Programming Windows Store Apps with HTML, CSS, and JavaScript

Second Edition

Professional

SECOND PREVIEW

Kraig Brockschmidt

SECOND PREVIEW

This excerpt provides early content from a book currently in development and is still in preview format. (See additional notices below.)

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

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

Contents

Introduction

In celebration of the one-year anniversary of this book's first edition, I'm delighted to offer you this second preview of *Programming Windows Store Apps in HTML, CSS, and JavaScript* (Second Edition).

It's been a huge year for both the book and the platform that it supports. With the general availability of Windows 8.1 earlier this month (October 2013), the capabilities of the Windows platform have grown dramatically, which is clearly in evidence by the expanded size of this second edition! Even in this preview (12 chapters and three appendices), the book is not far from the length of the first edition, and there are still eight chapters (and one appendix) to go. Nevertheless, it's my pleasure to provide you with a comprehensive volume on Microsoft's latest operating system.

This second preview takes us through nearly all of the Windows Library for JavaScript, which includes new controls like the Hub and Nav Bar. It also delves quite a bit into core WinRT APIs, such as those for working with HTTP requests, app state, and files. The bulk of WinRT coverage is yet to come in the final edition, as the remaining chapters include media, animations, contracts, live tiles, notifications, background tasks, device access, WinRT components, accessibility, localization, and the Windows Store. Fortunately, much is still the same as it was in Windows 8, and you can refer to the first edition of this book as a basis. Then check out the session videos from //build 2013 that you can find through http://buildwindows.com. The Windows Developer Center, http://dev.windows.com, also has updated documentation that covers Windows 8.1, so you can find much more there. I can specifically recommend Windows 8.1: New APIs and features for developers.

As I mentioned in the first preview, this second edition is intended to stand alone for developers who are starting with Windows 8.1. It represents the *state* of Windows 8.1 rather than trying to document the delta from Windows 8. For this reason I don't explain how to migrate apps from Windows 8 nor do I highlight many changes to APIs and behaviors (I do occasionally in specific cases). Check the Developer Center for such information.

That said, here's what you'll find in this second preview:

- Chapter 1, "The Life Story of a Windows Store App," covers the core characteristics of the Windows platform. This is much the same as in Windows 8, with the biggest exception being the view model for apps where we now have a variable sizing model.
- Chapter 2, "Quickstart," builds a first complete app, Here My Am!, that is gradually improved throughout the book. One significant change in this second preview is the use of thumbnails for loading and displaying images, which is a key performance consideration that you should always keep in mind. It's very seldom that you need to load full image data.
- Chapter 3, "App Anatomy and Performance Fundamentals," is much expanded from the first edition, especially including coverage of the WinJS Scheduler API for managing work on the UI thread, and a new section on debugging and profiling. I've also changed the title to reflect the

importance of performance considerations from the earliest stages of your app building.

- Chapter 4, "Web Content and Services," is a mixture of new content and networking topics from the first edition's Chapter 14. I moved these topics earlier in the book because using web content is increasingly important for apps, if not essential. This chapter covers network connectivity, hosting content (especially with the new `x-ms-webview` control), making HTTP requests (especially through the new `Windows.Web.Http.HttpClient` API), background transfers (which have been improved), authentication, and a little on Live Services.

- Chapter 5, "Controls and Control Styling," is updated from the first edition's Chapter 4 and includes the new `WinJS.UI.ItemContainer` control and discussed how to work with the WinJS dispose pattern.

- Chapter 6, "Data Binding, Templates, and Collections," combines material from the first edition's chapters on controls and collection controls. It's helpful to look at templates by themselves, especially given optimizations that are introduced with WinJS 2.0. This chapter also looks at collection data types both in WinRT (types like the *vector*) and the `WinJS.Binding.List`, setting us up well for Chapter 7.

- Chapter 7, "Collection Controls," focuses completely on the WinJS controls that provide UI for collections. These are the FlipView, ListView, Semantic Zoom, and Repeater controls. The ListView especially has many updates in WinJS 2.0, such as drag and drop support and better layouts (including custom layouts, see Appendix B).

- Chapter 8, "Layout and Views," is much rewritten from the first edition's chapter on layout, especially given the change from the distinct view states of Windows 8 to the variable sizing model of Windows 8.1. In addition, Windows 8.1 enables apps to have multiple views, which is a fun topic, and WinJS adds a new Hub control to support great home page experiences.

- Chapter 9, "Commanding UI," is much the same as the first edition's Chapter 7 and now includes the Nav Bar control that's new in WinJS 2.0. The App Bar control also have improvements for easier customization.

- Chapter 10, "The Story of State, Part 1: App Data and Settings," shares much of the first edition's content from Chapter 8 but tells the story of app data much more clearly. I have to admit that I wasn't wholly satisfied with the first edition's treatment of the subject, and after giving several //build talks that helped me organize the material, it was clear that I needed to cleanly separate app data (Chapter 10) and user data (Chapter 11). That opened up space in Chapter 10 here to also cover file I/O basics, as well as the matter of streams, buffers, and blobs.

- Chapter 11, "The Story of State, Part 2: User Data, Files, and SkyDrive," completes the discussion of working with files and folders, including the deep integration of SkyDrive that we have in Windows 8.1. This is another place we see the importance of using thumbnails when working with images, as it greatly helps to boost performance and reduce memory overhead, especially with collection controls.

- Chapter 12, "Input and Sensors," was one of the easier chapters to update, as there aren't many changes in these areas for Windows 8.1. The two main exceptions are the change from `MSPointer*` events to now-standard `pointer*` events, and the addition of geofencing APIs.
- Appendix A, "Demystifying Promises," completes the discussion of promises that are introduced in Chapters 2 and 3. After writing the first edition, I wanted to spend more time with promises for my own sake, but it's just my nature to leave a paper trail of my learnings! So, in this appendix we start from scratch about what promises are, see how promises are expressed in WinJS, explore how to create and source promises, and then pick apart some specific promise-heavy code.
- Appendix B, "WinJS Extras," is a collection of WinJS material that didn't fit cleanly into other chapters, including `WinJS.Namespace.define`, `WinJS.Class.define`, obscure WinJS APIs, and custom layouts for the ListView control (where we even do circular and spiral layouts!).
- Appendix C, "Additional Networking Topics," contains material that is related to Chapter 4 but didn't fit into that flow or that is more peripheral in nature. Note that `WinJS.xhr` is covered here because Chapter 4 emphasizes the preferred `Windows.Web.Http.HttpClient` API.

As you can see, in this second edition I'll be using appendices to go deeper into certain topics that would be too much of a distraction from the main flow of the chapters. Let me know what you think. Some of this material I've already posted on my blog, http://www.kraigbrockschmidt.com/blog, where I've been working on various topics since we published the first edition. I'll continue to be posting there, though a bit less frequently as I focus on completing this second edition.

Who This Book Is For

This book is about writing Windows Store apps using HTML5, CSS3, and JavaScript. Our primary focus will be on applying these web technologies within the Windows 8.1 platform, where there are unique considerations, and not on exploring the details of those web technologies themselves. For the most part, then, 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.

That said, much of this book is not specific to HTML, CSS, or JavaScript at all, because it's focused on the Windows platform and the Windows Runtime (WinRT) APIs. As such, at least half of this book will be useful to developers working in other languages (like C# or C++) who want to understand the system better. Much of Chapter 4 and Appendix C in this second preview, for example, is specific to WinRT. The subjects of app anatomy and promises in Chapter 3 and Appendix A, on the other hand, are very specific to the JavaScript option. In any case, this is a free ebook, so there's no risk, regardless of your choice of language and presentation technology!

In this book I'm 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, I've front-loaded the early chapters with the most important aspects of app development that also give you experience with the tools, the API, and some core platform features. 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.

Many insights have come to me 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 installed on your development machine, along with the Windows SDK for Windows 8.1 and the associated tools. All the tools, along with a number of other resources, are listed on the Windows 8.1 Downloads page. You'll specifically need Microsoft Visual Studio Express 2013 for Windows. We'll also acquire other tools along the way as we need them in this ebook. (Note that for all the screen shots 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.)

Also be sure to download the whole set of Windows app samples for JavaScript. 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.

I've seen some reviews of programming books that criticize authors for just pulling code samples from documentation rather than writing all their own code samples from scratch. Although I do provide a number of additional examples in this second preview's companion content, it's been one of my secondary goals 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. In most cases these samples are the same ones I would have written myself, and I'm very glad that wasn't necessary! In a few cases I felt the samples lacked certain scenarios, so you'll see some modified and extended samples in the companion content.

I've also 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.

A big part of the companion content are the many revisions of the app I call "Here My Am!" (a variant of "Hello World." We start building this in Chapter 2 and refine it throughout the course of the book. This includes localizing it into a number of different languages by the time we reach the end.

Something else I've done with this second preview is expand the use of video content. (All of the videos are also available in a folder with this preview's companion content). As in the first edition, I've made a few longer videos to demonstrate use of the Visual Studio and Blend tools. In a number of other cases, it's far easier to show dynamic effects in video than to explain them in text and screenshots. I'd also love to hear what you think about these.

Beyond all this, you'll find that the Windows samples gallery as well as the Visual Studio sample gallery also lets you search and browse additional 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 the Windows App Builder Blog, which is a great resource to follow. And if you're interested in the Windows 8 backstory—that is, how Microsoft approached this whole process of reimagining the operating system—check out the Building Windows 8 blog.

Some Formatting Notes

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, a fully qualified name like this—those that include the entire namespace—can become quite long and don't readily break across lines. In this second preview you'll see that these are occasionally hyphenated, as with `Windows.Security.-Cryptography.CryptographicBuffer.convertStringToBinary`. Generally speaking, the hyphen is not part of the identifier, though there are exceptions especially with CSS styles like `win-container`. In any case, Visual Studio's IntelliSense will help you, well, make sense of these!

For simplicity's sake (and because such hyphens produced some headaches in the first edition of this book), I've often omitted the namespace from identifiers because I trust you'll see it from the context. Plus, it's easy enough to search on the last piece of the identifier on http://dev.windows.com and find its reference page—or just click on the links I've included.

Occasionally, you'll also see an event name in 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 to prevent memory leaks, as discussed in 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.

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

The survey is short, and we read every one of your comments and ideas. Thanks in advance 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 now 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), and the Windows Store (including localization and accessibility). 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.

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.[1] 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!

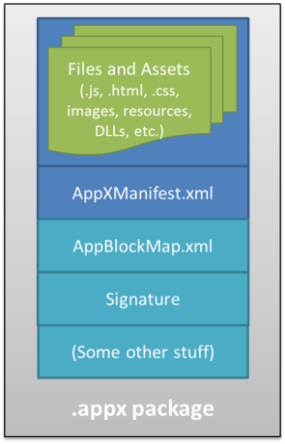


FIGURE 1-1 An appx package is simply a zip file that contains the app's files and assets, the app manifest, a signature, and a sort of table-of-contents called the blockmap. When uploading an app, the initial signature is provided by Visual Studio; the Windows Store will re-sign the app once it's certified.

[1] 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. 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.

Blockmaps make updates easy The blockmap is hugely important for the customer experience of app updates, 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 security functions (like detecting whether a package has been tampered with) and performance optimization, the blockmap is used to determine 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.[2]

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. As 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 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

[2] 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.

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 Preparing your app for the Store 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 Windows app certification requirements (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 `CurrentAppSimulator` 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 telemetry—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, screen shots,

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.

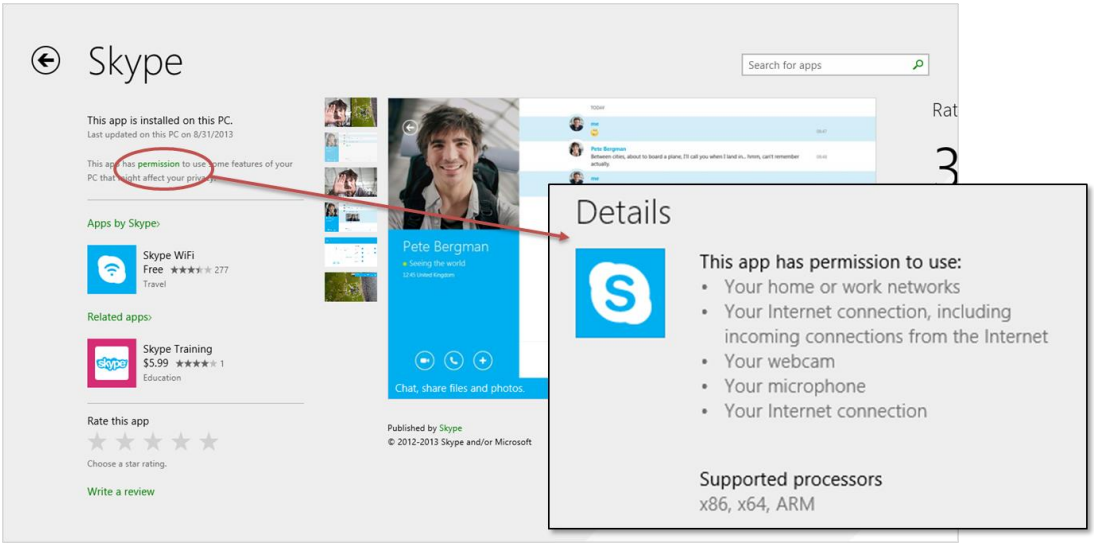


FIGURE 1-2 A typical app page in the Windows Store; by tapping the Permissions link at the upper left, the page pans to the Details section, which lists all the capabilities that are declared in the manifest (overlay). You can see here that Skype declares five different capabilities, all of which are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

The point here is that what you declare needs to 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.[3]

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

[3] The user always has the ability to disallow access to sensitive resources at run time for those apps that have declared the intent.

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 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!

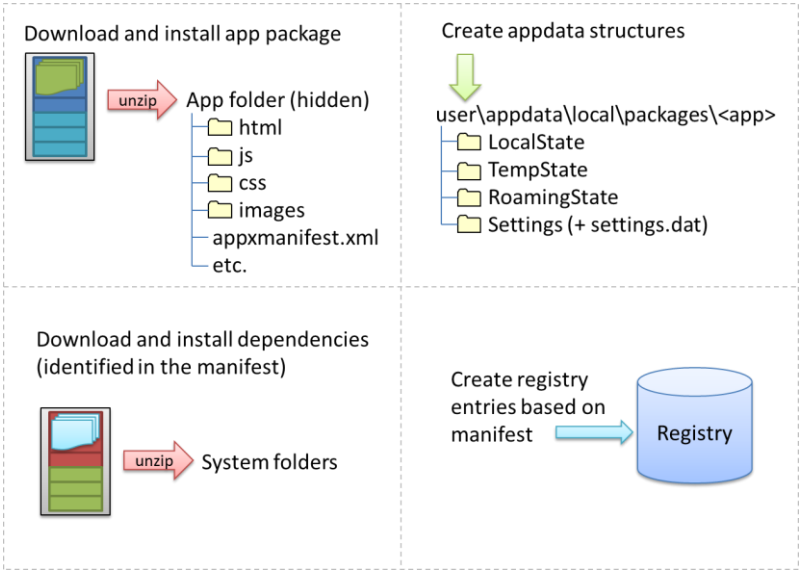


FIGURE 1-3 The installation process for Windows Store apps; the exact sequence is unimportant.

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 page 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, but you'll probably 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 many of its features throughout this book, though some won't cross our path. In any case, you can always explore what's available through the WinJS section of the Windows API reference.

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 Box2D, PhoneGap, and so forth. Apps can also use third-party binaries, known as WinRT components, that are again included in the 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 tries to automatically detect 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 page. 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[4], as shown in Figure 1-4.

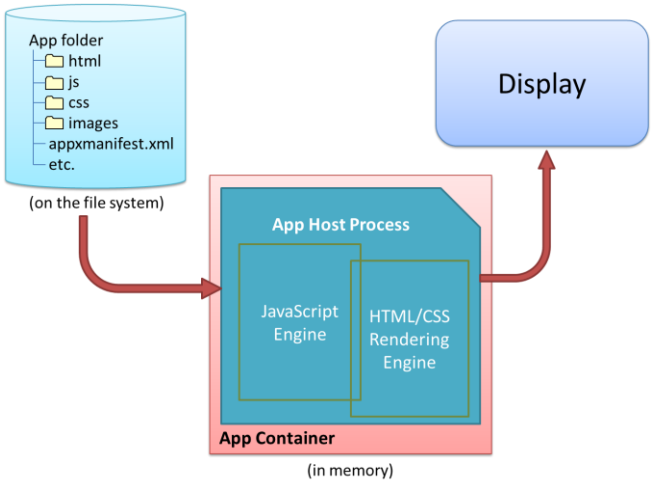


FIGURE 1-4 The app host is an executable (wwahost.exe) that loads, renders, and executes HTML, CSS, and JavaScript, in much the same way that a browser runs a web application.

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 engines support additional methods and properties that are specific to being an app as opposed to a website. Elements like `audio`, `video`, and `canvas` also have additional methods and properties. At the same time, objects like `MSApp` and methods like `requestAnimationFrame` that are available in Internet Explorer are also available to Store apps (`MSApp`, for its part, provides extra features too).
- 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

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

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.

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 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 (other than some that are built into Windows) 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) is similarly controlled—the WinRT APIs that work with those devices will fail if the broker blocks those calls. 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 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 (except in some debugging cases), between Store apps and desktop applications, and between Store apps and local services. 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.

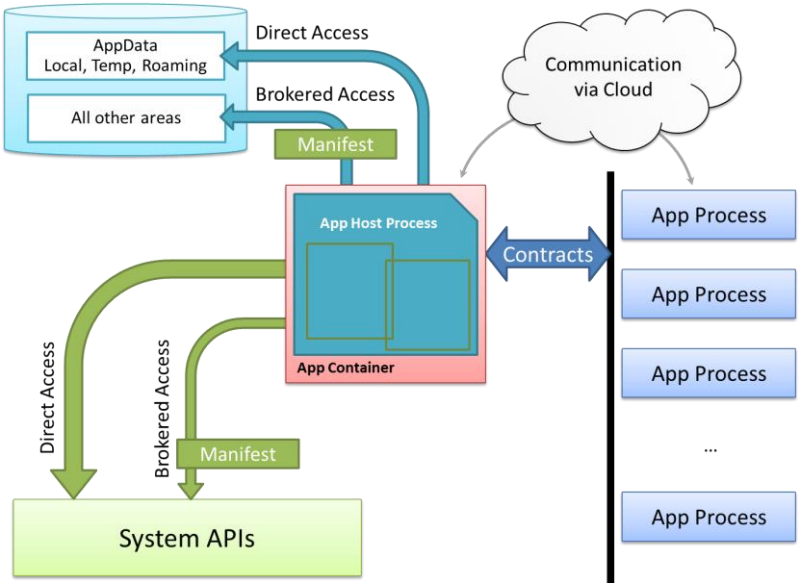


FIGURE 1-5 Process isolation for Windows Store apps.

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 compiled into binary dynamic-link libraries (DLLs), they will also typically run 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 DLLs. 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 a significant memory overhead and risks memory leaks across multiple garbage collectors.

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.

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 chrome 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.[5] (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, which increases the likelihood that the user will keep it visible.

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, where you accommodate 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 file or protocol associations, 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

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

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 also make good use of all the 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 scaling to screens and the Designing UX for apps topics in the docs.

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, or 8" devices 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 scales, contrasts (for accessibility), 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).

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).[6]	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*[7]	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.

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.

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

[7] 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.

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.

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 the user runs it 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 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.

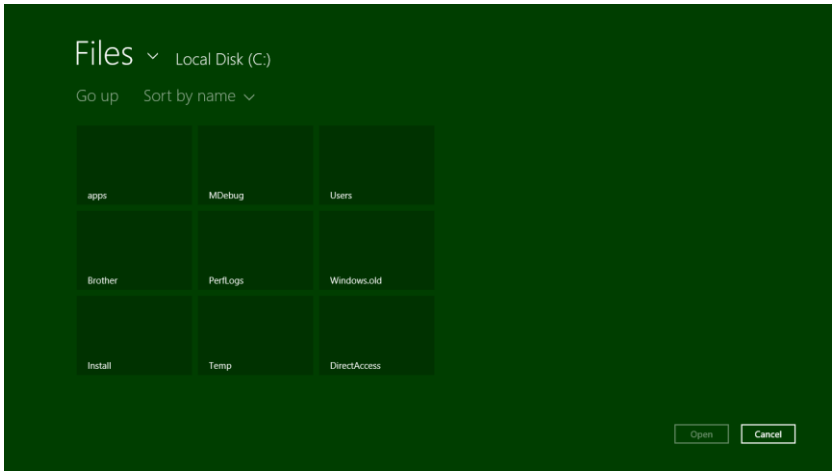


FIGURE 1-8 Using the file picker UI to access other parts of the file system from within a Store app, such as folders on a drive root (but not protected system folders). This is done by tapping the down arrow next to "Files." Typically, the file picker will look much more interesting when it's pointing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

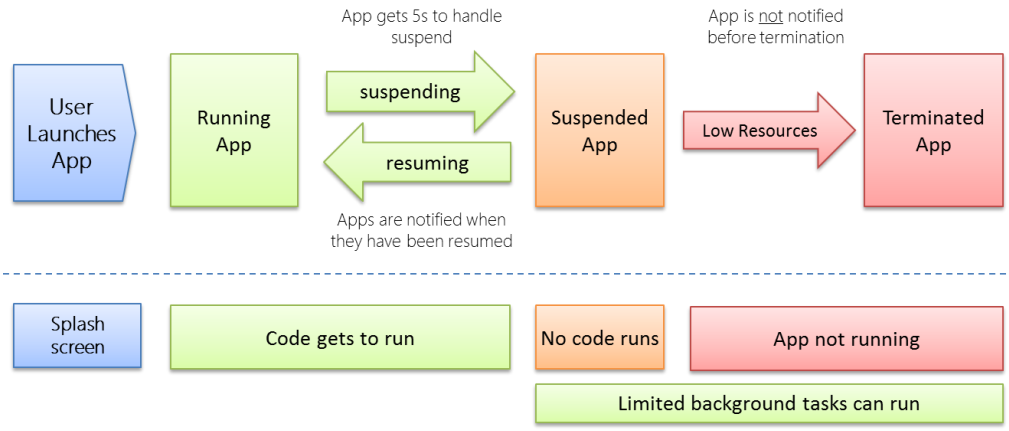


FIGURE 1-9 Process lifetime states for Windows Store apps.

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 page 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?"[8] 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 suspended. This way, if the app is terminated and restarted, it can reload that session state to return to

[8] 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?*

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 five different 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.

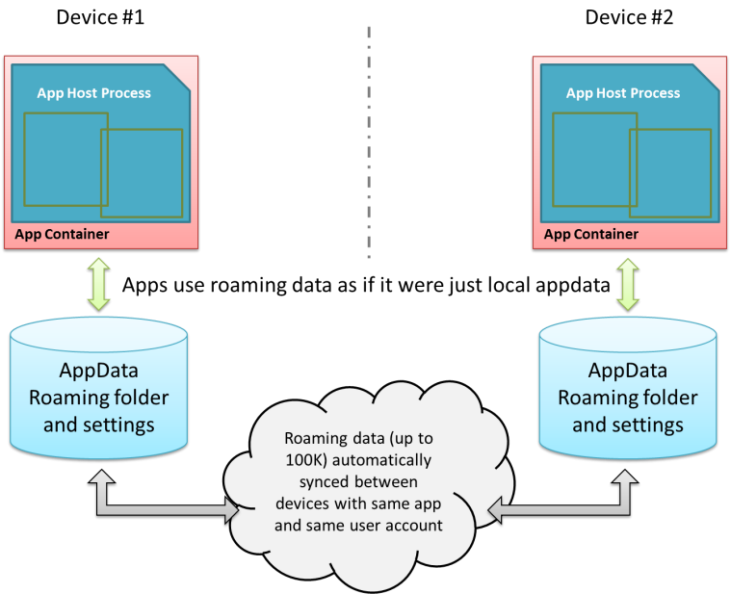


FIGURE 1-10 Automatic roaming of app roaming data (folder contents and settings) between devices.

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 to 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 SkyDrive.

So 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 has to configure the app on only one device. It 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 > SkyDrive > Sync 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. So, 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, then within "a reasonable time" reinstalls the app, the user 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 a user just happened to uninstall an app on all their 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 wrote 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. Devices 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.

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, you can still implement your own or use one of the third-party solutions found on the Windows Partner Directory.)

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 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, note that 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 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 Designing UX for apps 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 Inspiration section on the Windows Developer Center, 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 categories of apps.

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 back-end 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!

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.

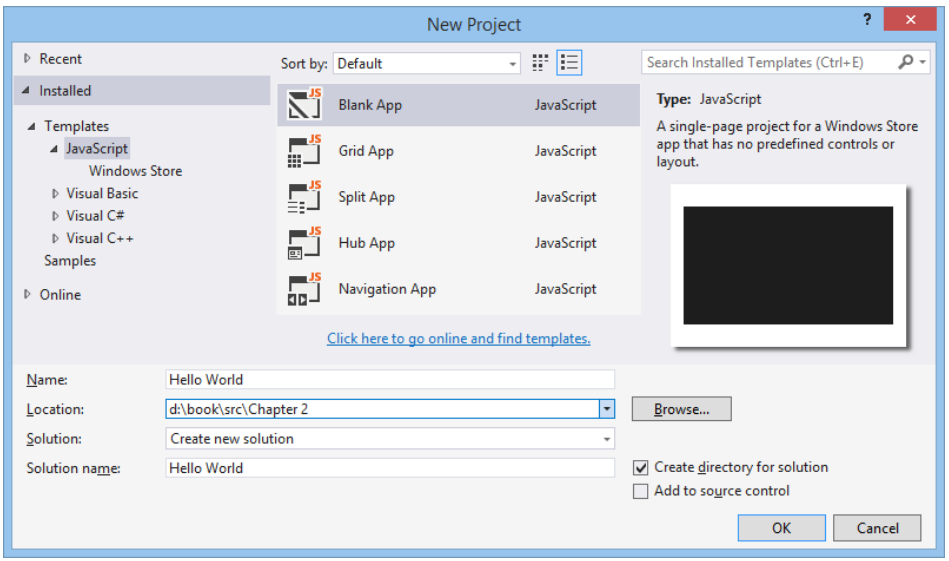


FIGURE 2-1 Visual Studio's New Project dialog using the light UI theme. (See the Tools > Options menu command, and then change the theme in the Environment/General section). I use the light theme in this book because it looks best against a white page background.

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.

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.[9]

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):

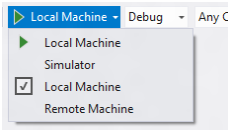


FIGURE 2-3 Visual Studio's debugging options on the toolbar.

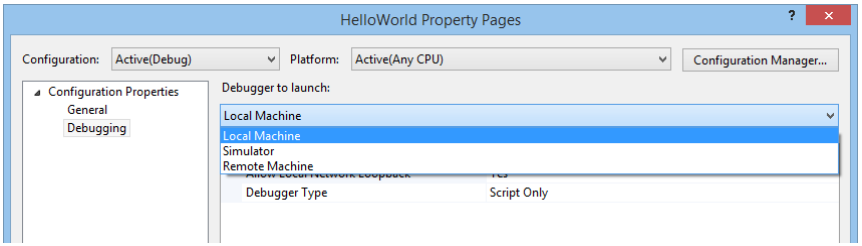


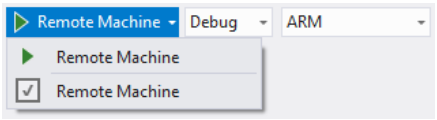
FIGURE 2-4 Visual Studio's debugging options in the app properties dialog.

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

[9] 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.

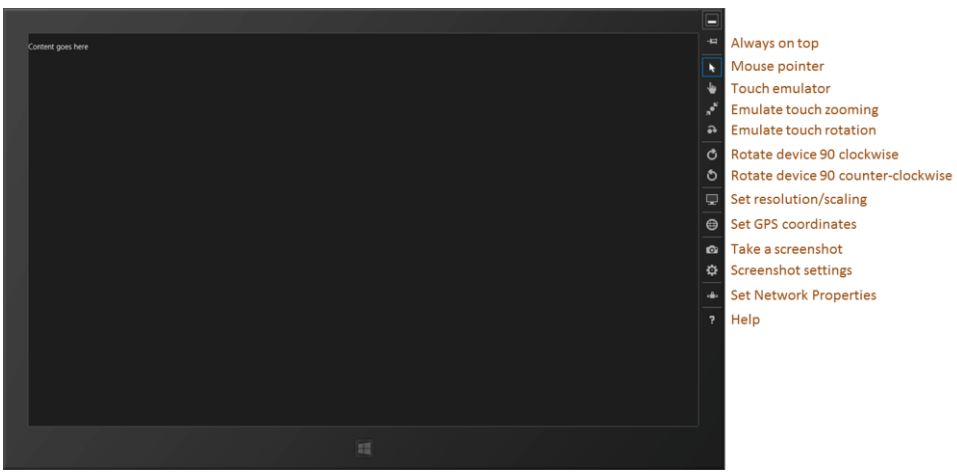


FIGURE 2-5 Hello World running in the simulator, with added labels on the right for the simulator controls. Truly, the "Blank App" template lives up to its name!

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/.

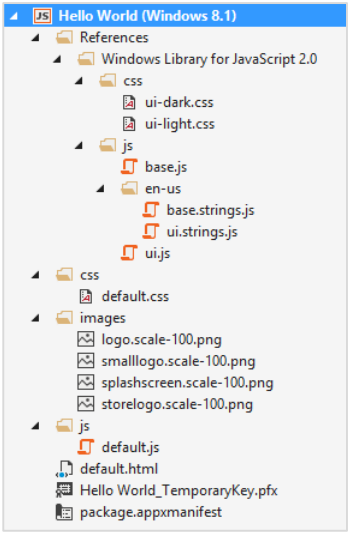


FIGURE 2-6 A Blank app project fully expanded in Solution Explorer.

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.ApplicationMode.-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.[10] 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.[11]

---

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

[11] 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 (reminder: also available with this preview'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 recommendations 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—designers can and do depart from when it makes sense.

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.

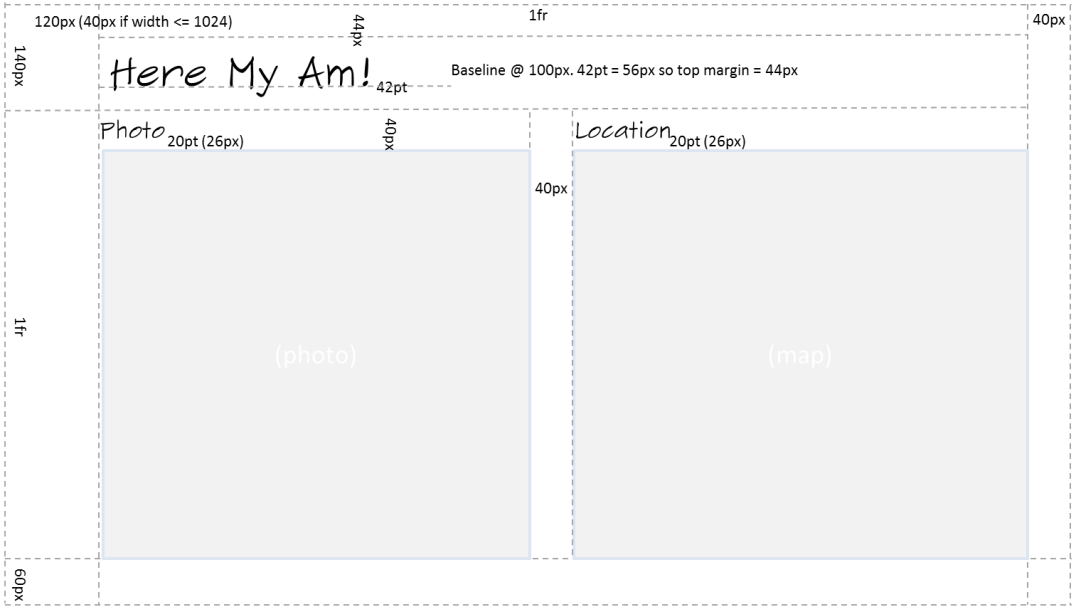


FIGURE 2-7 Wireframe for wide aspect ratios (width/height > 1). The left margin is nominally 120px, changing to 40px for smaller (<1024px) widths. The "1fr" labels denote proportional parts of the CSS grid (see Chapter 8, "Layout and Views") that occupy whatever space remains after the fixed parts are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

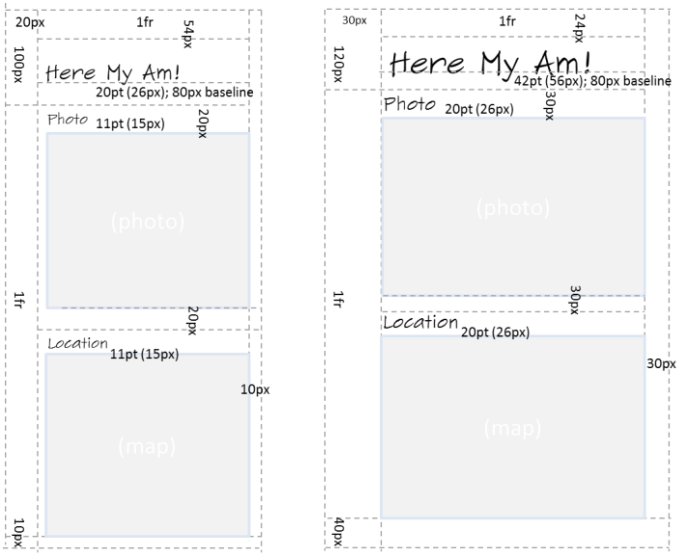


FIGURE 2-8 Wireframes for narrow (320–499px) and portrait (500px or higher) aspect ratios (width/height <= 1).

