click the Start Multipart Transfer. When the upload operation's `startAsync` is called, you should hit the server page breakpoint in Visual Studio for Web. You can step through that code if you want and examine the Request object; in the end, the code will copy the request into a file named *multipart-request.txt* on that server. This will contain the request contents as above, where you can see the relationship between how you set up the parts in the client and how they're received by the server.

Notes on Encryption, Decryption, Data Protection, and Certificates

The documentation on the Windows Developer Center along with APIs in the `Windows.Security` namespace are helpful to know about where protecting user credentials and other data is concerned. One key resource is the How to secure connections and authenticate requests topic; another is the Banking with strong authentication sample, which demonstrates secure authentication and communication over the Internet. A full writeup on this sample is found on Tailored banking app code walkthrough.

As for WinRT APIs, first is `Windows.Security.Cryptography`. Here you'll find the `CryptographicBuffer` class that can encode and decode strings in hexadecimal and base64 (UTF-8 or UTF-16) and also provide random numbers and a byte array full of such randomness. Refer to scenario 1 of the CryptoWinRT sample for some demonstrations, as well as scenarios 2 and 3 of the Web authentication broker sample. WinRT's base64 encoding is fully compatible with the JavaScript `atob` and `btoa` functions.

Next is `Windows.Security.Cryptography.Core`, which is truly about encryption and decryption according to various algorithms. See the Encryption topic, scenarios 2–8 of the CryptoWinRT sample, and again scenarios 2 and 3 of the Web authentication broker sample.

Third is `Windows.Security.Cryptography.DataProtection`, whose single class, `DataProtectionProvider`, deals with protecting and unprotecting both static data and a data stream. This applies only to apps that declare the *Enterprise Authentication* capability. For details, refer to Data protection API along with scenarios 9 and 10 of the CryptoWinRT sample.

Fourth, `Windows.Security.Cryptography.Certificates` provides several classes through which you can create certificate requests and install certificate responses. Refer to Working with certificates and the Certificate enrollment sample for more.

Fifth, some of the early W3C cryptography APIs have made their way into the app host, accessed through the `window.msCrypto` object.

And lastly it's worth at least listing the API under `Windows.Security.ExchangeActiveSync-Provisioning` for which there is the EAS policies for mail clients sample. I'm assuming that if you know why you'd want to look into this, well, you'll know!

Syndication: RSS, AtomPub, and XML APIs in WinRT

When we first looked at doing HTTP requests in Chapter 4, we grabbed the RSS feed from the Windows Developer Blog with the URI http://blogs.msdn.com/b/windowsappdev/rss.aspx. We learned then that `WinJS.xhr` returned a promise, the result of which contained a `responseXML` property, which is itself a

DomParser through which you can traverse the DOM structure and so forth.

Working with syndicated feeds by using straight HTTP requests is completely supported for Windows Store apps. In fact, the How to create a mashup topic in the documentation describes exactly this process, components of which are demonstrated in the Integrating content and controls from web services sample.

That said, WinRT offers additional APIs for dealing with syndicated content in a more structured manner, which could be better suited for some programming languages. One, Windows.Web.Syndication, offers a more structured way to work with RSS feeds. The other, Windows.Web.AtomPub, provides a means to publish and manage feed entries.

There is also an API for dealing with XML in Windows.Data.Xml.Dom (see the Windows Runtime XML data API sample), but this API is somewhat redundant with the built-in XML/DOM APIs present in JavaScript.

Reading RSS Feeds

The primary class within Windows.Web.Syndication is the SyndicationClient. To work with any given feed, you create an instance of this class and set any necessary properties. These are serverCredential (a PasswordCredential), proxyCredential (another PasswordCredential), timeout (in milliseconds; default is 30000 or 30 seconds), maxResponseBufferSize (a means to protect from potentially malicious servers), and bypassCacheOnRetrieve (a Boolean to indicate whether to always obtain new data from the server). You can also make as many calls to its setRequestHeader method (passing a name and value) to configure the HTTP request header.

The final step is to then call the SyndicationClient.retrieveFeedAsync method with the URI of the desired RSS feed (a Windows.Foundation.Uri). Here's an example derived from the Syndication sample, which retrieves the RSS feed for the Building Windows 8 blog:

```
uri = new Windows.Foundation.Uri("http://blogs.msdn.com/b/b8/rss.aspx");
var client = new Windows.Web.Syndication.SyndicationClient();
client.bypassCacheOnRetrieve = true;
client.setRequestHeader("User-Agent",
   "Mozilla/5.0 (compatible; MSIE 10.0; Windows NT 6.2; WOW64; Trident/6.0)");

client.retrieveFeedAsync(uri).done(function (feed) {
   // feed is a SyndicationFeed object

}
```

The result of retrieveFeedAsync is a Windows.Web.Syndication.SyndicationFeed object; that is, the SyndicationClient is what you use to talk to the service, and when you retrieve the feed you get an object though which you can then process the feed itself. If you take a look at SyndicationFeed by using the link above, you'll see that it's wholly stocked with properties that represent all the parts of the feed, such as authors, categories, items, title, and so forth. Some of these are represented themselves by other classes in Windows.Web.Syndication, or collections of them, where simpler types aren't sufficient: SyndicationAttribute, SyndicationCategory, SyndicationContent, SyndicationGenerator, SyndicationItem, SyndicationLink, SyndicationNode, SyndicationPerson, and SyndicationText. I'll leave the many details to the documentation.

We can see some of this in the sample, picking up from inside the completed handler for retrieveFeedAsync. Let me offer a more annotated version of that code:

```
client.retrieveFeedAsync(uri).done(function (feed) {
    currentFeed = feed;

    var title = "(no title)";

    // currentFeed.title is a SyndicationText object
    if (currentFeed.title) {
        title = currentFeed.title.text;
    }

    // currentFeed.items is a SyndicationItem collection (array)
    currentItemIndex = 0;
    if (currentFeed.items.size > 0) {
        displayCurrentItem();
    }
}

// ...

function displayCurrentItem() {
    // item will be a SyndicationItem

    var item = currentFeed.items[currentItemIndex];

    // Display item number.
    document.getElementById("scenario1Index").innerText = (currentItemIndex + 1) + " of "
        + currentFeed.items.size;

    // Display title (item.title is another SyndicationText).
    var title = "(no title)";
    if (item.title) {
        title = item.title.text;
    }
    document.getElementById("scenario1ItemTitle").innerText = title;

    // Display the main link (item.links is a collection of SyndicationLink objects).
    var link = "";
    if (item.links.size > 0) {
        link = item.links[0].uri.absoluteUri;
    }

    var scenario1Link = document.getElementById("scenario1Link");
    scenario1Link.innerText = link;
    scenario1Link.href = link;

    // Display the body as HTML (item.content is a SyndicationContent object, item.summary is
    // a SyndicationText object).
    var content = "(no content)";
    if (item.content) {
        content = item.content.text;
    }
    else if (item.summary) {
        content = item.summary.text;
    }
    document.getElementById("scenario1WebView").innerHTML = window.toStaticHTML(content);

    // Display element extensions. The elementExtensions collection contains all the additional
    // child elements within the current element that do not belong to the Atom or RSS standards
    // (e.g., Dublin Core extension elements). By creating an array of these, we can create a
    // WinJS.Binding.List that's easily displayed in a ListView.
    var bindableNodes = [];
    for (var i = 0; i < item.elementExtensions.size; i++) {
        var bindableNode = {
            nodeName: item.elementExtensions[i].nodeName,
            nodeNamespace: item.elementExtensions[i].nodeNamespace,
            nodeValue: item.elementExtensions[i].nodeValue,
        };
        bindableNodes.push(bindableNode);
    }
    var dataList = new WinJS.Binding.List(bindableNodes);
    var listView = document.getElementById("extensionsListView").winControl;
    WinJS.UI.setOptions(listView, { itemDataSource: dataList.dataSource });
}
```

It's probably obvious that the API, under the covers, is probably just using the XmlDocument API to retrieve all these properties. In fact, its getXmlDocument returns that XmlDocument if you want to access it yourself.

You can also create a SyndicationFeed object around the XML for a feed you might already have. For example, if you obtain the feed contents by using WinJS.xhr, you can create a new SyndicationFeed object and call its load method with the request's responseXML. Then you can work with the feed through the class hierarchy. When using the Windows.Web.AtomPub API to manage a feed, you also create a new or updated SyndicationItem to send across the wire, settings its values through the other objects in its hierarchy. We'll see this in the next section.

If retrieveFeedAsync throws an exception, by the way, which would be picked up by an error handler you provide to the promise's done method, you can turn the error code into a SyndicationErrorStatus value. Here's how it's used in the sample's error handler:

```
function onError(err) {
    // Match error number with a SyndicationErrorStatus value. Use
    // Windows.Web.WebErrorStatus.getStatus() to retrieve HTTP error status codes.
    var errorStatus = Windows.Web.Syndication.SyndicationError.getStatus(err.number);
    if (errorStatus === Windows.Web.Syndication.SyndicationErrorStatus.invalidXml) {
        displayLog("An invalid XML exception was thrown. Please make sure to use a URI that"
            + "points to a RSS or Atom feed.");
    }
}
```

As a final note, the Feed reader sample in the SDK provides another demonstration of the Windows.Web.Syndication API. Its operation is fully described on the Feed reader sample page in the documentation.

Using AtomPub

On the flip side of reading an RSS feed, as we've just seen, is the need to possibly add, remove, and edit entries on a feed, as with an app that lets the user actively manage a specific blog or site.

The API for this is found in Windows.Web.AtomPub and demonstrated in the AtomPub sample. The main class is the AtomPubClient that encapsulates all the operations of the AtomPub protocol. It has methods like createResourceAsync, retrieveResourceAsync, updateResourceAsync, and deleteResourceAsync for working with those entries, where each resource is identified with a URI and a SyndicationItem object, as appropriate. Media resources for entries are managed through createMediaResourceAsync and similarly named methods, where the resource is provided as an IInputStream.

The AtomPubClient also has retrieveFeedAsync and setRequestHeader methods that do the same as the SyndicationClient methods of the same names, along with a few similar properties like serverCredential, timeout, and bypassCacheOnRetrieve. Another method, retrieveServiceDocumentAsync, provides the workspaces/service documents for the feed (in the form of a Windows.Web.AtomPub.ServiceDocument object).

Again, the AtomPub sample demonstrates the different operations: retrieve (Scenario 1), create (Scenario 2), delete (Scenario 3), and update (Scenario 4). Here's how it first creates the AtomPubClient object (see js/common.js), assuming there are credentials:

```
function createClient() {
    client = new Windows.Web.AtomPub.AtomPubClient();
    client.bypassCacheOnRetrieve = true;

    var credential = new Windows.Security.Credentials.PasswordCredential();
    credential.userName = document.getElementById("userNameField").value;
    credential.password = document.getElementById("passwordField").value;
    client.serverCredential = credential;
}
```

Updating an entry (js/update.js) then looks like this, where the update is represented by a newly created SyndicationItem:

```
function getCurrentItem() {
    if (currentFeed) {
        return currentFeed.items[currentItemIndex];
    }
    return null;
}

var resourceUri = new Windows.Foundation.Uri( /* service address */ );
createClient();
```

```
var currentItem = getCurrentItem();

if (!currentItem) {
    return;
}

// Update the item
var updatedItem = new Windows.Web.Syndication.SyndicationItem();
var title = document.getElementById("titleField").value;
updatedItem.title = new Windows.Web.Syndication.SyndicationText(title,
    Windows.Web.Syndication.SyndicationTextType.text);
var content = document.getElementById("bodyField").value;
updatedItem.content = new Windows.Web.Syndication.SyndicationContent(content,
    Windows.Web.Syndication.SyndicationTextType.html);

client.updateResourceAsync(currentItem.editUri, updatedItem).done(function () {
    displayStatus("Updating item completed.");
}, onError);
```

Error handling in this case works with the `Window.Web.WebError` class (see js/common.js):

```
function onError(err) {
    displayError(err);

    // Match error number with a WebErrorStatus value, in order to deal with a specific error.
    var errorStatus = Windows.Web.WebError.getStatus(err.number);
    if (errorStatus === Windows.Web.WebErrorStatus.unauthorized) {
        displayLog("Wrong username or password!");
    }
}
```

Sockets

Sockets are a fundamental transport for networking and devices. Unlike HTTP requests, where a client sends a request to a server and the server responds—essentially an isolated transaction—sockets are a connection between client and server IP ports such that either one can send information to the other at any time. A similar mechanism can be found in the Windows Push Notification Service (WNS, see Chapter 16, "Alive with Activity"), but WNS is limited to notifications and is specifically designed to issue tile updates or notifications for apps that aren't running. Sockets, on the other hand, are for ongoing data exchange between a server (or a device acting as one)and a running client.

Sockets are generally used when there isn't a higher-level API or other abstraction for your particular scenario, when there's a custom protocol involved, when you need two-way communication, or when it makes sense to minimize the overhead of each exchange. Consider HTTP, a protocol that is itself built on lower-level sockets. A single HTTP request generally includes headers and lots of other information beyond just the bit of data involved, so it's an inefficient transport when you need to send lots of little bits. It's better to connect directly with the server and exchange data with a minimized custom protocol. VoIP is another example where sockets work well, as are multicast scenarios like multiplayer games. In the latter, one player's machine, acting as a server within a local subnet, can broadcast a message to all the other players, and vice versa, again with minimal overhead.

In the world of sockets, exchanging data can happen in two ways: as discrete packets/messages (like water balloons) or as a continuous stream (like water running through a hose). These are called datagram sockets and stream sockets, respectively, and both are supported through the WinRT API. WinRT also supports both forms of exchange through the WebSocket protocol, a technology originally created for web browsers and web servers that has become increasingly interesting for general purpose use within apps. All of the applicable classes can be found in the `Windows.Networking.Sockets` API (or occasionally in the parent `Windows.Networking` namespace), as we'll see in the following sections. Note that because some overlap between the different types of sockets exists, these sections are meant to be read in sequence so that I don't have to repeat myself too much!

> **Suspend and resume with sockets** As a general note, always close an open socket when your app is suspended, and reopen it when the app is resumed.

Datagram Sockets

In the language of sockets, a water balloon is called a datagram, a bundle of information sent from one end of the socket to the other—even without a prior connection—according to the User Datagram Protocol (UDP) standard. UDP, as I summarize here from its description on Wikipedia, is simple, stateless, unidirectional, and transaction-oriented. It has minimal overhead and lacks retransmission delays, and for these reasons it cannot guarantee that a datagram will actually be delivered. Thus, it's used where error checking and correction aren't necessary or where they are done by the apps involved rather than at the network interface level. In a VoIP scenario, for example, this allows data packets to just be dropped if they cannot be delivered, rather than having everything involved wait for a delayed packet. As a result, the quality of the audio might suffer, but it won't start stuttering or make your friends and colleagues sound like they're from another galaxy. In short, UDP might be unreliable, but it minimizes latency. Higher-level protocols like the Real-time Transport Protocol (RTP) and the Real Time Streaming Protocol (RTSP) are built on UDP.

A Windows Store app works with this transport—either as a client or a server—using the `DatagramSocket` class, which you instantiate with the `new` operator to set up a specific connection and listen for messages:

```
var listener = new Windows.Networking.Sockets.DatagramSocket();
```

On either side of the conversation, the next step is to listen for the object's `messagereceived` event:

```
// Event from WinRT: remember to call removeEventListener as needed
listener.addEventListener("messagereceived", onMessageReceived);
```

When data arrives, the handler receives a—wait for it!—DatagramSocketMessageReceivedEventArgs object (that's a mouthful). This contains localAddress and remoteAddress properties, both of which are a Windows.Networking.HostName object that contains the IP address, a display name, and a few other bits. See the "Network Information (the Network Object Roster)" section early in Chapter 4 for details. The event args also contains a remotePort string. More importantly, though, are the two methods from which you extract the data. One is getDataStream, which returns an IInputStream through which you can read sequential bytes. The other is getDataReader, which returns a Windows.Storage.Streams.DataReader object. As discussed in Chapter 10, "The Story of State, Part 1" in the section "Folders, Files, and Streams," this is a higher-level abstraction built on top of the IInputStream that helps you read specific data types directly. Clearly, if you know the data structure you expect to receive in the message, using the DataReader will relieve you from doing type conversions yourself.