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.[12]

[12] 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).[13] 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.

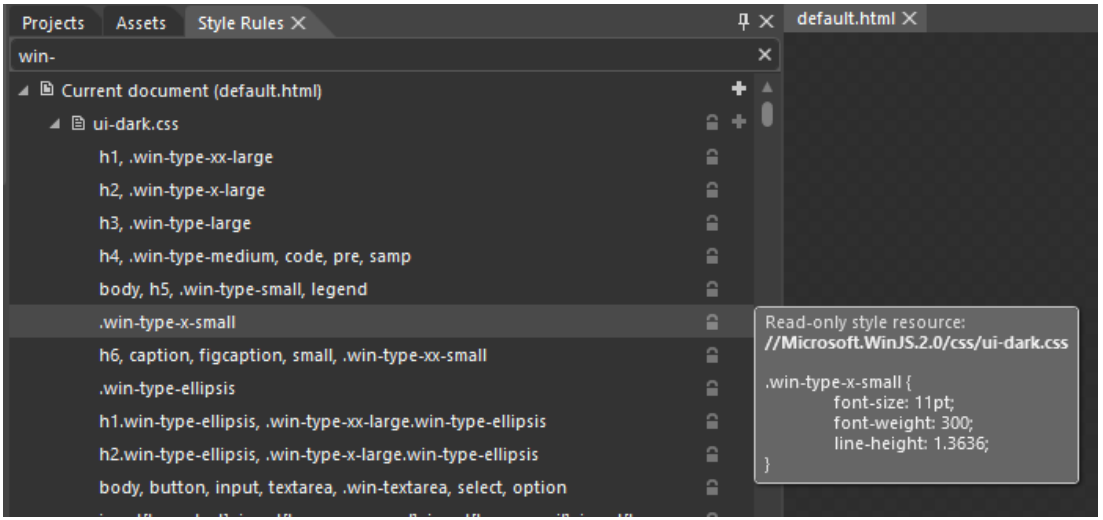


FIGURE 2-9 In Blend, the Style Rules tab lets you look into the WinJS stylesheet and see what each particular style contains. Take special notice of the search bar under the tabs where I've typed "win-". This helps you avoid visually scanning for a particular style—just start typing in the box, and let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

[13] 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.

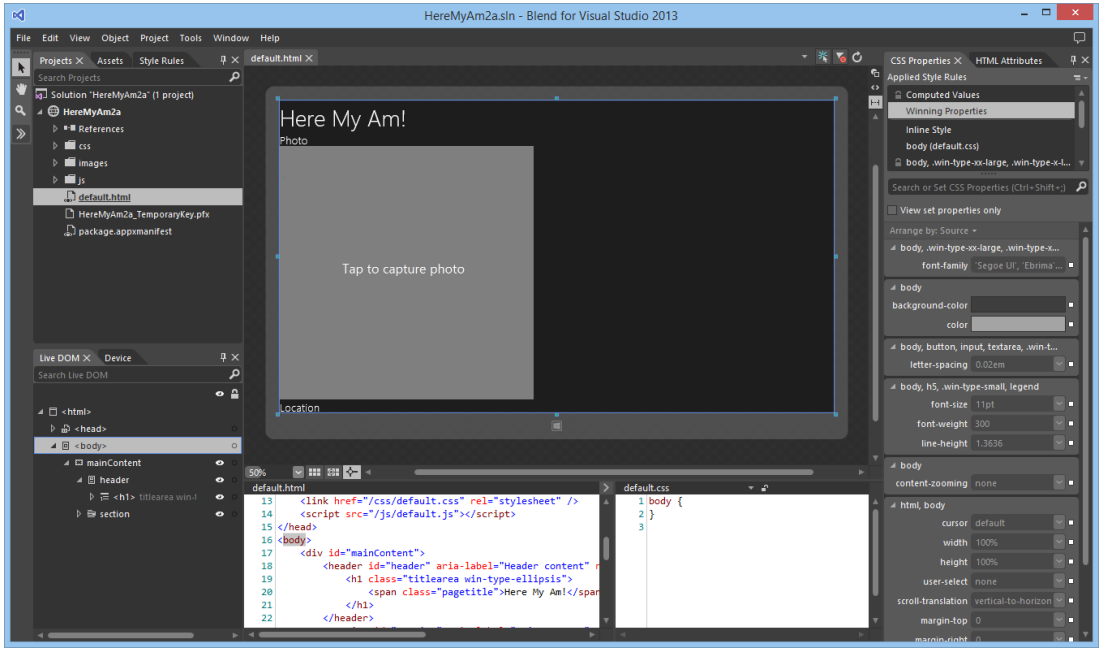


FIGURE 2-10 The app in Blend without styling, showing a view that is much like the Visual Studio simulator. If the taphere.png image doesn't show after adding it, use the View/Refresh menu command.

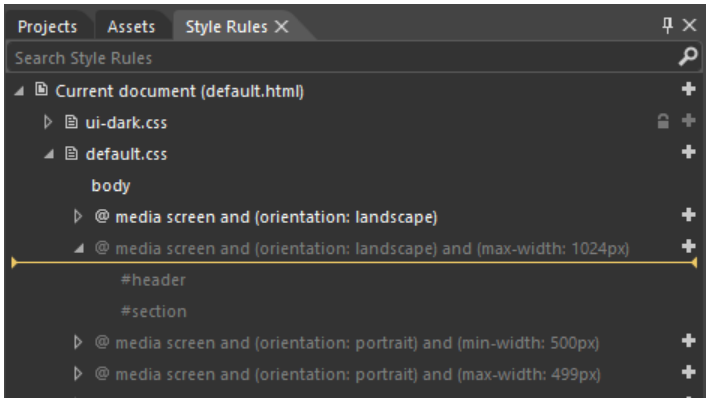
The tabs along the upper left give you access to your Project files, Assets like all the controls you can add to your UI, and a browser for all the Style Rules defined in the environment. On the lower left side, the Live DOM tab lets you browse your element hierarchy and the Device tab lets you set orientation, 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:



- 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.

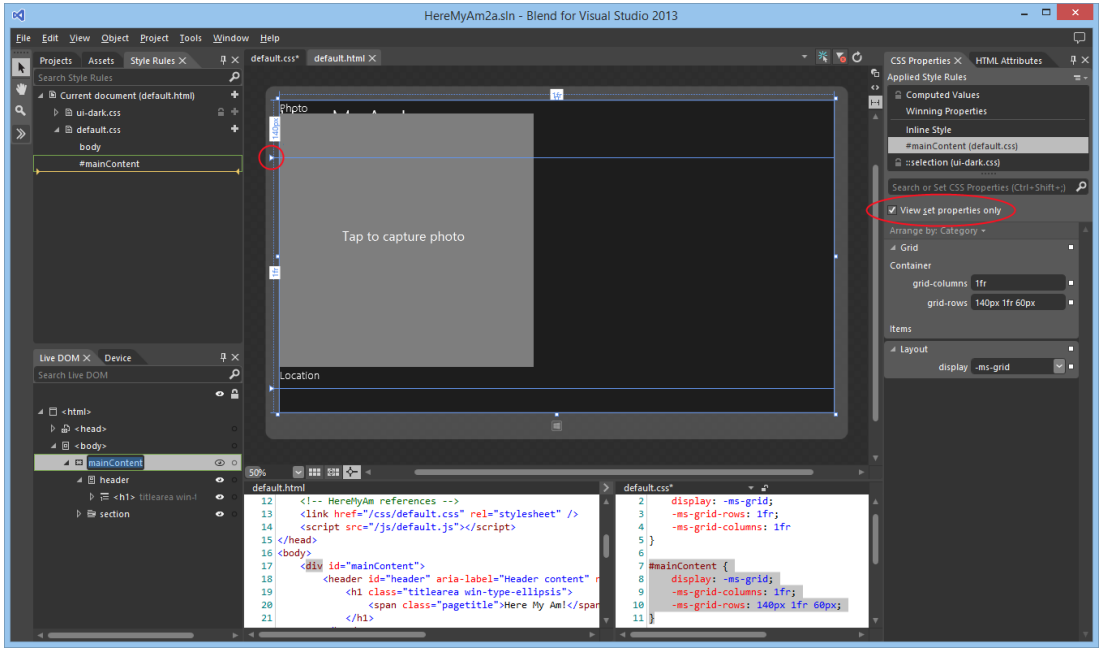


FIGURE 2-11 Setting the grid properties for the *mainContent* `div`. Notice how the View Set Properties Only checkbox (upper right) makes it easy to see what styles are set for the current rule. Also notice how the grid rows and columns appear on the artboard, including sliders (circled) to manipulate 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 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 and the //build 2013 talk linked above.)

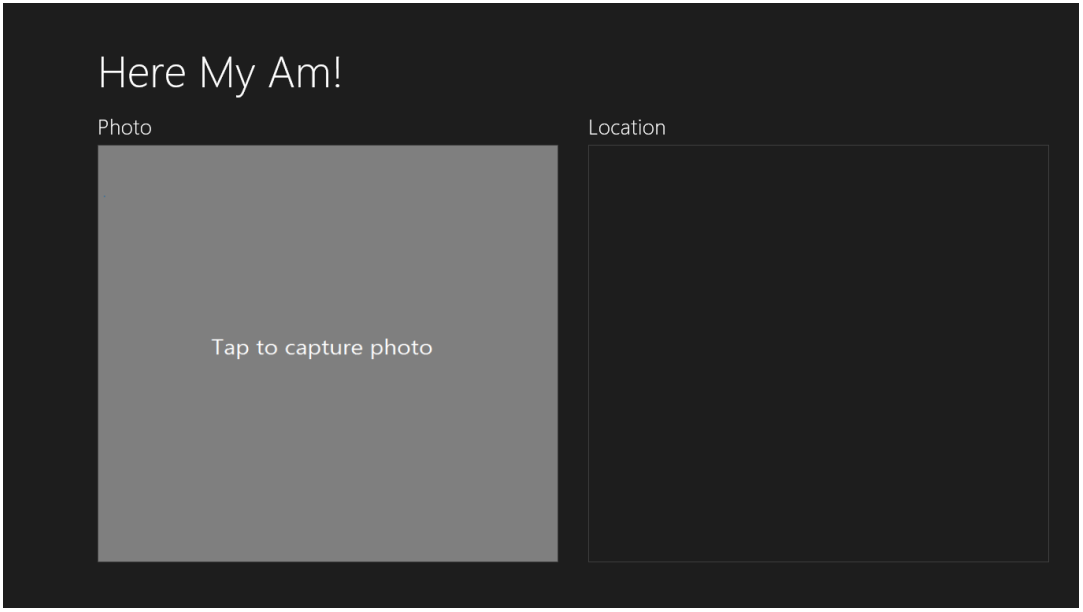


FIGURE 2-12 Full landscape view.

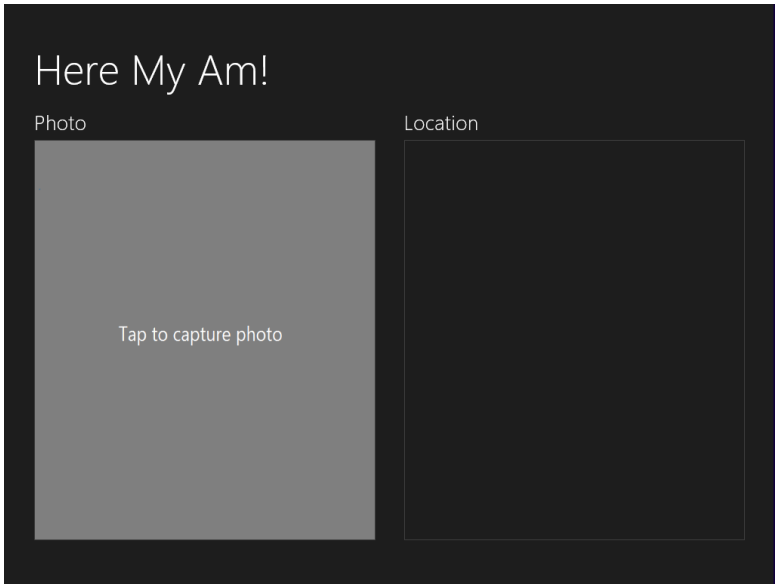


FIGURE 2-13 Partial landscape view (when sharing the screen with other apps).

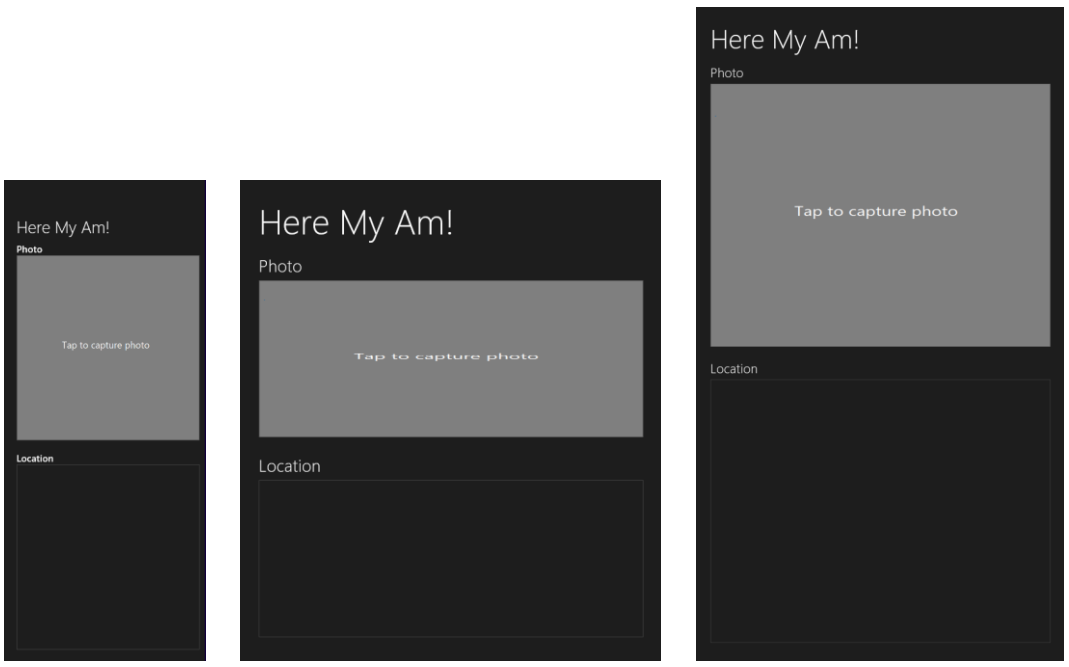


FIGURE 2-14 Narrow aspect ratio views: 320px wide (left), 50% wide (middle), and full portrait (right). These images are not to scale with one another. You can also see that the fixed placeholder image in the Photo section doesn't scale well to the 50% view; we'll solve this later in this chapter in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

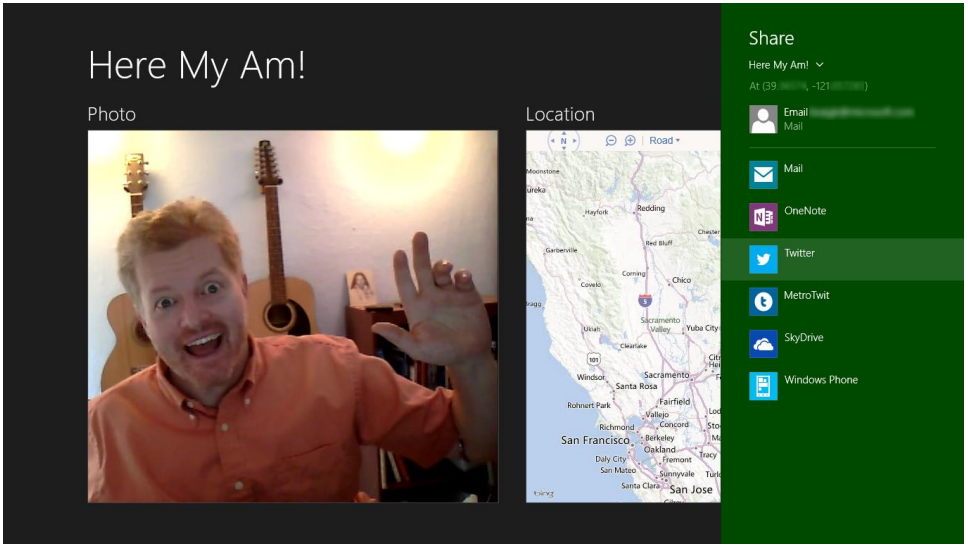


FIGURE 2-15 The completed Here My Am! app with the Share charm invoked (with my exact coordinates blurred out, because they do a pretty accurate job of pinpointing my house).

[14] 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.[15]

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`.

---

[15] 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.)

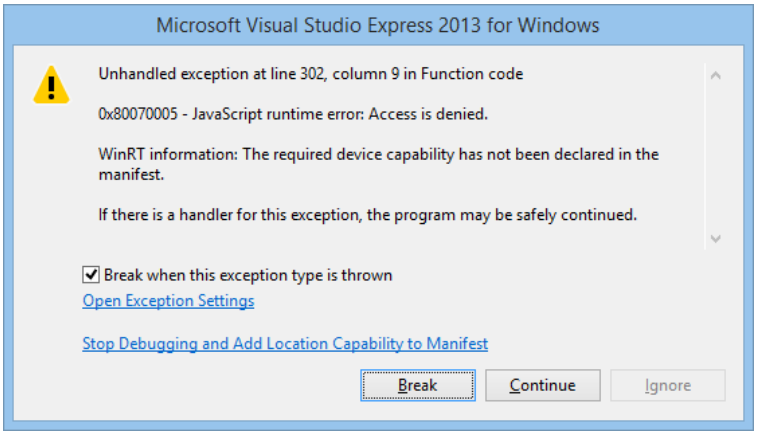
> **Async result types** When reading the docs for an async function, you'll see that the return type is listed like `IAsyncOperation<Geoposition>`; the name within `< >` indicates the actual data type of the results, so refer to the docs on that class for its details. Note also that the `IAsyncOperation` 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:



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.

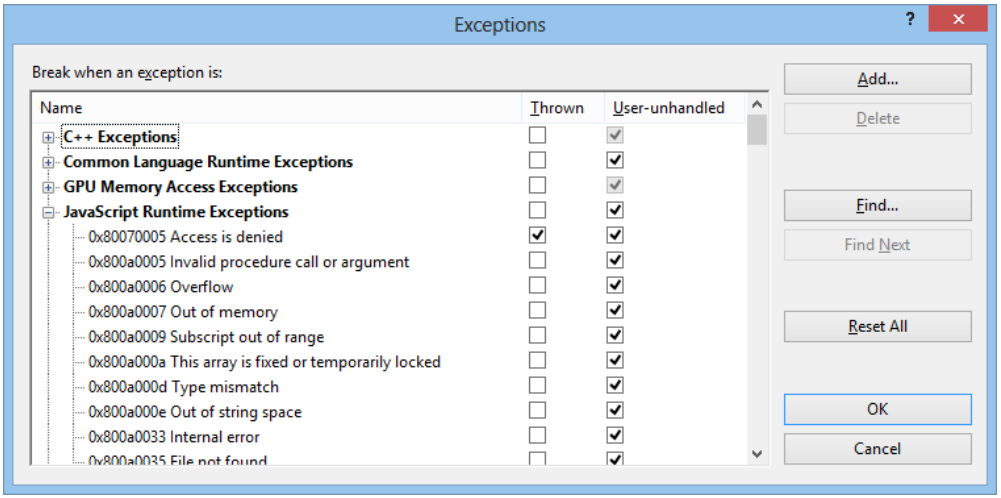


FIGURE 2-16 JavaScript run-time exceptions in the Debug/Exceptions dialog of Visual Studio.

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.

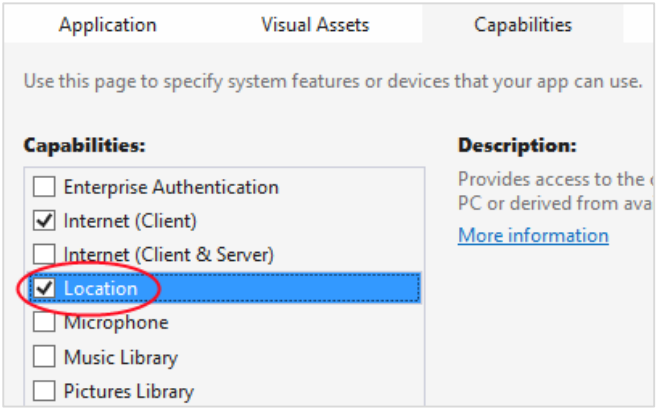


FIGURE 2-17 Setting the *Location* capability in Visual Studio's manifest editor. (Note that Blend supports editing the manifest only as XML.)

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:

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.

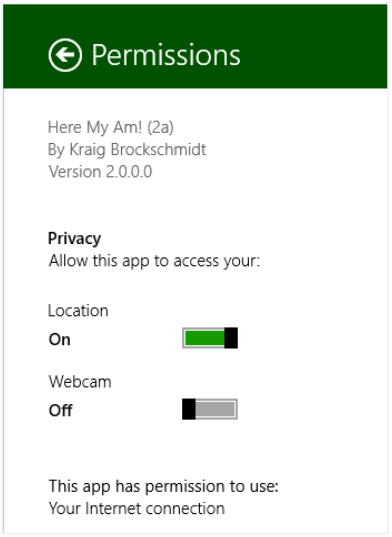


FIGURE 2-18 Any permissions that are subject to user consent can be changed at any time through the Settings Charm > Permissions pane.

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 if you like, as we'll see in Chapter 13, "Media.")

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.

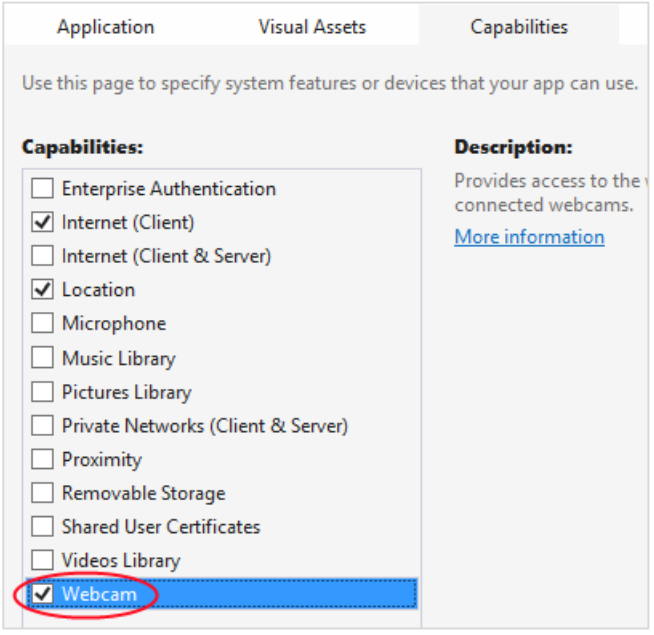


FIGURE 2-19 The camera capability in Visual Studio's manifest editor.

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):

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.)

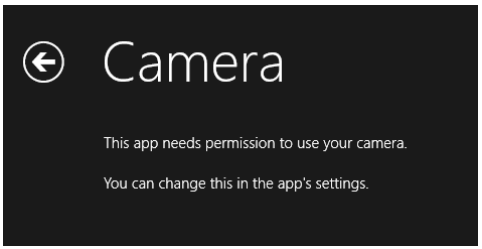
To invoke the camera UI, we only need create an instance of `Windows.Media.Capture.CameraCaptureUI` with `new` (a typical step to instantiate dynamic WinRT objects), configure it with the desired format and size (among many possibilities; see Chapter 13), and then call `captureFileAsync`. This will check 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:



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.[16] 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.

[16] 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!

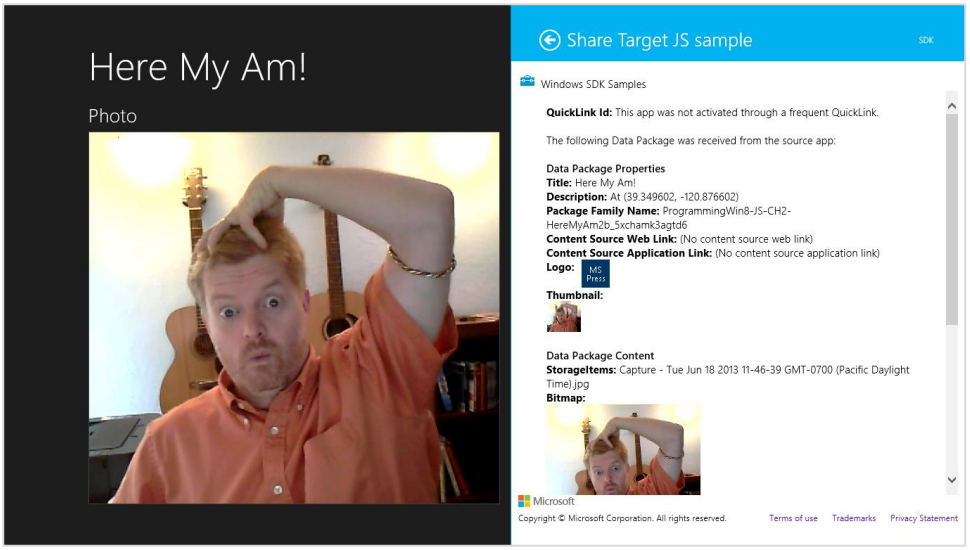


FIGURE 2-20 Sharing (monkey-see, monkey-do!) to the Share target sample in the Windows SDK, which is highly useful for debugging as it displays information about all the formats the source app has shared. (And if you still think I've given you coordinates to my house, the ones shown here will send you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

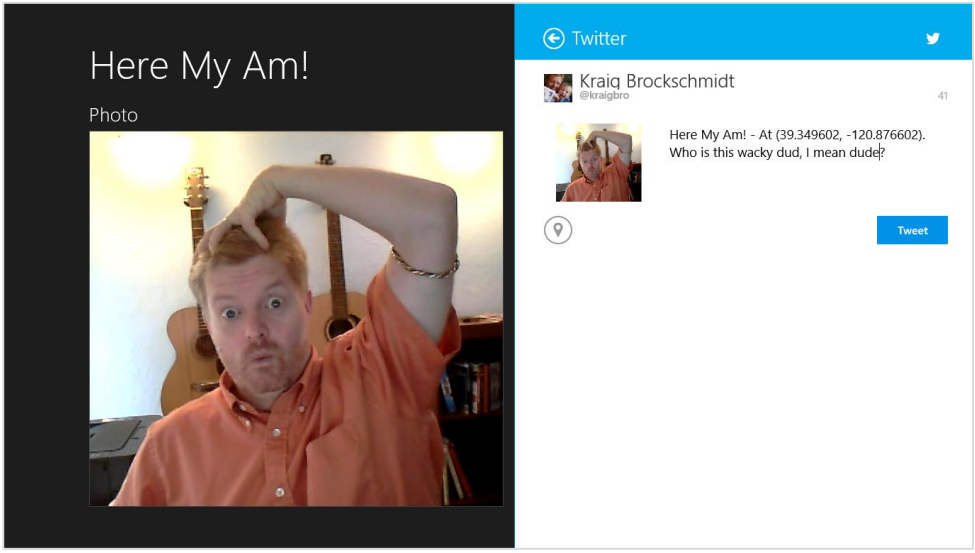


FIGURE 2-21 Sharing to Twitter. The fact that Twitter's brand color is nearly identical to the Windows SDK is sheer coincidence. The header color of the sharing pane always reflects the target app's specific color.

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.

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 one, 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).

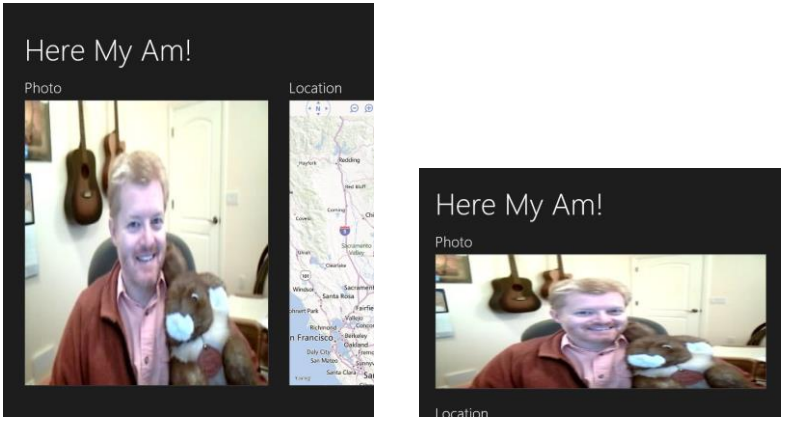


FIGURE 2-22 Poor image scaling with a low-resolution picture from the camera where the captured image isn't inherently large enough for the display area (left), and even worse results in the 50% view when the display area's aspect ratio changes significantly.

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.

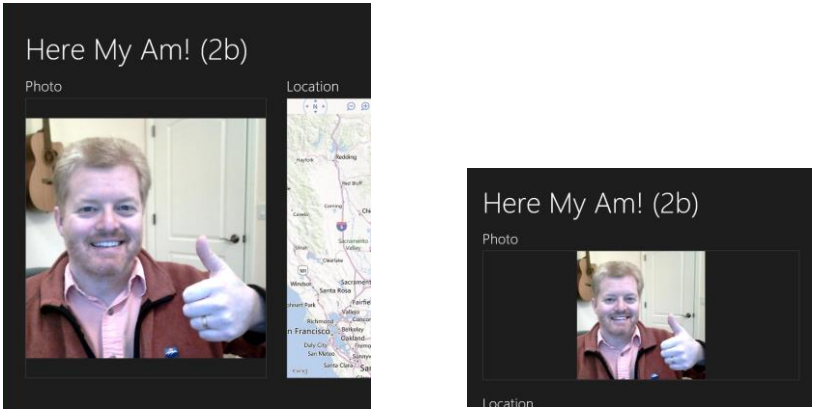


FIGURE 2-23 Proper image scaling after making the improvements.

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.getScaled-ImageAsThumbnailAsync`, 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 `getScaledImageAs-ThumbnailAsync` 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.

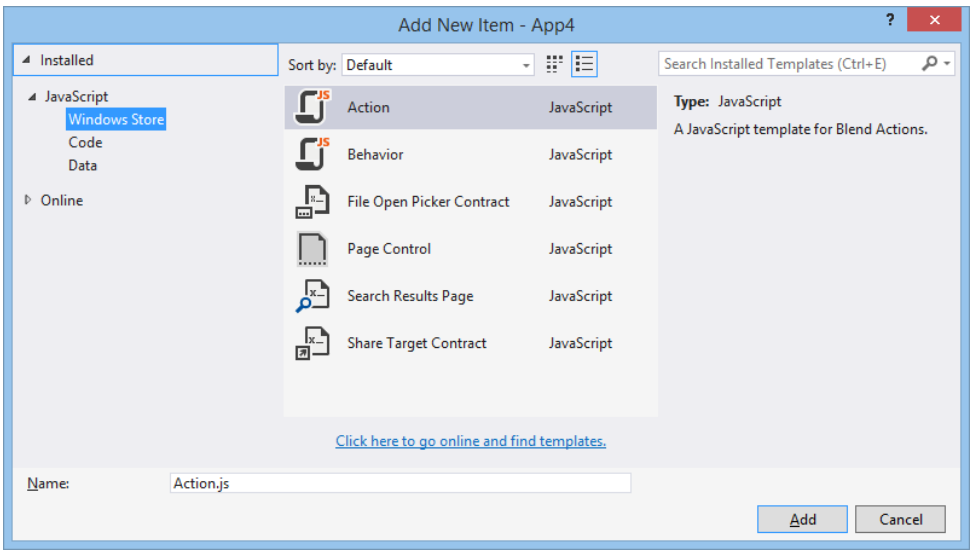


FIGURE 2-24 Available item templates for a Windows Store app written in JavaScript.