Of course, to get any kind of data from a socket, you need to connect it to something. For this purpose DatagramSocket includes a few methods for establishing and managing a connection:

- connectAsync Starts a connection operation given a HostName object and a service name (or UDP port, a string) of the remote network destination. This is used to create a one-way client to server connection.
- Another form of connectAsync takes a Windows.Networking.EndpointPair object that specifies host and service names for both local and remote endpoints. This is used to create a two-way client/server connection, as the local endpoint implies a call to bindEndpointAsync as below.
- bindEndpointAsync For a one-way server connection—that is, to only listen to but not send data on the socket—this method just binds a local endpoint given a HostName and a service name/port. Binding the service name by itself can be done with bindServiceNameAsync.
- joinMulticastGroup Given a HostName, connects the Datagram socket to a multicast group.
- close Terminates the connection and aborts any pending operations.

Tip To open a socket to a localhost port for debugging purposes, use connectAsync as follows:

```
var socket = new Windows.Networking.Sockets.DatagramSocket();
socket.connectAsync(new Windows.Networking.Sockets.DatagramSocket("localhost",
    "12345", Windows.Networking.Sockets.SocketProtectionLevel.plainSocket)
    .done(function () {
        // ...
    }, onError);
```

Note that any given socket can be connected to any number of endpoints—you can call connectAsync multiple times, join multiple multicast groups, and bind multiple local endpoints with bindEndpointAsync and bindServiceNameAsync. The close method, mind you, closes everything at once!

Once the socket has one or more connections, connection information can be retrieved with the DatagramSocket.information property (a DatagramSocketInformation). Also, note that the static DatagramSocket.getEndpointPairsAsync method provides (as the async result) a vector of available EndpointPair objects for a given remote hostname and service name. You can optionally indicate that you'd like the endpoints sorted according to the optimizeForLongConnections flag. See the documentation page linked here for details, but it basically lets you control which endpoint is preferred over others based on whether you want to optimize for a high-quality and long-duration connection that might take longer to connect to initially (as for video streaming) or for connections that are easiest to acquire (the default).

Control data can also be set through the DatagramSocket.control property, a DatagramSocketControl object with qualityOfService and outputUnicastHopLimit properties.

All this work, of course, is just a preamble to sending data on the socket connection. This is done through the DatagramSocket.outputStream property, an IOutputStream to which you can write whatever data you need using its writeAsync and flushAsync methods. This will send the data on every connection within the socket. Alternately, you can use one of the variations of getOutputStreamAsync to specify a specific EndpointPair or HostName/port to which to send the data. The result of both of these async operations is again an IOutputStream. And in all cases you can create a higher-level DataWriter object (see Chapter 10) around that stream:

```
var dataWriter = new Windows.Storage.Streams.DataWriter(socket.outputStream)
```

Here's how it's all demonstrated in the DatagramSocket sample, a little app in which you need to run each of the scenarios in turn. Scenario 1, for starters, sets up the server-side listener of the relationship on the localhost, using port number 22112 (the service name) by default. To do this, it creates the sockets, adds the listener, and calls bindServiceNameAsync (js/startListener.js):

```
socketsSample.listener = new Windows.Networking.Sockets.DatagramSocket();
// Reminder: call removeEventListener as needed; this can be common with socket relationships
// that can come and go through the lifetime of the app.
socketsSample.listener.addEventListener("messagereceived", onServerMessageReceived);

socketsSample.listener.bindServiceNameAsync(serviceName).done(function () {
    // ...
}, onError);
```

When a message is received, this server-side component takes the contents of the message and writes it to the socket's output stream so that it's reflected in the client side. This looks a little confusing in the code, so I'll show the core path of this process with added comments:

```
function onServerMessageReceived(eventArgument) {
    // [Code here checks if we already got an output stream]
    socketsSample.listener.getOutputStreamAsync(eventArgument.remoteAddress,
        eventArgument.remotePort).done(function (outputStream) {
            // [Save the output stream with some other info, omitted]
            socketsSample.listenerOutputStream = outputStream;
        }

        // This is a helper function
        echoMessage(socketsSample.listenerOutputStream, eventArgument);
    });
}

// eventArgument here is a DatagramSocketMessageReceivedEventArgs with a getDataReader method
function echoMessage(outputStream, eventArgument) {
    // [Some display code omitted]

    // Get the message stream from the DataReader and send it to the output stream
    outputStream.writeAsync(eventArgument.getDataReader().detachBuffer()).done(function () {
        // Do nothing - client will print out a message when data is received.
    });
}
```

In most apps using sockets, the server side would do something more creative with the data than just send it back to the client! But this just changes what you do with the data in the input stream.

Scenario 2 sets up a listener to the localhost on the same port. On this side we also create a DatagramSocket and set up a listener for messagereceived. Those messages—such as the one written to the output stream on the server side, as we've just seen—are picked up in the event handler below (js/connectToListener.js), which uses the DataReader to extract and display the message:

```
function onMessageReceived(eventArgument) {
    try {
        var messageLength = eventArgument.getDataReader().unconsumedBufferLength;
        var message = eventArgument.getDataReader().readString(messageLength);
        socketsSample.displayStatus("Client: receive message from server \"" + message + "\"");
    } catch (exception) {
        status = Windows.Networking.Sockets.SocketError.getStatus(exception.number);
        // [Display error details]
    }
}
```

Note that when an error occurs on a socket connection, you can pass the error number to the getStatus method of the SocketError object and get back a more actionable SocketErrorStatus value. There are many possible errors here, so see its reference page for details.

Even with all the work we've done so far, nothing has yet happened because we've sent no data! So switching to scenario 3, pressing its Send 'Hello' Now button does the honors from the client side (js/sendData.js):

```
// [This comes after a check on the socket's validity]
socketsSample.clientDataWriter =
    new Windows.Storage.Streams.DataWriter(socketsSample.clientSocket.outputStream);
```

```
var string = "Hello World";
socketsSample.clientDataWriter.writeString(string);

socketsSample.clientDataWriter.storeAsync().done(function () {
    socketsSample.displayStatus("Client sent: " + string + ".");
}, onError);
```

The `DataWriter.storeAsync` call is what actually writes the data to the stream in the socket. If you set a breakpoint here and on both `messagereceived` event handlers, you'll then see that `storeAsync` generates a message to the server side, hitting `onServerMessageReceived` in js/startListener.js. This will then write the message back to the socket, which will hit `onMessageReceived` in js/connectToListener.js, which displays the message. (And to complete the process, scenario 4 gives you a button to call the socket's `close` method.)

The sample does everything with the same app on localhost to make it easier to see how the process works. Typically, the server will be running on another machine entirely, but the steps of setting up a listener apply just the same. As noted in Chapter 4, localhost connections work only on a machine with a developer license and will not work for apps acquired through the Windows Store.

Stream Sockets

In contrast to datagram sockets, streaming data over sockets uses the Transmission Control Protocol (TCP). The hallmark of TCP is accurate and reliable delivery—it guarantees that the bytes received are the same as the bytes that were sent: when a packet is sent across the network, TCP will attempt to retransmit the packet if there are problems along the way. This is why it's part of TCP/IP, which gives us the World Wide Web, email, file transfers, and lots more. HTTP, SMTP, and the Session Initiation Protocol (SIP) are also built on TCP. In all cases, clients and servers just see a nice reliable stream of data flowing from one end to the other.

Unlike datagram sockets, for which we have a single class in WinRT for both sides of the relationship, stream sockets are more distinctive to match the unique needs of the client and server roles. On the client side is `Windows.Networking.Sockets.StreamSocket`; on the server it's `StreamSocketListener`.

Starting with the latter, the `StreamSocketListener` object looks quite similar to the `DatagramSocket` we've just covered, with these methods, properties, and events:

- `information` Provides a `StreamSocketListenerInformation` object containing a `localPort` string.
- `control` Provides a `StreamSocketListenerControl` object with a `qualityOfService` property.
- `connectionreceived` A WinRT event that's fired when a connection is made to the listener. Its event arguments are a `StreamSocketListenerConnectionReceivedEventArgs` that contains a single property, `socket`. This is the `StreamSocket` for the client, in which is an `outputStream` property where the listener can obtain the data stream.

- bindEndpointAsync and bindServiceNameAsync   Binds the listener to a HostName and service name, or binds just a service name.

- close   Terminates connections and aborts pending operations.

On the client side, StreamSocket again looks like parts of the DatagramSocket. In addition to the control (StreamSocketControl) and information properties (StreamSocketInformation) and the ubiquitous close method, we find a few other usual suspects and one unusual one:

- connectAsync   Connects to a HostName/service name or to an EndpointPair. In each case you can also provide an optional SocketProtectionLevel object that can be plainSocket, ssl, or sslAllowNullEncryption. There are, in other words, four variations of this method.

- inputStream   The IInputStream that's being received over the connection.

- outputStream   The IOutputStream into which data is written.

- upgradeToSslAsync   Upgrades a plainSocket connection (created through connectAsync) to use SSL as specified by either SocketProtectionLevel.ssl or sslAllowNullEncryption. This method also required a HostName that validates the connection.

For more details on using SSL, see How to secure socket connections with TLS/SSL.

In any case, you can see that for one-way communications over TCP, an app creates either a StreamSocket or a StreamSocketListener, depending on its role. For two-way communications an app will create both.

The StreamSocket sample, like the DatagramSocket sample, has four scenarios that are meant to be run in sequence on the localhost: first to create a listener (to receive a message from a client, scenario 1), then to create the StreamSocket (scenario 2) and send a message (scenario 3), and then to close the socket (scenario 4). With streamed data, the app implements a custom protocol for how the data should appear, as we'll see. Also note that scenario 5 demonstrates evaluating certificate validity and displaying certificate properties, which you'd need to do when making connections over SSL.

Starting in scenario 1 (js/startListener.js), here's how we create the listener and event handler. Processing the incoming stream data is trickier than with a datagram because we need to make sure the data we need is all there. This code shows a good pattern of waiting for one async operation to finish before the function calls itself recursively. Also note how it creates a DataReader on the input stream for convenience:

```
socketsSample.listener = new Windows.Networking.Sockets.StreamSocketListener(serviceName);
// Match with removeEventListener as needed
socketsSample.listener.addEventListener("connectionreceived", onServerAccept);

socketsSample.listener.bindServiceNameAsync(serviceName).done(function () {
   // ...
   }, onError);
}
```

```
// This has to be a real function; it will "loop" back on itself with the call to
// acceptAsync at the very end.
function onServerAccept(eventArgument) {
    socketsSample.serverSocket = eventArgument.socket;
    socketsSample.serverReader =
         new Windows.Storage.Streams.DataReader(socketsSample.serverSocket.inputStream);
    startServerRead();
}

// The protocol here is simple: a four-byte 'network byte order' (big-endian) integer that
// says how long a string is, and then a string that is that long. We wait for exactly 4 bytes,
// read in the count value, and then wait for count bytes, and then display them.
function startServerRead() {
    socketsSample.serverReader.loadAsync(4).done(function (sizeBytesRead) {
        // Make sure 4 bytes were read.
        if (sizeBytesRead !== 4) { /* [Show message] */ }

        // Read in the 4 bytes count and then read in that many bytes.
        var count = socketsSample.serverReader.readInt32();
        return socketsSample.serverReader.loadAsync(count).then(function (stringBytesRead) {
            // Make sure the whole string was read.
            if (stringBytesRead !== count) { /* [Show message] */ }

            // Read in the string.
            var string = socketsSample.serverReader.readString(count);
            socketsSample.displayOutput("Server read: " + string);

            // Restart the read for more bytes. We could just call startServerRead() but in
            // the case subsequent read operations complete synchronously we start building
            // up the stack and potentially crash. We use WinJS.Promise.timeout() to invoke
            // this function after the stack for current call unwinds.
            WinJS.Promise.timeout().done(function () { return startServerRead(); });
        }); // End of "read in rest of string" function.
    }, onError);
}
```

This code is structured to wait for incoming data that isn't ready yet, but you might have situations in which you want to know if there's more data available that you haven't read. This value can be obtained through the `DataReader.unconsumedBufferLength` property.

In scenario 2, the data-sending side of the relationship is simple: create a `StreamSocket` and call `connectAsync` (js/connectToListener.js; note that `onError` uses `StreamSocketError.getStatus` again):

```
socketsSample.clientSocket = new Windows.Networking.Sockets.StreamSocket();
socketsSample.clientSocket.connectAsync(hostName, serviceName).done(function () {
    // ...
}, onError);
```

Sending data in scenario 3 takes advantage of a `DataWriter` built on the socket's output stream (js/sendData.js):

```
var writer = new Windows.Storage.Streams.DataWriter(socketsSample.clientSocket.outputStream);
var string = "Hello World";
var len = writer.measureString(string); // Gets the UTF-8 string length.
writer.writeInt32(len);
writer.writeString(string);

writer.storeAsync().done(function () {
    writer.detachStream();
}, onError);
```

And closing the socket in scenario 4 is again just a call to `StreamSocket.close`.

As with the DatagramSocket sample, setting breakpoints within `openClient` (js/connectToListener.js), `onServerAccept` (js/startListener.js), and `sendHello` (js/sendData.js) will let you see what's happening at each step of the process.

Web Sockets: MessageWebSocket and StreamWebSocket

Having now seen both Datagram and Stream sockets in action, we can look at their equivalents on the WebSocket side. As you might already know, WebSockets is a standard created to use HTTP (and thus TCP) to set up an *initial* connection after which the data exchange happens through sockets over TCP. This provides the simplicity of using HTTP requests for the first stages of communication and the efficiency of sockets afterwards.

As with regular sockets, the WebSocket side of WinRT supports both water balloons and water hoses: the `MessageWebSocket` class provides for discrete packets as with datagram sockets (though it uses TCP and not UDP), and `StreamWebSocket` clearly provides for stream sockets. Both classes are similar to their respective `DatagramSocket` and `StreamSocket` counterparts, so much so that their interfaces are very much the same (with distinct secondary types like `MessageWebSocketControl`):

- Like `DatagramSocket`, `MessageWebSocket` has `control`, `information`, and `outputStream` properties, a `messagereceived` event, and methods of `connectAsync` and `close`. It adds a `closed` event along with a `setRequestHeader` method.
- Like `StreamSocket`, `StreamWebSocket` has `control`, `information`, `inputStream`, and `outputStream` properties, and methods of `connectAsync` and `close`. It adds a `closed` event and a `setRequestHeader` method.

Notice that there isn't an equivalent to `StreamSocketListener` here. This is because the process of establishing that connection is handled through HTTP requests, so such a distinct listener class isn't necessary. This is also why we have `setRequestHeader` methods on the classes above: so that you can configure those HTTP requests. Along these same lines, you'll find that the `connectAsync` methods take a `Windows.Foundation.Uri` rather than hostnames and service names. But otherwise we see the same kind of activity going on once the connection is established, with streams, `DataReader`, and `DataWriter`.

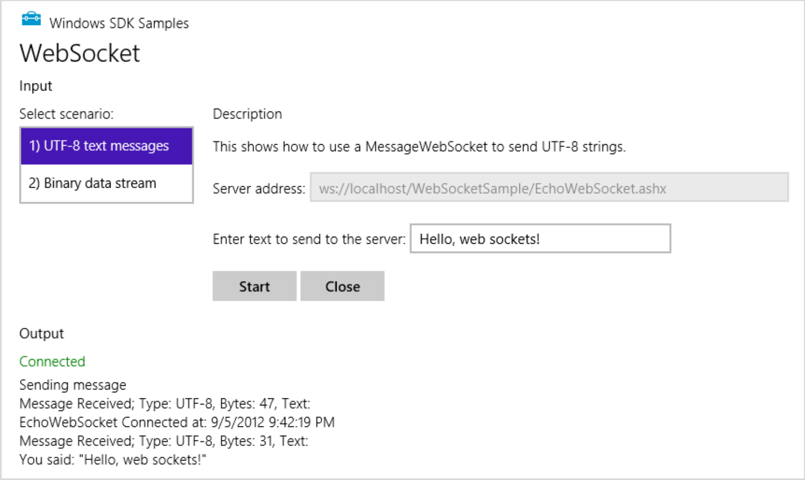
Sidebar: Comparing W3C and WinRT APIs for WebSockets

Standard WebSockets, as they're defined in the W3C API, are entirely supported for Windows Store apps. However, they support only a transaction-based UDP model like `DatagramSocket` and only text content. The `MessageWebSocket` in WinRT, however, supports both text and binary, plus you can use the `StreamWebSocket` for a streaming TCP model as well. The WinRT APIs also emit more detailed error information and so are generally preferred over the W3C API.

Let's look more closely at these in the context of the Connecting with WebSockets sample. This sample is dependent upon an ASP.NET server page running in the localhost, so you must first go into its Server folder and run **powershell.exe -ExecutionPolicy unrestricted -file setupserver.ps1** from an Administrator command prompt. (For more on setting up Internet Information Services and the localhost, refer to the "Using the localhost" sidebar in the "Using Windows.Web.Http.HttpClient" section in Chapter 4.) If the script succeeds, you'll see a WebSocketSample folder in *c:\inetpub\wwwroot* that contains an EchoWebService.ashx file. Also, as suggested in Chapters 4 and 16, you can run the Web platform installer to install Visual Studio Express for Web that will allow you to run the server page in a debugger. Always a handy capability!

Within EchoWebService.ashx you'll find an `EchoWebSocket` class written in C#. It basically has one method, `ProcessRequest`, that handles the initial HTTP request from the web socket client. With this request it acquires the socket, writes an announcement message to the socket's stream when the socket is opened, and then waits to receive other messages. If it receives a text message, it echoes that text back through the socket with "You said" prepended. If it receives a binary message, it echoes back a message indicating the amount of data received.

Going to scenario 1 of the Connecting with WebSockets sample, we can send a message to that server page by using `MessageWebSocket` and get back a message of our own as shown below. In this case the output in the sample reflects information known to the app and nothing from the service itself.



Description

This shows how to use a MessageWebSocket to send UTF-8 strings.

Server address: ws://localhost/WebSocketSample/EchoWebSocket.ashx

Enter text to send to the server: Hello, web sockets!

Output

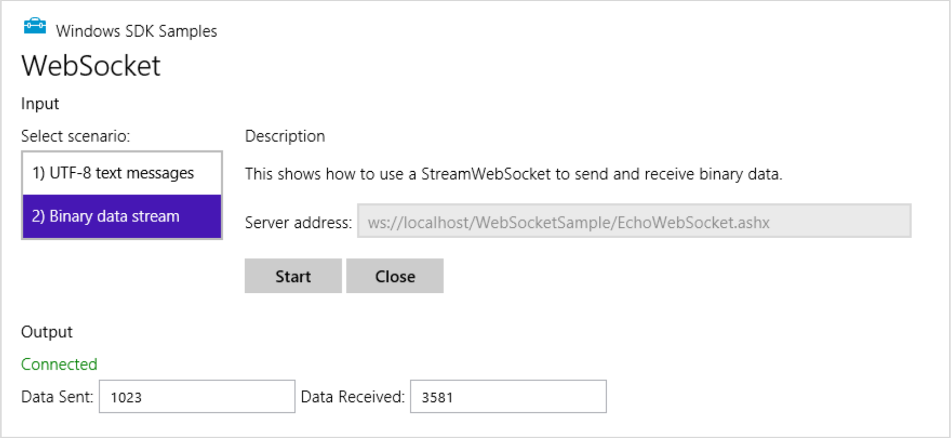
```
Connected
Sending message
Message Received; Type: UTF-8, Bytes: 47, Text:
EchoWebSocket Connected at: 9/5/2012 9:42:19 PM
Message Received; Type: UTF-8, Bytes: 31, Text:
You said: "Hello, web sockets!"
```

In the sample, we first create a MessageWebSocket, call its connectAsync, and then use a DataWriter to write some data to the socket. It also listens for the messagereceived event to output the result of the send, and it listens to the closed event from the server so that it can do the close from its end. The code here is simplified from js/scenario1.js:

```
var messageWebSocket;
var messageWriter;

var webSocket = new Windows.Networking.Sockets.MessageWebSocket();
webSocket.control.messageType = Windows.Networking.Sockets.SocketMessageType.utf8;
webSocket.onmessagereceived = onMessageReceived;
webSocket.onclosed = onClosed;

// The server URI is obtained and validated here, and stored in a variable named uri.

webSocket.connectAsync(uri).done(function () {
    messageWebSocket = webSocket;
    // The default DataWriter encoding is utf8.
    messageWriter = new Windows.Storage.Streams.DataWriter(webSocket.outputStream);
    sendMessage();    // Helper function, see below
}, function (error) {
    var errorStatus = Windows.Networking.Sockets.WebSocketError.getStatus(error.number);
    // [Output error message]
});

function onMessageReceived(args) {
    var dataReader = args.getDataReader();
    // [Output message contents]
}

function sendMessage() {
    // Write message in the input field to the socket
    messageWriter.writeString(document.getElementById("inputField").value);
    messageWriter.storeAsync().done("", sendError);
}

function onClosed(args) {
    // Close our socket if the server closes [simplified from actual sample; it also closes
    // the DataWriter it might have opened.]
    messageWebSocket.close();
}
```

Similar to what we saw in previous sections, when an error occurs you can turn the error number into a `SocketErrorStatus` value. In the case of WebSockets you do this with the `getStatus` method of `Windows.Networking.Sockets.WebSocketError`. Again, see its reference page for details.

Scenario 2, for its part, uses a `StreamWebSocket` to send a continuous stream of data packets, a process that will continue until you close the connection:



This shows how to use a StreamWebSocket to send and receive binary data.

Server address: ws://localhost/WebSocketSample/EchoWebSocket.ashx

Connected

Data Sent: 1023 Data Received: 3581

Here's the process in code, simplified from js/scenario2.js, where we see a similar pattern to what we just saw for `MessageWebSocket`, only sending a continuous stream of data:

```
var streamWebSocket;
var dataWriter;
var dataReader;
var data = "Hello World";
var countOfDataSent;
var countOfDataReceived;

var webSocket = new Windows.Networking.Sockets.StreamWebSocket();
webSocket.onclosed = onClosed;

// The server URI is obtained and validated here, and stored in a variable named uri.

webSocket.connectAsync(uri).done(function () {
    streamWebSocket = webSocket;
    dataWriter = new Windows.Storage.Streams.DataWriter(webSocket.outputStream);
    dataReader = new Windows.Storage.Streams.DataReader(webSocket.inputStream);
    // When buffering, return as soon as any data is available.
    dataReader.inputStreamOptions = Windows.Storage.Streams.InputStreamOptions.partial;
    countOfDataSent = 0;
    countOfDataReceived = 0;

    // Continuously send data to the server
    writeOutgoing();

    // Continuously listen for a response
    readIncoming();
}, function (error) {
    var errorStatus = Windows.Networking.Sockets.WebSocketError.getStatus(error.number);
    // [Output error message]
});

function writeOutgoing() {
    try {
        var size = dataWriter.measureString(data);
        countOfDataSent += size;
        }
        dataWriter.writeString(data);
        dataWriter.storeAsync().done(function () {
            // Add a 1 second delay so the user can see what's going on.
            setTimeout(writeOutgoing, 1000);
        }, writeError);
    }
    catch (error) {
        // [Output error message]
    }
}

function readIncoming(args) {
    // Buffer as much data as you require for your protocol.
    dataReader.loadAsync(100).done(function (sizeBytesRead) {
        countOfDataReceived += sizeBytesRead;
        // [Output count]

        var incomingBytes = new Array(sizeBytesRead);
        dataReader.readBytes(incomingBytes);

        // Do something with the data. Alternatively you can use DataReader to
        // read out individual booleans, ints, strings, etc.

        // Start another read.
        readIncoming();
    }, readError);
}

function onClosed(args) {
    // [Other code omitted, including closure of DataReader and DataWriter]
    streamWebSocket.close();
}
```

As with regular sockets, you can exercise additional controls with WebSockets, including setting credentials and indicating supported protocols through the control property of both `MessageWebSocket` and `StreamWebSocket`. For details, see How to use advanced WebSocket controls in the documentation. Similarly, you can set up a secure/encrypted connection by using the `wss://` URI scheme instead of `ws://` as used in the sample. For more, see How to secure WebSocket connections with TLS/SSL.

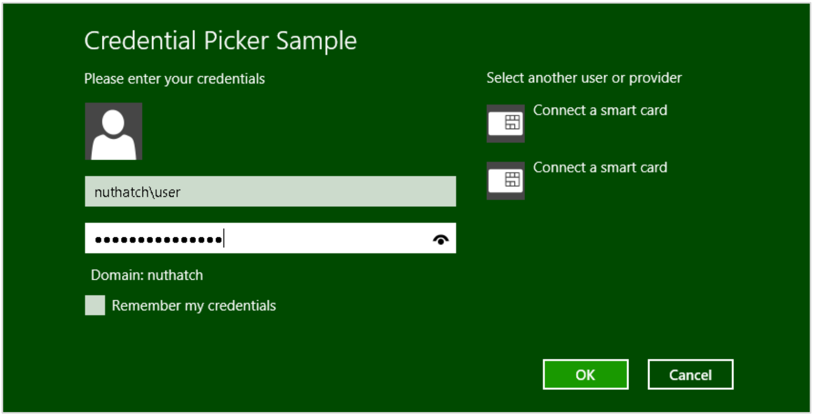
The ControlChannelTrigger Background Task

In Chapter 16 in the "Lock Screen Dependent Tasks and Triggers" section, we took a brief look at the `Windows.Networking.Sockets.ControlChannelTrigger` class that can be used to set up a background task for real-time notifications as would be used by VoIP, IM, email, and other "always reachable" scenarios. To repeat, working with the control channel is not something that can be done from JavaScript, so refer to How to set background connectivity options in the documentation along with the following C#/C++ samples:

- ControlChannelTrigger StreamSocket sample
- ControlChannelTrigger XmlHttpRequest sample
- ControlChannelTrigger StreamWebSocket sample
- ControlChannelTrigger HTTP client sample

The Credential Picker UI

For enterprise scenarios where the Web Authentication Broker won't suffice for authentication, WinRT provides a built-in, enterprise-ready UI for entering credentials: `Windows.Security.Credentials.UI.CredentialsPicker`. When you instantiate this object and call its `pickAsync` method, as does the Credential Picker sample, you'll see the UI shown below. This UI provides for domain logins, supports, and smart cards (I have two smart card readers on my machine, as you can see), and it allows for various options such as authentication protocols and automatic saving of the credential.


Credential Picker Sample

Please enter your credentials

nuthatch\user

•••••••••••••••

Domain: nuthatch

□ Remember my credentials

Select another user or provider

Connect a smart card

Connect a smart card

The result from `pickAsync`, as given to your completed handler, is a `CredentialPickerResults` object with the following properties (when you enter some credentials in the sample, you'll see these values reflected in the sample's output):

- `credentialuserName` A string containing the entered username.
- `credentialPassword` A string containing the password (typically encrypted depending on the authentication protocol option).
- `credentialDomainName` A string containing a domain if entered with the username (as in <domain>\<username>).
- `credentialSaved` A Boolean indicating whether the credential was saved automatically; this depends on picker options, as discussed below.
- `credentialSavedOption` A `CredentialSavedOption` value indicating the state of the Remember My Credentials check box: `unselected`, `selected`, or `hidden`.
- `errorCode` Contains zero if there is no error, otherwise an error code.
- `credential` An `IBuffer` containing the credential as an opaque byte array. This is what you can save in your own persistent state if need be and pass back to the picker at a later time. We'll see how at the end of this section.

The three scenarios in the sample demonstrate the different options you can use to invoke the credential picker. For this there are three separate variants of `pickAsync`. The first variant accepts a target name (which is ignored) and a message string that appears in the place of "Please enter your credentials" shown in the previous screenshot:

```
Windows.Security.Credentials.UI.CredentialPicker.pickAsync(targetName, message)
    .done(function (results) {
    }
```

The second variant accepts the same arguments plus a caption string that appears in the place of "Credential Picker Sample" in the screenshot:

```
Windows.Security.Credentials.UI.CredentialPicker.pickAsync(targetName, message, caption)
    .done(function (results) {
    }
```

The third variant accepts a `CredentialPickerOptions` object that has properties for the same `targetName`, `message`, and `caption` strings, along with the following:

- `previousCredential` An `IBuffer` with the opaque credential information as provided by a previous invocation of the picker (see `CredentialPickerResults.credential` above).
- `alwaysDisplayDialog` A Boolean indicating whether the picker is displayed. The default is `false`, but this applies only if you also populate `previousCredential` (with an exception for domain-joined machines—see table below). The purpose here is to show the dialog when a stored credential might be incorrect and the user is expected to provide a new one.

- errorCode The numerical value of a Win32 error code (default is ERROR_SUCCESS) that will be formatted and displayed in the dialog box. You would use this when you obtain credentials from the picker initially but find that those credentials don't work and need to invoke the picker again. Instead of providing your own message, you just choose an error code and let the system do the rest. The most common values for this are 1326 (login failure), 1330 (password expired), 2202 (bad username), 1907 or 1938 (password must change/password change required), 1351 (can't access domain info), and 1355 (no such domain). There are, in fact, over 15,000 Win32 error codes, but that means you'll have to search the reference linked above (or search within the winerror.h file typically found in your *Program Files (x86)\Windows Kits\8.0\Include\shared* folder). Happy hunting!
- callerSavesCredential A Boolean indicating that the app will save the credential and that the picker should not. The default value is false. When set to true, credentials are saved to a secure system location (not the credential locker) if the app has the *Enterprise Authentication* capability (see below).
- credentialSaveOption A CredentialSaveOption value indicating the initial state of the Remember My Credentials check box: unselected, selected, or hidden.
- authenticationProtocol A value from the AuthenticationProtocol enumeration: basic, digest, ntlm, kerberos, negotiate (the default), credSsp, and custom (in which case you must supply a string in the customAuthenticationProcotol property). Note that with basic and digest, the CredentialPickerResults.credentialPassword will *not* be encrypted and is subject to the same security needs as a plain text password you collect from your own UI.

Here's an example of invoking the picker with an errorCode indicating a previous failed login:

```
var options = new Windows.Security.Credentials.UI.CredentialPickerOptions();
options.message = "Please enter your credentials";
options.caption = "Sample App";
options.targetName = "Target";
options.alwaysDisplayDialog = true;
options.errorCode = 1326;  // Shows "The username or password is incorrect."
options.callerSavesCredential = true;
options.authenticationProtocol =
    Windows.Security.Credentials.UI.AuthenticationProtocol.negotiate;
options.credentialSaveOption = Windows.Security.Credentials.UI.CredentialSaveOption.selected;

Windows.Security.Credentials.UI.CredentialPicker.pickAsync(options)
    .done(function (results) {
    }
```

To clarify the relationship between the callerSavesCredential, credentialSaveOption, and the credentialSaved properties, the following table lists the possibilities:

Enterprise Auth capability	callerSavesCredential	credentialSaveOption	Credential Picker saves credentials	Apps saves credentials to credential locker
No	true	Selected	No	**Yes**
		unselected or hidden	No	No
	false	Selected	No	**Yes**
		unselected or hidden	No	No
Yes	true	Selected	No	**Yes**
		unselected or hidden	No	No
	false	Selected	**Yes** (credentialSaved will be true)	**Optional**
		unselected or hidden	No	No

The first column refers to the *Enterprise Authentication* capability in the app's manifest, which indicates that the app can work with Intranet resources that require domain credentials (and assumes that the app is also running on the Enterprise Edition of Windows). In such cases the credential picker has a separate secure location (apart from the credential locker) in which to store credentials, so the app need not save them itself. Furthermore, if the picker saves a credential and the app invokes the picker with alwaysDisplayDialog set to false, previousCredential can be empty because the credential will be loaded automatically. But without a domain-joined machine and this capability, the app must supply a previousCredential to avoid having the picker appear.

This brings us to the question about how, exactly, to persist a CredentialPickerResults.-credential and load it back into CredentialPickerOptions.previousCredential at another time. The credential is an IBuffer, and if you look at the IBuffer documentation you'll see that it doesn't in itself offer any useful methods for this purpose (in fact, you'll really wonder just what the heck it's good for!). Fortunately, other APIs understand buffers. To save a buffer's content, pass it to the writeBufferAsync method in either Windows.Storage.FileIO or Windows.Storage.PathIO. To load it later, use the readBufferAsync methods of the FileIO and PathIO objects.