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. While I wasn't on site every day managing the whole 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 entirely 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 is something 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 must be 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, 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 app 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.[17]

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.

[17] 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.

Manifest Editor Tab	Text Item or Setting	Use
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	24x24	30x30	42x42	54x54

	exist for the app. We'll cover this in Chapter 15, "Contracts."				
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 runtime.	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.

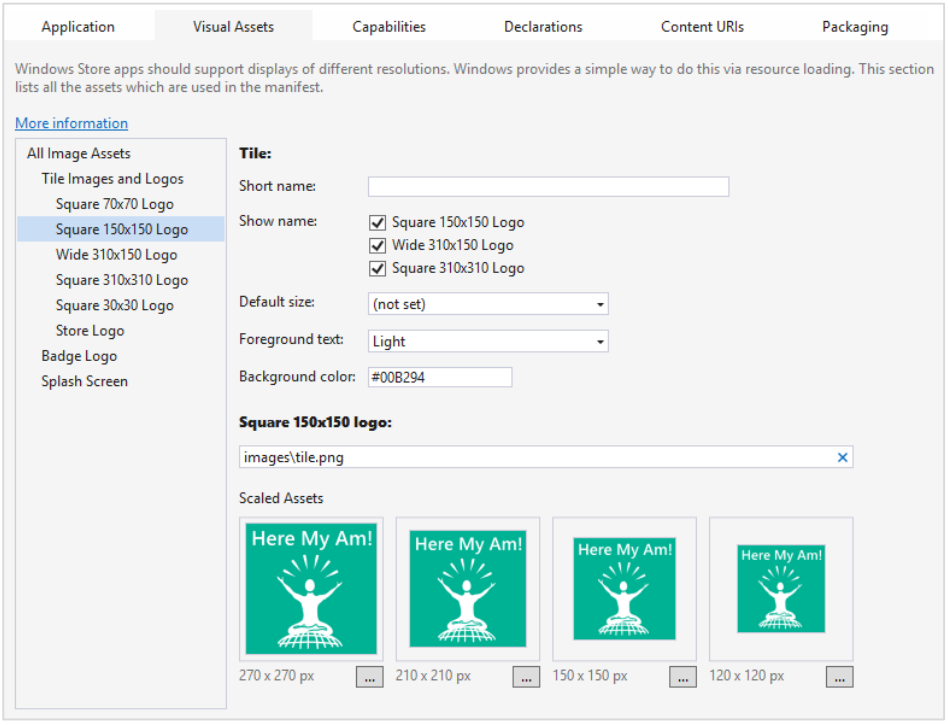


FIGURE 3-1 Visual Studio's Visual Assets tab of the manifest editor. It automatically detects whether a scaled asset exists for the base filename (such as images\tile.png).

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 also 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 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. Fortunately, the Windows Store (for Windows 8.1) transparently restructures your resources into separate packs that are downloaded only as a user needs them, provided that you structure your resources as we'll discuss in Chapter 19, "Apps for Everyone, Part 1." 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.[18]
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

[18] 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.

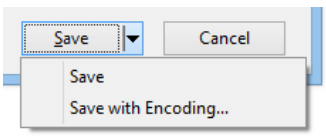
Once the `activated` event handler returns, the default splash screen is dismissed unless the app has requested a deferral, as discussed later in the "Activation Deferrals and setPromise" section.With the latter four events, your app's handling of these very much determines how quickly it comes up and 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."

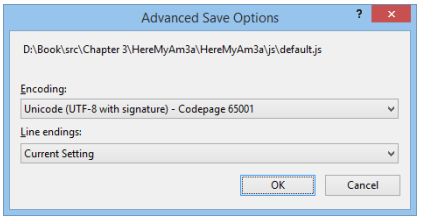
What's also different between an app and a website is that an app can again be activated for many different purposes, such as contracts and associations, even while it's already running. As we'll see in later chapters, the specific page that gets loaded (step 3) can vary by contract, and if a particular 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.





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.ApplicationMode.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.[19] 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

[19] 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."[20]

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

[20] 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 (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!)

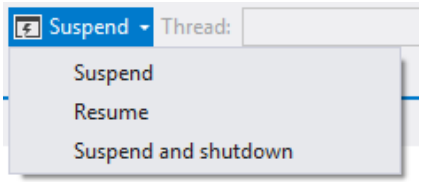


FIGURE 3-2 The Visual Studio toolbar drop-down to simulate suspend, resume, and terminate.

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.

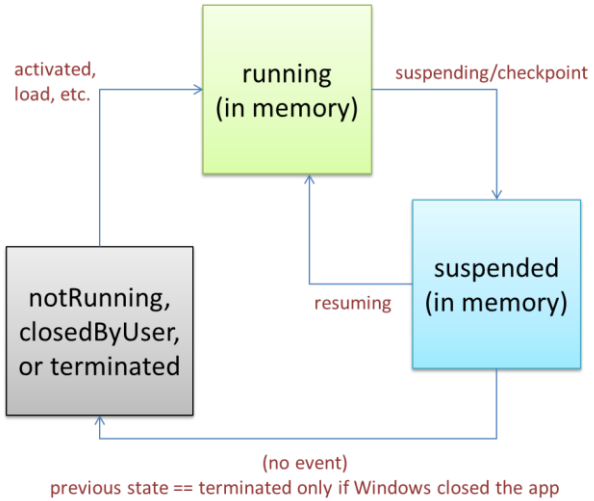


FIGURE 3-3 Process lifecycle events and `previousExecutionState` values.

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, provide 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.[21] 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`).[22]

> **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

---

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

[22] 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 ever see both pages on-screen together.

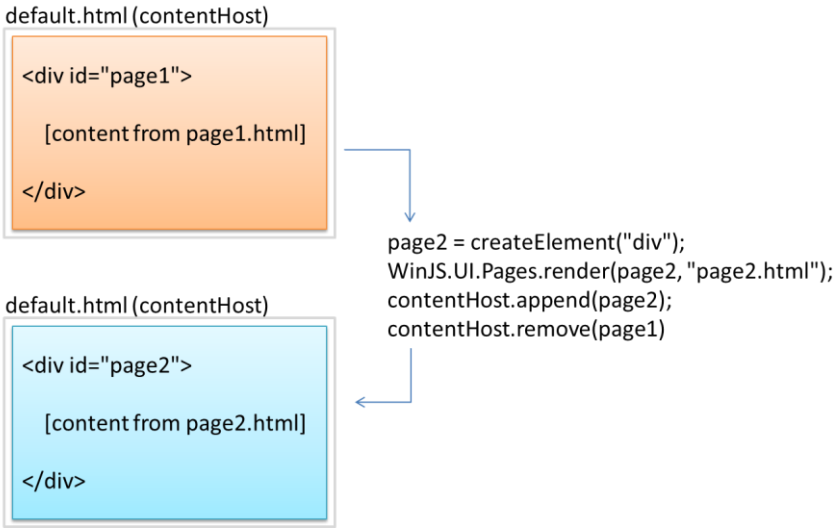


FIGURE 3-4 Performing page navigation in the context of a single host (typically default.html) by replacing appending the content from page2.html and removing that from page1.html. Typically, each page occupies the whole display area, but page controls can just as easily be used for smaller areas.

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.[23] 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 create 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.

---

[23] 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.

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

```
<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.
`ready`	Called after the page have been added to the DOM.
`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.[24]

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

[24] 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, 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.[25]

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");
```

[25] 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.

```
    } else {
        linkEl.setAttribute('href', "//Microsoft.WinJS.2.0 /css/ui-light.css");
    }
}
```

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.[26]

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

[26] 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.

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.

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.[27]

> **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.[28] 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.

---

[27] 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.

[28] 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.

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"!

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 on 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 that page, you'll want to change the priority 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.[29]

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, execute in the expected order:

```
Scheduling Jobs...
```

[29] The `Scheduler.execHigh` method is also a shortcut for directly calling `MSApp.execAtPriority` with `Priority.high`. This method does not accommodate any added arguments.

```
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
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 these 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.

Clearly, 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 you create through `Scheduler.createOwnerToken`, then assign 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 doesn't 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 job 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 methods that exists on the Scheduler is `retrieveState`, a diagnostic aid that returns a string that describes 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 we 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, we'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, we could encapsulate the process of setting a new priority within another promise that we can then insert into the chain. The best way to do this is to 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 that are generated by calling code like the above with different priority arguments. They are called `schedulePromiseHigh`, `schedulePromiseAboveNormal`, `schedulePromiseNormal`, `schedulePromiseBelowNormal`, or `schedulePromiseIdle`, each of which

also automatically cancels the scheduled job if there's an error (the code above does not).

As pre-made completed handlers, we use them simply by inserting the appropriate name at those points in the promise chain where we'd like 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);
});
```

Inside a Task

When you call `schedule` for a particular function, there's some additional information that's available inside that function.

First of all, the value of `this` will be whatever you pass as the third (optional) argument to `schedule`, with the default being the global context, of course.

Second, inside that task (and anywhere else, for that matter) you can call `Scheduler.currentPriority` to determine the priority at which your running. (Note: the present documentation suggests that you can set this value, but it's actually read-only.)

Third, the first argument to the task function will be an object with four methods and properties defined by `Scheduler.IJobInfo`. One property is `job`, the same object that was returned from the original call to `schedule`. Another member is the `setPromise` method whose purpose is straightforward. When a task is waiting on the completion of some async operation, it can clearly yield until that operation is complete. At that time it will then want to schedule the next step in its work (at the same priority). What you pass to the job info object's `setPromise`, then, is a promise that's fulfilled with that next task (that is, a function). This is easy enough to obtain with code like this, where *nextWorker* is the function to reschedule.

```
function firstWorker(jobInfo) {
    var nextWorkerPromise = someOperationAsync().then(function (results) {
        return nextWorker;
    });

    jobInfo.setPromise(nextWorkerPromise);
}
```

That is, the operation promise's then method returns another promise that's fulfilled with the return value of the completed handler, which is *nextWorker* in this case, which is exactly what we need.

The other two members of the job info object are a Boolean flag `shouldYield` and another method named `setWork`. These, along with `setPromise`, are meant to help long-running tasks—such as those at idle priority—break up their work in without having to manually call `setImmediate`, `setInterval`, or `setTimeout`.

Simply said, for jobs at `aboveNormal` priority and lower, the Scheduler already has a time interval in place much like you'd create manually. When it invokes a task function, it will set the job info object's `shouldYield` to `false` if the task can to a little work and then return. The task will continue to be called in this way so long as the Scheduler's interval hasn't passed or the task hasn't yielded through `setPromise`. When it's time to yield, however, the Scheduler will call the task with `shouldYield` set to `true`. In this case the task should call the job info object's `setWork` method with the function to reschedule (at the same priority). Typically this is the same function that is already executing, provided that it has some termination condition built in.

Scenario 4 of the HTML Scheduler sample shows this. When you press the Execute a Yielding Task button, it schedules a function called *worker* at `idle` priority (js/yieldingscenario.js):

```
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 + 2000)) {
                // do nothing;
            }
        }
    }

    if (taskCompleted) {
        window.output("Completed yielding task.");
        taskCompleted = false;
    }
}, S.Priority.idle);
```

Provided that the task is active (another button in the sample sets `taskCompleted` to `true`) If asked to yield, *worker* just calls `jobInfo.setWork` with itself. If `shouldYield` is false, then *worker* can do something (looping for a bit).

While this is going on, you can press the Add Higher Priority Tasks to Queue and see that those tasks are run before the next call to *worker*. In addition, you can poke around in the UI to see that the idle task is not blocking the UI thread.

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.

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 will send 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 directly 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! After all, Store apps should not provide their own close affordances, as described in requirement 3.6 of the Store certification policy.

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.[30] 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)

[30] 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.

and select Enable Log. This will trace any calls to `console.log` as well as other diagnostic information coming from the app host.

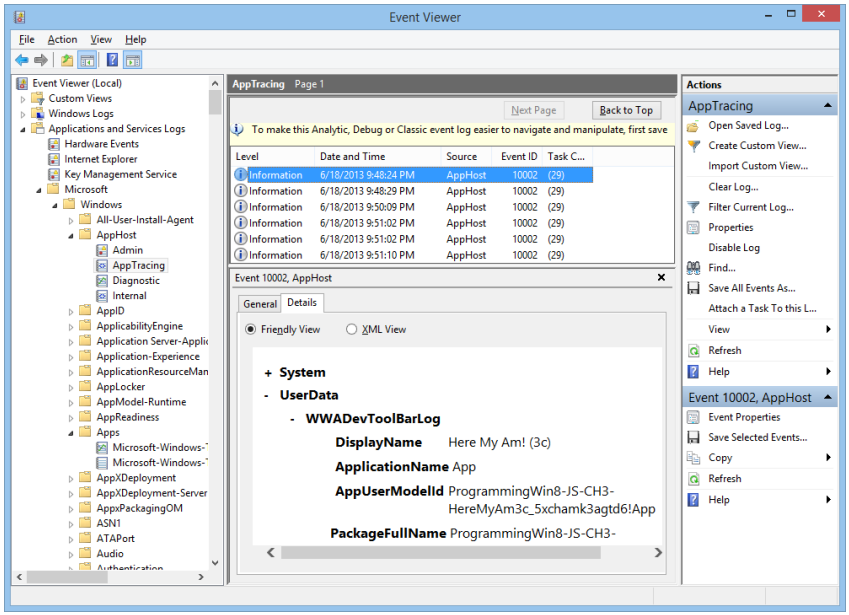


FIGURE 3-5 App host events, such as unhandled exceptions, load errors, and logging can be found in Event Viewer.

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-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.

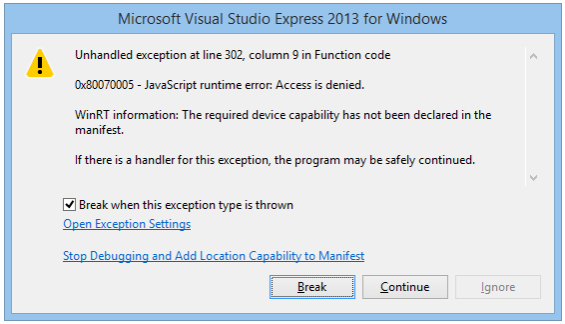


FIGURE 3-6 Visual Studio's exception dialog. As the dialog indicates, it's safe to press Continue if you have an error handler in the app; otherwise the app will terminate. Note that the checkbox in this dialog is a shortcut to toggle the Thrown checkbox for this exception type in the Exceptions dialog.

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.

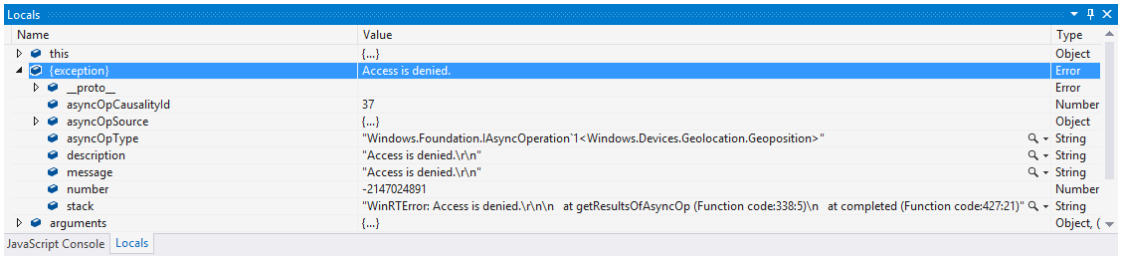


FIGURE 3-7 Information in Visual Studio's Locals pane when you break on an exception.

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 only for 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.

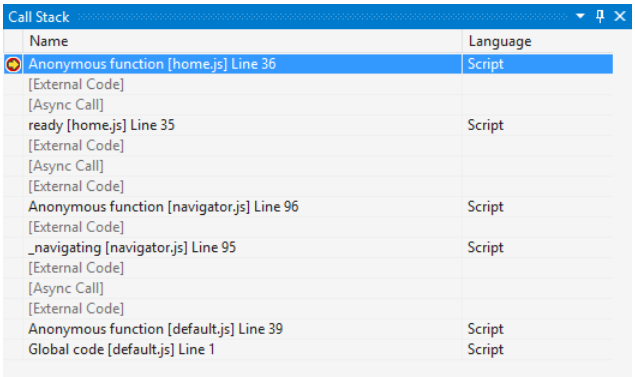
> **WinJS.validation** Speaking of exceptions, if you set `WinJS.validation` to `true` in your app, you'll instruct WinJS to perform a few extra checks on arguments and internal state, and throw exceptions if something is amiss. Just search on "validation" in the WinJS source files for where it's used.

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.

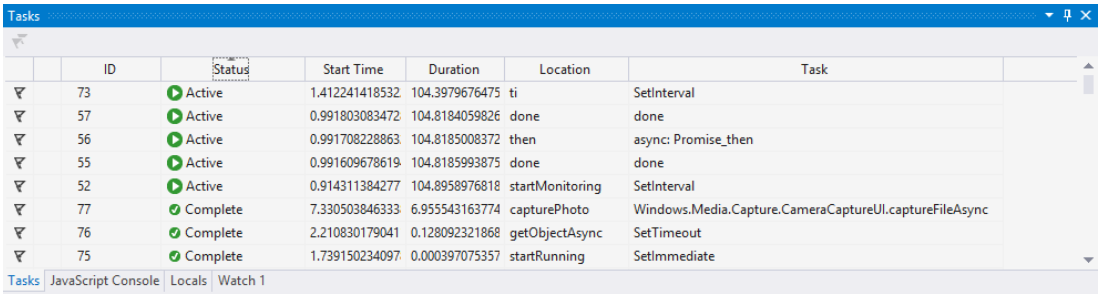
When you stop at a breakpoint, or when you hit an exception within an async process, take a look at the debugger's Call Stack pane (typically in the lower right of Visual Studio), as shown here:



Generally speaking, 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.



Performance and Memory Analysis

Alongside its excellent debugging tools, Visual Studio also offers additional tools to help evaluate the performance of an app, analyze its memory usage, and otherwise discover and diagnose problems that affect the user experience of an app and its 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.[31]

[31] DeMarco tells this amusing story as an example 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

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

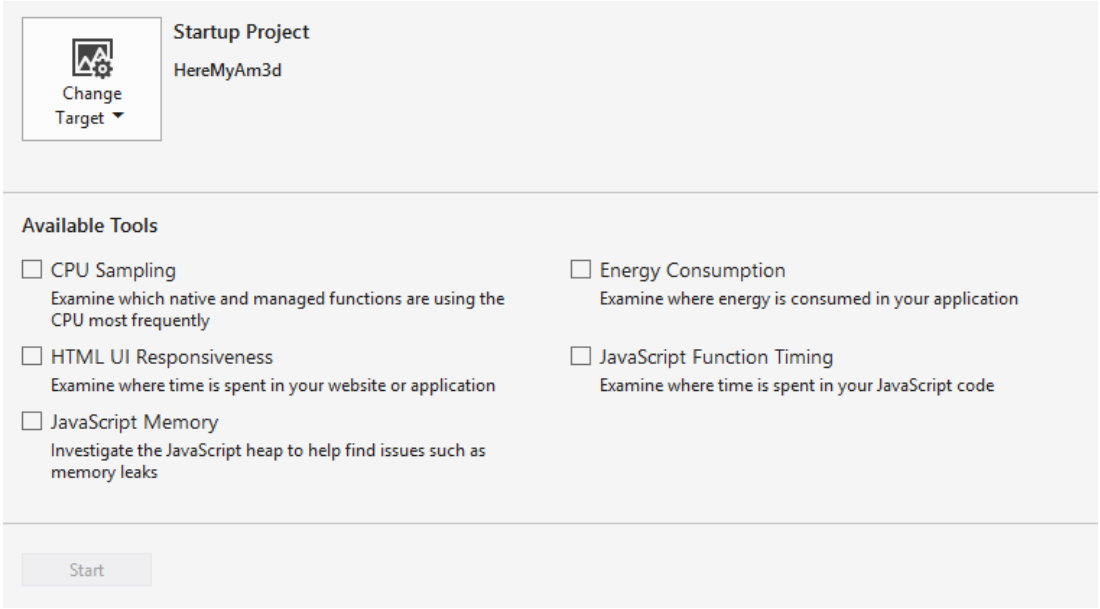
performance problems that you fixed a long time ago to rear their ugly heads once again. As the character Alistor Moody of the Harry Potter books says, "Constant vigilance!"

> **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.

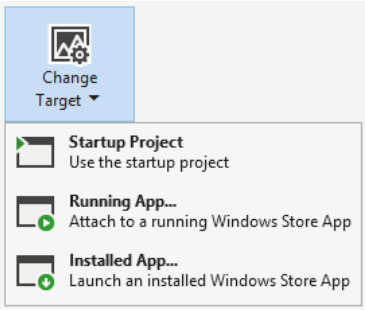
So, on to the tools. These are found on the Debug > Performance And Diagnostics… menu, which brings up the hub shown below:

---

the other."



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:



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 project and Visual Studio are 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.

It's worth noting 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 rendering becomes slow. 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, 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.

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.[32] 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 timeline on the scale of *seconds*, so use marks to indicate only significant user interaction events rather than every function entry and exit.

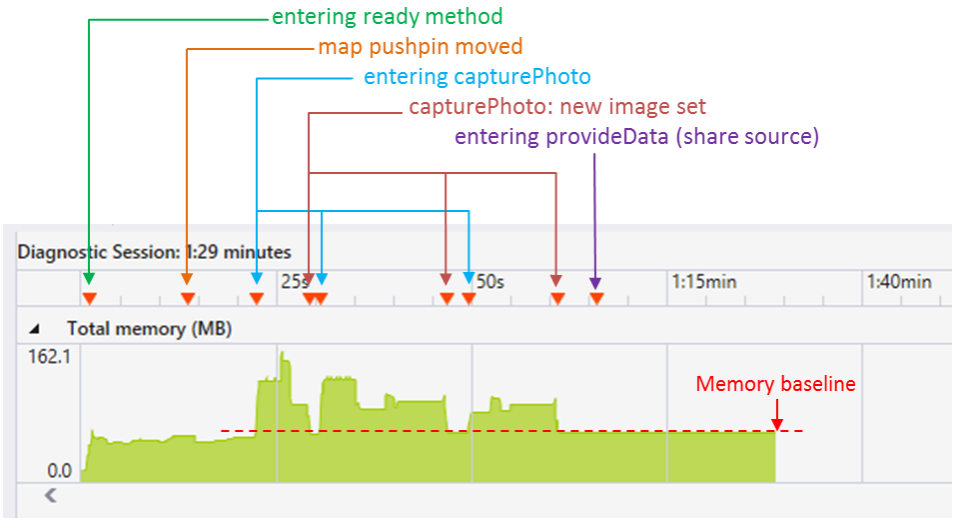


FIGURE 3-8 Output of the JavaScript Memory analyzer annotated with different marks. The red dashed line is also added in this figure to show the ongoing memory footprint; it is not part of the tool's output.

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 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

[32] 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`.

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 image when it get replaced. Finally I invoked the Share charm (last mark), and we can see that this caused 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 had 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. In other words, if this toolkit reports failures on your local machine, you can be certain that you'd 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 performance 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 9000 separate APIs, a number that continues to grow monthly. This means not only that there are over 9000 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.[33]

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

[33] 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 sockets, because that's a lower-level mechanism that has more context in device-to-device communication. We'll come back to that in Chapter 17, "Devices and Printing." Similarly, setting up service connections for live tiles and push notifications 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, 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 once 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. Window Store policy, in fact, requires that apps are 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 network, as in a home or business, 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.

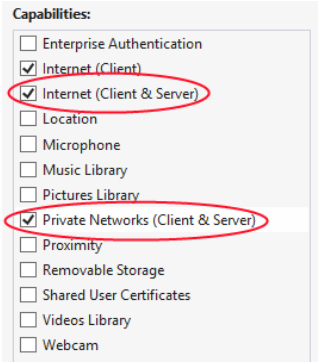


FIGURE 4-1 Additional network capabilities in the manifest.

Sidebar: Localhost Loopback

Regardless of the capabilities declared in the manifest, local loopback—that is, using http://localhost URIs—is blocked for Windows Store apps. An exception is made 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). This 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 `getConnectivityIntervalsAsync`, 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.

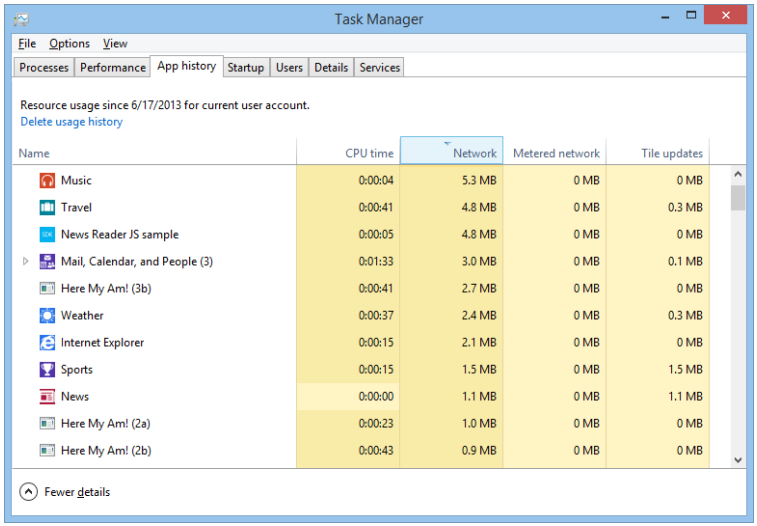
Cost Awareness

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:



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. Windows Store policy, for its part (section 4.5), requires 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`. I like to think of that policy as a minimum requirement—your customers will clearly appreciate more 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.

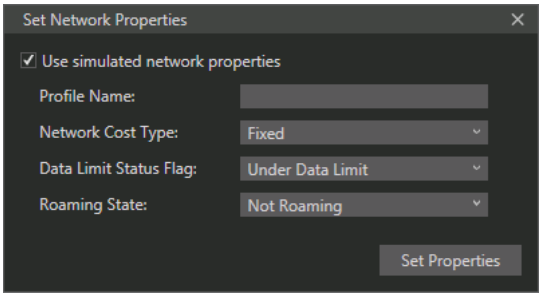
On a `fixed` type network, where data is unrestricted up to `dataPlanLimitInMegabytes`, we find cases where a number of the other properties become interesting. For example, if `overDataLimit` is already `true`, you can ask the user to confirm additional network traffic or just defer certain operations 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.dataPlan-LimitInMegabytes`. 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

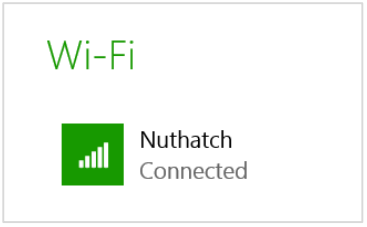
You may be thinking, "OK, so I get the need for my app to behave properly with metered networks, but how do I test such conditions?" You can, of course, use a real metered network through a mobile provider, such as the Internet Sharing feature on my phone. However, I do have a data limit and I certainly 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.

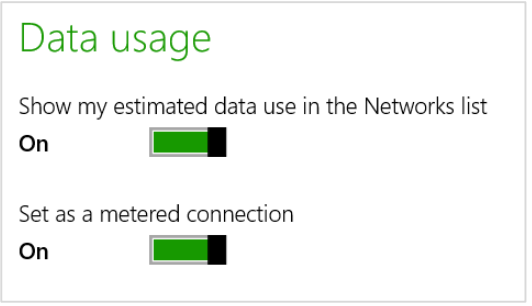
In the simulator, click the Change Network Properties button on the lower right side of the simulator's frame (it's the command above Help—refer back to Figure 2-5 in Chapter 2, "Quickstart"). This brings up the following dialog:



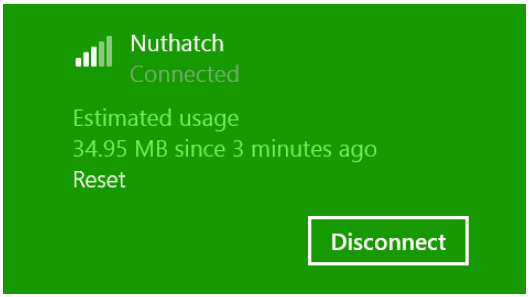
In this dialog you can create a profile with whatever name and options you'd like. The variations for cost type, data limit status, and roaming allow you to test all conditions that your app might encounter. As such, this is your first choice for working with cost awareness.

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):





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:



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 30,000 feet. 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 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 two caveats with the webview is that it does not at present support IndexedDB or 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.

Caution `iframe` and `x-ms-webview` elements are *not* intended to let you just build an app out of remote web pages. Section 2 of the Windows Store app certification requirements, in fact, specifically disallows this: "the primary app experience must take place within the app," meaning that it doesn't happen within hosted websites. A few key reasons for this are that many websites aren't set up well for touch interaction (which violates requirement 3.5) and often won't work well in different views (violating requirement 3.6). You also want to make sure that the user can navigate easily without getting lost in random websites and that the app still follows the design guidelines, uses the app bar, supports contracts like Share, and so on.

In short, overuse of web content will likely mean that the app won't be accepted by the Store, though web content that's specifically designed for use with an app and behaves like native app content won't be so scrutinized.

Requirement 3.9 also 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, as downloading data to configure a game level, for instance, doesn't quite fall into this category. Nevertheless, this requirement is taken seriously so be very careful about making assumptions here.

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.ApplicationMode.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 again of Windows Store certification requirement 3.9 that specifically disallows dynamic script that changes your app logic or its interaction with WinRT.

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, 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.[34]

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.)

[34] At whatever point the webview supports IndexedDB or AppCache, these features will likely require such permissions as well.

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. 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.[35] 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 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 already 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.

---

[35] 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.

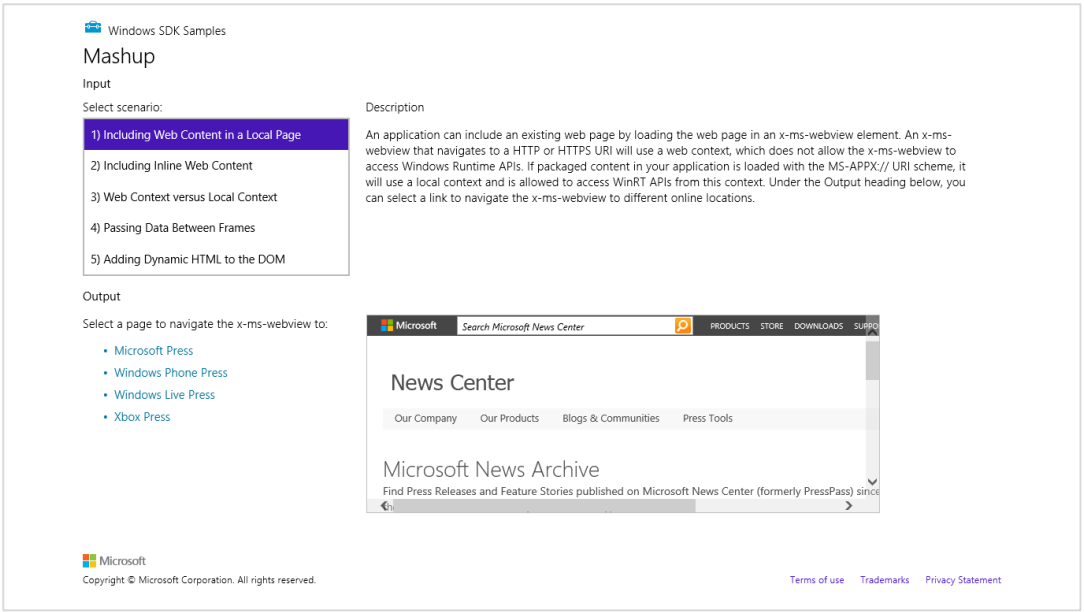


FIGURE 4-2 Displaying a webview, which is an HTML element like any others within an app layout. The webview runs within the web context and allows navigation within its own content.

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_ProgrammingWin8.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:

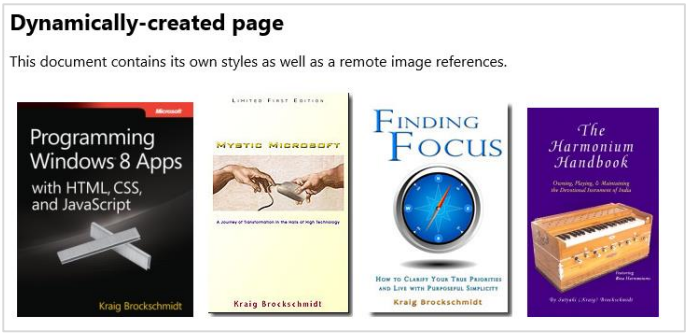
```
var webview = document.getElementById("webview");
webview.navigateToString(content);
```

We could just as easily write this text to a file in our appdata and use `navigate` with an `ms-appdata` URI (also 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:



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 file. 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 perhaps 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_4e61766967617465546f53747265616d/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 `MSWebViewUnviewableContent-Identified` 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 `MSWebViewFrame-NavigationCompleted`, 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.