This is demonstrated in the modified Credential Picker sample in the appendices' companion content. In js/scenario3.js we save credential within the completed handler for CredentiaPicker.-pickAsync:

```
//results.credential will be null if the user cancels
if (results.credential != null) {
    //Having retrieved a credential, write the opaque buffer to a file
    var option = Windows.Storage.CreationCollisionOption.replaceExisting;

    Windows.Storage.ApplicationData.current.localFolder.createFileAsync("credbuffer.dat",
        option).then(function (file) {
        return Windows.Storage.FileIO.writeBufferAsync(file, results.credential);
    }).done(function () {
        //No results for this operation
        console.log("credbuffer.dat written.");
    }, function (e) {
        console.log("Could not create credbuffer.dat file.");
    });
}
```

I'm using the local appdata folder here; you could also use the roaming folder if you want the credential to roam (securely) to other devices as if it were saved in the Credential Locker.

To reload, we modify the `launchCredPicker` function to accept a buffer and use that for `previousCredential` if given:

```
function launchCredPicker(prevCredBuffer) {
    try {
        var options = new Windows.Security.Credentials.UI.CredentialPickerOptions();

        //Set the previous credential if provided
        if (prevCredBuffer != null) {
            options.previousCredential = prevCredBuffer;
        }
```

We then point the `click` handler for *button1* to a new function that looks for and loads the credbuffer.dat file and calls `launchCredPicker` accordingly:

```
function readPrevCredentialAndLaunch() {
    Windows.Storage.ApplicationData.current.localFolder.getFileAsync("credbuffer.dat")
        .then(function (file) {
        return Windows.Storage.FileIO.readBufferAsync(file);
    }).done(function (buffer) {
        console.log("Read from credbuffer.dat");
        launchCredPicker(buffer);
    }, function (e) {
        console.log("Could not reopen credbuffer.dat; launching default");
        launchCredPicker(null);
    });
}
```

Other Networking SDK Samples

Sample	Description (from the Windows Developer Center)
Connectivity Manager Sample	Demonstrates how an app can activate and use a secondary packet device protocol (PDP) context on a mobile broadband device, employing the `Windows.Networking.Connectivity.ConnectivityManager` class.
HomeGroup app sample	Demonstrates how to use a HomeGroup to open, search, and share files. This sample uses some of the HomeGroup options. In particular, it uses `Windows.Storage.Pickers.PickerLocationId` enumeration and the `Windows.Storage.KnownFolders.homeGroup` property to select files contained in a HomeGroup.
Remote desktop app container client sample	Demonstrates how to use the Remote Desktop app container client objects in an app.
RemoteApp and desktop connections workspace API sample	Demonstrates how to use the `WorkspaceBrokerAx` object in a Windows Store app.

SMS message send, receive, and SIM management sample	Demonstrates how to use the Mobile Broadband SMS API (`Windows.Devices.Sms`). This API can be used only from mobile broadband device apps and is not available to apps generally.
SMS background task sample	Demonstrates how to use the Mobile Broadband SMS API (`Windows.Devices.Sms`) with the Background Task API (`Windows.ApplicationModel.Background`) to send and receive SMS text messages. This API can be used only from mobile broadband device apps and is not available to apps generally.
USSD message management sample	Demonstrates network account management using the USSD protocol with GSM-capable mobile broadband devices. USSD is typically used for account management of a mobile broadband profile by the Mobile Network Operator (MNO). USSD messages are specific to the MNO and must be chosen accordingly when used on a live network. (That sample is applicable only to those building mobile broadband device apps; it draws on the API in `Windows.Networking.NetworkOperators`.)

Appendix D

Provider-Side Contracts

In this appendix:

- File picker providers
- The cached file updater
- Contact card action providers
- Contact picker providers
- Appointment providers

File Picker Providers

In Chapter 11, "The Story of State, Part 2," we looked at how the file/folder picker can be used to reference not only locations on the file system but also content that's managed by other apps or even created on-the-fly within other apps. Let's be clear on this point: the app that's *using* the file picker is doing so to obtain a `StorageFile` or `StorageFolder` for some purpose. But this does not mean that *provider* apps that can be invoked through the file picker necessary manage their data *as* files or folders. Their role is to take whatever kind of data they manage and package it up so that it *looks* like a file/folder to the picker.

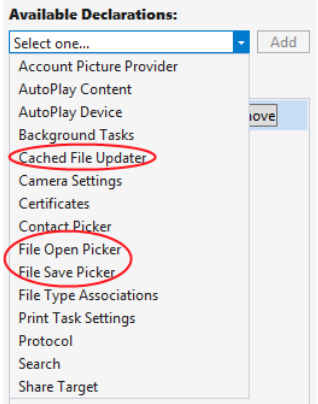
In the "The File Picker UI" section of Chapter 11, for instance, we saw how the Windows Sound Recorder app can be used to record and audio track and return it through the file picker. Such a recording did not exist at the time the target app was invoked; instead, it displayed its UI through which the user could create a file that was then passed back through the file picker. In this way, the Sound Recorder app shortcuts the whole process of creating a new recording: it provides that function exactly when the user is trying to select an audio file. Otherwise the user would have to start the Sound Recorder app separately, make a recording, store it locally, and switch to the original app to invoke the file picker, and locate that new file.

The file picker is not limited to audio or other media, of course: it works with any file type, depending on what the caller indicates it wants. One app might let the user go into an online music library, purchase and download a track, and then return that file to the file picker. Another app might perform some kind of database query and return the results as a file, and still others might allow the user to browse online databases of file-type entities, again hiding the details of downloading and packaging that data as a file such that the user's experience of the file picker is seamless across the local file system, online resources, and apps that just create or acquire data dynamically. In such cases, however, note that the file picker contracts are designed for relatively quick in-and-out experiences. For this reason an app should provide only basic editing capabilities in this context (like cropping a photo or trimming the audio).

As with the Search and Share target contracts, Visual Studio and Blend provide an item template for file picker providers, specifically the File Open Picker Contract item in a project's Add > New Item dialog. This gives you a basic selection structure built around a ListView control, but not much else. For our purposes we won't be using this template; we'll draw on samples instead. Generally speaking, when servicing the file picker contracts, an app should use the same views and UI as it does when launched normally, thereby keeping the app experience consistent in both scenarios.

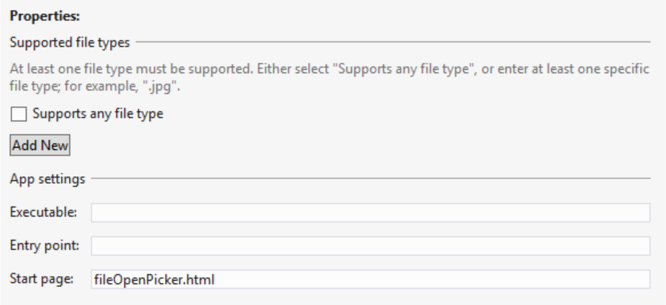
Manifest Declarations

To be a provider for the file picker, an app starts by—what else!—adding the appropriate declaration to its manifest. In this case there are actually three declarations: File Open Picker, File Save Picker, and Cached File Updater, as shown below in Visual Studio's manifest designer. Each of these declarations can be made once within any given app.



The File Open Picker and File Save Picker declarations are what make a provider app available in the dialogs invoked through the `Windows.Storage.Pickers.FileOpenPicker` and `FileSavePicker` API. The calling app in both cases is completely unaware that another app might be invoked—all the interaction is between the picker and the provider app through the contract, with the contract broker being responsible first for displaying a UI through which to select an object and second for returning a `StorageFile` object for that item.

With both the File Open Picker and File Save Picker contracts, the provider app indicates in its manifest those file types that it can service. This is done through the Add New button in the image below; the file picker will then make that app available as a choice only when the calling app indicates a matching file type. The Supports Any File Type option that you see here will make the app always appear in the list, but this is appropriate only for apps like OneDrive that provide a general storage location. Apps that work only with specific file types should indicate only those types.



Properties:

Supported file types

At least one file type must be supported. Either select "Supports any file type", or enter at least one specific file type; for example, ".jpg".

□ Supports any file type

App settings

Executable:

Entry point:

Start page: fileOpenPicker.html

The provider app indicates a Start Page for the open and save provider contracts separately—the operations are distinct and independent. In both cases, as we've seen for other contracts, these are the pages that the file picker will load when the user selects this particular provider app. As with Share targets, these pages are typically independent of the main app and will have their own script contexts and activation handlers, as we'll see in the next section. (Again, the Executable and Entry Point options are there for other languages.)

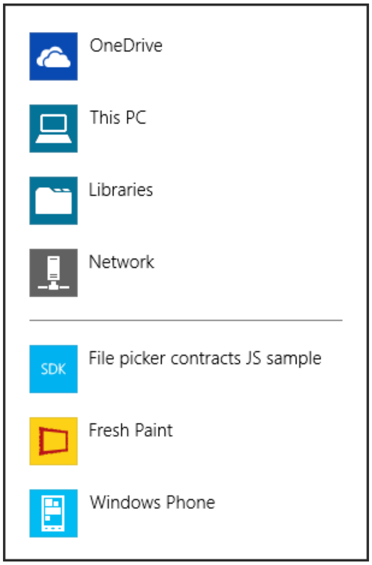
You might be asking: why are the open and save contracts separate? Won't most apps generally provide both? Well, not necessarily. If you're creating a provider app for a web service that is effectively read-only (like the image results from a search engine), you can serve only the file open case. If the service supports the creation of new files and updating existing files, such as a photo or document management service would, then you can also serve the file save case. There might also be scenarios where the provider would serve only the save case, such as writing to a sharing service. In short, Windows cannot presume the nature of the data sources that provider apps will work with, so the two contracts are kept separate.

The next main section in this Appendix covers the Cached File Updater contract, but it's good to know how it relates to the others here. This contract allows a provider app to synchronize local and remote copies of a file, essentially to subscribe to and manage change/access notifications for provided files. This is primarily of use to apps that represent a file repository where the user will frequently open and save files, like OneDrive or a database app. It's essentially a two-way binding service for files when either local or remote copies can be updated independently. As such, it's always implemented in conjunction with the file picker provider contracts.

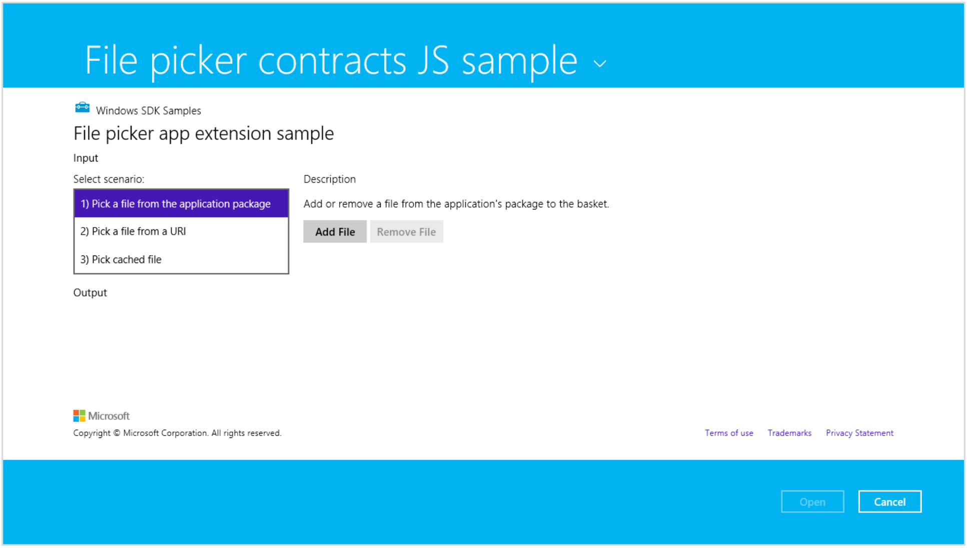
> **Tip** The Sharing and exchanging data topic has some helpful guidance about when you might choose to be a provider for the file save picker contract and when being a share target is more appropriate.

Activation of a File Picker Provider

Demonstrations of the file picker provider contracts—for open and save—are found in the File picker contracts sample, which I'll refer to as the *provider sample* for clarity. Declarations for both are included in the manifest with Supports Any File Type, so the sample will be listed with other apps in all file pickers:



When the provider app is invoked, the Start page listed in the manifest for the appropriate contract (open or save) is loaded. These are fileOpenPicker.html and fileSavePicker.html, found in the root of the project. Both of these pages are again loaded independently of the main app and appear as shown in Figure D-1 and Figure D-2. Note that the title of the app and the color scheme is determined by the Visual Assets section in the provider app's manifest. In particular, the text comes from the Tile > Short Name field and the colors come from the Tile > Foreground Text and Tile > Background Color settings. Note that the system automatically adds the down chevron (⌄) next to the title (at the top of each figure) through which the user can select a different picker location or provider app.


File picker contracts JS sample ⌄

Windows SDK Samples

File picker app extension sample

Input

Select scenario:

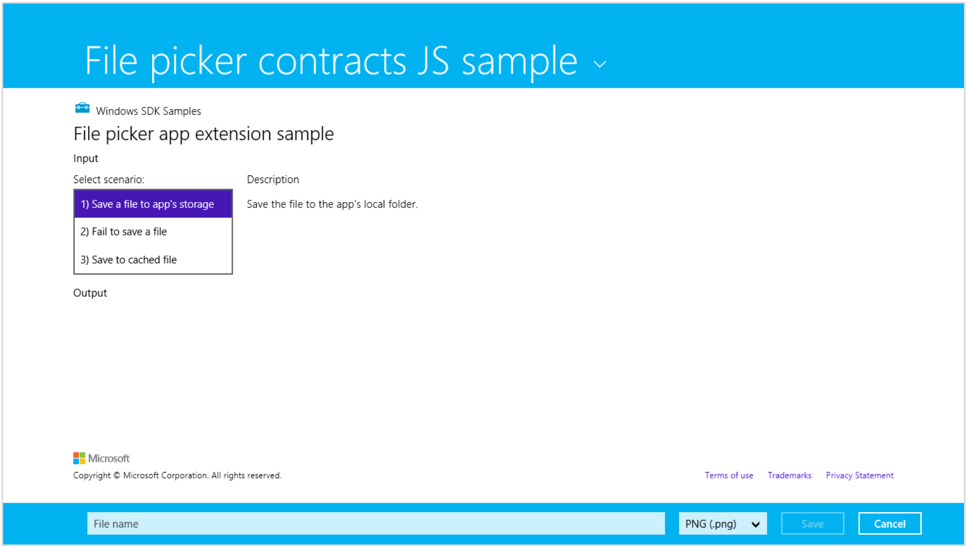
1) Pick a file from the application package
2) Pick a file from a URI
3) Pick cached file

Description

Add or remove a file from the application's package to the basket.

Copyright © Microsoft Corporation. All rights reserved.

FIGURE D-1 The Open UI as displayed by the sample.



File picker contracts JS sample

Windows SDK Samples

File picker app extension sample

Input

Select scenario:

1) Save a file to app's storage

2) Fail to save a file

3) Save to cached file

Description

Save the file to the app's local folder.

Output

Microsoft

Copyright © Microsoft Corporation. All rights reserved.

FIGURE D-2 The Save UI as displayed by the sample.

When you first run this sample, you won't see either of these pages. Instead you'll see a page through which you can invoke the file open or save pickers and then choose this app as a provider. You can do this if you like, but I recommend using a different app to invoke the pickers, just so we're clear on which app is playing which role. For this purpose you can use the sample we used in Chapter 11, the File picker sample (this is the consumer side). You can even use something like the Windows Music app where the Open File command on its app bar will invoke a picker wherein the provider sample will be listed.

Whatever your choice, the important parts of the provider sample are its separate pages for servicing its contracts, which are again fileOpenPicker.html and fileSavePicker.html. In the first case, the code is contained in js/fileOpenPicker.js where we can see the `activated` event handler with the activation kind of `fileOpenPicker`:

```
function activated(eventObject) {
    if (eventObject.detail.kind ===
        Windows.ApplicationModel.Activation.ActivationKind.fileOpenPicker) {
        fileOpenPickerUI = eventObject.detail.fileOpenPickerUI;

        eventObject.setPromise(WinJS.UI.processAll().then(function () {
            // Navigate to a scenario page...
        }));
    }
}
```

Here `eventObject.detail` is a `WebUIFileOpenPickerActivatedEventArgs` object, whose `fileOpenPickerUI` property (a `FileOpenPickerUI` object) provides the means to fulfill the provider's responsibilities with the contract.

In the second case, the code is in js/fileSavePicker.js where the activation kind is fileSavePicker:

```
function activated(eventObject) {
    if (eventObject.detail.kind ===
        Windows.ApplicationModel.Activation.ActivationKind.fileSavePicker) {
        fileSavePickerUI = eventObject.detail.fileSavePickerUI;

        eventObject.setPromise(WinJS.UI.processAll().then(function () {
            // Navigate to a scenario page
        }));
    }
}
```

where eventObject.detail is a WebUIFileSavePickerActivatedEventArgs object. As with the open contract, the fileSavePickerUI property of this (a FileSavePickerUI object) provides the means to fulfill the provider's side of the contract.

In both open and save cases, the contents of the contract's Start page is displayed within the letterboxed area between the system-provided top and bottom bands. If that content overflows the provided space, scrollbars would be provided only within that area—the top and bottom bands always remain in place. In both cases, WinRT also provides the usual features for activation, such as the splashScreen and previousExecutionState properties, just as we saw in Chapter 3, "App Anatomy and Performance Fundamentals," meaning that you should reload necessary session state and use extended splash screens as needed.

What's most interesting, though, are the contract-specific interactions that are represented in the different scenarios for these pages (as you can see in Figure D-1 and Figure D-2). Let's look at each.

Note For specific details on designing a file picker experience, see Guidelines for file pickers.

File Open Provider: Local File

The provider for file open works through the FileOpenPickerUI object supplied with the fileOpenPicker activation kind. Simply said, whatever kind of UI the provider offers to select some file or data will be wired to the various methods, properties, and events of this object. First, the UI will use the allowedFileTypes property to filter what it displays for selection—clearly, the provider should not display items that don't match what the file picker is being asked to pick! Next, the UI can use the selectionMode property (a FileSelectionMode value) to determine if the file picker was invoked for single or multiple selection.

When the user selects an item within the UI, the provider calls the addFile method with the StorageFile object as appropriate for that item. Clearly, the provider has to somehow create that StorageFile object. In the sample's open picker > scenario 1 (js/fileOpenPickerScenario1.js), this is accomplished with a StorageFolder.-getFileAsync (where the StorageFolder is the package location).

```
Windows.ApplicationModel.Package.current.installedLocation
    .getFileAsync("images\\squareTile-sdk.png").then(function (fileToAdd) {
    addFileToBasket(localFileId, fileToAdd);
}
```

where addFileToBasket just calls FileOpenPickerUI.addFile and displays messages for the result. That result is a value from the AddFileResult enumeration: added (success), alreadyAdded (redundant operations, so the file is already there), notAllowed (adding is denied due to a mismatched file type), and unavailable (app is not visible). These really just help you report the result to users in your UI. Note also that the canAddFile method might be helpful for enabling or disabling add commands in your UI as well, which will prevent some of these error cases from ever arising in the first place.

The provider app must also respond to requests to remove a previously added item, as when the user removes a selection from the "basket" in the multi-select file picker UI. To do this, listen for the FileOpenPickerUI object's fileRemoved event, which provides a file ID as an argument. You pass this ID to containsFile followed by removeFile as in the sample (js/fileOpenPickerScenario1.js):

```
// Wire up the event in the page's initialization code
fileOpenPickerUI.addEventListener("fileremoved", onFileRemovedFromBasket, false);

function removeFileFromBasket(fileId) {
    if (fileOpenPickerUI.containsFile(fileId)) {
        fileOpenPickerUI.removeFile(fileId);
    }
}
```

If you need to know when the file picker UI is closing your page (such as the user pressing the Open or Cancel buttons shown in Figure D-1), listen for the closing event. This gives you a chance to close any sessions you might have opened with an online service and otherwise perform any necessary cleanup tasks. In the eventArgs you'll find an isCanceled property that indicates whether the file picker is being canceled (true) or being closed due to the Open button (false). The eventArgs.closingOperation object also contains a getDeferral method and a deadline property that allows you to carry out async operations as well, similar to what we saw in Chapter 3 for the suspending event.

A final note is that a file picker provider should respect the FileOpenPickerUI.settingsIdentifier to relaunch the provider to a previous state (that is, a previous picker session). If you remember from the other side of this story, an app that's using the file picker can use the settingsIdentifier to distinguish different use cases within itself—perhaps to differentiate certain file types or feature contexts. The identifier can also differ between different apps that invoke the file picker. By honoring this property, then, a provider app can maintain a case-specific context each time it's invoked (basically using settingsIdentifier in its appdata filenames and the names of settings containers), which is how the built-in file pickers for the file system works.

It's also possible for the provider app to be suspended while displaying its UI and could possibly be shut down if the calling app is closed. However, if you manage picker state based on settingsIdentifier values, you don't need to save or manage any other session state where your picker functionality is concerned.

File Open Provider: URI

For the most part, scenario 2 of the open file picker case in the provider sample is just like we've seen in the previous section. The only difference is that it shows how to create a StorageFile from a nonfile source, such as an image that's obtained from a remote URI. In this situation we need to obtain a data stream for the remote URI and convert that stream into a StorageFile. Fortunately, a few WinRT APIs make this very simple, as shown in js/fileOpenPickerScenario2.js within its onAddFileUri method:

```
function onAddUriFile() {
    // Respond to the "Add" button being clicked
    var imageSrcInput = document.getElementById("imageSrcInput");

    if (imageSrcInput.value !== "") {
        var uri = new Windows.Foundation.Uri(imageSrcInput.value);
        var thumbnail =
            Windows.Storage.Streams.RandomAccessStreamReference.createFromUri(uri);

        // Retrieve a file from a URI to be added to the picker basket
        Windows.Storage.StorageFile.createStreamedFileFromUriAsync("URI.png", uri,
            thumbnail).then(function (fileToAdd) {
            addFileToBasket(uriFileId, fileToAdd);
        },
        function (error) {
            // ...
        });
    } else {
        // ...
    }
}
```

Here Windows.Storage.StorageFile.createStreamedFileFromUriAsync (a static method) does the honors to give us a StorageFile for a URI, and addFileToBasket is again an internal method that just calls the addFile method of the FileOpenPickerUI object.

If you need to perform authentication or take other special steps to obtain content from a web service, you'll generally want to use the Windows.Networking.BackgroundTransfer API to acquire the content (where you can provide credentials), followed by StorageFile.createStreamedFile to serve that file up through the contract. StorageFile.createStreamedFileFromUriAsync does exactly this but doesn't provide for authentication.

File Save Provider: Save a File

Similar to how the file open provider interacts with a FileOpenPickerUI object, a provider app for saving files works with the specific methods, properties, and events FileSavePickerUI class. Again, the open and save contracts are separate concerns because the data source for which you might create a provider app might or might not support save operations independently of open. If you do support both, you will likely reuse the same UI and would thus use the same Start page and activation path.

Within the FileSavePickerUI class, we first have the allowedFileTypes as provided by the app that invoked the file save picker UI in the first place. As with open, you'll use this to filter what you show in your own UI so that users can clearly see what items for these types already exist. You'll also typically want to populate a file type drop-down list with these types as well.

For restoring the provider's save UI for the specific calling app from a previous session, there is again the settingsIdentifier property.

Referring back to Figure D-2, notice the controls along the bottom of the screen, the ones that are automatically provided by the file picker UI when the provider app is invoked. When the user changes the filename field, the provider app can listen for and handle the FileSavePickerUI object's filenameChanged event; in your handler you can get the new value from the fileName property. If the provider app has UI for setting the filename, it cannot write to this property, however. It must instead call trySetFileName, whose return value from the SetFileNameResult enumeration is either succeeded, notAllowed (typically a mismatched file type), or unavailable. This is typically used when the user taps an item in your list, where the expected behavior is to set the filename to the name of that item.

The most important event, of course, happens when the user finally taps the Save button. This will fire the FileSavePickerUI object's targetFileRequested event. You must provide a handler for this event, in which you must create an empty StorageFile object in which the app that invoked the file picker UI can save its data. The name of this StorageFile must match the fileName property.

The eventArgs for this event is a TargetFileRequestedEventArgs object. This contains a single property named request, which is a TargetFileRequest. Its targetFile property is where you place the StorageFile you create (or null if there's an error). You must set this property before returning from the event handler, but of course you might need to perform asynchronous operations to do this at all. For this purpose, as we've seen many times, the request also contains a getDeferral method. This is used in scenario 1 of the provider sample's save case (js/fileSavePickerScenario1.js):

```
function onTargetFileRequested(e) {
    var deferral = e.request.getDeferral();

    // Create a file to provide back to the Picker
    Windows.Storage.ApplicationData.current.localFolder.createFileAsync(
        fileSavePickerUI.fileName).done(function (file) {
        // Assign the resulting file to the targetFile property and complete the deferral
        e.request.targetFile = file;
        deferral.complete();
    }, function () {
        // Set the targetFile property to null and complete the deferral to indicate failure
        e.request.targetFile = null;
        deferral.complete();
    });
};
```

In your own app you will, of course, replace the `createFileAsync` call in the local folder with whatever steps are necessary to create a file or data object. Where remote files are concerned, on the other hand, you'll need to employ the Cached File Updater contract (see "Cached File Updater" below).

File Save Provider: Failure Case

Scenario 2 of the provider sample's save UI shows one other aspect of the process: displaying errors in case there is a real failure to create the necessary `StorageFile`. Generally speaking, you can use whatever UI you feel is best and consistent with the app in general, to let the user know what they need to do. The sample uses a `MessageDialog` like so (js/fileSavePickerScenario2.js):

```
function onTargetFileRequestedFail(e) {
    var deferral = e.request.getDeferral();

    var messageDialog = new Windows.UI.Popups.MessageDialog("If the app needs the user to
correct a problem before the app can save the file, the app can use a message like this to
tell the user about the problem and how to correct it.");

    messageDialog.showAsync().done(function () {
        // Set the targetFile property to null and complete the deferral to indicate failure
        // once the user has closed the dialog.  This will allow the user to take any
        // necessary corrective action and click the Save button once again.
        e.request.targetFile = null;
        deferral.complete();
    });
};
```

Cached File Updater

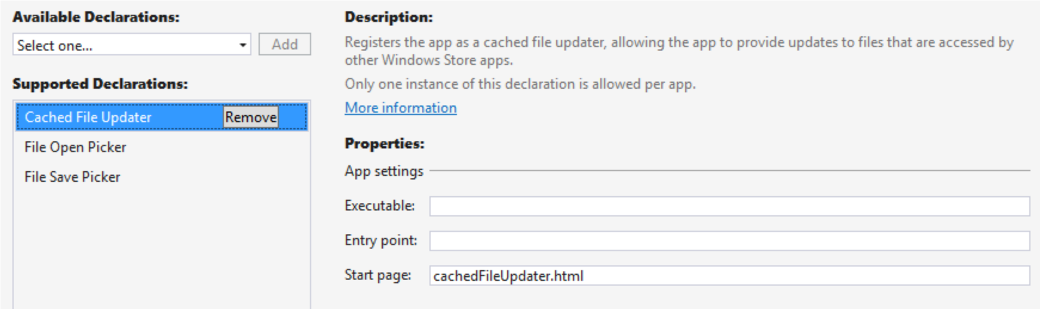
Using the cached file updater contract provides for keeping a local copy of a file in sync with one managed by a provider app on some remote resource. This contract is specifically meant for apps that provide access to a storage location where users regularly save, access, and update files. The OneDrive app in Windows is a good example of this.

Back in Chapter 11, we saw some of the method calls that are made by an app that uses the file picker: `Windows.Storage.CachedFileManager.deferUpdates` and `completeUpdatesAsync`. This usage is shown in scenarios 4 and 6 of the File picker sample we worked with in that chapter. These are the calls that a file-*consuming* app makes if and when it writes to a file that it obtained from a file picker. It does this because it won't know (and shouldn't care) whether the file provider has another copy in a database, on a web service, etc., that needs to be kept in sync. If the provider needs to handle synchronization, the consuming app's calls to these methods will trigger the necessary cached file updater UI of the provider app, which might or might not be shown, depending on the need. Even if the consuming app doesn't call these methods, the provider app will still be notified of changes but won't be able to show any UI.

There are two directions with which this contract works, depending on whether it's updating a *local* (cached) copy of a file or the *remote* (source) copy. In the first case, the provider is asked to update the local copy, typically when the consuming app attempts to access that file (pulling it from the `FutureAccessList` or `MostRecentlyUsed` list of [`Windows.Storage.AccessCache`](); it does not explicitly ask for an update). In the second case, the consuming app has modified the file such that the provider needs to propagate those changes to its source copy.

From a provider app's point of view, updates come into play whenever it supplies a file to another app. This can happen through the file picker contracts, as we've seen in the previous section, but also through file type associations and the share contract. In the latter case a share source app is, in a sense, a file provider and might make use of the cached file updater contract. In short, if you want your file-providing app to be able to track and synchronize updates between local and remote copies of a file, this is the contract to use.

Supporting the contract begins with a manifest declaration as shown below, where the Start page indicates the page implementing the cached file updater UI. That page will handle the necessary events to update files and might or might not actually be displayed to the user, as we'll see later.



Description:

Registers the app as a cached file updater, allowing the app to provide updates to files that are accessed by other Windows Store apps.

Only one instance of this declaration is allowed per app.

Properties:

App settings

Executable:

Entry point:

Start page: cachedFileUpdater.html

The next step for the provider is to indicate when a given `StorageFile` should be hooked up with this contract. It does so by calling [`Windows.Storage.Provider.CachedFileUpdater.setUpdateInformation`]() on a provided file, as shown in scenario 3 of the [File picker contracts sample](), which I'll again refer to as the *provider sample* for simplicity (js/fileOpenPickerScenario3.js):

```
function onAddFile() {
    // Respond to the "Add" button being clicked
    Windows.Storage.ApplicationData.current.localFolder.createFileAsync("CachedFile.txt",
        Windows.Storage.CreationCollisionOption.replaceExisting).then(function (file) {
        Windows.Storage.FileIO.writeTextAsync(file, "Cached file created...").then(
            function () {
                Windows.Storage.Provider.CachedFileUpdater.setUpdateInformation(
                    file, "CachedFile",
                    Windows.Storage.Provider.ReadActivationMode.beforeAccess,
                    Windows.Storage.Provider.WriteActivationMode.notNeeded,
                    Windows.Storage.Provider.CachedFileOptions.requireUpdateOnAccess);
                addFileToBasket(localFileId, file);
            }, onError);
    }, onError);
};
```

Note Be careful to not confuse the CachedFileUpdater class, which we're using here from the provider side, with the CachedFileManager class, which is used from the consumer side of the contract.

The setUpdateInformation method takes the following arguments:

- A StorageFile for the data in question.
- A content identifier string that identifies the remote resource to keep in sync.
- A ReadActivationMode indicating whether the calling app can read its local file without updating it; values are notNeeded and beforeAccess.
- A WriteActivationMode indicating whether the calling app can write to the local file and whether writing triggers an update; values are notNeeded, readOnly, and afterWrite.
- One or more values from CachedFileOptions (that can be combined with bitwise-OR) that describes the ways in which the local file can be accessed without triggering an update; values are none (no update), requireUpdateAccess (update on accessing the local file), useCachedFileWhenOffline (update on access if the calling app desires, and access is allowed if there's no network connection), and denyAccessWhenOnline (triggers an update on access and requires a network connection).