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 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, as 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) { /* console output */ };
op.onerror = function (e) { /* console output */ };

//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 that you must be sure to 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`):

```
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.

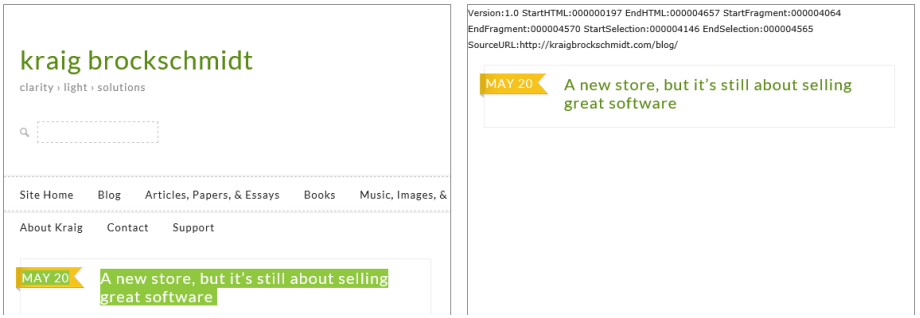


FIGURE 4-3 Example output from the WebviewExtras example, showing that the captured selection from a webview includes information about the selection as well as the HTML itself.

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.

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.

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.[36]

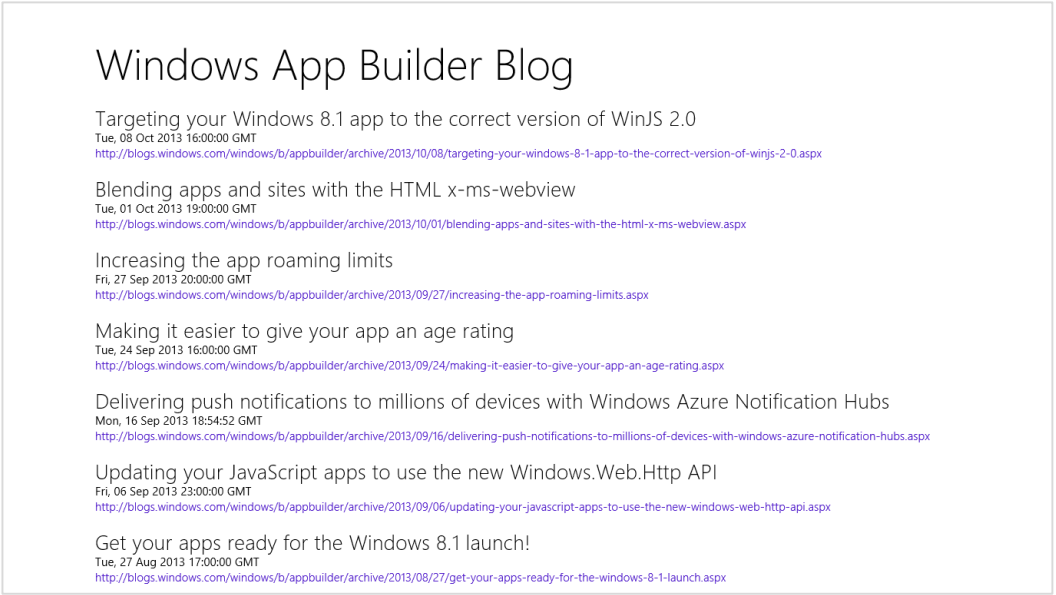


FIGURE 4-4 The output of the SimpleXhr1 and SimpleXhr2 apps.

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);
```

[36] 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.

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.

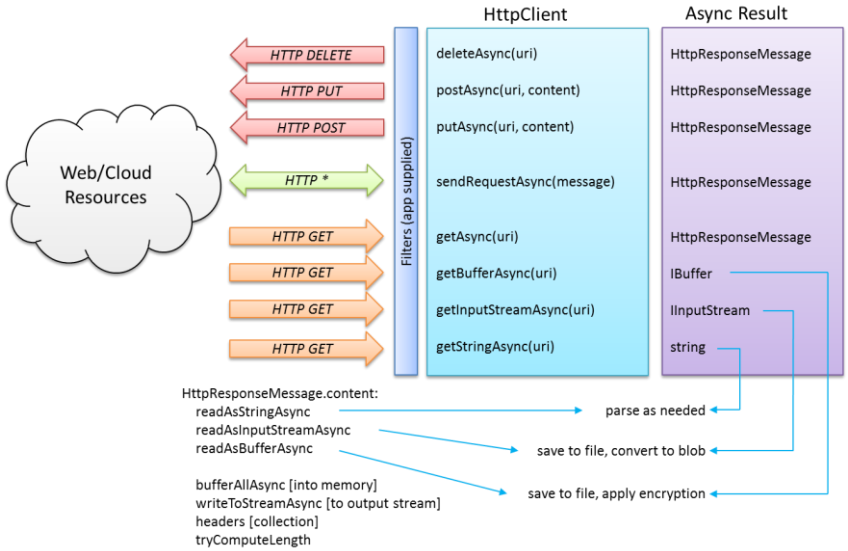


FIGURE 4-5 The methods in the `HttpClient` object and their associated HTTP traffic. Note how all traffic is routed through an app-supplied filter (or a default), which allows fine-grained control on a level underneath the API.

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`,[37] and other arbitrary `properties` that you want to communicate to the filter and possibly the server. The completed handler for `getAsync` will receive an `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`).

[37] 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`.

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 a 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 `HttpBaseProtocol-Filter` 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:

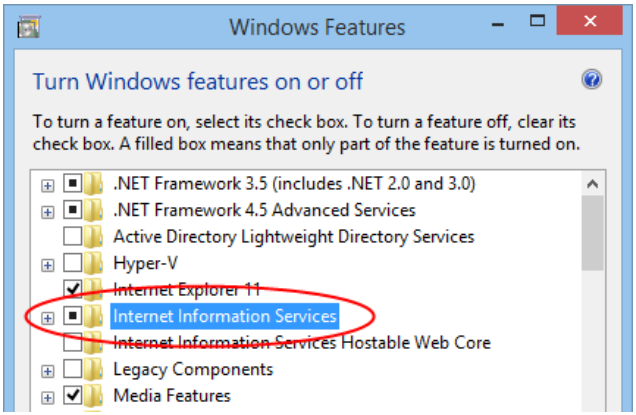
- **Scenarios 1–3** GET requests for text (with cache control), stream, and an XML list.
- **Scenarios 4–7** POST requests for text, stream, multipart MIME form, and a stream with progress.
- **Scenarios 8–10** Getting, setting, and deleting cookies.
- **Scenario 11** A metered connection filter that 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:



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.[38]

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.

[38] 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.

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 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.[39]

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.

[39] For readers familiar with .NET languages, note that the .NET `System.Net.HttpClient` API does *not* benefit from the cache or precaching.

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-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 by default 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[40] 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.

[40] Might I suggest http://kraigbrockschmidt.com/images/photos/kraigbrockschmidt-dot-com-122-10-S.jpg?

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 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:[41]

```
// 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);
```

[41] 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.

```
    // 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`).

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:[42]

[42] As with downloads, the code in the sample has more structure than shown here and again defines its own

```
// 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`, `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."

`UploadOperation` class with the same name as the one in WinRT, so I'm omitting mention of it.

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.[43]

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

[43] The optional `group` argument for the other methods is obsolete and replaced with these that work with a `transferGroup` argument.

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:

```
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 and checklist 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!

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 might 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 > SkyDrive > Sync Settings > Other Settings > Passwords). This help to create a seamless experience with your app as the user moves between devices.[44]

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

[44] 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.

user/multiple resources (scenario 2), multiple users/multiple resources (scenario 3), and clearing out the locker entirely (scenario 4).

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.

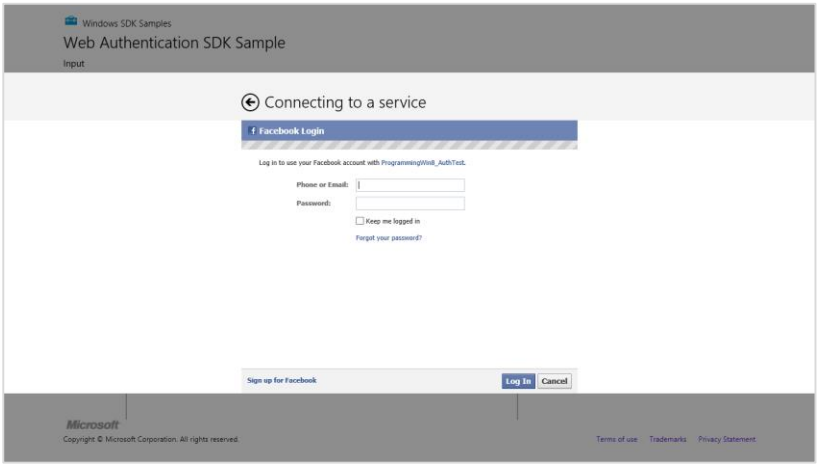


FIGURE 4-6 The Web authentication broker sample using a Facebook login page.

> **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.

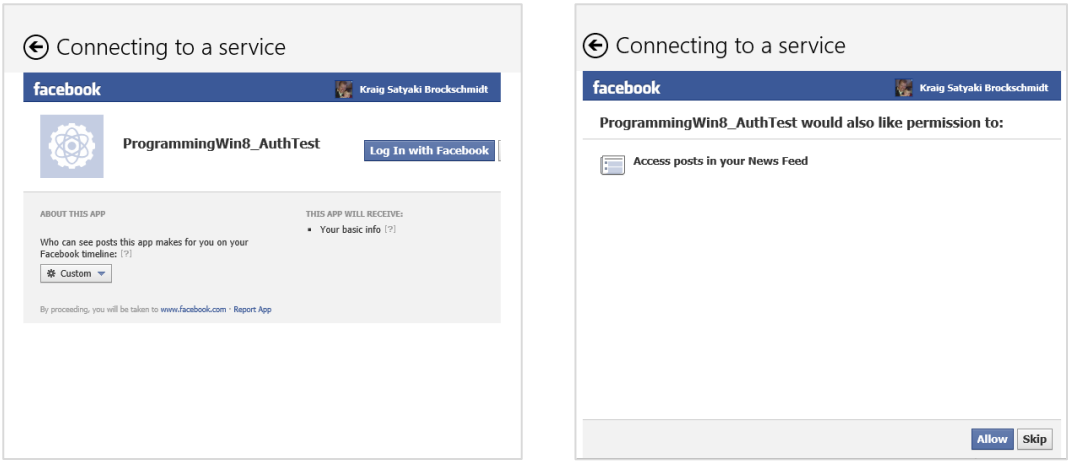


FIGURE 4-7 Additional authentication steps for Facebook within the web authentication broker.

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.

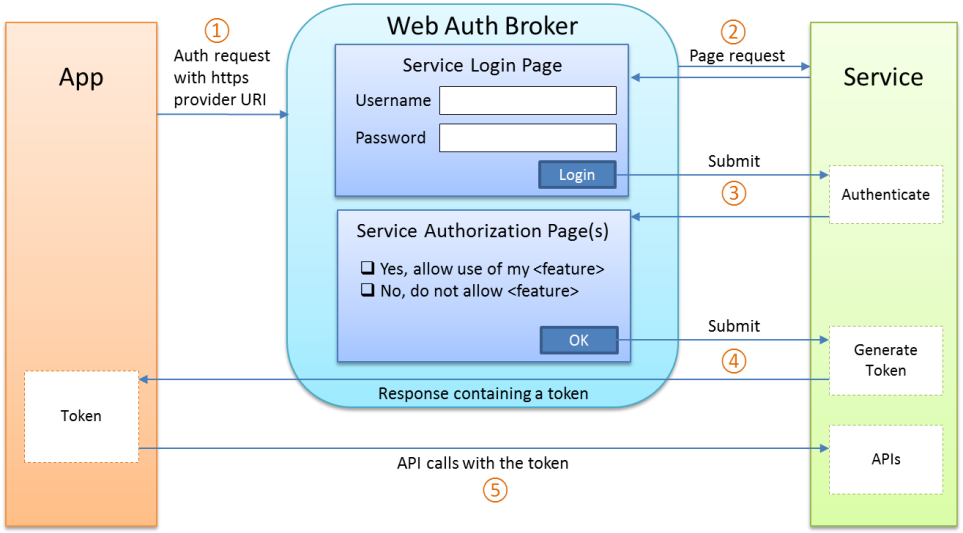


FIGURE 4-8 The authentication flow with the web authentication broker.

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.[45]

[45] As described on How the web authentication broker works, `requestUri` and `callbackUri` "correspond to an

The results given to the completed handler for `authenticateAsync` is a `WebAuthentication-Result` 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):

Authorization Endpoint URI and Redirection URI in the OAuth 2.0 protocol. The OpenID protocol and earlier versions of OAuth have similar concepts."

```
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.

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:

1. 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.
2. 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.
3. The app calls `authenticateAsync` with the `silentMode` option.
4. 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.
5. 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.
6. 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, SkyDrive, 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.[46] 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 SkyDrive accounts) and can also work with provider like Windows Azure Active Directory and SkyDrive 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.)

---

[46] 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 SkyDrive 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).

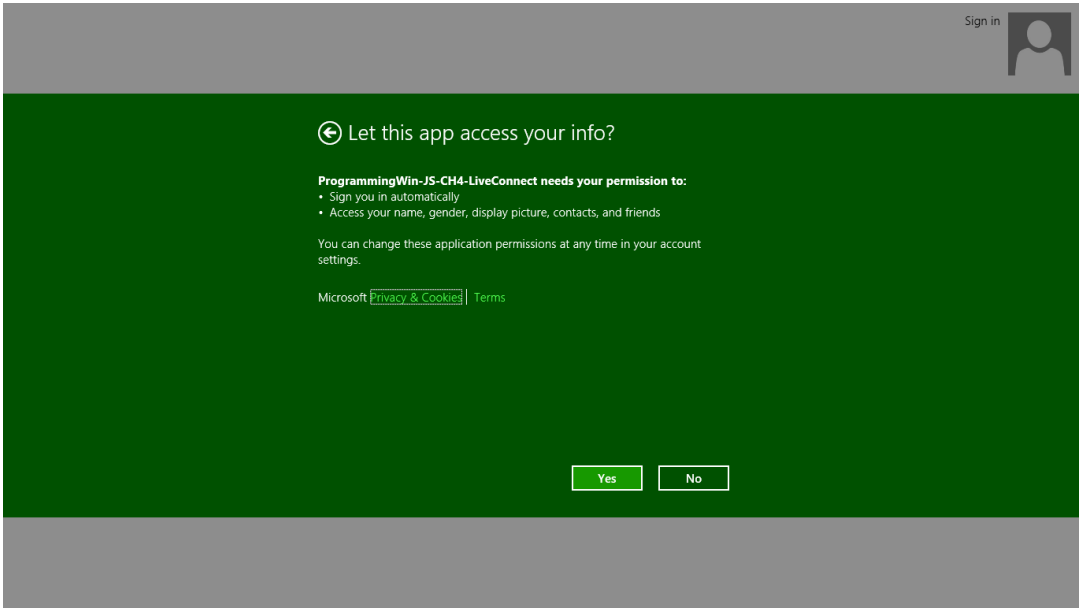


FIGURE 4-9 The Live Connect consent dialog that appears when you first attempt to log in.

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.

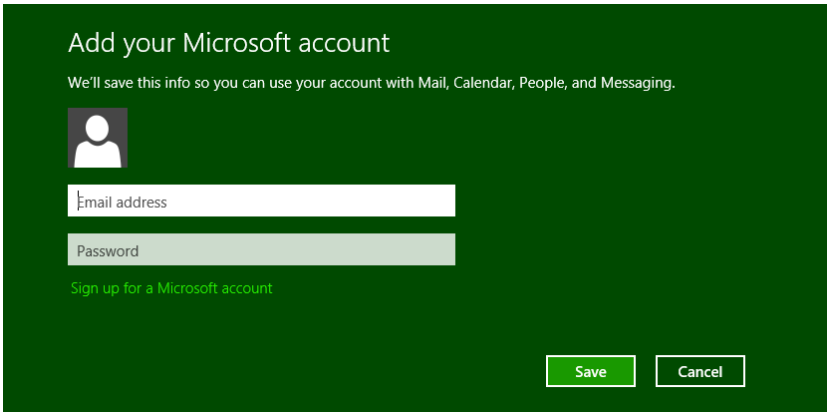
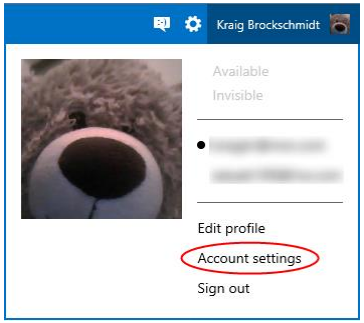
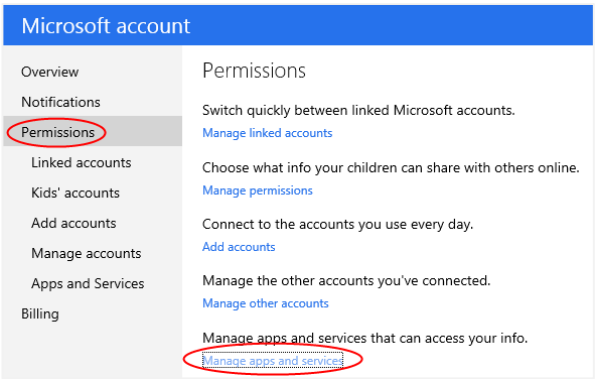


FIGURE 4-10 The Microsoft account login dialog if the user logged in to Windows with a domain account.

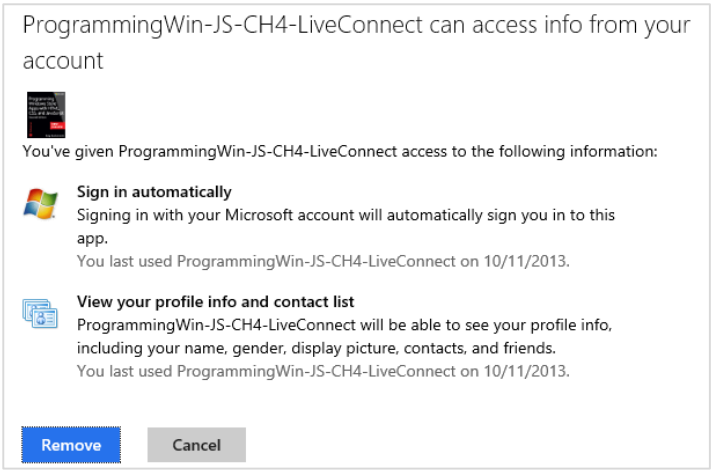
Once you've consented to any request from an app, those permissions can be managed through the Microsoft Account portal, https://account.live.com. You can also get there from http://www.live.com by clicking your name on the upper right. This will pop up some options (as shown below), where Account Settings 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:



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:



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.

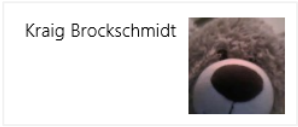
Anyway, a successful login will also trigger `sessionChange` events as well as the `login` event. In the LiveConnect example, the `login` handler (a function called `onLoginComplete`) retrieves the user's name and profile picture by using the Live API as follows (js/default.js, code condensed and error 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:

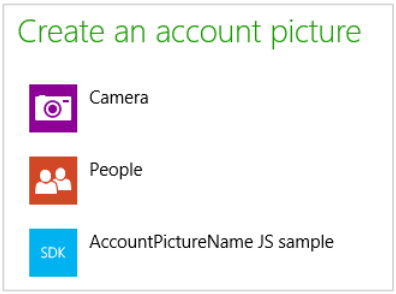
- **User name** If the `nameAccessAllowed` property is `true`, an app can then call `getDisplayNameAsync`, `getFirstNameAsync`, and `getLastNameAsync`, all of which provide a string to your completed handler. If `nameAccessAllowed` is false, these methods will complete but provide an empty result. Also 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 Accounts, under Create an Account Picture:



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, "Contracts"), 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 SkyDrive, 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.

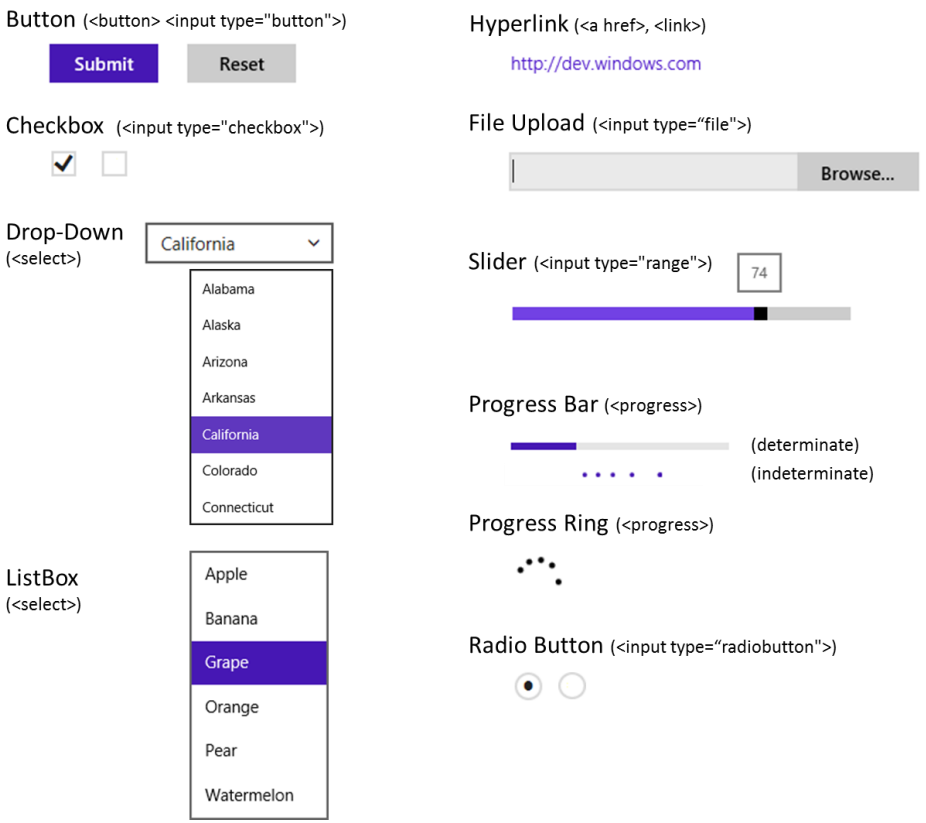


FIGURE 5-1 Standard HTML5 controls with default "light" styles (the ui-light.css stylesheet of WinJS).

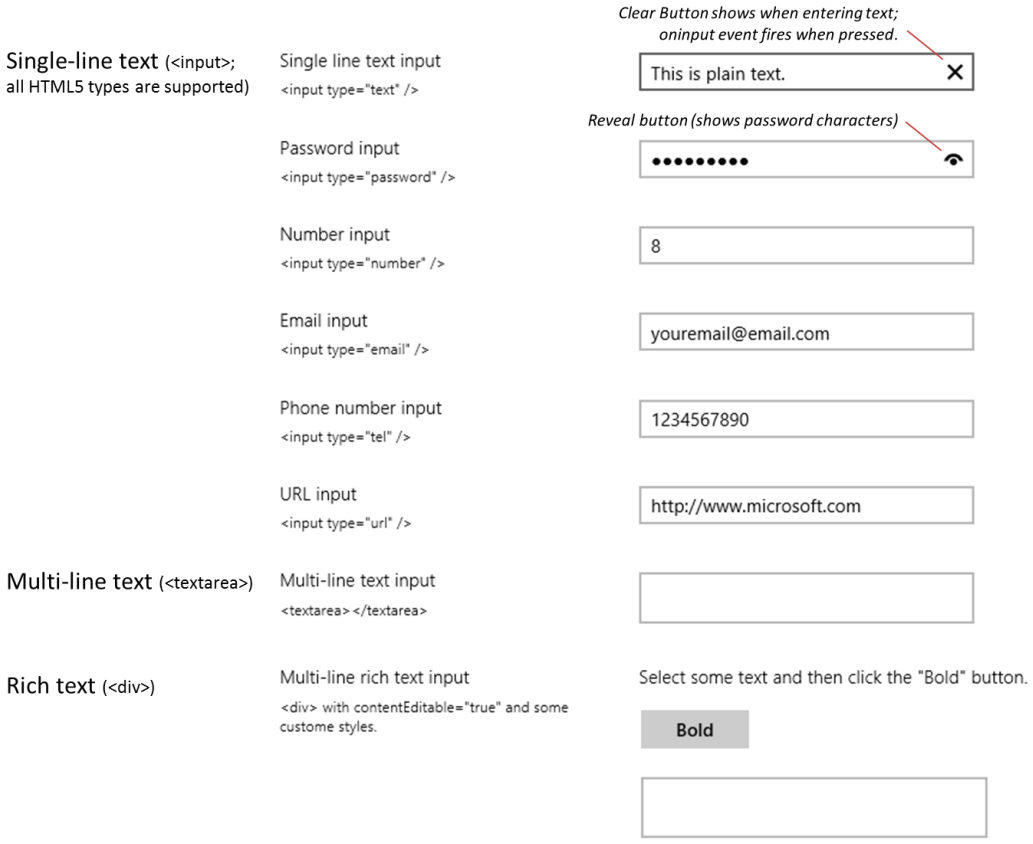


FIGURE 5-2 Standard HTML5 text input controls with default "light" styles (the ui-light.css stylesheet of WinJS).

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		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!

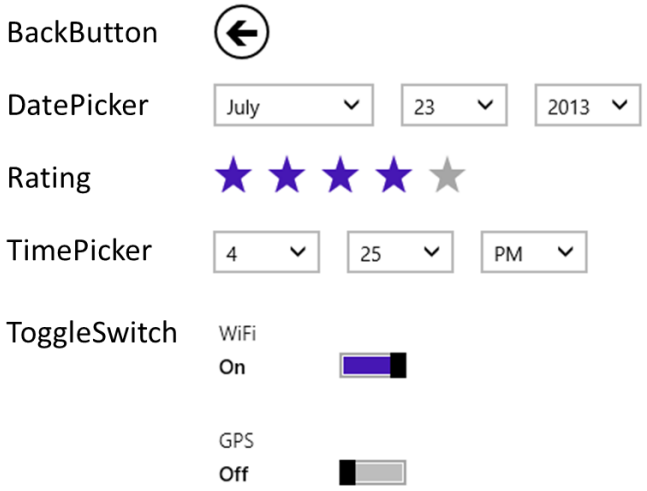


FIGURE 5-3 Default (light) styles for the simple WinJS controls.

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`.[47]

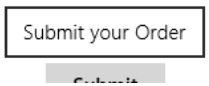
The `HtmlControl` is a simpler form of the page controls we saw in Chapter 3, allowing you to easily load some HTML fragment from your app package (along with its referenced CSS and JavaScript) as a

[47] 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.

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):



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:

7. Parse the `data-win-options` string into an options object, throwing exceptions if parsing fails.
8. Extract the constructor specified in `data-win-control` and call `new` on that function passing the root element and the options object.
9. The constructor creates whatever child elements it needs within the root element.
10. 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>
```

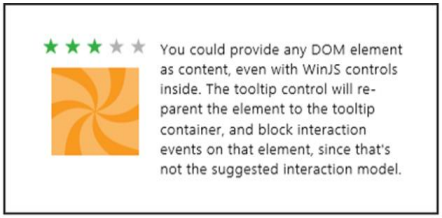
Second, the `contentElement` property of the tooltip control can name another element altogether, which will be displayed when the tooltip is invoked. For example, consider this piece of hidden HTML in our markup that contains other controls:

```
<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:



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 (also available with the companion content):

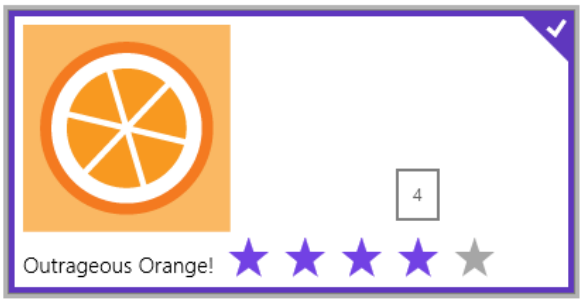
In all these cases the default `swipeOrientation` is `vertical`; the other option in `WinJS.UI.Orientation` is `horizontal`, which is demonstrated in scenario 3 of the sample.