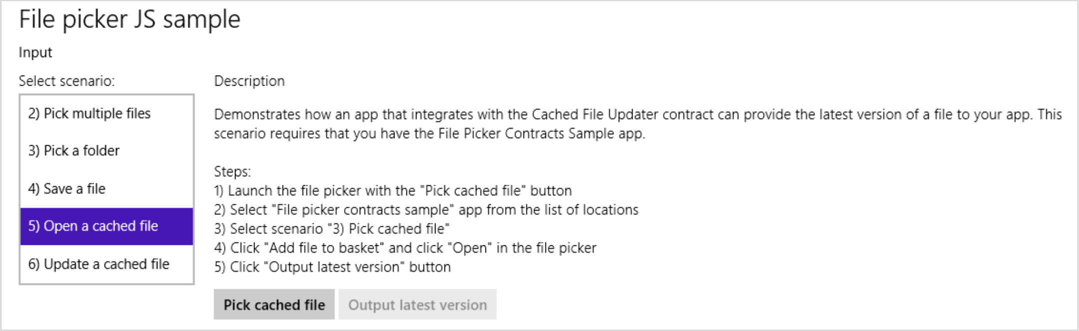
It's through this call, in other words, that the provider specifically controls how and when it should be activated to handle updates when a local file is accessed.

So, together we have two cases where the provider app will be invoked and might be asked to show its UI: one where the calling app updates the file, and another when the calling app attempts to access the file but needs an update before reading its contents.

Before going into the technical details, let's see how these interactions appear to the user. To see the cached file updater in action using the sample, invoke it by using the file picker from another app. First, then, run the provider sample to make sure its contracts are registered. Then run the aforementioned File picker sample. In the latter, scenarios 4, 5, and 6 cause interactions with the cached file updater contract. Scenarios 4 and 6 write to a file to trigger an update to the remote copy; scenario 5 accesses a local file that will trigger a local update as part of the process.

Updating a Local File: UI

In scenario 5 of the File picker sample (updating a local file), follow the instructions that it gives to tap the Pick Cached File button and select the File Picker Contracts JS Sample from the location list:



File picker JS sample

Input

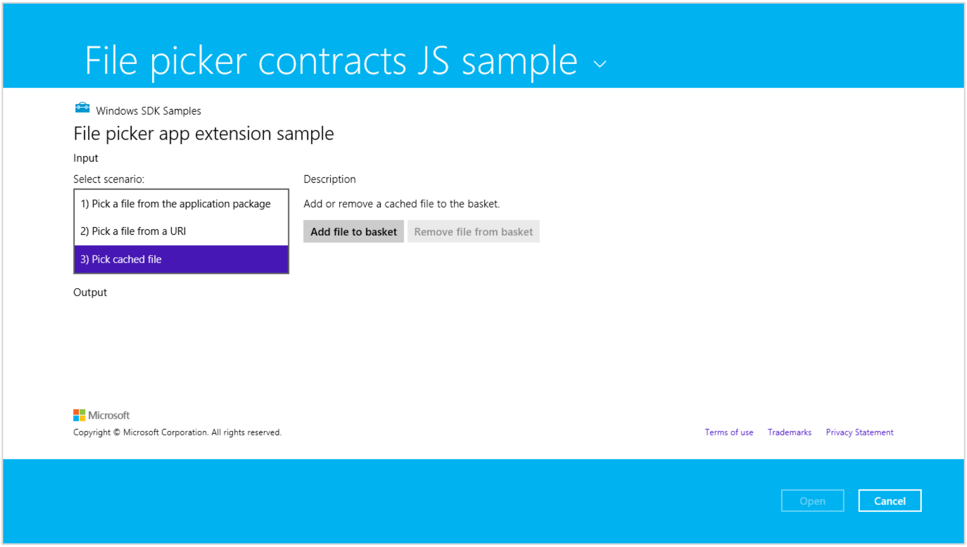
Select scenario:

Description

Demonstrates how an app that integrates with the Cached File Updater contract can provide the latest version of a file to your app. This scenario requires that you have the File Picker Contracts Sample app.

Steps:
1) Launch the file picker with the "Pick cached file" button
2) Select "File picker contracts sample" app from the list of locations
3) Select scenario "3) Pick cached file"
4) Click "Add file to basket" and click "Open" in the file picker
5) Click "Output latest version" button

This will launch the provider sample. In that view, select scenario 3 so that you see the UI shown in Figure D-3. This is the mode of the provider sample that is just a file picker *provider*, (js/fileOpenPickerScenario3.js) where it calls `setUpdateInformation`. This is *not* the UI for the cached file updater yet. Tap the Add File to Basket button, and tap the Open button. This will return you to the first app (the picker sample in the above graphic) where the Output Latest Version button will now be enabled. Tapping *that* button will then invoke the provider sample through the cached file updater contract, as shown in Figure D-4. This is what appears when there's a need to update the local copy of the cached file.



File picker contracts JS sample

Windows SDK Samples

File picker app extension sample

Input

Select scenario:

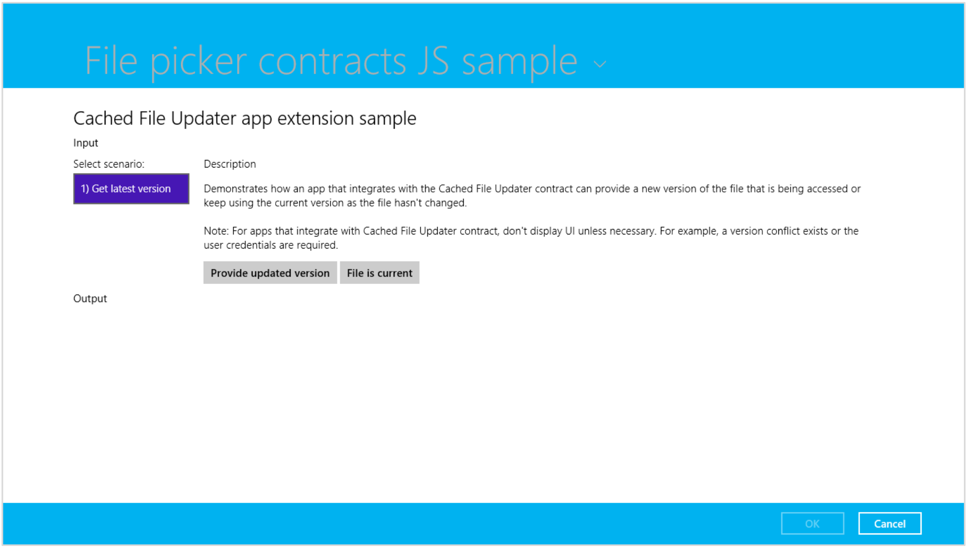
Description

Add or remove a cached file to the basket.

Output

Copyright © Microsoft Corporation. All rights reserved.

FIGURE D-3 The provider sample's UI for picking a file; the `setUpdateInfomation` method is called on the provided file to set up the cached file updater relationship.



File picker contracts JS sample

Cached File Updater app extension sample

Input

Select scenario:

Description

Demonstrates how an app that integrates with the Cached File Updater contract can provide a new version of the file that is being accessed or keep using the current version as the file hasn't changed.

Note: For apps that integrate with Cached File Updater contract, don't display UI unless necessary. For example, a version conflict exists or the user credentials are required.

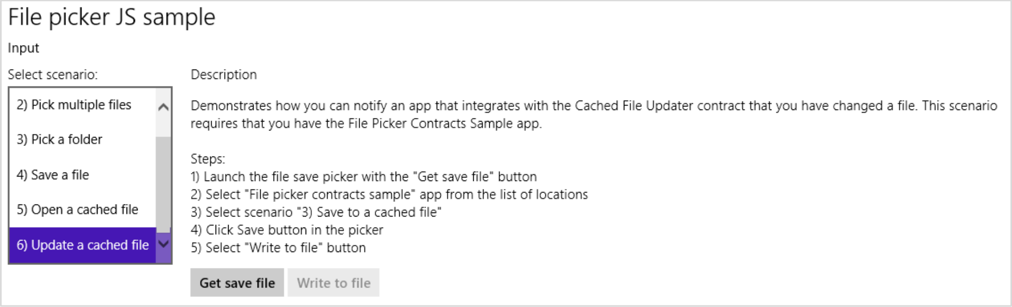
Output

FIGURE D-4 The provider sample's UI for the cached file updater contract on a local file.

Take careful note of the description in the sample. While the sample shows this UI by default, a cached file updater app will *not* show it unless it's necessary to resolve conflicts or collect credentials. Oftentimes no such interaction is necessary and the provider silently updates the local file or indicates that it's current. The sample's UI here choose instead to show both those options as explicit choices (and be sure to choose one of them because selecting Cancel will throw an exception).

Updating a Remote File: UI

In scenario 6 of the File picker sample (updating a remote file), we can see the interactions that take place when the consuming app writes changes to its local copy, thereby triggering an update to the remote copy. Start by tapping the Get Save File button in the UI shown below, and then select the File Picker Contracts JS Sample again from the location list:



File picker JS sample

Input

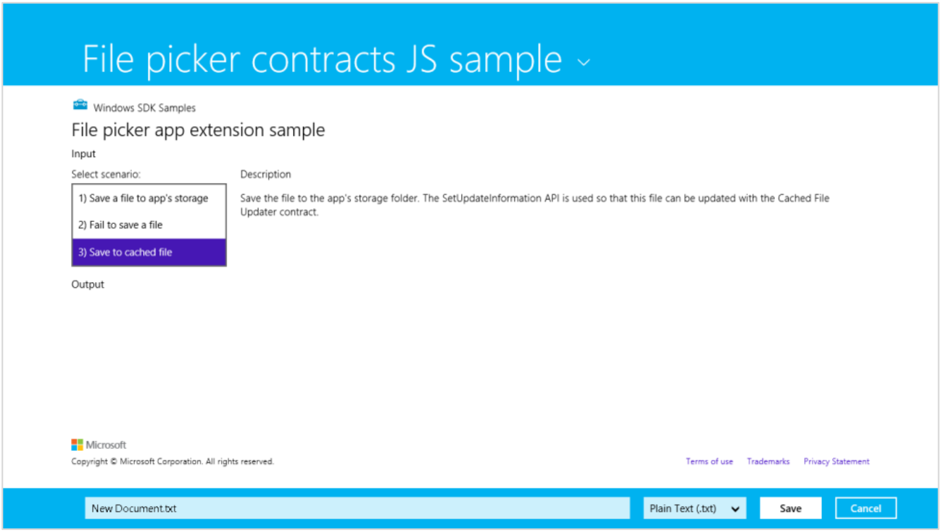
Select scenario:

Description

Demonstrates how you can notify an app that integrates with the Cached File Updater contract that you have changed a file. This scenario requires that you have the File Picker Contracts Sample app.

Steps:
1) Launch the file save picker with the "Get save file" button
2) Select "File picker contracts sample" app from the list of locations
3) Select scenario "3) Save to a cached file"
4) Click Save button in the picker
5) Select "Write to file" button

This will invoke the provider sample's UI of Figure D-5 through the file save picker contract, where you should select scenario 3 (implemented in html/fileSavePickerScenario3.html and js/fileSavePickerScenaro3.js). If you look in the JavaScript file, you'll again see a call to `setUpdateInformation` that's called when you enter a file name and tap Save. Doing so also returns you to the picker sample above where Write To File should now be enabled. Tapping Write To File then reinvokes the provider sample through the cached file updater contract with the UI shown in Figure D-6. This UI is intended to demonstrate how such a provider app would accommodate overwriting or renaming the remote file.



File picker contracts JS sample

Windows SDK Samples

File picker app extension sample

Input

Select scenario:

1) Save a file to app's storage
2) Fail to save a file
3) Save to cached file

Description

Save the file to the app's storage folder. The SetUpdateInformation API is used so that this file can be updated with the Cached File Updater contract.

Output

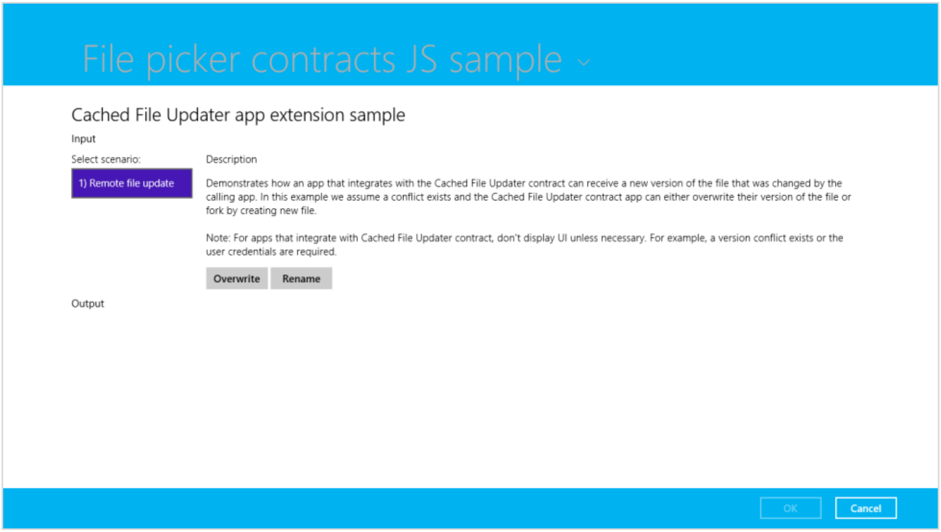
Microsoft

Copyright © Microsoft Corporation. All rights reserved.

New Document.txt

Plain Text (.txt)

FIGURE D-5 The provider sample's UI for saving a file; the `setUpdateInfomation` method is again called on the provided file to set up the cached file updater relationship.



File picker contracts JS sample

Cached File Updater app extension sample

Input

Select scenario:

1) Remote file update

Description

Demonstrates how an app that integrates with the Cached File Updater contract can receive a new version of the file that was changed by the calling app. In this example we assume a conflict exists and the Cached File Updater contract app can either overwrite their version of the file or fork by creating new file.

Note: For apps that integrate with Cached File Updater contract, don't display UI unless necessary. For example, a version conflict exists or the user credentials are required.

Output

FIGURE D-6 The provider sample's UI for the cached file updater contract on a remote file.

Update Events

Let's see how the cached file updater contract looks in code. As you will by now expect, the provider app is launched, the Start page (cachedFileUpdater.html in the project root) is loaded, and the activated handler is called with the activation kind of cachedFileUpdater. This will happen for both local and remote cases, and as we'll see here, you use the same activation code for both. Here eventObject.detail is a WebUICachedFileUpdaterActivatedEventArgs that contains a cachedFileUpdaterUI property (a CachedFileUpdaterUI) along with the usual roster of kind, previousExecutionState, and splashScreen. Here's how it looks in js/cachedFileUpdater.js of the provider sample:

```
function activated(eventObject) {
    if (eventObject.detail.kind ===
        Windows.ApplicationModel.Activation.ActivationKind.cachedFileUpdater) {
        cachedFileUpdaterUI = eventObject.detail.cachedFileUpdaterUI;

        cachedFileUpdaterUI.addEventListener("fileupdaterequested", onFileUpdateRequest);
        cachedFileUpdaterUI.addEventListener("uirequested", onUIRequested);

        switch (cachedFileUpdaterUI.updateTarget) {
            case Windows.Storage.Provider.CachedFileTarget.local:
                // Code omitted: configures sample to show cachedFileUpdaterScenario1
                // if needed.
                break;

            case Windows.Storage.Provider.CachedFileTarget.remote:
                // Code omitted: configures sample to show cachedFileUpdaterScenario2
                // if needed.
                break;
        }
    }
}
```

When the provider app is invoked to update a local file from the remote source, the cachedFileUpdaterUI.updateTarget property will be local, as you can see above. When the app is being asked to update a remote file with local changes, the target is remote. All the sample does in these cases is point to either html/cachedFileUpdaterScenario1.html (Figure D-4) or html/cachedFileUpdaterScenario2.html (Figure D-6) as the update UI.

The UI is not shown initially. What happens first is that the CachedFileUpdaterUI object fires its fileUpdateRequested event to attempt a silent update. Here the eventArgs is a FileUpdateRequestedEventArgs object with a single request property (FileUpdateRequest), an object that you'll want to save in a variable that's accessible from your update UI.