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>
```

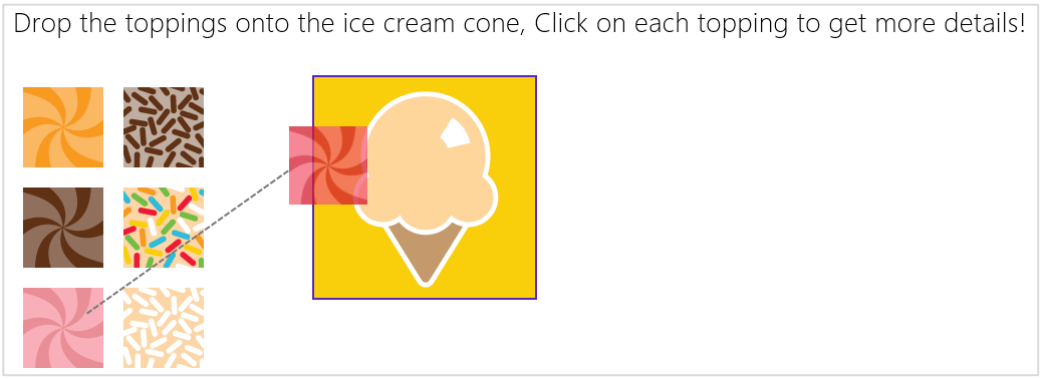
Take special note of the `win-interactive` class given to the `Rating` control—this is what tells the parent element (the `ItemContainer`) to pass interaction events down to the child control. This allows you to directly change the rating, where the rating's tooltip appears as it should for mouse hover:



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:



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 additional 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.)

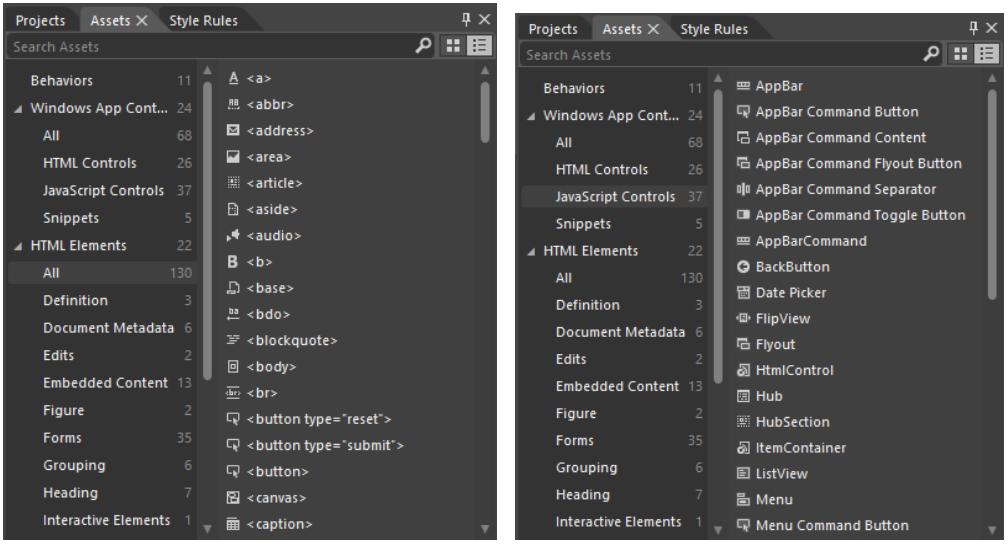


FIGURE 5-4 HTML elements (left) and WinJS control (right) as shown in Blend's Assets tab.

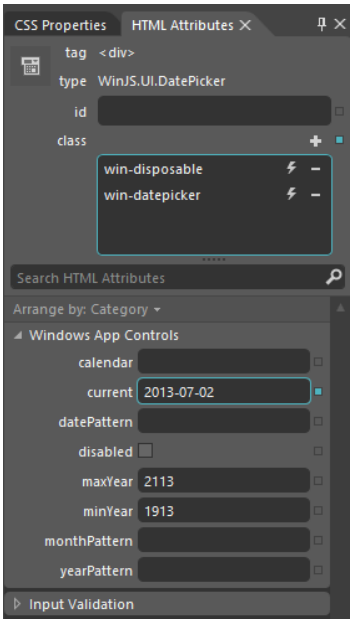


FIGURE 5-5 Blend's HTML Attributes tab shows WinJS control options, and editing them will affect the `data-win-options` attribute in markup.

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.

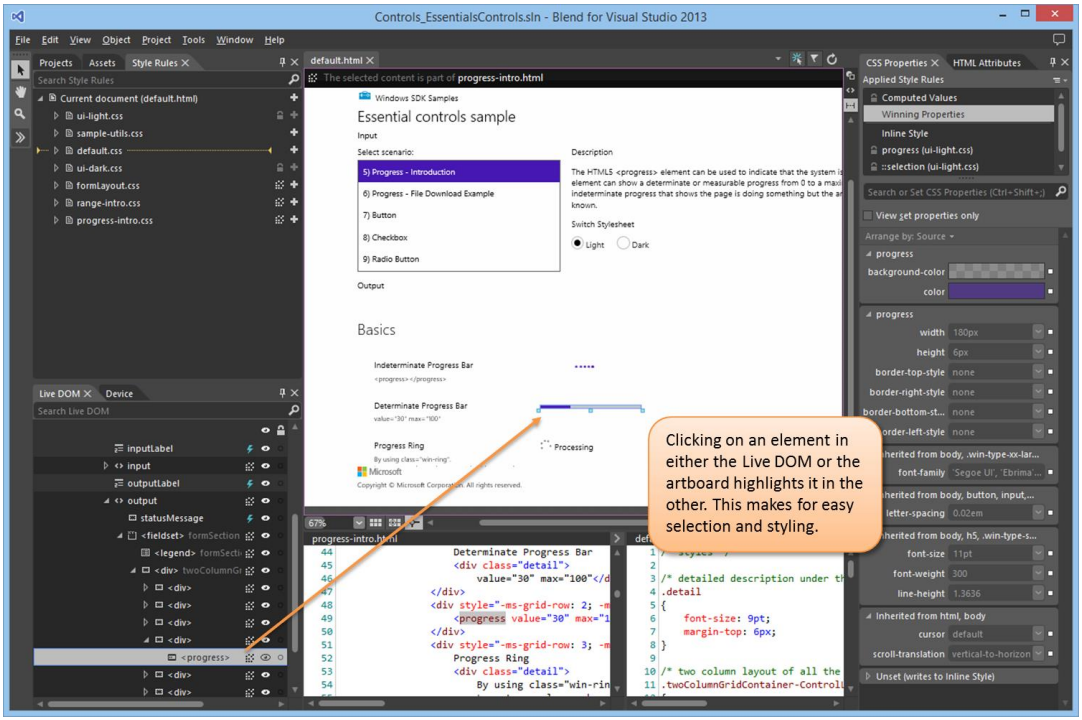


FIGURE 5-6 Blend's interaction between the artboard and the Live DOM.

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.

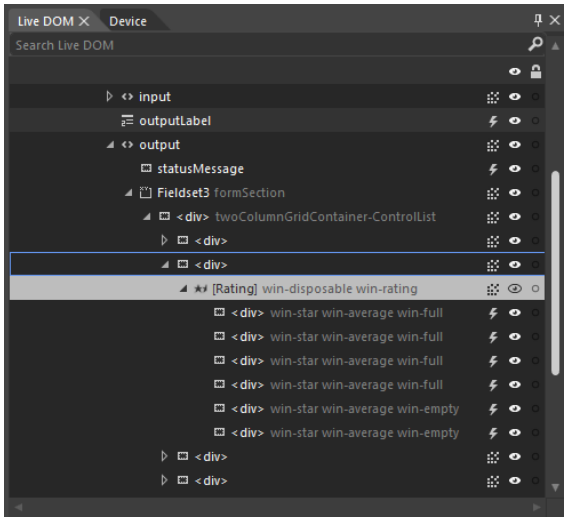


FIGURE 5-7 Expanding a WinJS control in Blend's Live DOM reveals the elements that are used to build it.

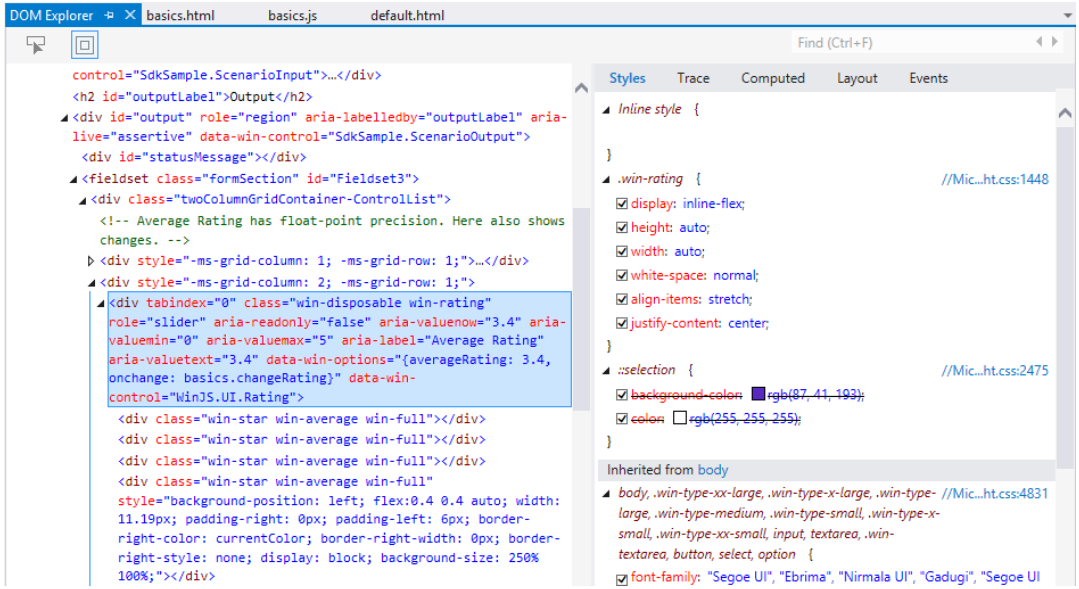


FIGURE 5-8 Expanding a WinJS control in Visual Studio's DOM Explorer also shows complete details for a control.

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

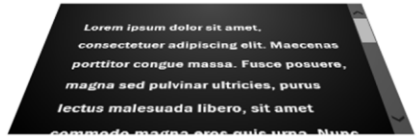
Now we get to enjoy a visual tour of styling capabilities for Windows Store apps. Much can be done with standard styles, and then there are all the things you can do with special styles and pseudo-elements as shown in the graphics in this section. The specifics of all these examples can be seen in the 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)



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)

Checkbox
Checkbox
Radio button

File upload

CSS pseudo-element: input[type="file"].<class>::-ms-value

CSS pseudo-element: input[type="file"].<class>::-ms-browse

Text Input (most forms)

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

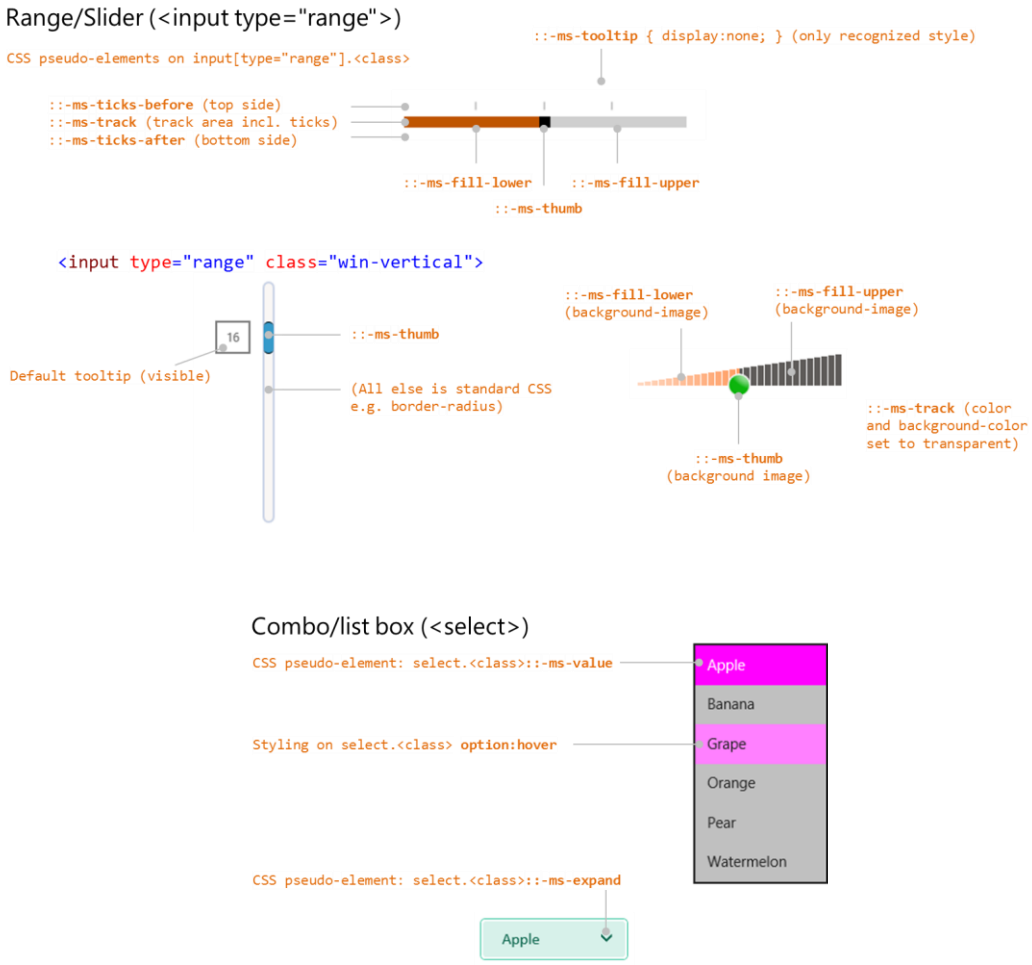
Progress

```
CSS pseudo-element:
    progress.<class>::-ms-fill {
        -ms-animation-name: -ms-ring;
    }
```

CSS pseudo-element: progress.<class>::-ms-fill (background-image, etc)

Text Input (password)

CSS pseudo-element: input[type="password"].<class>::-ms-reveal

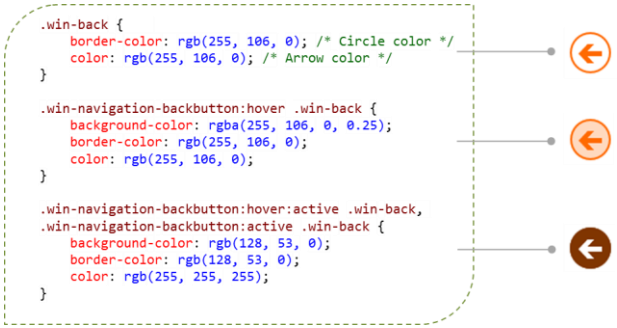


Note Though not shown here, you can also use the `-ms-scrollbar-*` styles for scrollbars that appear on pannable content in your app.

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

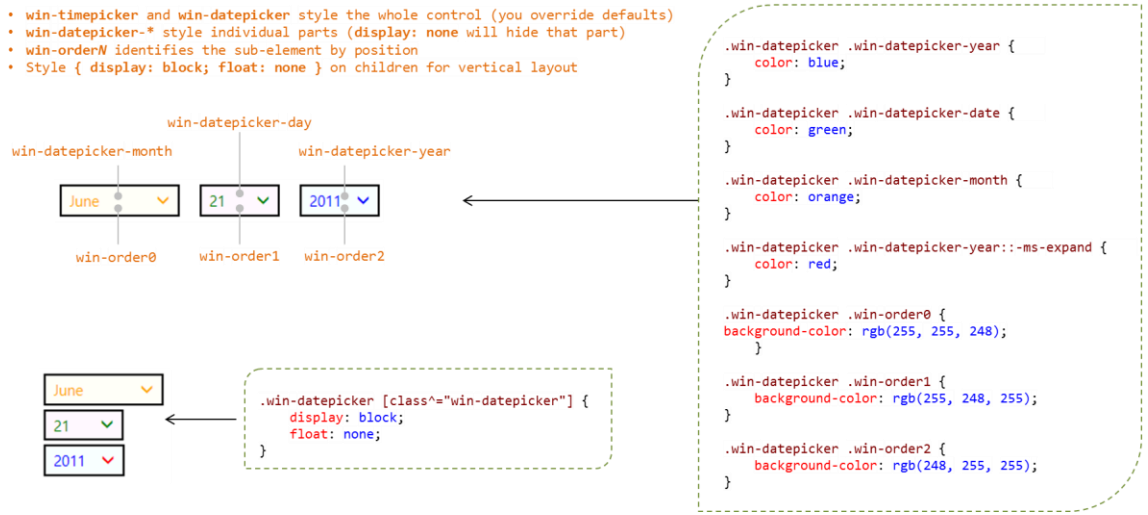
itself is made up of a `<button>` with the class `win-navigation-backbutton` (for the overall control, including pseudo-selectors) and a `<span>` with the class `win-back` (that isolates the ring and the arrow character). You can use these to change the coloration of the control:

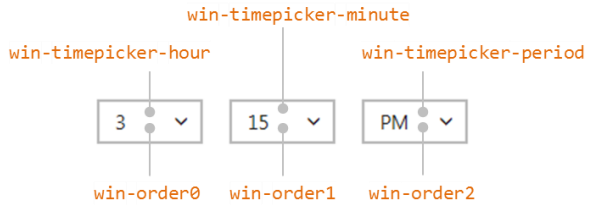


```
.win-back {
    border-color: rgb(255, 106, 0); /* Circle color */
    color: rgb(255, 106, 0); /* Arrow color */
}

.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:





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:[48]

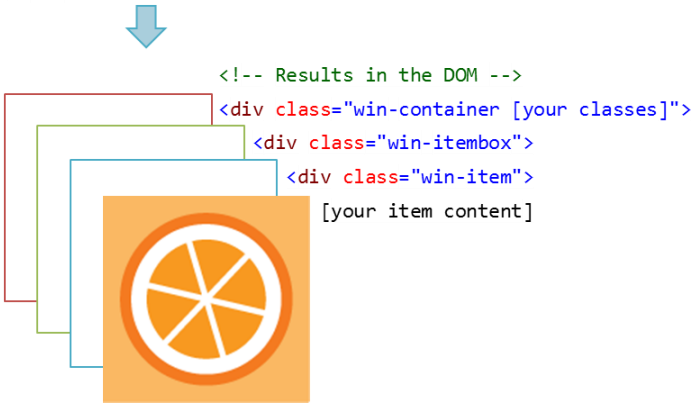
Typically you'll use these selection style classes to create specific selectors for styling the control's individual parts, which are also identified with specific `win-*` classes:

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.

[48] Technically speaking, the WinJS stylesheets contain no references to `win-selectionstyleoutline` as all selectors simply use `:not(.win-selectionstylefilled)`.

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 `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:

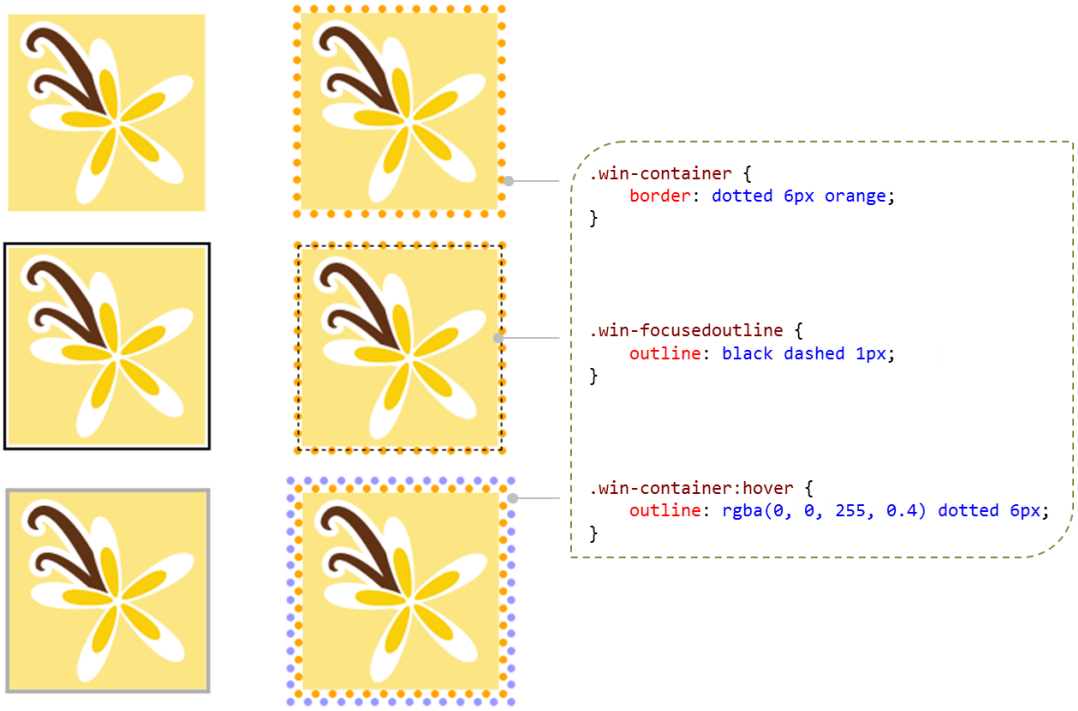
```
<!-- Markup in your HTML file -->
<div class="[your classes]" data-win-control="WinJS.UI.ItemContainer">
    [your item content]
</div>
```



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.)



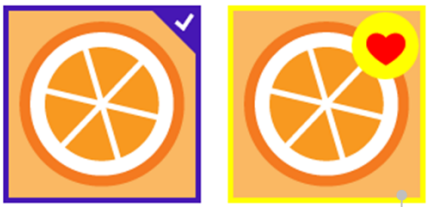
```
.win-container {
    border: dotted 6px orange;
}

.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.

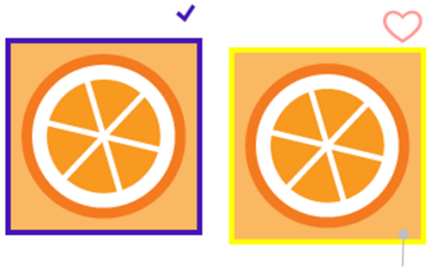


```
.win-container:not(.win-selectionstylefilled).win-selected .win-selectionborder {
    border-color: yellow;
}

.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 (take a look at Video 5-5 for the full experience):

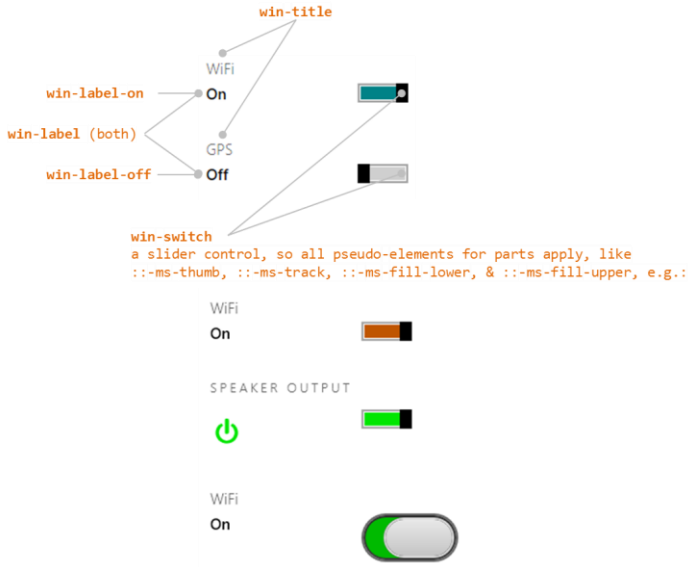


```
.win-container:not(.win-selectionstylefilled) .win-selectionhint {
    font-size: 0px; /* Hide the default checkmark */
}

.win-container:not(.win-selectionstylefilled) .win-selectionhint::before {
    content: '\E006';
    font-size: 30px;
    color: rgba(255, 0, 0, 0.4);
}
```

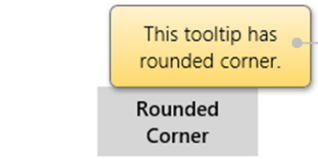
The `Rating` control similarly has states that can be styled in addition to its stars and the overall control. Again, `win-*` classes identify these individually and combinations style the variations:

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`

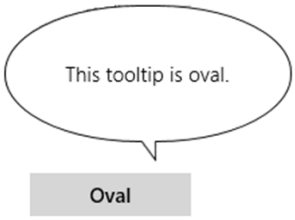
For the `ToggleSwitch`, `win-*` classes identify parts of the control; states are implicit. Note that the `win-switch` part is just an HTML slider control (`<input type="range">`), so you can utilize all the pseudo-elements for its parts as shown in the "Styling Gallery: HTML Controls" section earlier.



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:




```
#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;
}
```




```
#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.

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`.[49]

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.

[49] 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; 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.

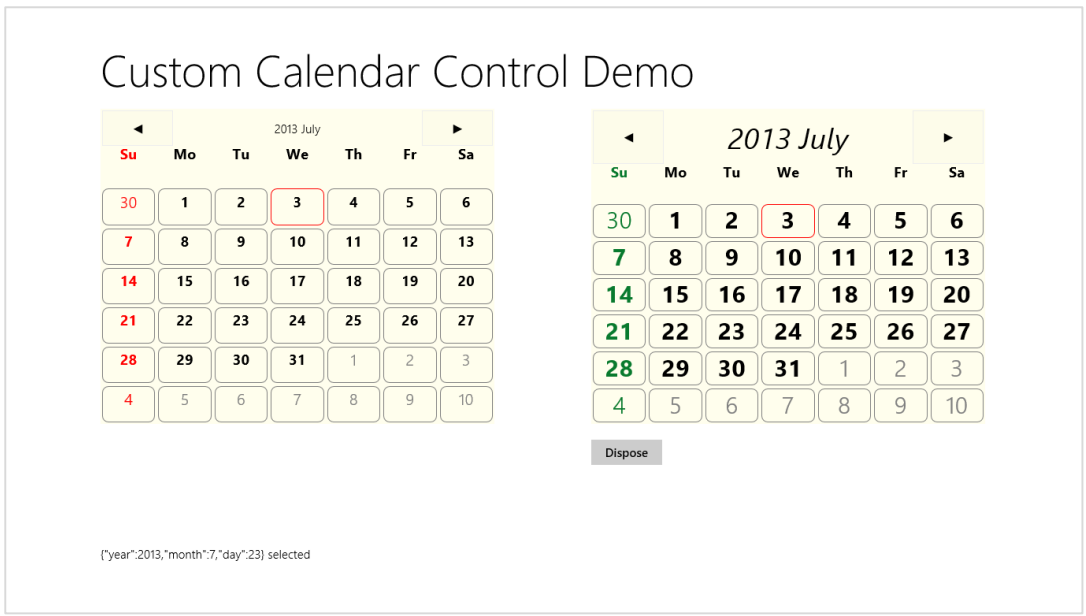


FIGURE 5-9 Output of the Calendar Control example.

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`:[50]

[50] 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`.

```
function disposeImpl () {
    //Reverse the constructor's steps
    this._cal = null;

    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![51]

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

[51] 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.

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 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;
    }
});
```

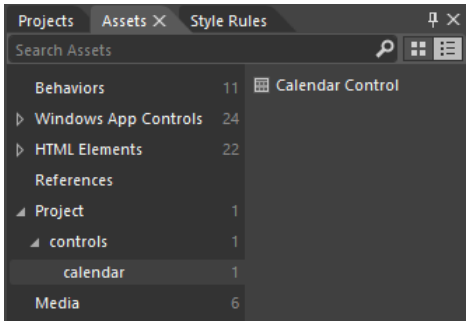
Finally, it is possible to have custom controls show up in the Assets tab alongside the HTML elements and the WinJS controls. For this you'll first need an OpenAjax Metadata XML (OAM) file that provides all the necessary information for the control, and you already have plenty of references to draw from. 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):



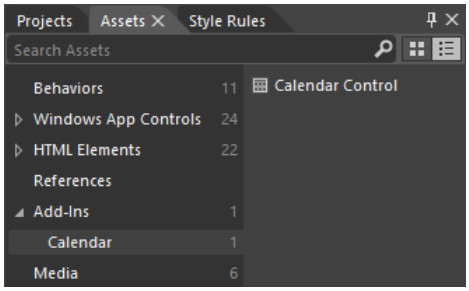
If you drag and drop that control onto an HTML page, you'll then see the different bits added in:

```
<!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 really 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, and 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 it 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.)

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 will give us the basis for understanding how to work 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.

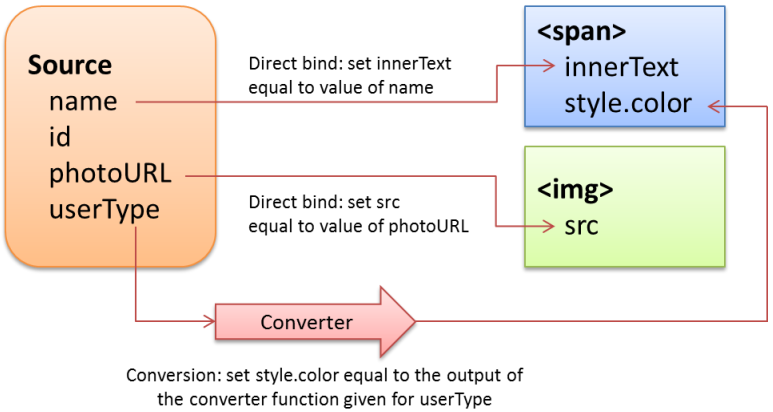
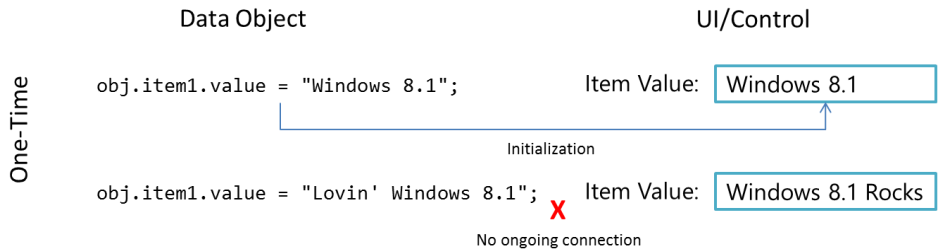


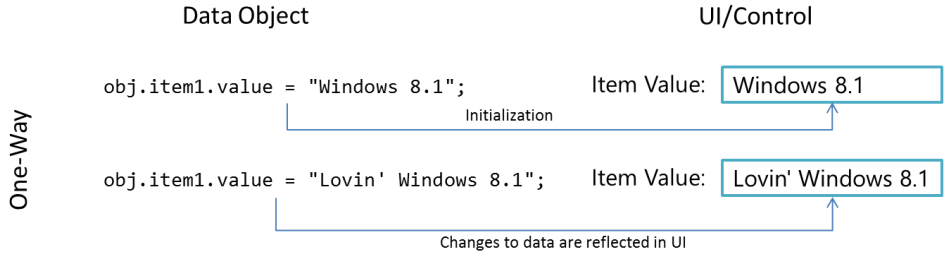
FIGURE 6-1 A common data-binding scenario between a source data object and two target UI elements, involving two direct bindings and one binding with a conversion function.

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):

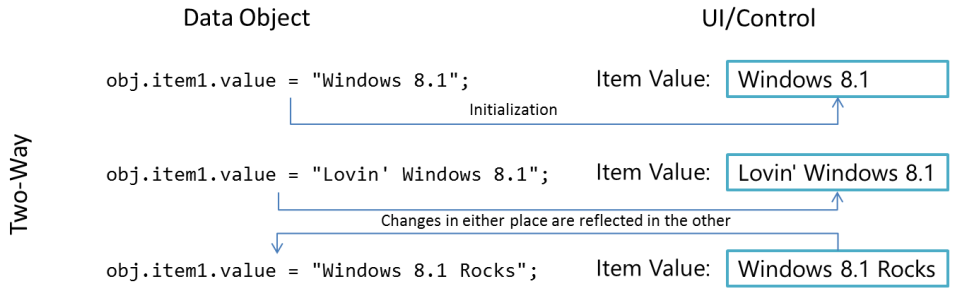
One-time: the value of the source property is copied to the target property at some point, after which there is no further relationship. This is what you automatically do when passing variables to control constructors, for instance, or simply initializing target property values with source properties. 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).



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 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:

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 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.[52] 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>
```

[52] 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.[53]

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

[53] 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):

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";
```

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.[54] 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).
- `getProperty` Retrieves a property value as an observable object itself, which makes it possible

[54] `expandProperties` also includes properties of the object's prototype.

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.

> **Preview note** The current documentation for `WinJS.Binding.bind` is wrong, as it's just a copy of the mixin's `bind` documentation. The true documentation can be found in the base.js file of WinJS if you search on the full name. I have filed a bug on this.

I referred to `WinJS.Binding.bind` just now as 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.[55] 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

[55] The actual code uses `this.onPositionChange.bind(this)` which I've simplified to avoid confusion between all the `binds`!

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, namely `WinJS.Binding.defaultBind`.[56] 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, as described later in this chapter.

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>
```

[56] 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.

Doing anything more requires that you implement the initializer function directly. This is necessary, 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 `data-win-options` string, if needed. The contents of the template—what will be 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.

> **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.

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 otherwise dramatically improve performances. 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`):

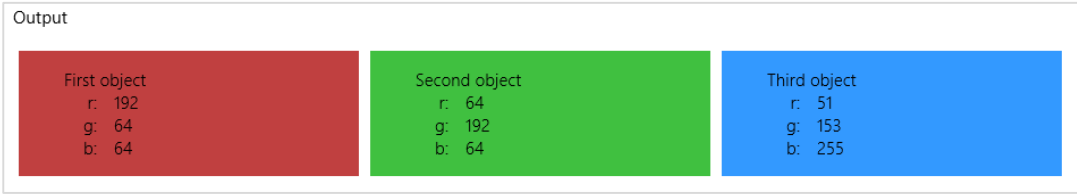
```
var DataSource = WinJS.Binding.define({
    text: "", color: { r: 0, g: 0, b: 0 }
});

// 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:



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`:[57]

- `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`.

[57] 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 especially helpful 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 on the control of the same names: `bindingInitializer`, `debugBreakOnRender`, `disableOptimizedProcessing`, and `extractChild`.[58] 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 is 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 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

[58] 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.

that it breaks some existing code somewhere—perhaps yours! If you find that's the case, or if for some reason want your app to just run slower, you can 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 contents disables compilation implicitly.

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 involved. The same is true for WinJS controls in the template that would normally go through the async `UI.processAll` method but are instead just instantiated directly with `new` and a pre-parsed options object. Avoiding async calls and extra parsing is another reason why the compiled template runs so 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 it 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` type—whose various capabilities are all supported in Windows Store apps, as are the typed JavaScript arrays 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 it 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.[59]

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>`. `IVector<String>`, for instance, 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 enabled 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 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 wherever a WinRT API calls for an `IIterable`. Second, when a WinRT APIs produces an iterator as a result, you can pretty much 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.

---

[59] 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 a formal construct for them. 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.

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.

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 an 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 only has methods to go through the collection in 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`.[60]

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.

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.

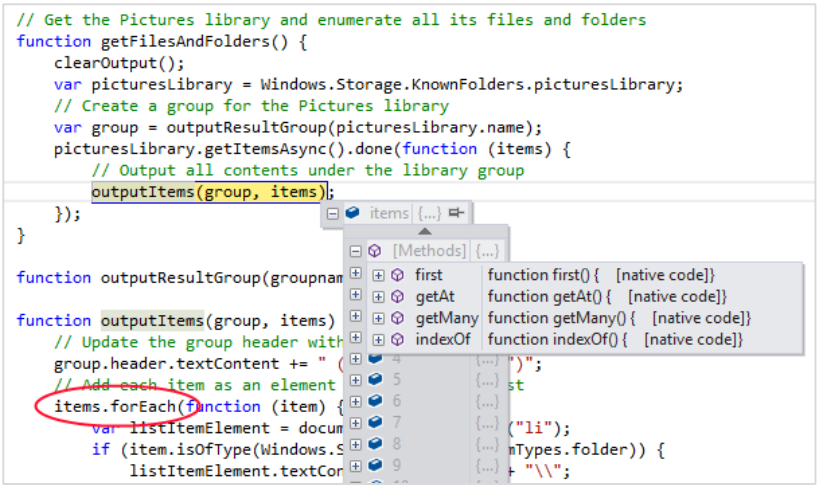
Within WinRT you'll most often encounter vectors with a read-write collection property on some object. For example, the `Windows.UI.Popups.MessageDialog` object, which we'll meet in Chapter 9, "Commanding UI," has a `commands` property of type `IVector<IUICommand>`, which translates into JavaScript simply 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 is what you get back from

[60] The `IObservableVector<T>` interface in `Windows.Foundations.Collections` exists for other languages and is not ever seen in JavaScript.

`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 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):



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` element. 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.[61] 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.

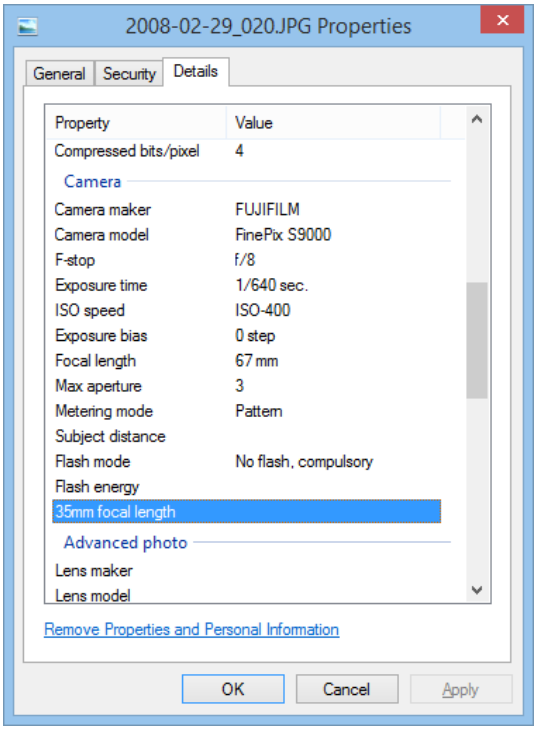
It's important to note right up front that maps and property sets are *not* projected as arrays. They *do* support value lookup through the `[ ]` operator, but otherwise do not have array methods. A map, instead, has methods (from `IMap`) called `lookup` (same as `[ ]`), `insert`, `remove`, `clear`, 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:

[61] 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.



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 will be 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 typically build a `List` with a sparse array meaning that you can have noncontiguous indices within the array (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.

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 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 will begin to 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 the these 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.

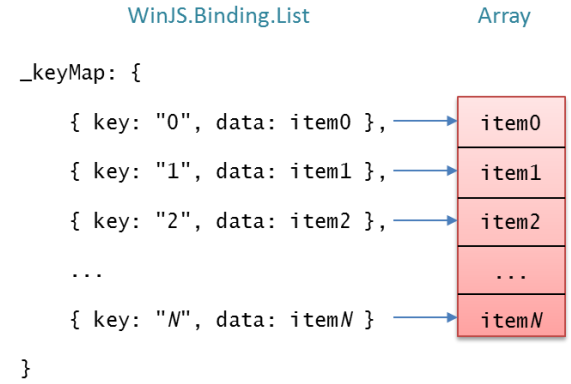


FIGURE 6-2 The default relationship between a `WinJS.Binding.List` and its underlying array.

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.

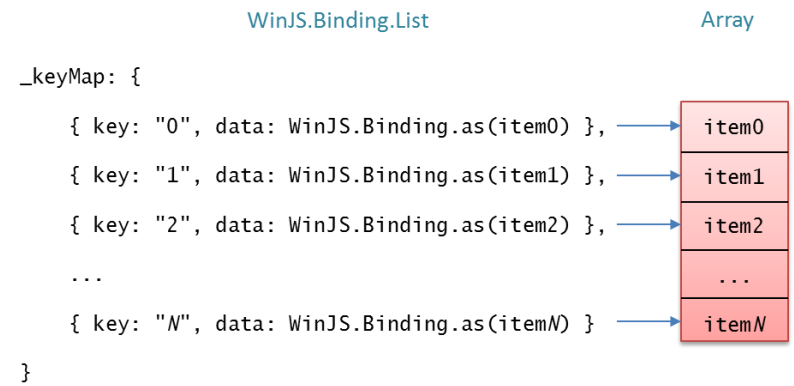


FIGURE 6-3 The relationship between a `WinJS.Binding.List` and its underlying array with the `binding` option set to `true`. If the array is sparse, the keys will not be contiguous, of course.

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's obvious why changes to the `List` are reflected in the array, as scenario 1 of the example demonstrates.

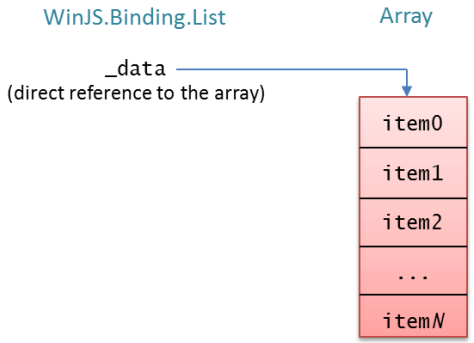


FIGURE 6-4 The relationship between a `WinJS.Binding.List` and its underlying array with the `proxy` option set to `true`. The `binding` option is ignored in this case.

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).

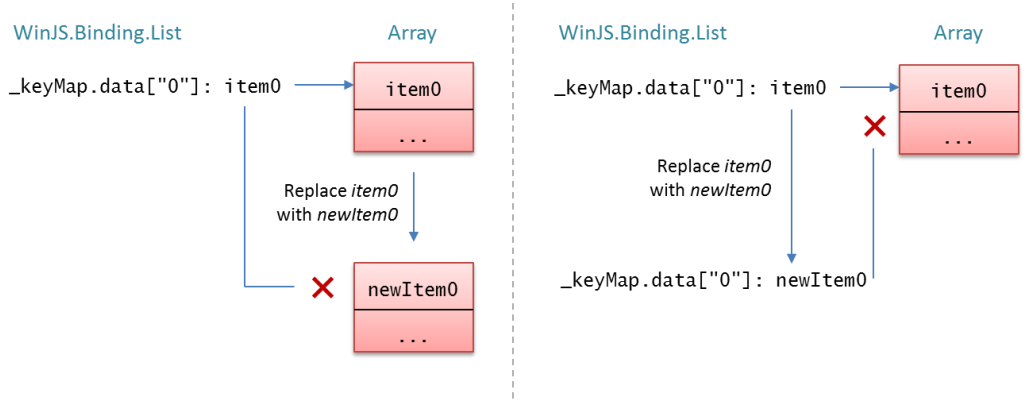


FIGURE 6-5 Replacing a whole object in either the array or the list will disconnect the related item in the other.

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.

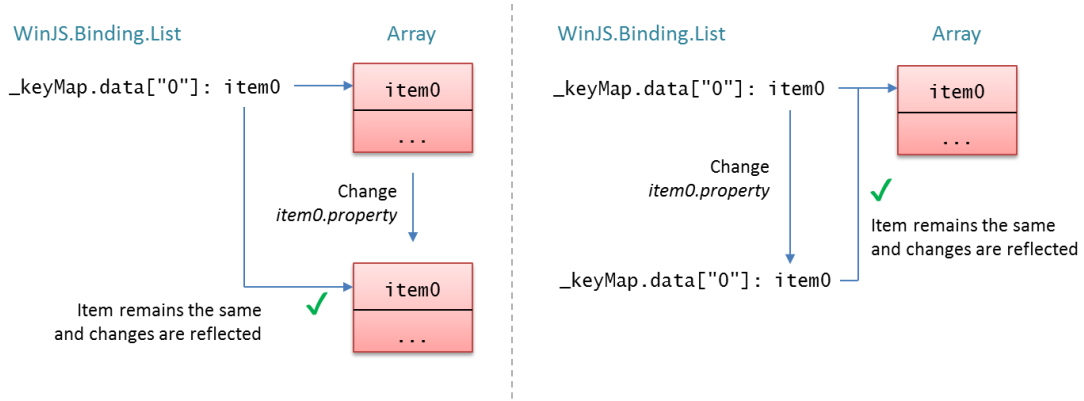


FIGURE 6-6 Changing properties of objects in either the array or the list will change the properties in the other, because both are still referencing the same object.

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 the `span` element that's data-bound to the item from `List.getAt(0)` automatically updates when changes came through the `List` but not when they came 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 yet 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 resort those contacts by one or more fields in each records. 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 across the board (and the same data binding code).

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 the output on and off separately from your other logging.

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!

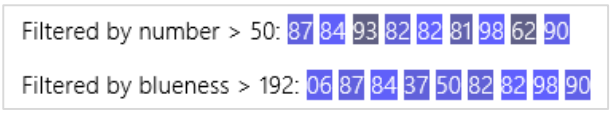
Scenario 3 (js/scenario3.js) of the example creates an empty `List` without an underlying array and uses `push` to populate it with objects containing a random number and color (like scenario 2):

```
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:

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;
}
```



Then we have three sorted projections, one of the original list sorted by number (then blueness in case of repeats), one sorted just by blueness, and a sorted from the filtered by blueness list:

```
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;
}
```

Add to this 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}
```

And a last one that's 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) but one 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 very much what one studies in college-level anatomy courses. Looking around my office and my home, I see even more collections: a book shelf with books; scrapbooks with pages and pages with pictures; cabinets with cans, boxes, and bins of food; my son's innumerable toys; the DVD case…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 means 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 very much 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, namely the features of the `WinJS.Binding` namespace, including binding templates and the observable `List` class. Now we can turn our attention to collection controls through which we can visualize and manipulate that data.

In this chapter we'll explore the three collection controls provided by WinJS 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. As it's really 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 and implementation 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. It's 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");
```

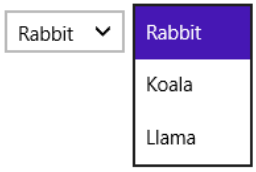
```
    o.value = item.id;
    o.textContent = item.description;
    e.appendChild(o);
});
```

And you'd generally repeat this process, of course, whenever the array contents changed, clearing out the `<select>` and recreating each `<option>`.

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 can really use WinJS controls and custom control on 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 as 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.

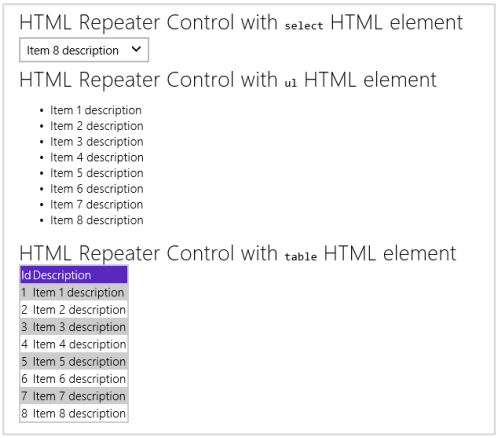
Here's how it's used in scenario 1 of the HTML Repeater control sample with `<select>`, `<ul>`, and `<tbody>` elements; note that each Repeater has only one immediate child element (html/scenario1.html):

```
<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.

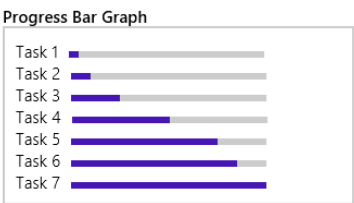


Because the Repeater turns its child element (and there must be only one) into a `Template` using the `extractChild` option, those elements are removed from the DOM. Rendering the template for each item in the collection will then create individual copies bound to those items. And because the Repeater 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 really goes to show that the Repeater is perfect of creating things like graphs and charts where repeated elements are involved:



Note that with the Repeater and others control that can declaratively reference a template (like the FlipView and ListView), 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 this control 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 on how best to use the control.

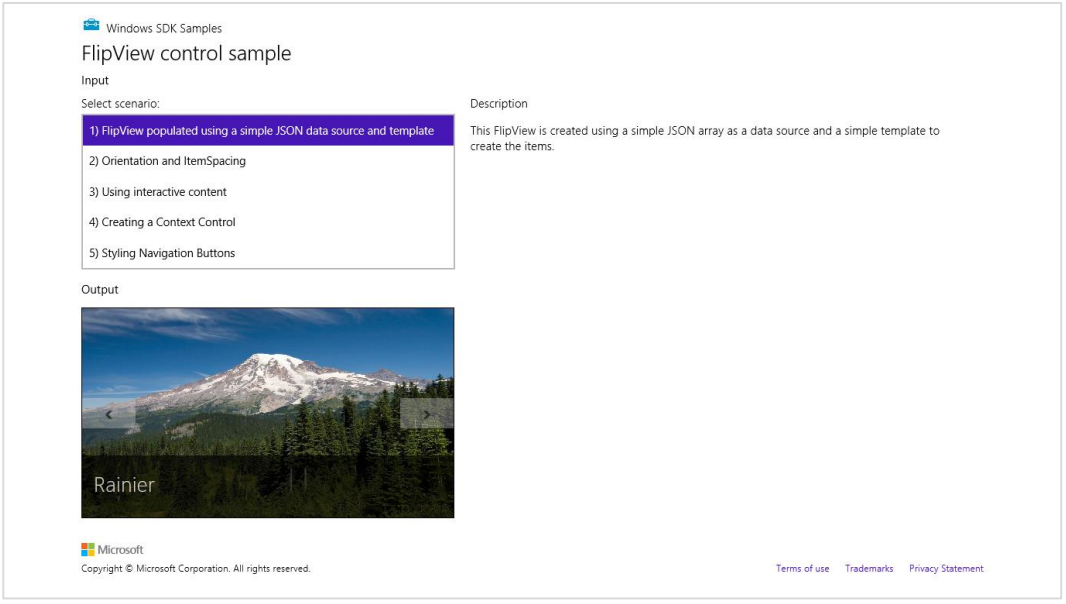


FIGURE 7-1 The HTML FlipView control sample; the FlipView is the control displaying the picture.

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.IList-DataSource` 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>
```

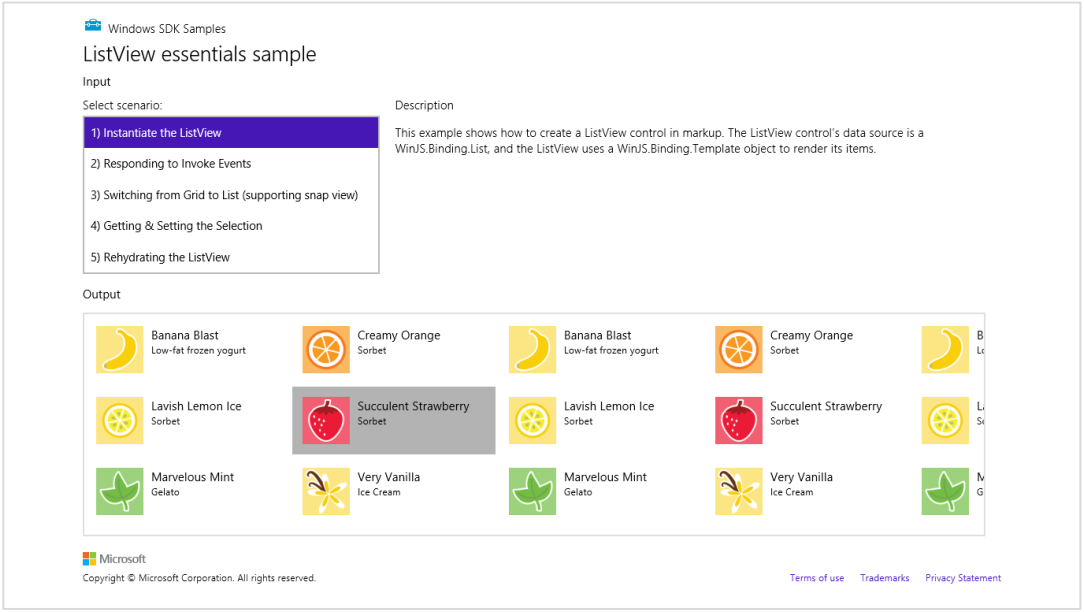


FIGURE 7-2 The HTML ListView essentials sample.

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 here 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 the custom layouts, which is found in Appendix B, "WinJS Extras."

> **Tip** A number of errors will cause a ListView to fail to instantiate. 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 from 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, since 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 back to the UI thread first.

Sidebar: Item Hover Styling

While 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 using 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 really 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.

How we make use of these with the ListView 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`.

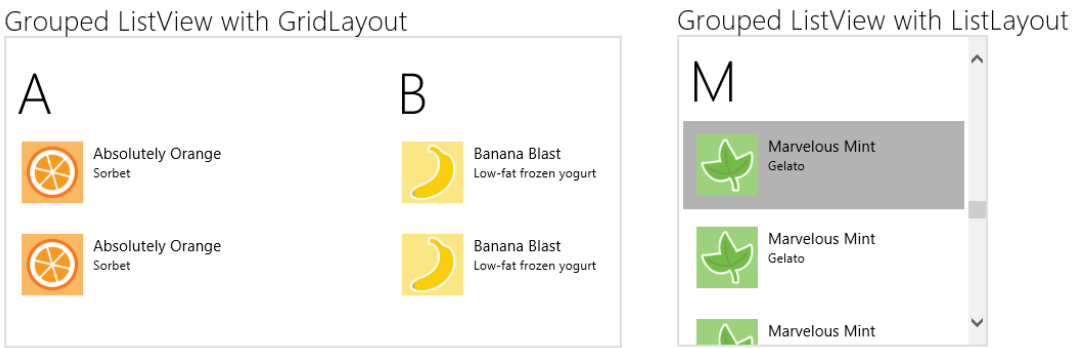


FIGURE 7-3 The output of scenario 1 of the HTML ListView grouping and Semantic Zoom sample.

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` property. 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 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 really only the most basic such summary.) Since this function is only called 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.

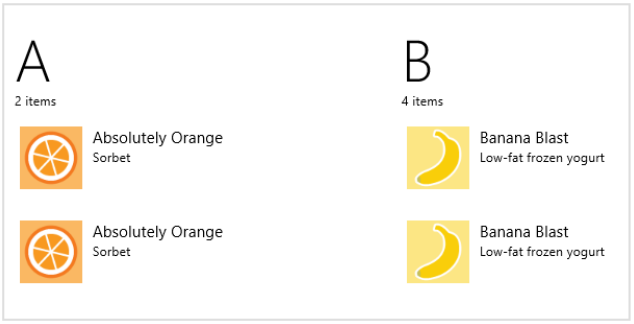
For example, in a slightly modified version of this sample in this chapter's companion code I use `createFiltered` to obtain a projection of the list filtered by the current key.[63] The `length` property of this projection is then the number of items in the group:

```
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:



One other note for scenario 1 is that although it doesn't use a group sorter function with

[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.

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 generates a groupheaderinvoked event where the eventArgs.detail object will contain 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 also 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.

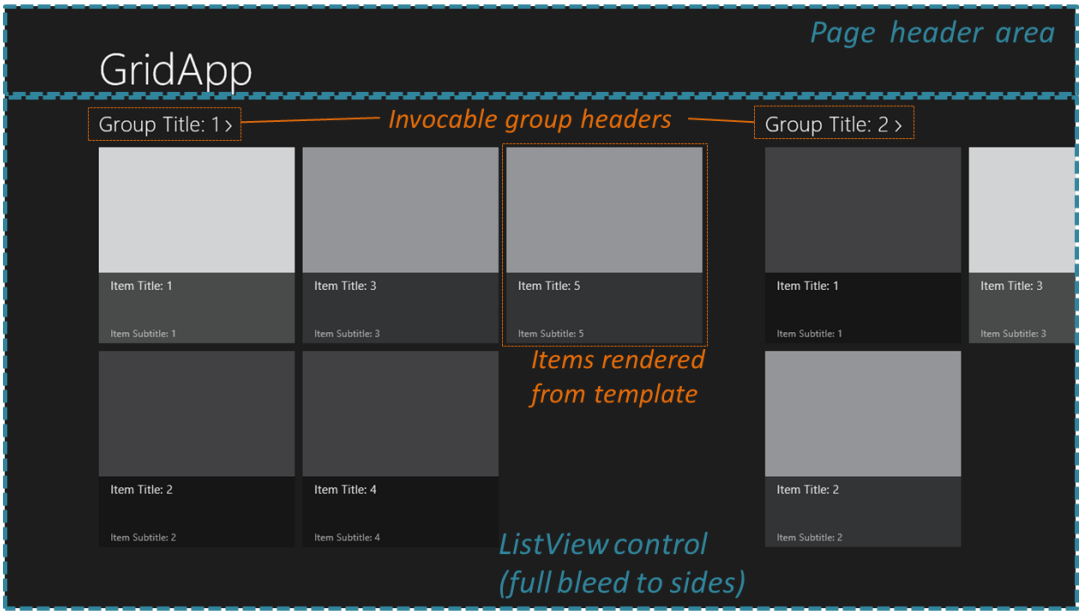


FIGURE 7-4 ListView elements as shown in the Grid App template home page. (All colored items are added labels and lines.)

There's quite a bit going on with the ListView in this project, so we'll 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 sprinkled.

As for the data itself (that you'll likely replace), this is again 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 get 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 sometimes 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, is intended to be wholly replaced with real data that you obtain from other sources, like a file or some web API, and wrap 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 really applicable to both.

The Semantic Zoom Control

Since 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 (especially ungrouped items), where a user will likely get really 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 a 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 (reminder: all the videos are also available with the companion content), where I show the effects both at normal and slow speeds.

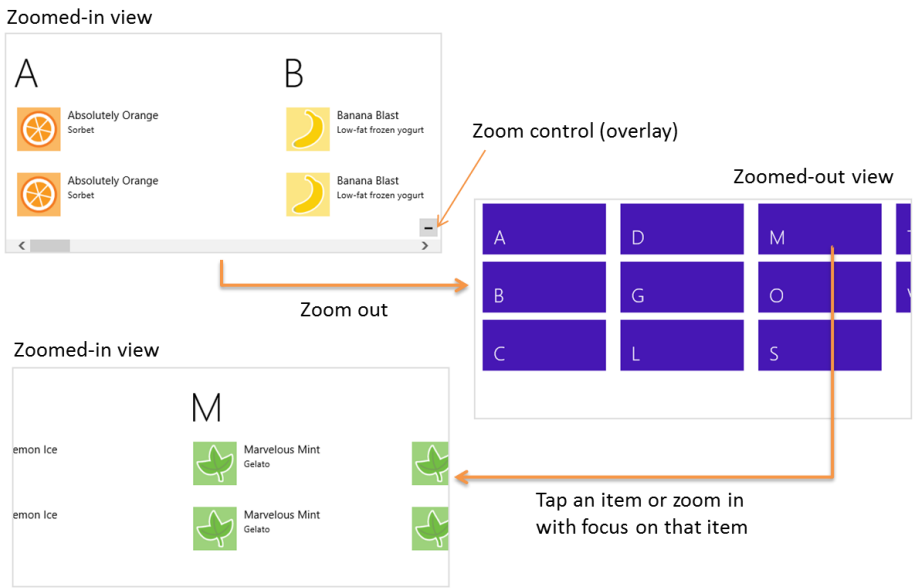


FIGURE 7-5 Semantic zoom between the two views in the ListView grouping and Semantic Zoom sample. The zoom control overlay appears only for the mouse (as does the scrollbar). See Video 7-1 for the dynamic effect.

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.

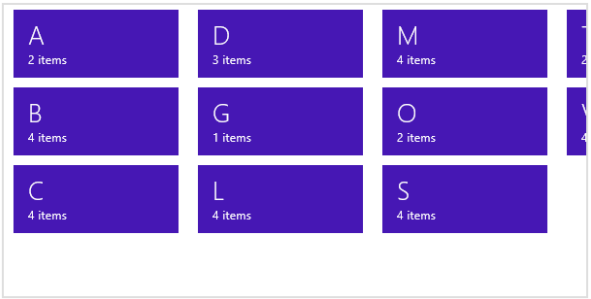
The control has a few other properties, such as `enableButton` (a Boolean to control the visibility of the overlay button; default is `true`), `locked` (a Boolean that disables zooming in either direction and can be set dynamically to lock the current zoom state; default is `false`), and `zoomedOut` (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):



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 methods of the interface. 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 noted that 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 the same for each item. 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

As I noted before, 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, we don't actually 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 assign a rendering function to those properties instead, 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 another 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 properties we need from `item.data` and apply whatever conversions you need.

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 piece of optimized C++ code in the app host and doesn't need to go through any other layers to get 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, inserting markup into your HTML file where you can go right into styling. The process is described here, which I walk through in Video 7-2.

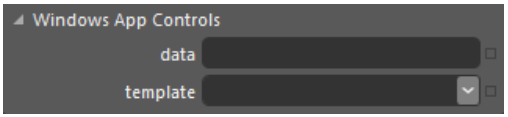
First create your data source in code as you normally would, making sure it's accessible from markup. In early stages of development you can use placeholder data, of course. As in the video, here's one that lives in a data.js file and is accessible via *Data.seasonalItems*:

```
var slTitle = "Item Title";
var slSubtitle = "Item Sub Title";
var slSubtext = "Quisque in porta lorem dolor amet sed consectetuer ising elit, ...";

var seasonalList = [
    { title: slTitle, subtitle: slSubtitle, description: slSubtext,
      image: "/images/assets/section2_1a.jpg" },
    { title: slTitle, subtitle: slSubtitle, description: slSubtext,
      image: "/images/assets/section2_1b.jpg" },
    { title: slTitle, subtitle: slSubtitle, description: slSubtext,
      image: "/images/assets/section2_1c.jpg" },
    { title: slTitle, subtitle: slSubtitle, description: slSubtext,
      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*:

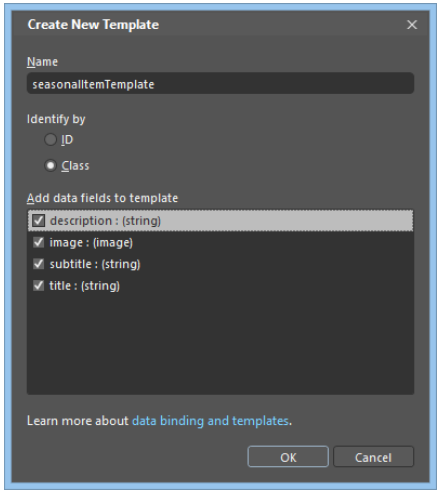


In the data property, enter the identifier for your data source (*Data.seasonalItems* in the example). This will create a `data-win-options` string in your markup, and you should see the untemplated results in the control:

```
<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"}

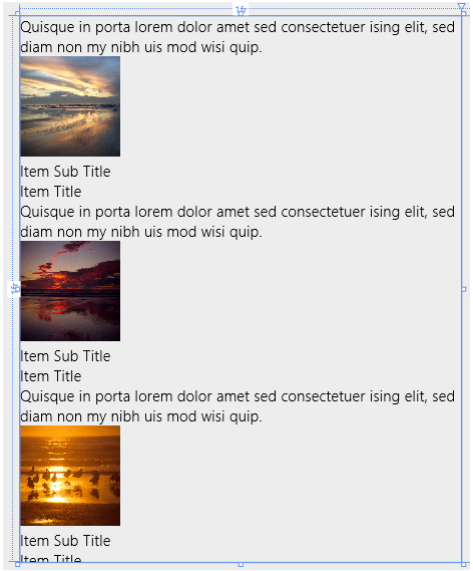
Next, click the drop-down next to the *template* property and select <Create New Template…>, which brings up the dialog below wherein you'll conveniently see the members of your data source. Check those you need and give your template a name:

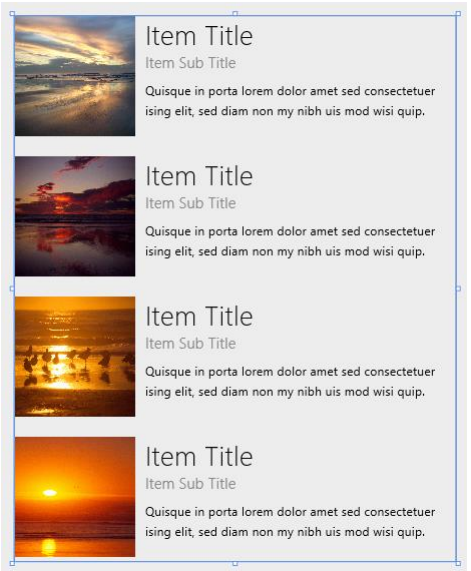


When you press OK, Blend will create unstyled markup in your HTML file and insert the appropriate reference in the control's options. Because I selected to identify the template with a class, the reference uses the select syntax:

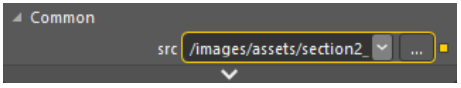
```
<div class="seasonalItemTemplate" data-win-control="WinJS.Binding.Template">
    <div>
        <div data-win-bind="textContent:description"></div>
        <img data-win-bind="src:image" height="100" width="100">
        <div data-win-bind="textContent:subtitle"></div>
        <div data-win-bind="textContent:title"></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):

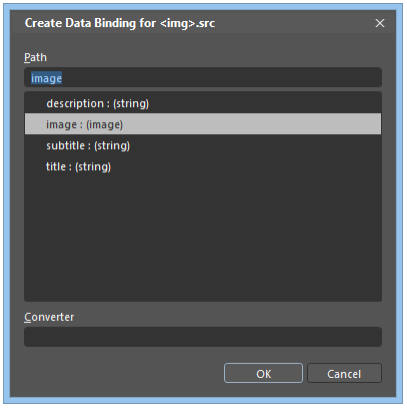




Blend also provides a shortcut to create and edit `data-win-bind` entries for individual properties, which works especially well inside a template. With a specific control selected (whether inside a template or anywhere else), click the little square to the right of a property in the HTML attributes pane, as shown below for the `img` element in the template we just created:



The yellow highlight here means that the property is data-bound already. Anyway, 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:



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.

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:

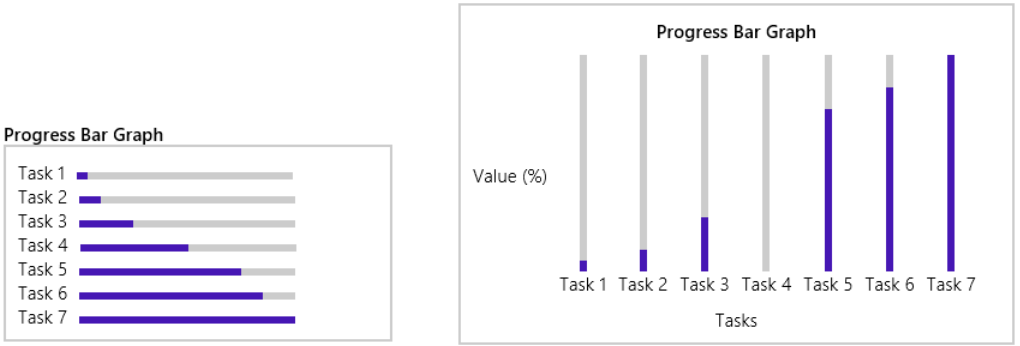
Event trigger	Before DOM is updated	After DOM is updated
The List in the `data` property fires its reload event	`itemsreloading`	`itemsreloaded`
Individual item in the data source changed	`itemchanging`	`itemchanged`
Item added to the data source	`iteminserting`	`iteminserted`
Item removed from the data 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 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 `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. Nevertheless, you can use the class to style those elements as desired.