If it's possible to silently update a local file, follow these steps:

- Because you'll likely be doing async operations to perform the update, obtain a deferral from request.getDeferral.
- To do the update, use one of these options:

  - If you already have a StorageFile with the new contents, just call request.updateLocalFile. This is a synchronous call, in which case you do not need to obtain a deferral.
  - The local file's StorageFile object will be in request.file. You can open this file and write whatever contents you need within it. This will typically start an async operation, after which you return from the event handler.
- To update the contents of a remote file, copy the contents from request.file to the remote source.
- Depending on the outcome of the update, set request.status to a value from FileUpdateStatus: complete (the copies are sync'd), incomplete (sync didn't work but the local copy is still available), userInputNeeded (the update failed for need of credentials or conflict resolution), currentlyUnavailable (source can't be reached, and the local file is inaccessible), failed (sync cannot happen now or ever, as when the source file has been deleted), and completeAndRenamed (the source version has been renamed, generally to resolve conflicts).
- If you asked for a deferral and processed the outcome within completed and error handlers, call the deferral's complete method to finalize the update.

Now the provider might know ahead of time that it can't do a silent update at all—a user might not be logged into the back-end service (or credentials are needed each time), there might be a conflict to resolve, and so forth. In these cases the event handler should check the value of cachedFileUpdaterUI.uiStatus (a UIStatus) and set the request.status property accordingly:

- If the UI status is visible, switch to that UI and return from the event handler. Complete the deferral when the user has responded through the UI.
- If UI status is hidden, set request.status to userInputNeeded and return. This will trigger the CachedFileUpdaterUI.onuiRequested event followed by another fileUpdateRequested event where uiStatus will be visible, in which case you'll switch to your UI.
- If the UI status is unavailable, set request.status to currentlyUnavailable.

You can see some of this in the sample's onFileUpdateRequest handler (js/cachedFileUpdater.js); it really handles only the uiStatus check because it doesn't attempt silent updates at all (as described in the comments I've added below):

```
function onFileUpdateRequest(e) {
    fileUpdateRequest = e.request;
    fileUpdateRequestDeferral = fileUpdateRequest.getDeferral();

    // Attempt a silent update using fileUpdateRequest.file silently, or call
    // fileUpdateRequest.updateLocalFile in the local case, setting fileUpdateRequest.status
    // accordingly, then calling  fileUpdateRequestDeferral.complete(). Otherwise, if you
    // know that user action will be required, execute the following code.
    switch (cachedFileUpdaterUI.uiStatus) {
        case Windows.Storage.Provider.UIStatus.hidden:
            fileUpdateRequest.status =
                Windows.Storage.Provider.FileUpdateStatus.userInputNeeded;
            fileUpdateRequestDeferral.complete();
            break;

        case Windows.Storage.Provider.UIStatus.visible:
            // Switch to the update UI (configured in the activated event)
            var url = scenarios[0].url;
            WinJS.Navigation.navigate(url, cachedFileUpdaterUI);
            break;

        case Windows.Storage.Provider.UIStatus.unavailable:
            fileUpdateRequest.status = Windows.Storage.Provider.FileUpdateStatus.failed;
            fileUpdateRequestDeferral.complete();
            break;
    }
}
```

Again, if a silent update succeeds, the provider app's UI never appears to the user. In the case of the provider sample, it never attempts to do a silent update and so always does the check on uiStatus. When the app was just launched to service the contract, we'll end up in the hidden case and return userInputNeeded, as would happen if you attempted a silent update but returned the same status. Either way, the CachedFileUpdateUI object will fire its uiRequested event, telling the provider app that the system is making the UI visible. The app, in fact, can defer initializing its UI until this event occurs because there's no need to do so for a silent update.

After this, the fileUpdateRequested event will fire again with uiStatus now set to visible. Notice how the code above will have called request.getDeferral in this case but has not called its complete. We save that step for when the UI has done what it needs to do (and, in fact, we save both the request and the deferral for use from the UI code).

The update UI is responsible for gathering whatever user input is necessary to accomplish the task: collecting credentials, choosing which copy of a file to keep (the local or remote version), allowing for renaming a conflicting file (when updating a remote file), and so forth. When updating a local file, it writes to the StorageFile within request.file or calls request.updateLocalFile; in the remote case it copies data from the local copy in request.file.

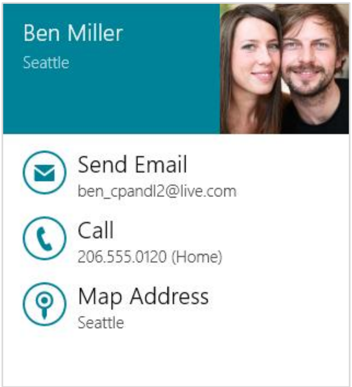
To complete the update, the UI code then sets request.status to complete (or any other appropriate code if there's a failure) and calls the deferral's complete method. This will change the status of the system-provided buttons along the bottom of the screen (see Figure D-4 and Figure D-6), enabling the OK button and disabling Cancel. In the provider sample, both buttons just execute these two lines for this purpose:

```
fileUpdateRequest.status = Windows.Storage.Provider.FileUpdateStatus.complete;
fileUpdateRequestDeferral.complete();
```

All in all, the interactions between the system and the app for the cached file updater contract are simple and straightforward in themselves: handle the events, copy data around as needed, and update the request status. The real work with this contract is first deciding when to call `setUpdateInformation`, providing the UI to support updates of local and remote files under the necessary circumstances, and interacting with your backend storage system.

Contact Cards Action Providers

In Chapter 15, "Contracts," we saw that contact cards can appear with up to three actions, depending on the data contained in the contact. To repeat a few images from that chapter below, we see actions such as Email, Call, Send Message, and Map:

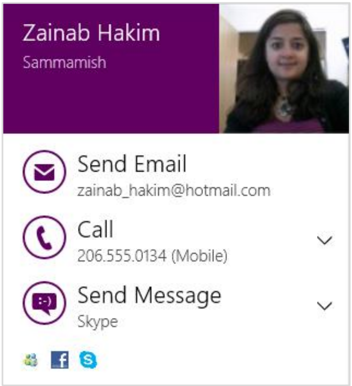


Ben Miller
Seattle

Send Email
ben_cpandl2@live.com

Call
206.555.0120 (Home)

Map Address
Seattle

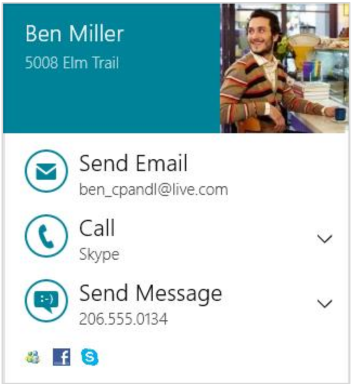


Zainab Hakim
Sammamish

Send Email
zainab_hakim@hotmail.com

Call
206.555.0134 (Mobile)

Send Message
Skype



Ben Miller
5008 Elm Trail

Send Email
ben_cpandl@live.com

Call
Skype

Send Message
206.555.0134

Furthermore, when a contact is found via the Search charm, or opened in the People app (by tapping More Details in the lower right of the contact card, not shown in the images above), these actions are available along with other less common ones, again depending on the information that's available for that contact. To round out the list, the full set of supported actions are Call, Map, Send Message, Post To, and Video Call. (A View Profile might also appear in the People app specifically for Facebook contacts as well.)

The question we want to answer here is how to create an app that can handle one or more of these actions.

The answer has two parts. First, some actions work through URI associations. The two primary ones are Email that uses `mailto:` and Call, which uses `tel:` for simple phone numbers.[144] These actions will just launch a URI with either scheme and let Windows take care of the rest. Providers that handle email and phone-number calling thus register themselves to handle these URI schemes, as generally described in Chapter 15 in the section "Launching Apps with URI Scheme Associations." For more details on the URIs schemes themselves, refer to the URI Scheme page on Wikipedia, where you'll find links to all the specifications.

For all other actions, and as an optional way to handle Call, there are special entries in the app manifest and special cases of app activation. You can find an example of this in the Handling Contact Actions sample as well as in the Quickstart: Handling contact actions topic in the documentation.

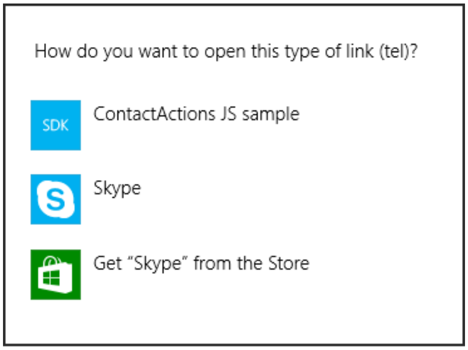
In these cases you'll need to edit your manifest manually as XML (right-click package.appxmanifest in Visual Studio, and select View Code). Within the `Application` element, make sure there's an `Extensions` section, within which you specify the `windows.contact` extension with one or more actions. Here's the XML from the SDK sample (noting that the `m2` namespace is for the Windows 8.1 manifest additions):

```
<Application Id="App" StartPage="default.html">
  <!-- Other entries omitted -->
  <Extensions>
    <m2:Extension Category="windows.contact">
      <m2:Contact>
        <m2:ContactLaunchActions>
          <m2:LaunchAction Verb="call">
            <m2:ServiceId>telephone</m2:ServiceId>
          </m2:LaunchAction>
          <m2:LaunchAction Verb="message">
            <m2:ServiceId>skype.com</m2:ServiceId>
          </m2:LaunchAction>
          <m2:LaunchAction Verb="map"/>
        </m2:ContactLaunchActions>
      </m2:Contact>
    </m2:Extension>
  </Extensions>
</Application>
```

Each supported action is described by an `m2:LaunchAction` element, whose `Verb` attribute identifies the action and must be a value from the `Windows.ApplicationModel.Contacts.Contact-LaunchActionVerbs` class: `call`, `message`, `map`, `post`, and `videoCall`. The `m2:ServiceId` element then describes the service that the app uses to complete the action (not needed for the `map` verb).

[144] Two other URI schemes are for Facebook-specific messaging and View Profile actions: `message-facebook-com:` and `viewprofile-facebook-com:`. You'd use these only if you're making a dedicated Facebook app.

Once an app with such manifest entries is installed, it shows up as a target when the user selects an action. For example, run the Handling Contact Actions sample from Visual Studio and select a Call action from a contact card or the Search charm. If you haven't yet chosen a default handler for calls, you'll see something like this:



How do you want to open this type of link (tel)?

ContactActions JS sample

Skype

Get "Skype" from the Store

Tip Users control these associations through PC Settings > Search and Apps > Defaults > Choose Default Apps By Protocol.

Assuming that your app—and we're using the sample here—is the default provider for that action, it will be launched with `ActivationKind.contact`, as shown here in the sample's `activated` handler alongside the `protocol` and `launch` activations for reference (js/default.js):

```
if (eventObject.detail.kind === Windows.ApplicationModel.Activation.ActivationKind.contact) {
    arg = eventObject.detail;

    if (arg.verb === Windows.ApplicationModel.Contacts.ContactLaunchActionVerbs.call) {
        // Handle calls
    } else if (arg.verb === Windows.ApplicationModel.Contacts.ContactLaunchActionVerbs.message) {
        // Handle messaging
    } else if (arg.verb === Windows.ApplicationModel.Contacts.ContactLaunchActionVerbs.map) {
        // Handle mapping
    }
} else if (eventObject.detail.kind ===
Windows.ApplicationModel.Activation.ActivationKind.protocol) {
    // Protocol activation for URI associations
} else if (eventObject.detail.kind ===
Windows.ApplicationModel.Activation.ActivationKind.launch) {
    // Normal launch
}
```

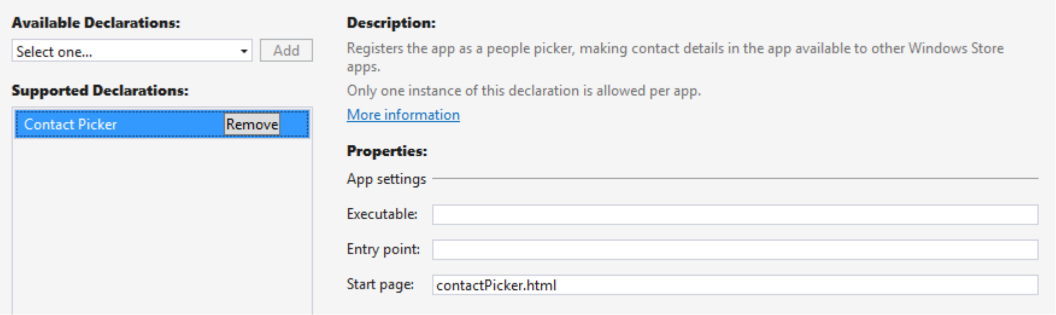
In a contact activation, `eventObject.detail.verb` will contain one of the `ContactLaunchActionVerb` values, so you'll then take appropriate action for each one individually. Here, `eventObject.detail` will be one of the following object types, depending on the verb. Note that each object type includes the standard properties of `kind`, `previousExecutionState`, and `splashScreen`, which I won't show individually.

Verb	Event object type	Properties
call	ContactCallActivatedEventArgs	contact, serviceId, serviceUserId, verb
map	ContactMapActivatedEventArgs	address, contact, verb
message	ContactMessageActivatedEventArgs	contact, serviceId, serviceUserId, verb
post	ContactPostActivatedEventArgs	contact, serviceId, serviceUserId, verb
videoCall	ContactVideoCallActivatedEventArgs	contact, serviceId, serviceUserId, verb

With each activation verb and a Contact object in the contact field, you have all the information you need in hand including display names, thumbnails, and so forth. In the case of the map verb, the other field of interest is ContactAddress, giving you what you need to do the mapping. In all other cases, the serviceId is whatever string is specified in the manifest entry for this action, and serviceUserId is the appropriate data from the contact for that service. For example, when serviceId is telephone, the serviceUserId will be the telephone number. For a video call it could be the user's Skype ID, and for messaging it would be the appropriate user ID for that service. In short, these two fields supply what you need to complete the basic action, and you build your UI with the other details in contact.

Contact Picker Providers

On the provider side of the Contact Picker, which is demonstrated in the Contact Picker app sample that we worked with in Chapter 15, we see the same pattern as for file picker providers. First, a provider app declares the Contact Picker contract in its manifest, indicating the Start page to load within the context of the picker. In the sample, the Start page is contactPicker.html that in turn loads html/contactPickerScenario.html (with their associated JavaScript files):



Available Declarations:

Select one... | Add

Supported Declarations:

Contact Picker | Remove

Description:

Registers the app as a people picker, making contact details in the app available to other Windows Store apps.

Only one instance of this declaration is allowed per app.

More information

Properties:

App settings

Executable:

Entry point:

Start page: contactPicker.html

As with the file picker, having a separate Start page means having a separate activated handler, and in this case it looks for the activation kind of contactPicker (js/contactPicker.js):

```
function activated(e) {
    if (e.detail.kind === Windows.ApplicationModel.Activation.ActivationKind.contactPicker) {
        contactPickerUI = e.detail.contactPickerUI;
        e.setPromise(WinJS.UI.processAll().then(function () {
            // ...
        }));
    }
}
```

The `e.detail` here is a ContactPickerActivatedEventArgs (these names are long, but at least they're predictable!). As with all activations, it contains `kind`, `previousExecutionState`, and `splashScreen` properties for the usual purposes. Its `contactPickerUI` property, a ContactPickerUI, contains the information specific to the picker contract:

- The `selectionMode` and `desiredFields` properties as supplied by the calling app.
- Three methods—`addContact`, `removeContact`, and `containsContact`—for managing what's returned to the calling app. These methods correspond to the actions of a typical selection UI.
- One event, `contactsRemoved`, which informs the provider when the user removes an item from the basket along the bottom of the screen. (Refer to Figure 15-15 in Chapter 15.)

Within a provider, each contact is represented by a Contact object. A provider will create an object for each contact it supplies. In the sample (js/contactPickerScenario.js), there's an array called `sampleContacts` that simulates what would more typically come from a database. That array just contains JSON records like this:

```
{
    displayName: "David Jaffe",
    firstName: "David",
    lastName: "Jaffe",
    personalEmail: "david@contoso.com",
    workEmail: "david@cpandl.com",
    workPhone: "",
    homePhone: "248-555-0150",
    mobilePhone: "",
    address: {
        full: "",
        street: "",
        city: "",
        state: "",
        zipCode: ""
    },
    id: "761cb6fb-0270-451e-8725-bb575eeb24d5"
},
```

Each record is shown as a check box in the sample's UI (generated in the `createContactUI` function), but your own provider app will likely use a ListView for this purpose. The sample is just trying to keep things simple so that you can see what's happening with the contract itself.