The repeater's children won't receive any styling classes of their own either, so it's really 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 simple 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 aren'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 itself 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)
    ])
});
```

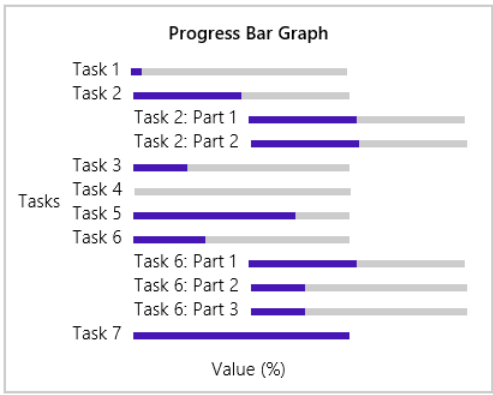
Take a close look now at the template structure in html/scenario6.html:

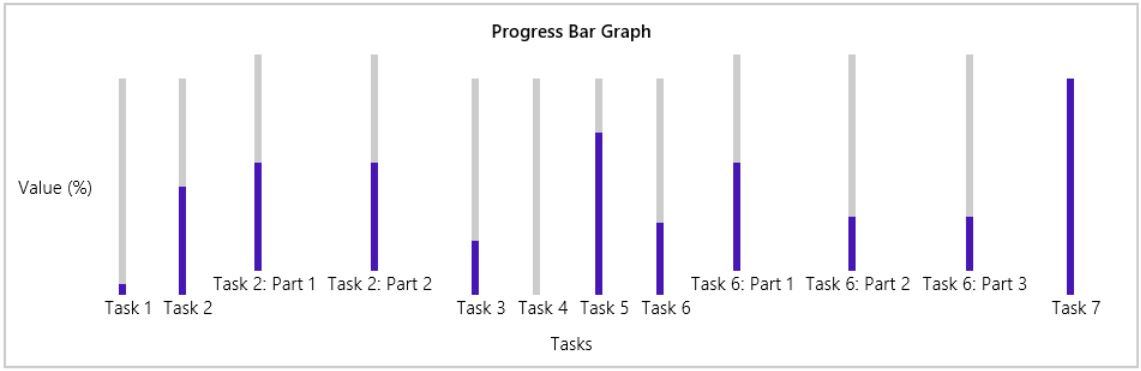
```
<div class="template" data-win-control="WinJS.Binding.Template">
    <div class="bar">
        <label class="label" data-win-bind="textContent: description"></label>
        <div class="barClip">
            <progress class="progress" data-win-bind="value: value" max="100"></progress>
        </div>
        <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. Remember that 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:



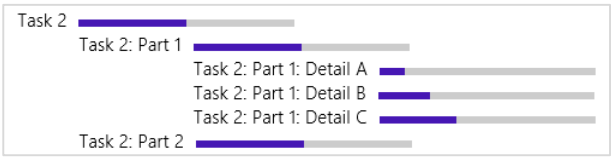


Nesting the second-level `Repeater` that refers to the same template is perfectly legal—a template control just renders its child elements when asked, and if that happens to contain a copy of itself, then you'll have some recursive rendering, but nothing that's going to confuse WinJS! In fact, the nested 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*:



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

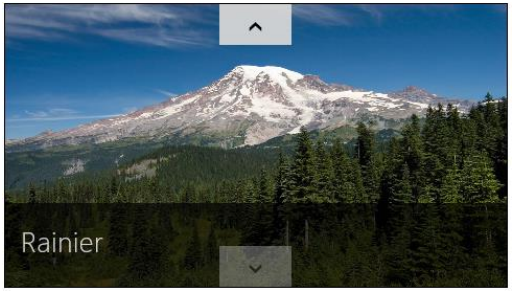
One reason that the `ItemContainer` and `Repeater` elements were created for WinJS 2.0 was that many developers wanted a UI that worked a lot like a `ListView`, but without all the `ListView` features. Thus instead of trying to make a `ListView` in which all those features could be disabled, the WinJS 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.

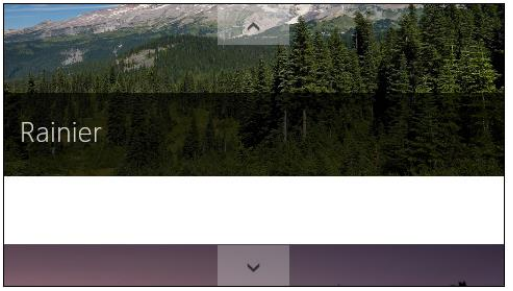
As a result, it's very straightforward in WinJS 2.0 to create a lightweight type of `ListView` where you have fully interactive items (select, swipe, drag, and invoke behaviors) but without any other layout policy or list-level interactivity (panning, incremental loading, keyboard navigation, reordering, 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

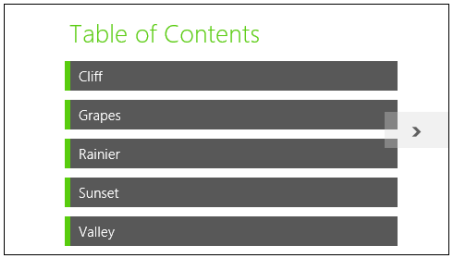
The `WinJS.UI.FlipView` is a very capable and flexible control for any situation where you want to page through items in a data source one at a time. We saw the basics earlier, in "Quickstart #2," so let's now see the rest of its features through the other scenarios of the HTML FlipView control sample. (It's worth mentioning again 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 in 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.



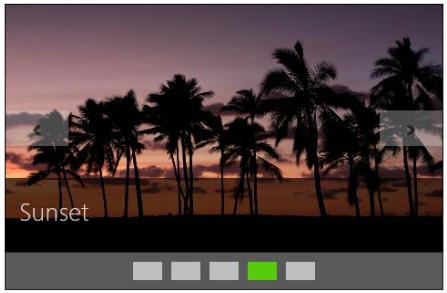


Scenario 3 ("Using interactive content") shows the use of a renderer function instead of a declarative template, as we learned about in "How Templates Work with Collection Controls." Scenario 3 uses a renderer (a function called `mytemplate` in js/interactiveContent.js) to create a "table of contents" for the item in the data source marked with a "contentArray" type:



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:



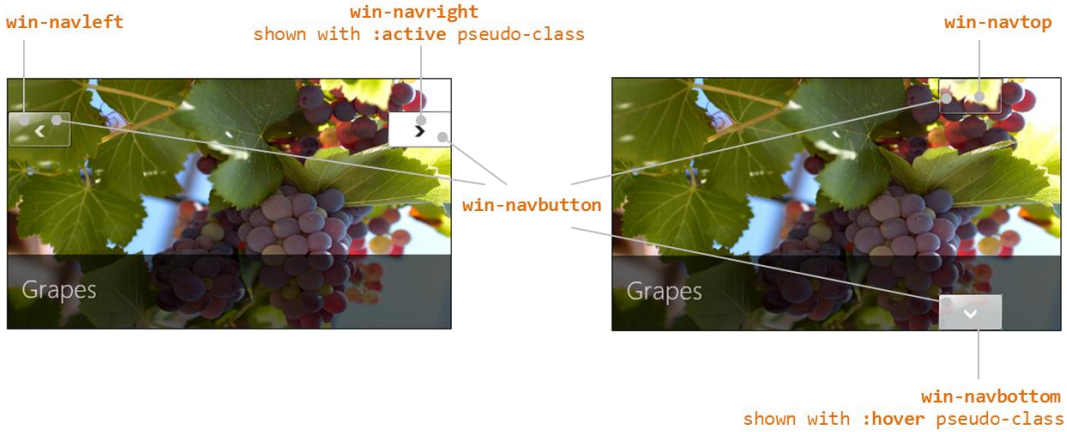
The items themselves are again rendered using a declarative template, which in this case just contains a placeholder `div` called *ContextContainer* for the navigation control (html/context-Control.html):

```
<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 while a flip is happening. (The event occurs when an item changes visibility and is fired twice for each flip, once for the previous item, and again for the new one.)

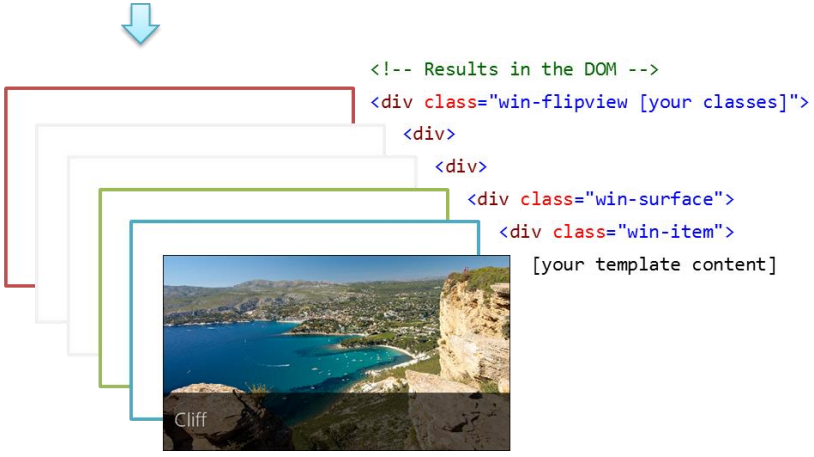
Scenario 5 (“Styling Navigation Buttons”) demonstrates the styling features of the FlipView, which involves various `win-*` styles and pseudo-classes as shown here (also documented on Styling the FlipView and its items):



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>
```



The `win-surface` class is where you'd 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 completed (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`—they instead work against an abstract data source defined by a set of interfaces.[64] That abstraction allows you to implement any kind of data source you want and plug it right 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 control 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*), it's important to avoid loading image files to generate the thumbnail. Instead, *always* use `Windows.Storage.StorageFile.getThumbnailAsync` or `getScaledImageAsThumbnailAsync` 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 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. 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 SkyDrive 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

[64] The Repeater, though, works with only a `List`, meaning it works with only in-memory sources.

[65] See "Windows.Foundation.Collection Types" in Chapter 6 for an overview of what interfaces are in JavaScript, which basically amounts to a kind of documentation. For custom sources, you typically get the methods from the `VirualizedDataSource` class.

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 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.

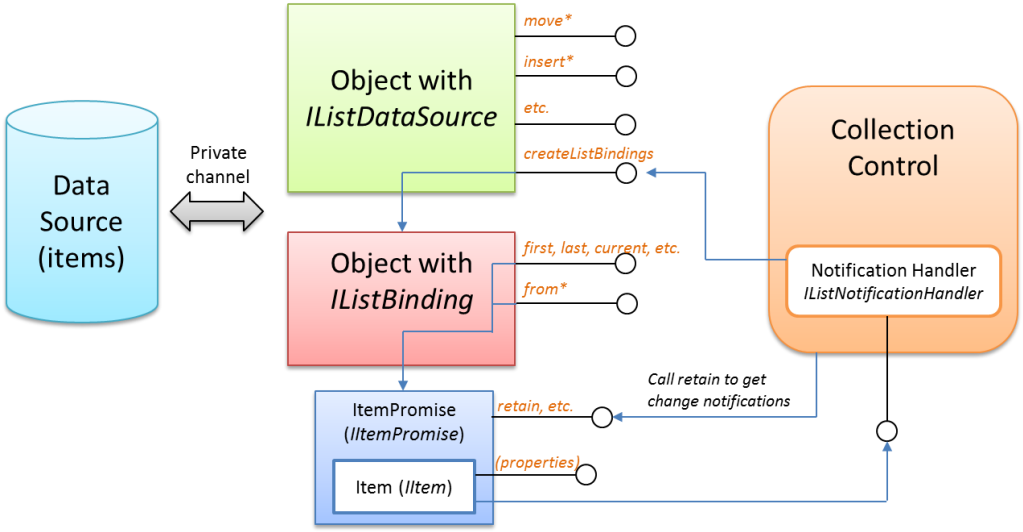


FIGURE 7-6 The general relationships between a data source, the `IListDataSource` and `IListBinding` interfaces, and a collection control like the ListView that uses that data source through those interfaces and provides a notification handler with `IListNotificationHandler` methods.

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, all 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 simply 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 to retain every item in the source: 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`/`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 then 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);
}
```

[66] The sample calls `beginEdits` and `endEdits` in this method, which are not necessary for a single change and are therefore omitted. The same is true for removing one item, but the methods are effective when swapping items or shuffling the whole set.

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`, 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;

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 just 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.

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 as a data source instead of an in-memory array. It provides automatic change detection as well, 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.

The `StorageDataSource` works with something called a *query*, but we won't learn about that until 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");
```

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

[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.

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 afore-mentioned 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 plenty of optimizations like caching and virtualization. To make the whole job easier, WinJS accordingly 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 an 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 source 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`,	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

`itemsFromKey`, `itemsFromStart`	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 a number of these methods won't make sense. 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`.

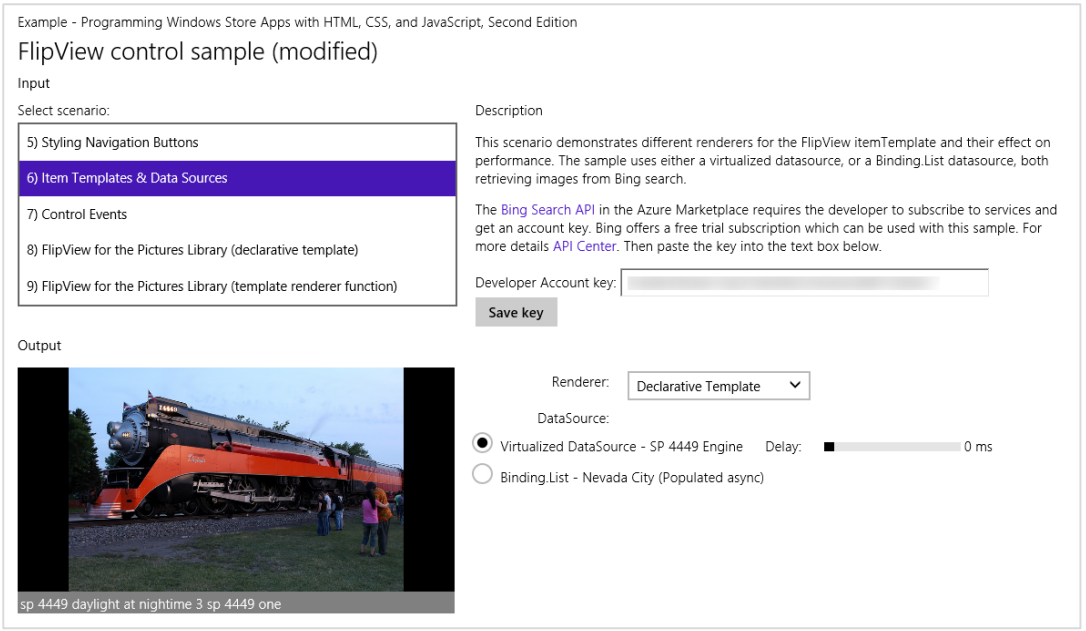


FIGURE 7-7 Scenario 6 of the modified HTML FlipView sample in the companion content (cropped).

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/bingImage-SearchDataSource.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 `VirtiualizedDataSource`. 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 the bulk of the code in js/bingImageSearchDataSource.js. In this case we have a read-only, index-based, nonorderable source without change notification, so 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 that 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.

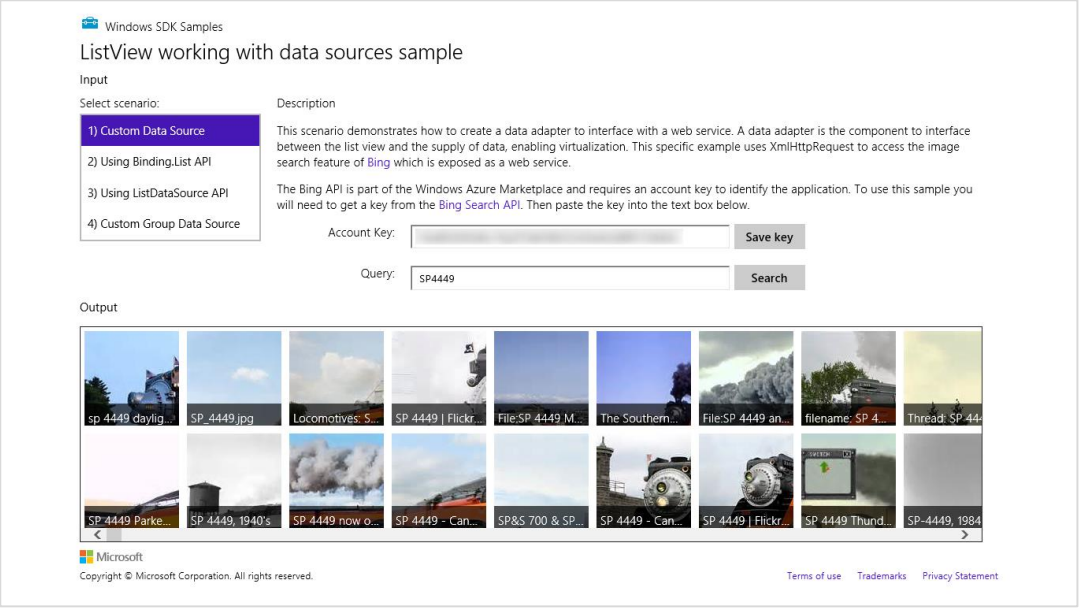


FIGURE 7-8 Scenario 1 of the modified HTML ListView working with data sources sample, which allows you to enter a search term of your choice.

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 rather 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 = 100, countBefore = 0, countAfter =0
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 = 371, countBefore = 0, countAfter =31
itemsFromIndex: requestIndex = 557, countBefore = 0, countAfter =303
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. That is, 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++. For more, see 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 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 in 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, such that app bar commands could act 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.

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, which is to say that there isn't a similarity of items across the groups. Because of this, using a ListView is more complicated than just using a single pannable `div` with a CSS grid in which you can lay out whatever sections you need or than using the Hub control that was created for these scenarios.

Within those sections, of course, you might use ListView controls to display an item collection, but for the overall page, a simple `div` is all you need. I've illustrated these choices in Figure 7-9 using an image from the Navigation design for Windows Store apps topic, since 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 there is essentially fixed content (like a single item), the best choice is a `div`.

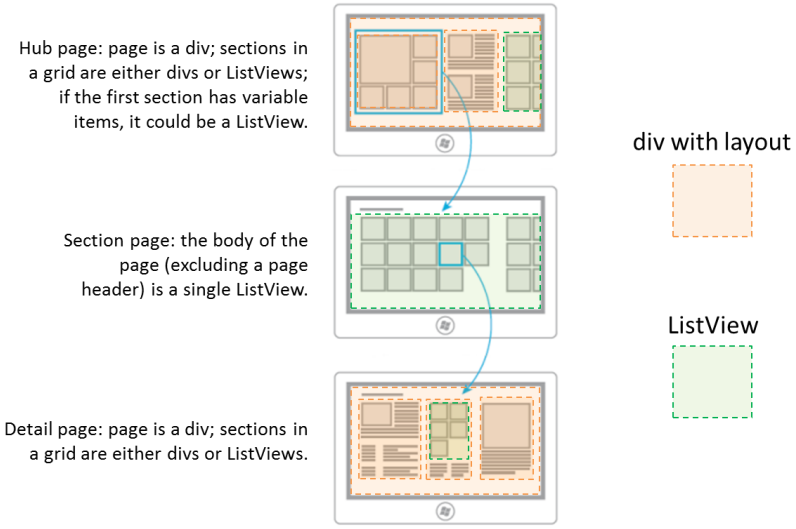


FIGURE 7-9 Breaking down typical hub-section-detail page designs into `div` elements and ListView controls. The Hub control is typically useful for pannable regions that contain different types of sections.

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 and checklist 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.

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 what indices are visible, or 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. There is also the `scrollPosition` property that 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 recommended

[68] I'll remind you again that much changed in the ListView control between WinJS 1.0 and WinJS 2.0, as I describe on ListView Changes between WinJS 1.0 and WinJS 2.0. In this chapter I describe only the WinJS 2.0 control, so if you see a method, property, or event in the docs that isn't included here, it's probably deprecated.

that you 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.

- **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, allowing a vertically oriented ListView or its surrounding page to pan. If this is set to `select`, the gesture is processed by the item to select it.

- **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 generally 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 yet 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.

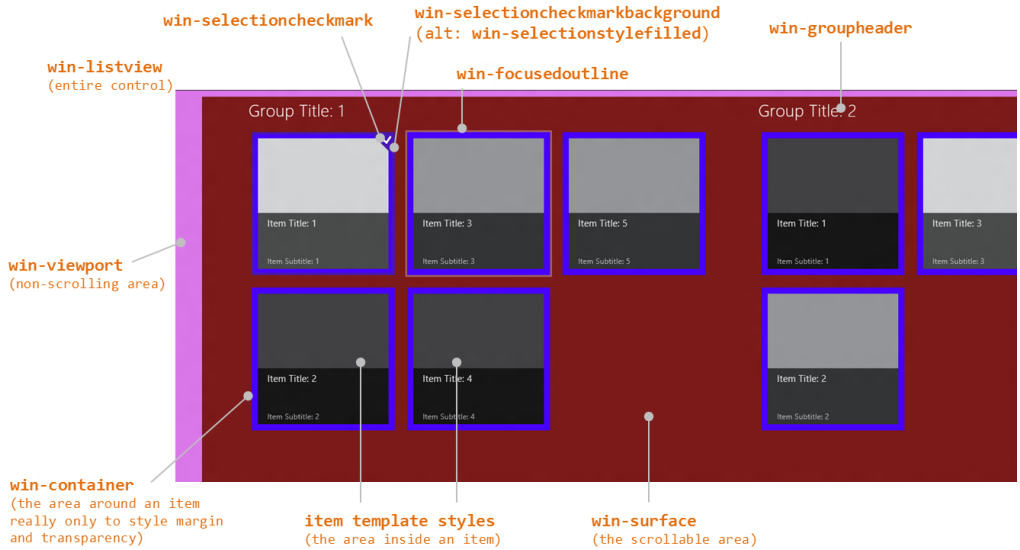


FIGURE 7-10 Most of the style classes as utilized by the ListView control.

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 that style in CSS or set it from JavaScript when you know the space that the ListView should occupy, as we'll cover in Chapter 8.
- 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, as shown below. If you want the filled look, 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 all of these as the same ones that apply to the `WinJS.UI.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 get tired 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, of course.)

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. It's simply a matter of watching for item requests near the end of the list (however far you want to make it) and using that as a trigger to load more items. Within an item renderer, you'd 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, you'd 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, of course, such as drop effects that can be 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.

There are four ways the ListView can participate in drag and drop. 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; otherwise 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 project 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 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`.

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)

[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.

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}
```

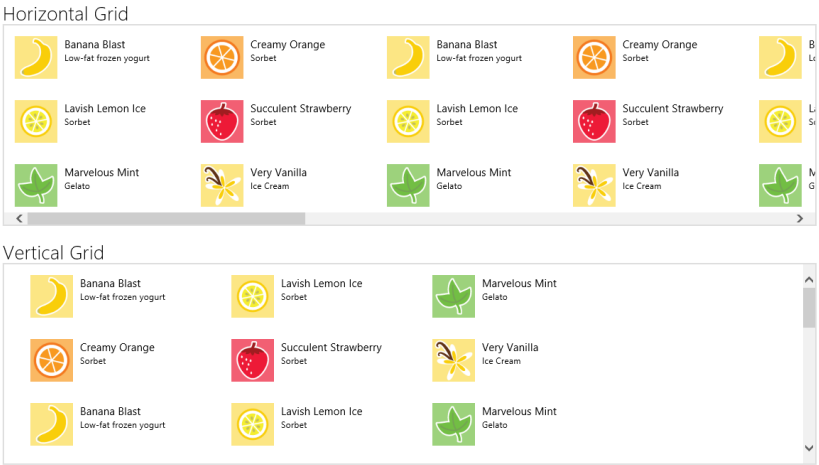


FIGURE 7-11 Horizontal (default) and vertical orientations for the GridLayout; notice the directions in which the items are laid out: top to bottom in horizontal, right to left in vertical (which is reversed with right-to-left languages). I've included the scrollbars here to show the panning direction more clearly.

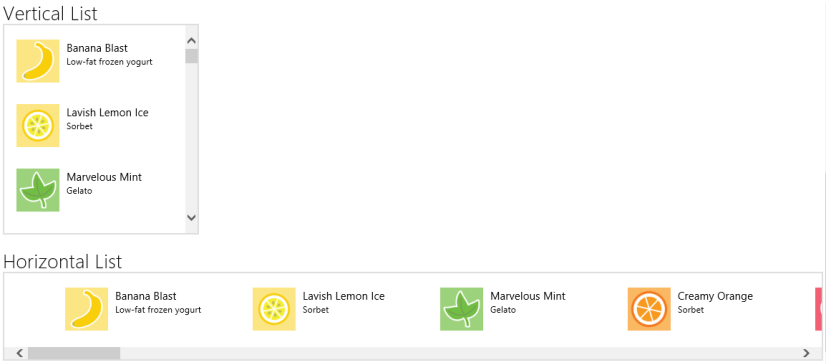


FIGURE 7-12 Vertical (default) and horizontal orientations for the `ListLayout`. I've again included the scrollbars here to show the panning direction more clearly.

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 screen shot. On a larger monitor where the control is wider, we'll 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' }
```

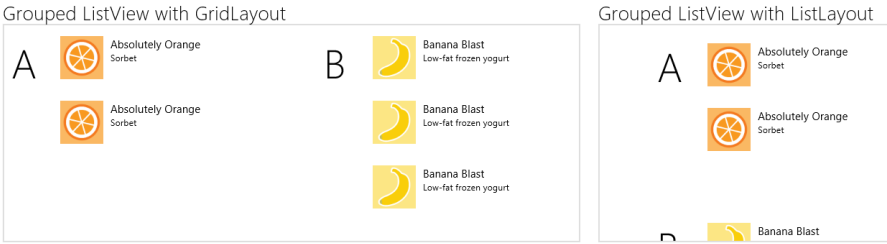


FIGURE 7-13 Using `groupHeaderPosition: 'left'` with `GridLayout` and `ListLayout`. With a right to left language, the left position will shift to the right. Compare this to Figure 7-3, and note that I made the controls a little bigger in CSS so that the items would show fully.

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.)

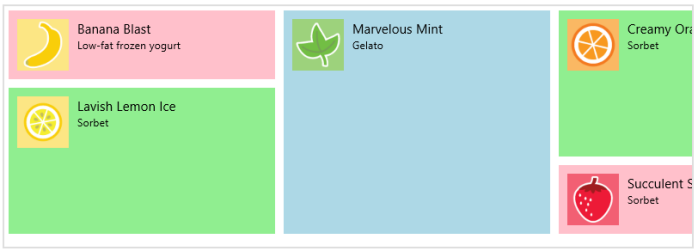


FIGURE 7-14 The HTML ListView item templates sample showing multisize items through cell spanning.

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 can mean that styles from the most recently loaded scenario are 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 there are three styles in the WinJS stylesheets that 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.

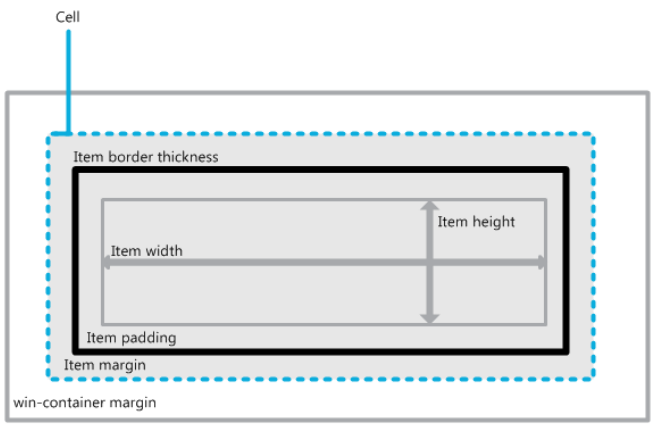
In the sample, we have items with the same widths but spanning one, two, or three rows, so we want to do the calculations for height. Using the first formula, *cellSize* is 80 (the height from `groupInfo`); and the top and bottom margins from `win-container` for a horizontal grid are both 5px, so *containerMargin* 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).

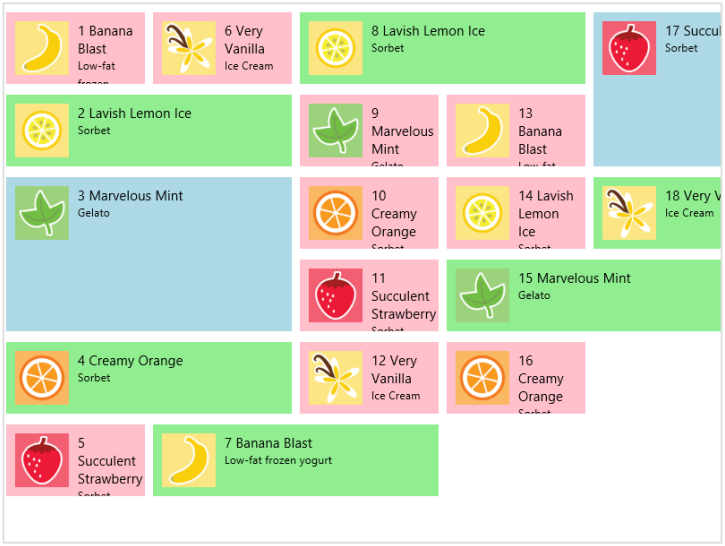


FIGURE 7-15 A horizontal `CellSpanningLayout` with varying item widths, showing infill of gaps.

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.

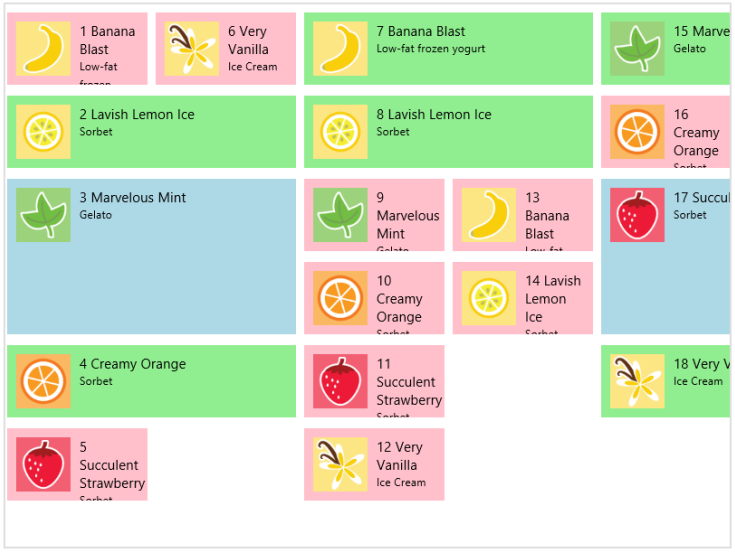


FIGURE 7-16 Using new columns in cell spanning on items 7 and 15.

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 there is one part that 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 image loading. 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.

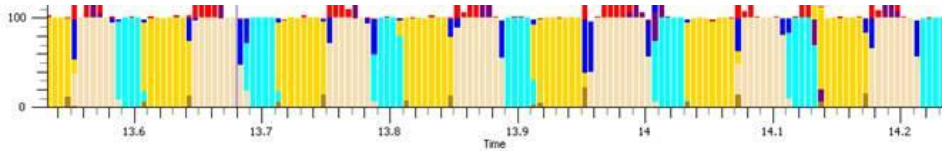
Our context for this discussion is the HTML ListView optimizing performance sample that demonstrates all these levels and allows you to see their respective effects (also using the Bing Search API data source that we've seen elsewhere, so you'll need your API key again). Here's an overview:

- 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 expansive visual operations, like loading images and 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 come to 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, 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 creating 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 there is a data source and an item template used to render each item in that source. 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.
- 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 only need a few reference points like 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 really 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 actually 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 they design 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 space, habits that hopefully make their movement more efficient and productive.

Good app design clearly follows the same principles, which is exactly why Microsoft recommends following consistent patterns with 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 of many years of research and investigation into what would really work 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 anatomy, 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—about 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 showed that users will actually 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 was similar to the silhouette but behaves differently), users must expend much more energy just 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 all my furniture around!

The bottom line is that there are very good reasons behind *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 in the first section of 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!)

The most important point to understand here is this: *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*. That is, 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 as well 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 work *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, they can close a view at any time). With multiple views, apps can, in other words, 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, in fact, 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 there some UI elements like the app bar and flyouts don't participate in layout; I'll cover these in other chapters. There are also auxiliary app pages that service contracts (such as Settings and Search) and exist outside your main navigation flow. These will employ the same layout principles covered in this chapter, but how and when they appear will be covered later.

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 done 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.)

As an example, 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 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 an app bar); Settings through the Settings charm; adding feeds, refresh, and navigation through commands on to 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?

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 LOB apps to Windows Store apps. Definitely worth a few minutes of your time to at least see what's available!

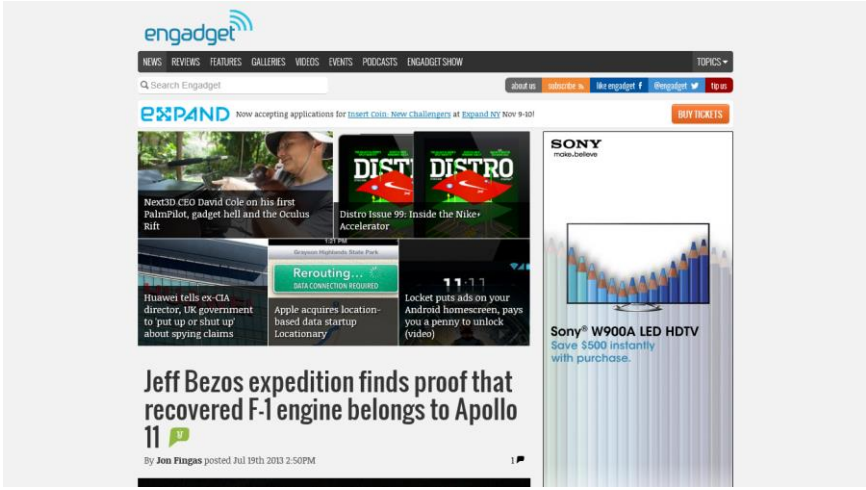


FIGURE 8-1 A typical desktop or web application design that emphasizes chrome at the expense of content.