When a contact is selected, the sample's `addContactToBasket` function is called. This is the point at which we create the actual `Contact` object and call `ContactPickerUI.addContact`. The process here for each field follows a chain of other function calls, so let's see how it works for the single *homeEmail* field in the source record, starting with addContactToBasket (again in js/contactPickerScenario.js). The rest of the field values are handled pretty much the same way:

```
function addContactToBasket(sampleContact) {
    var contact = new Windows.ApplicationModel.Contacts.Contact();
    contact.firstName = sampleContact.firstName;
    contact.lastName = sampleContact.lastName;
    contact.id = sampleContact.id;

    if (sampleContact.personalEmail) {
        var personalEmail = new Windows.ApplicationModel.Contacts.ContactEmail();
        personalEmail.address = sampleContact.personalEmail;
        personalEmail.kind = Windows.ApplicationModel.Contacts.ContactEmailKind.personal;
        contact.emails.append(personalEmail);
    }

    // Add other fields...

    // Add the contact to the basket
    switch (contactPickerUI.addContact(sampleContact.id, contact)) {
        // Show various messages based on the result, which is of type
        // Windows.ApplicationModel.Contacts.Provider.AddContactResult
    }
}
```

Now, when an item is unselected in the list, it needs to be removed from the basket:

```
function removeContactFromBasket(sampleContact) {
    // Programmatically remove the contact from the basket
    if (contactPickerUI.containsContact(sampleContact.id)) {
        contactPickerUI.removeContact(sampleContact.id);
    }
}
```

Similarly, when the user removes an item from the basket, the contact provider needs to update its selection UI by handling the contactremoved event:

```
contactPickerUI.addEventListener("contactremoved", onContactRemoved, false);

function onContactRemoved(e) {
    // Add any code to be called when a contact is removed from the basket by the user
    var contactElement = document.getElementById(e.id);
    var sampleContact = sampleContacts[contactElement.value];
    contactElement.checked = false;
}
```

You'll notice that we haven't said anything about closing the UI, and in fact the ContactPickerUI object does not have an event for this. Simply said, when the user selects the commit button (with whatever text the caller provided), it gets back whatever the provider has added to the basket. If the user taps the cancel button, the operation returns a null contact. In both cases, the provider app will be suspended and, if it wasn't running prior to being activated for the contact, closed automatically.

Do note that as with file picker providers, a contact provider should save its session state when suspended and restore that state when relaunched with previousExecutionState==terminated. Although not demonstrated in the sample, a real provider app should save its current selections and viewing position within its list, along with whatever else, to session state and restore that in its activated handler when necessary.

Appointment Providers

An appointment provider app registers itself to handle the four distinct appointment actions apropos to the Appointments API discussed in Chapter 15. Those actions are adding an appointment, removing an appoint-ment, updating (replacing) an appointment, and showing a time frame.

To demonstrate at least the structure of a provider app, I've included the AppointmentsProvider example in the companion content for the appendices. This example does not maintain any kind of calendar—it only shows how to handle the activations for the different verbs and the data that's passed with each one.

To register itself, the provider app must declare appropriate extensions in its manifest XML. As with contact actions described earlier, the Visual Studio manifest editor does not presently have a UI for these entries, so you'll need to right-click package.appxmanifest in Visual Studio and select View Code to edit the XML directly. Then within Application > Extensions, add an Extension for the windows.appointmentProvider category:

```
<Application>
  <Extensions>
    <m2:Extension Category="windows.appointmentsProvider" StartPage="default.html">
      <m2:AppointmentsProvider>
        <m2:AppointmentsProviderLaunchActions>
          <m2:LaunchAction Verb="addAppointment" StartPage="html/manageAppointment.html"/>
          <m2:LaunchAction Verb="removeAppointment" StartPage=" html/manageAppointment.html"/>
          <m2:LaunchAction Verb="replaceAppointment" StartPage=" html/manageAppointment.html"/>
          <m2:LaunchAction Verb="showTimeFrame" />
        </m2:AppointmentsProviderLaunchActions>
      </m2:AppointmentsProvider>
    </m2:Extension>
  </Extensions>
</Application>
```

Each action, as you can see, is represented by a LaunchAction entry that maps a verb to a particular page in the app that handles that kind of activation.

> **Note** An appointments provider *must* implement all four verbs because users set a default calendar app for all verb together. If you don't handle one of the verbs, you'll break apps that attempt to use the appointments API.

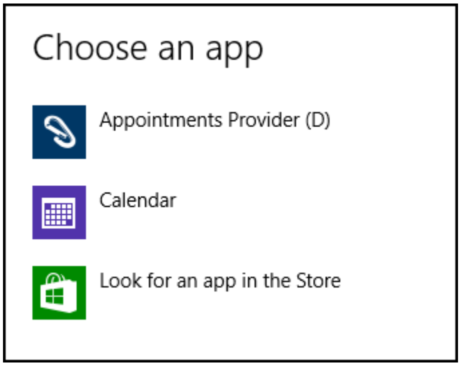
If a LaunchAction doesn't specify its own StartPage attribute, the activation will use the

StartPage specified with the Extension element. In the declarations above, we use html/manageAppointment.html for add, remove, and replace cases, whereas the showTimeFrame verb will go to default.html.

I've chosen to show it this way because if you refer back to Chapter 15, the APIs to add, remove, and replace appointments display the provider's UI in a flyout within the calling app, and so it makes sense to have a dedicated HTML page for that purpose (or you can use distinct pages if you like). The call that triggers the showTimeFrame verb, on the other hand, launches the provider app in its own view, so it makes some sense to use the same view as the app would if it's launched standalone and then to navigate to the specific time frame.

The verbs themselves, by the way, come from the AppointmentsProviderLaunchActionVerbs, which is found in the Windows.ApplicationModel namespace, down under Appointments.AppointmentProvider, which, put altogether in code, forms about the longest fully-qualified name I've encountered in all of WinRT! Its values are exactly those you see in the manifest above.

Installing an app with these entries makes it available to handle calendar actions, and the act of installation will cause a prompt to appear the next time the user performs an action that invokes the default calendar. This way your users will have the opportunity to select your app that they just installed and use it in favor of the built-in Calendar. The user can also change the default in PC Settings > Search and Apps > Defaults > Calendar:



When the provider app is launched to handle an appointment action, Windows will load the StartPage specified for that verb and call the activated handler in that page. The activation kind is set to appointmentsProvider, and the appropriate verb in contained in eventObject.detail.verb. In the example's *manageAppointments* page, then, we have this basic structure (html/manageAppointments.js):

```
var app = WinJS.Application;
var activation = Windows.ApplicationModel.Activation;
var verbs =

Windows.ApplicationModel.Appointments.AppointmentsProvider.AppointmentsProviderLaunchActionVerbs
;
```

```
app.onactivated = function (args) {
    if (args.detail.kind === activation.ActivationKind.appointmentsProvider) {
        switch (args.detail.verb) {
            case verbs.addAppointment:
                break;

            case verbs.removeAppointment:
                break;

            case verbs.replaceAppointment:
                break;
        }

        args.setPromise(WinJS.UI.processAll());
    }
};

app.start();
```

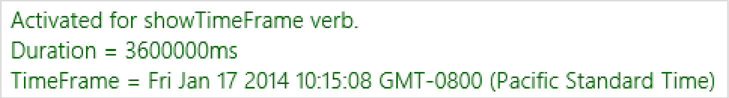
I told you the fully-qualified name of the verbs class was long! Anyway, notice that because this page is being launched by itself, we have the same structure that we have in default.js, including a call to `app.start()` at the bottom. And speaking of default.js, here's how we differentiate the `showTimeFrame` launch case from normal startup (js/default.js):

```
var verbs =

Windows.ApplicationModel.Appointments.AppointmentsProvider.AppointmentsProviderLaunchActionVerbs
;

app.onactivated = function (args) {
    if (args.detail.kind === activation.ActivationKind.appointmentsProvider) {
        if (args.detail.verb == verbs.showTimeFrame) {
            // Launched to show appointments time frame
        }
        args.setPromise(WinJS.UI.processAll());
    }

    if (args.detail.kind === activation.ActivationKind.launch) {
        // Normal launch

        args.setPromise(WinJS.UI.processAll());
    }
};
```

The type of object in `eventObject.detail` is unique in each case, as described in the following table. I can't stand typing out the really long types directly, so I'm going to rely on your inherent sense of wildcards with which you can substitute the type noted into `AppointmentsProvider*ActivatedEventArgs`. Each of these objects contains the usual `kind`, `previousExecutionState`, and `splashScreen` properties, which I won't list. The other properties are then typically `verb` and an operation-specific object, except with `showTimeFrame`:

Verb	Event object type	Properties (in eventObject.detail)
addAppointment	AddAppointment	verb, addAppointmentOperation (an AddAppointmentOperation)
removeAppointment	RemoveAppointment	verb, removeAppointmentOperation (a RemoveAppointmentOperation)
replaceAppointment	ReplaceAppointment	verb, replaceAppointmentOperation (a ReplaceAppointmentOperation)
showTimeFrame	ShowTimeFrame	verb, timeToShow (the starting Date), duration (period to short from the starting date in milliseconds)

With showTimeFrame, the duration and timeToShow values are the same ones the client passed to the AppointmentsManager.showTimeFrameAsync method, so you can just use these to configure your calendar view. The example, for its part, just shows these values in its output. Enough said!



```
Activated for showTimeFrame verb.
Duration = 3600000ms
TimeFrame = Fri Jan 17 2014 10:15:08 GMT-0800 (Pacific Standard Time)
```

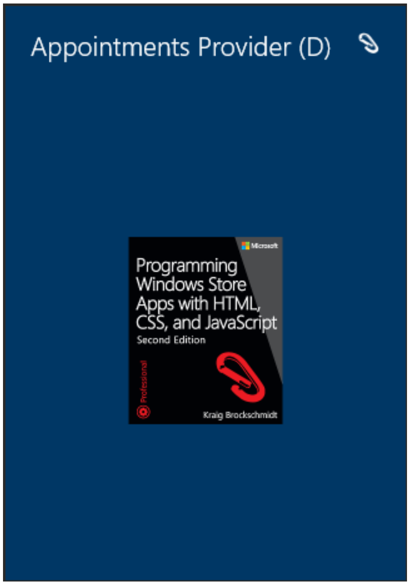
The other three verbs each receive an *operation* object, whose members are described in the following table. Note that there are a number of common members across all three, and then a few unique members for each:

Common Member	Type	Description
dismissUI	method	Closes the flyout UI in the client. The provider typically uses this to dismiss the UI while it continues to complete the operation in the background. The provider has 15 seconds from the time it calls dismissUI to call reportCompleted or reportError. Calling reportCanceled will automatically dismiss the UI.
reportCanceled	method	Informs the client that the user canceled the operation through the flyout UI.
reportCompleted	method	Informs the client that the operation was carried out. In add and remove cases, this is called with an app-defined *appointmentId* for the new entry. In the replace operation object, this method takes no arguments.
reportError	method	Informs the client that an error occurred and the operation failed. This method takes a string that describes the error.
sourcePackageFamilyName	property	The package family name of the client app.
Add/Replace operations		
appointmentInformation	property	The Appointment object containing details for the new or replacement entry.
Remove/Replace operations		
appointmentId	property	The id of the appointment to remove or replace, as was returned by the reportCompleted method via add and replace verbs.
instanceStartDate	property	A Date indicating the specific appointment to remove or replace. This is included because the appointmentId could represent a recurring appointment but the client is asking to manage a specific instance in that series.

With all this it's fairly obvious that you handle each verb with an appropriate bit of UI, displaying the relevant change to the calendar with the provided Appointment object and perhaps including a button to save the changes. The AppointmentsProvider example demonstrates the basics, again not showing a calendar but at least displaying the information it receives. Refer to the code in html/manageAppointment.js for details.

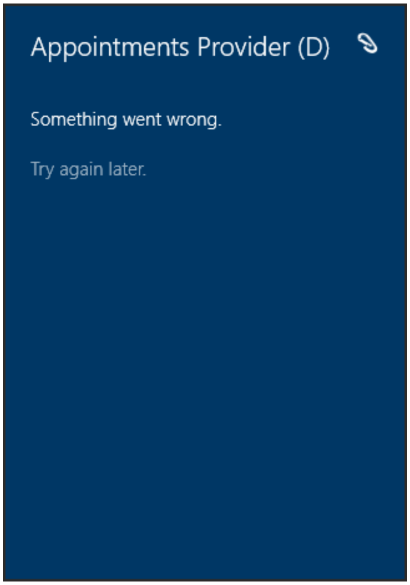
What's interesting about activating the provider app within a flyout is that you'll be working with a smaller amount of screen real-estate than you ever encounter in normal views: 320x400 pixels. Windows also provides a header with your app's Display Name (from the Application tab in the manifest) and square 30x30 logo (from Visual Assets), using The Tile > Background Color (from Visual Assets) as the background.

When the provider is first launched, the flyout (below left) will show this header along with a scaled-down splash screen image against your splash screen background color (which in this example is the same as the tile color). And because of the downscaling of your splash screen image, make sure that your image looks good at about 50% of its normal resolution. If your provider app gets stuck on this splash screen, it means that Windows could not find the StartPage for that verb. Double-check that your manifest is pointing to the right HTML page. Windows will eventually time out and show the error below right. The same error will appear if your app crashes while handling the verb.



Appointments Provider (D)

Microsoft
Programming Windows Store Apps with HTML, CSS, and JavaScript
Second Edition
Professional
Kraig Brockschmidt



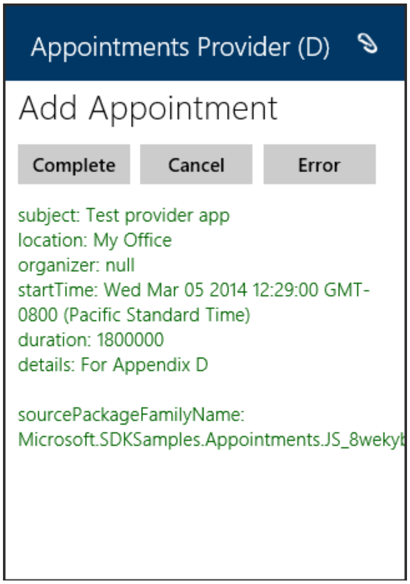
Appointments Provider (D)

Something went wrong.

Try again later.

To debug your provider, run the app in the Visual Studio debugger and let it go through its normal startup path. Later activations through appointments verbs will hit any breakpoints you set in those code paths, including any activations that hit default.html and call your activated handler there again. The caveat here is that if you're debugging on the local machine, the flyout in the client will be dismissed as soon as you switch focus to the debugger. It works better, then, to debug in the simulator or on a remote machine where focus won't be affected like this.

Anyway, if all goes well, you'll see the UI you've defined in your verb-specific pages, such as the image below from the example app. Of course, I wholly expect you to design a much better UI than this!



subject: Test provider app
location: My Office
organizer: null
startTime: Wed Mar 05 2014 12:29:00 GMT-0800 (Pacific Standard Time)
duration: 1800000
details: For Appendix D

sourcePackageFamilyName:
Microsoft.SDKSamples.Appointments.JS_8wekyb

About the Author

Kraig Brockschmidt has worked with Microsoft since 1988, focusing primarily on helping developers through writing, education, public speaking, and direct engagement. Kraig is currently a Senior Program Manager in the Windows Ecosystem team working directly with the developer community as well as key partners on building apps for Windows. Through work like *Programming Windows Store Apps in HTML, CSS, and JavaScript,* he brings the knowledge gained through that direct experience to the worldwide developer audience. His other books include *Inside OLE* (two editions), *Mystic Microsoft*, *The Harmonium Handbook,* and *Finding Focus.* His website is www.kraigbrockschmidt.com.

Now that you've read the book...

Tell us what you think!

Was it useful?
Did it teach you what you wanted to learn?
Was there room for improvement?

Let us know at http://aka.ms/tellpress

Your feedback goes directly to the staff at Microsoft Press, and we read every one of your responses. Thanks in advance!

Microsoft