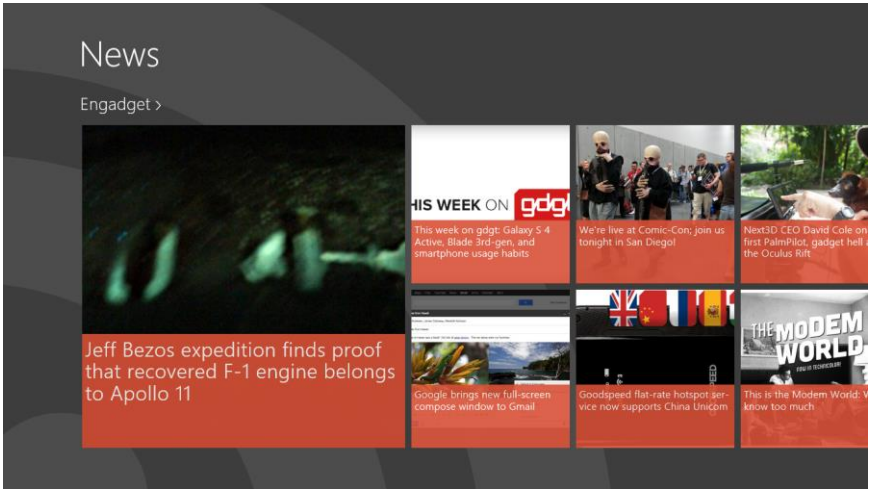


FIGURE 8-2 The Feed reader sample drawing on the same feed as the web app in Figure 8-1. Most of the chrome has disappeared, leaving much more space for content or visually-relaxing space.

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.

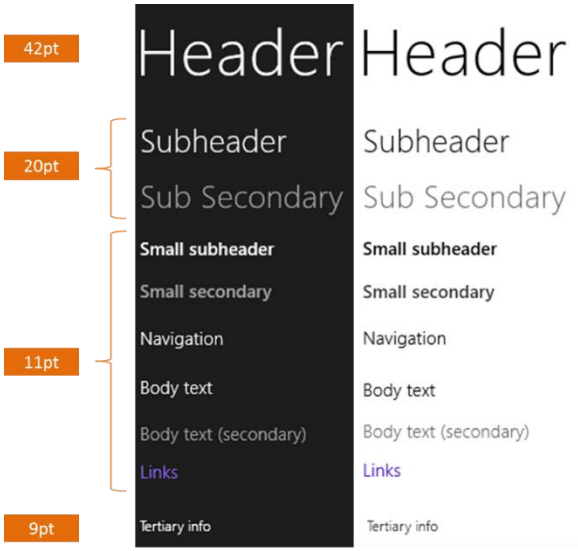


Figure 8-3 The typographic ramp of Windows Store app design, shown in both the ui-dark.css (left) and ui-light.css (right) stylesheets.

Within the body of content, then, 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." That is, 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, namely 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 in the next section.

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 there is also a dynamic relationship between pages and between content on a page. Keep these in mind with your designs, because they 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 very 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 simply 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. 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 number of 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 primarily for landscape and multimonitor scenarios). Portrait mode is very common with small tablet 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 (typically horizontally panning) to a portrait-oriented model (typically vertical 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 allowable 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 full screen.
- 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]
- An app can specify Supported Rotations in the manifest—these specifically affect whether an

[70] The `tryUnsnap` API that existed in Windows 8 is deprecated in Windows 8.1 and does nothing.

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 in the previous section, 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.

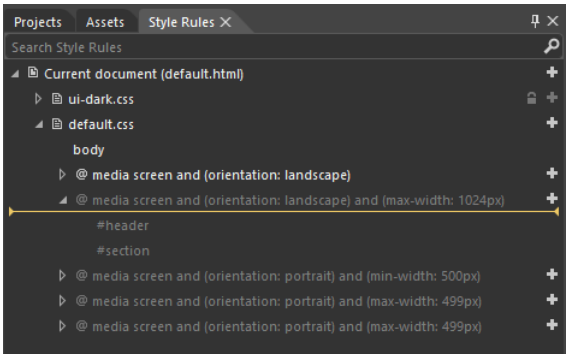
Responsive design has two parts. First are those things you can handle declaratively in CSS:

- Place all size-independent and default styles in CSS 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:



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 means is 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 2560px by 1440px. 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, you 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 that 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.

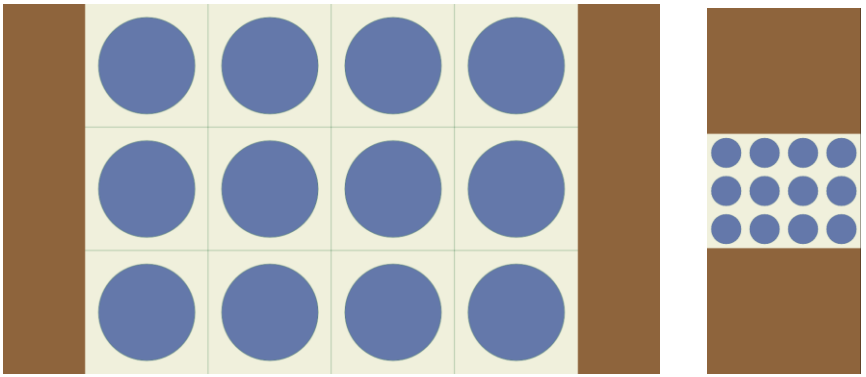


FIGURE 8-4 Fixed layout scaling with a simple CSS transform, showing letterboxing on a full-screen 16:9 ratio display (left) and in a narrow 320px view (right).

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.

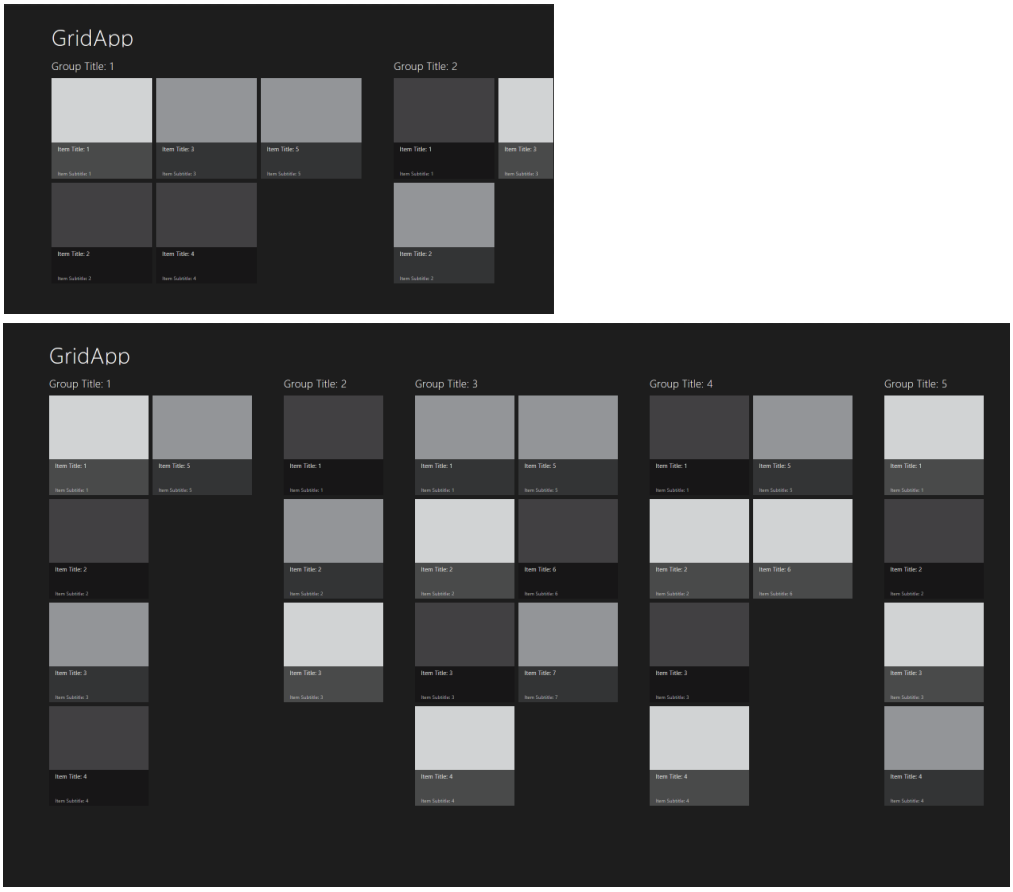


FIGURE 8-5 Adaptive layout in the Grid App project template shown for a 1366x768 display (top) and a 2560x1440 display (bottom).

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 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 really 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%`.

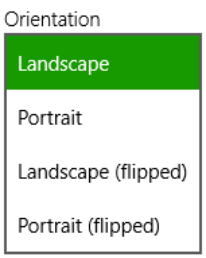
Finally, let me again recommend the Guidelines for scaling to screens topic in the documentation, which goes into many more design questions, such as:

- 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:



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, make sure the app you want is 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 only about 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`.

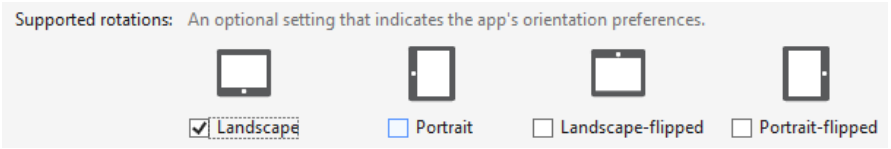
A more specific value comes from the `Windows.Graphics.Display.DisplayInformation` class (obtained via `DisplayInformation.getForCurrentView()`) and its `currentOrientation` property. This identifies one of the four rotation quadrants that are possible for a device relative to its native orientation: `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:



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 basically ignored). The rotations that the simulator performs happen on the level of the display driver, which is different than the WinRT display orientation. To really 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 (reminder: the videos are also available with the companion content). 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 really 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 really 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 the "Graphics that Scale Well" section 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 match, 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.

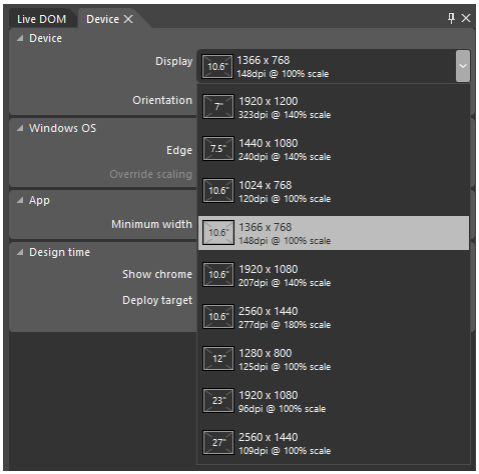
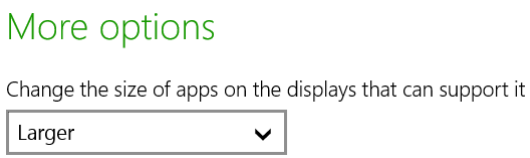


FIGURE 8-6 Options for display sizes and pixel densities in Blend's Device tab.

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.



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. There is also a `logicaldpichanged` event that 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 the other graphics 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 can then just refer to the base name in an element or in CSS, 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 Chapter 16, "Alive with Activity," in the section "Using Local and Web Images" 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 there are certainly app scenarios where a single app wants to manage multiple independent views, possibly and oftentimes on separate monitors, and the user has the ability to rearrange and resize 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 only share data 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 used 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. Within the secondary view, it will receive message 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.

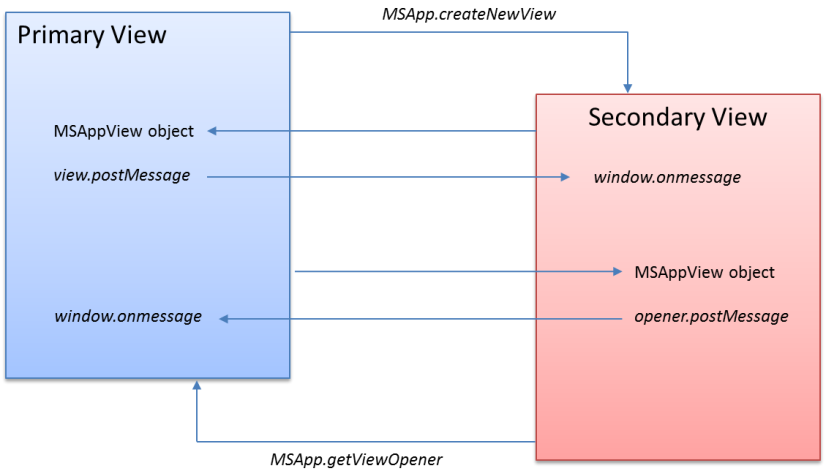


FIGURE 8-7 A primary view creates a secondary view through `MSApp.createNewView`. The secondary view retrieves a reference to the primary view through `MSApp.getViewOpener`. The result in both cases is an `MSAppView` object whose `postMessage` call sends a message to the other view.

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 work 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 didn't show 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, then, 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, then in the main app switch to a secondary view in the space. Then you'll see two secondary views at once. If you switch to the main app in either, you'll then 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 really isn't 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 in the next main section. 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 8-4. 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 added headings for each section, we'd 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 then 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*:

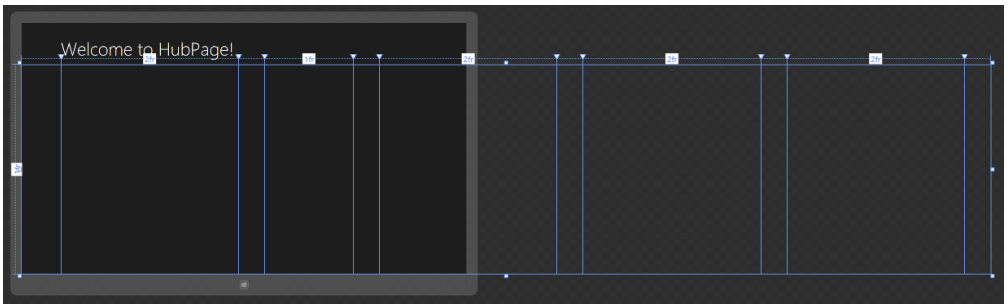
```
<section aria-label="Main content" role="main">
    <div class="hubSections">
    </div>
</section>
```

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 as 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 then 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 say, 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), then bi-directional panning can go into effect, but the user would typically have to try to make this happen.

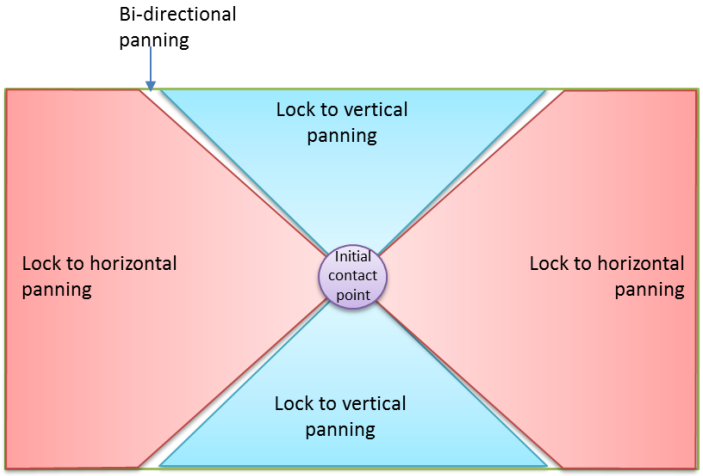


FIGURE 8-9 The concept of rails, where the white areas indicate regions where the touch point is moved equally along both axes and bi-directional panning can happen. Note that railing is determined along 45-degree lines from the touch point and has nothing to do with the shape of the region itself.

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 such as ARM devices.

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, then, 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. Note that we 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. Do note that 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. Do note also that 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, we might as well 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 in the previous section, 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. 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 then 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 children 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.

There are two places to find reference code for the Hub: 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 different 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, then, 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. There are four sections labeled Images, ListView, Video, and Form Controls that describe exactly what kind of content that section contains. In this particular case, the headers are static and there is no 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!

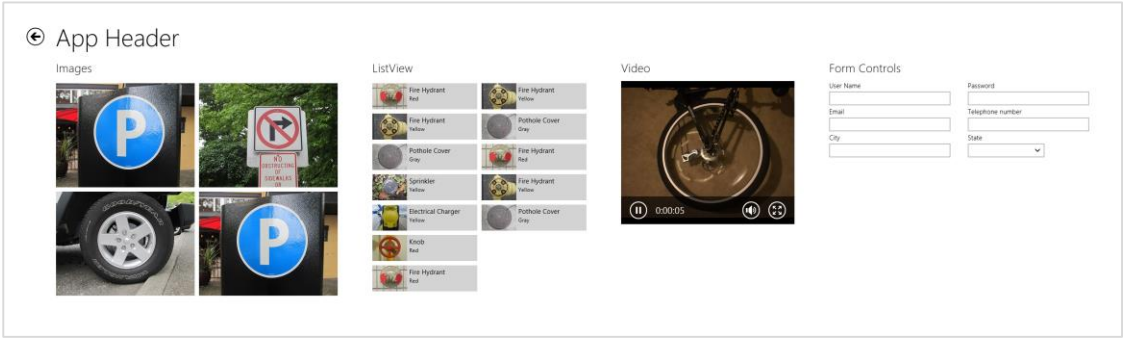


FIGURE 8-10 The full layout for scenario 1 of the HTML Hub Control sample.

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 `myData` property defined within 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 in the next section 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.

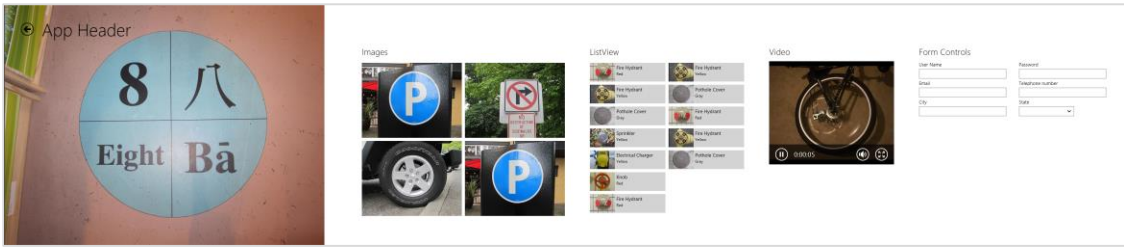


FIGURE 8-11 The full layout for scenario 2 of the HTML Hub Control sample. The hero image is just another HubSection added to the beginning of the Hub control, where the image, width, and margins are defined in CSS. The header for that section is also hidden via CSS.

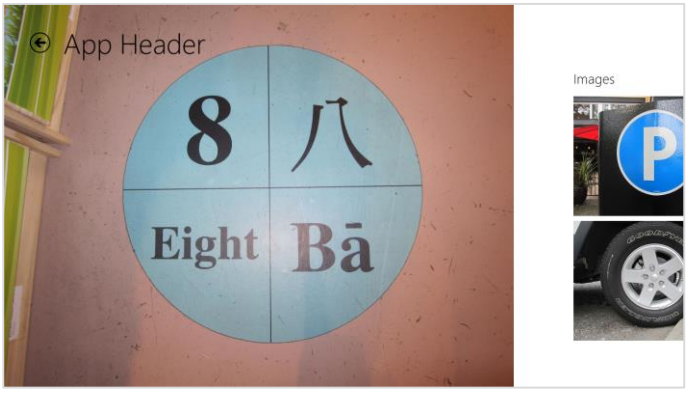
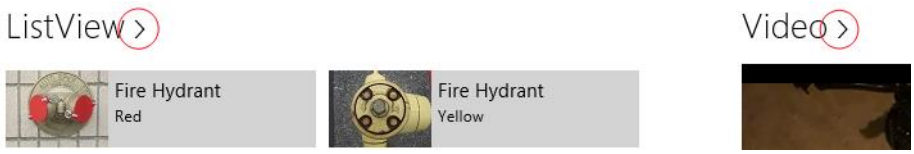


FIGURE 8-12 The appearance of the hero image of scenario 2 on a 1366x768 display.

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>
```

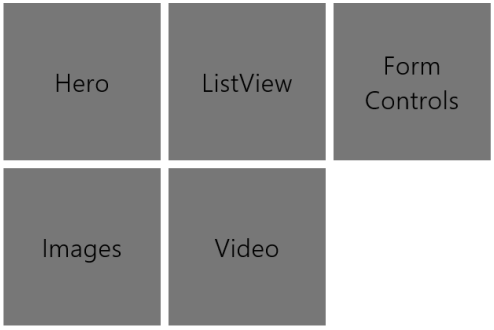
This makes it possible to navigate to the header with the Tab key, and it adds chevrons to the header text (circled in red):



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) then 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 the sections out 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.

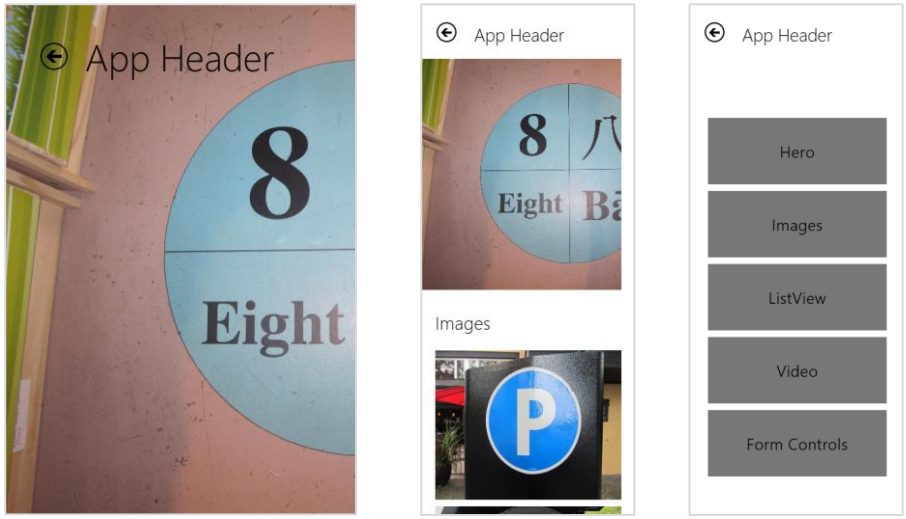


FIGURE 8-13 Scenario 5 at 500px wide (left), 320px wide (middle) with the vertical Hub layout, and 320px wide zoomed out (right) with a vertical ListView.

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 window 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

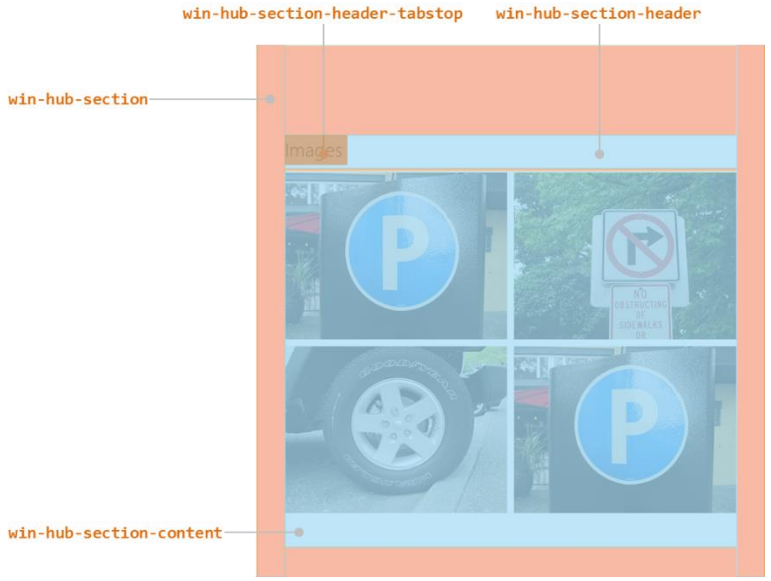
Styling the hub control is a matter of styling its various parts, like we've seen in earlier chapters with the ListView, the FlipView, and the ItemContainer controls. To lay the `div` structure out clearly, a bit of markup like this:

```
<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 would be `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 really 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. Really 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 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 the same 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 is also used to demonstrate 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, then 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`. Then 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, so 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. In this section, then, let's look 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 this section 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 really 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:

[75] At the time of writing, the `-ms-*` prefixes on these styles were no longer needed but are still supported.

> *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: -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 and in the project templates, as we saw with the Fixed Layout template earlier. 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

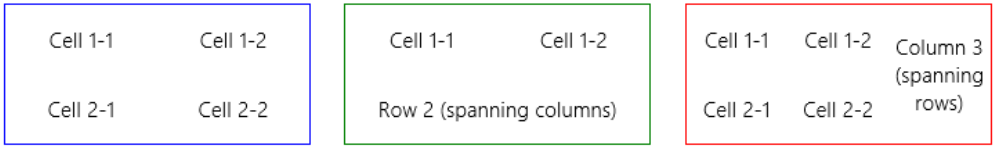
Just as the flexbox has both block level and inline models, there is also an inline grid: `display: -ms-inline-grid`. Unlike the block level grid, the inline variant allows you to place several grids on the same line. This is shown in the InlineGrid example for this chapter, where we have three `div` 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);
});
```

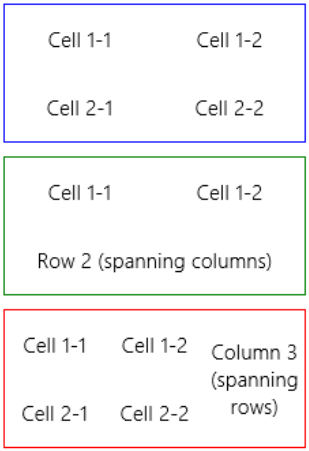
[76] If you're accustomed to the `-ms-box*` styles for flexbox, Microsoft aligned to the W3C specifications as they existed at that time for Windows 8 and Internet Explorer 10. As of Windows 8.1 and Internet Explorer 11, the `[-]ms` prefixes are apparently no longer needed, but as of this writing they are still what's recognized by Visual Studio's IntelliSense, so I'm showing them as such here.

```
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;
}


/* default.css */
.inline {
    display: -ms-inline-grid;
}

.block {
    display: -ms-grid;
}
```

When using the inline grid, the elements appear as follows:



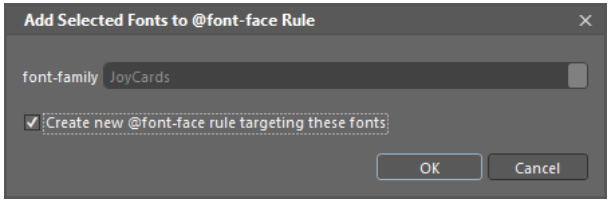
When using the block level grid, we see this instead:

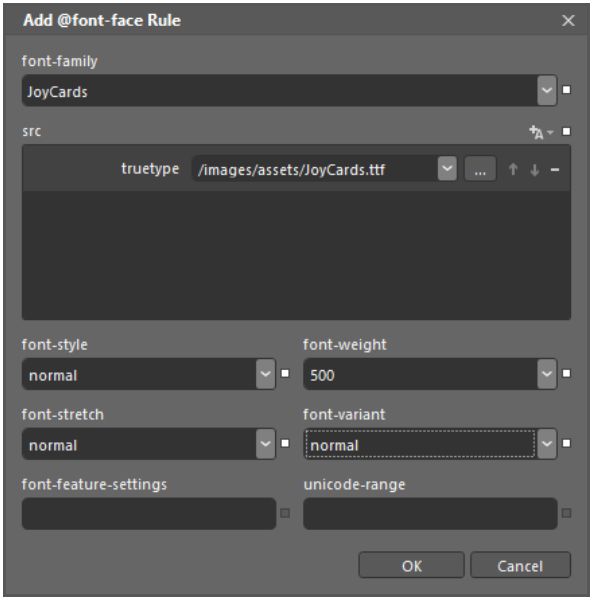


Fonts and Text Overflow

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. In the Windows SDK there is a very helpful CSS typography JS sample that 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:





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;
}
```

This will make 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. So at present you'll need to shorten the text and insert ellipsis manually if it spans multiple lines.

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 coming online and 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.

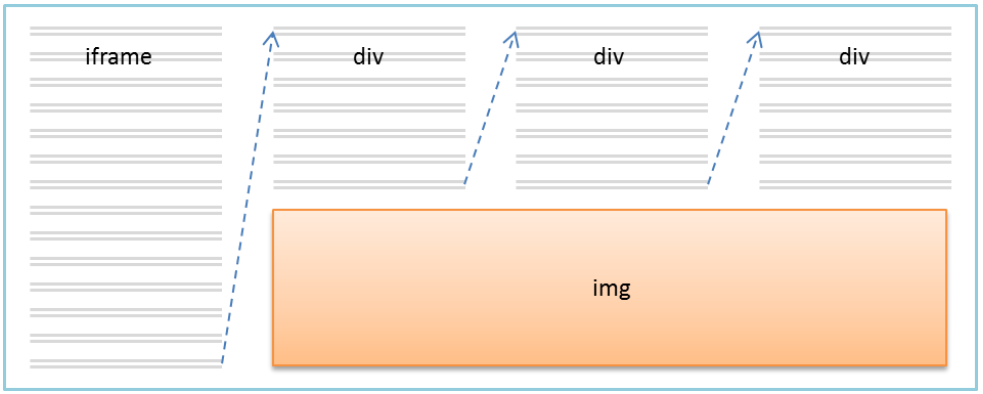


FIGURE 8-14 CSS region chains to flow content across multiple elements.

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; the Internet Explorer team technically owns them and has not yet updated them for 8.1, but they work just fine.) In all cases, though, be aware that styling for regions is limited to properties that 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 a 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 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 generally 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.

The app 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. Settings are specifically handled through the Settings charm, as we'll see in Chapter 10, "The Story of State, Part 1."

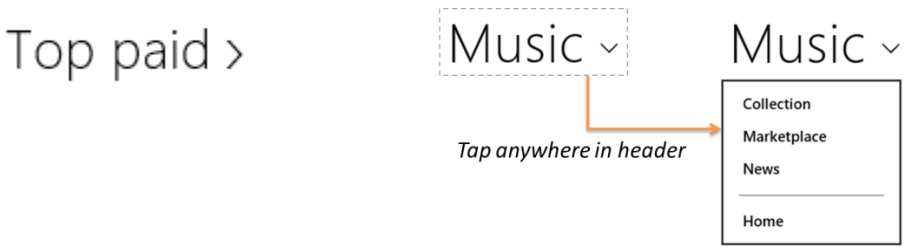
An important point with all of these command controls is that they don't participate in page layout: they instead "fly out" and remain on top of the current page. This means we thankfully don't need to worry about their impact on layout…with one small exception that I'll keep secret for now.

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

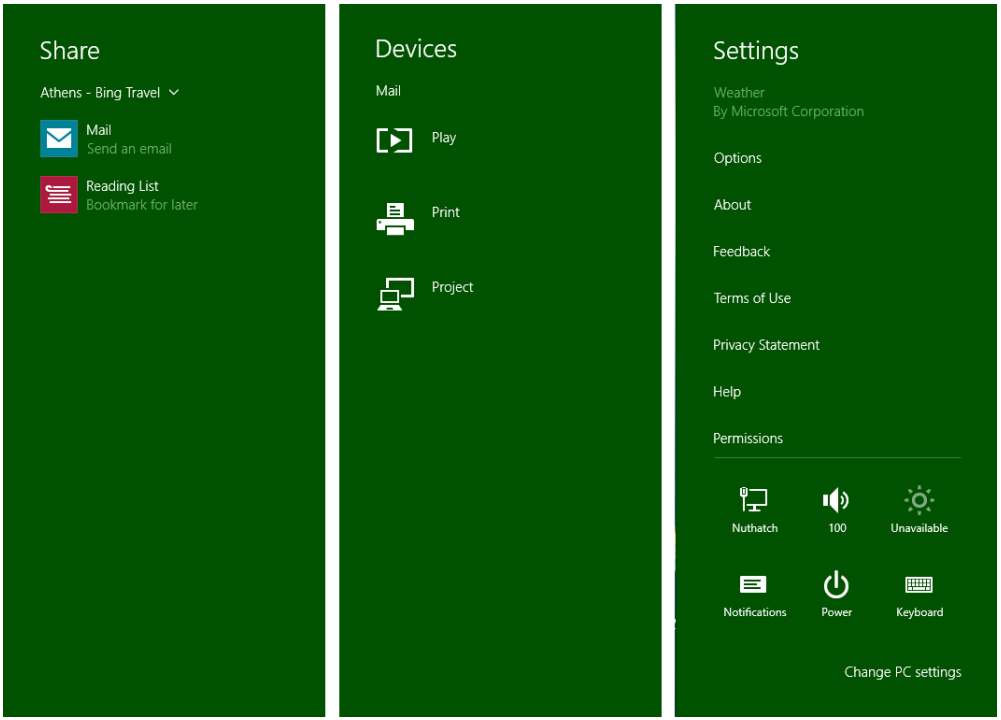
The placement of commands is really quite central to app design. Unlike the guidelines—or lack thereof!—for desktop application commands, which has resulted in quite a jumble, the Windows Developer Center offers two rather extensive topics on this subject: Commanding design and Laying out your UI. These 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):

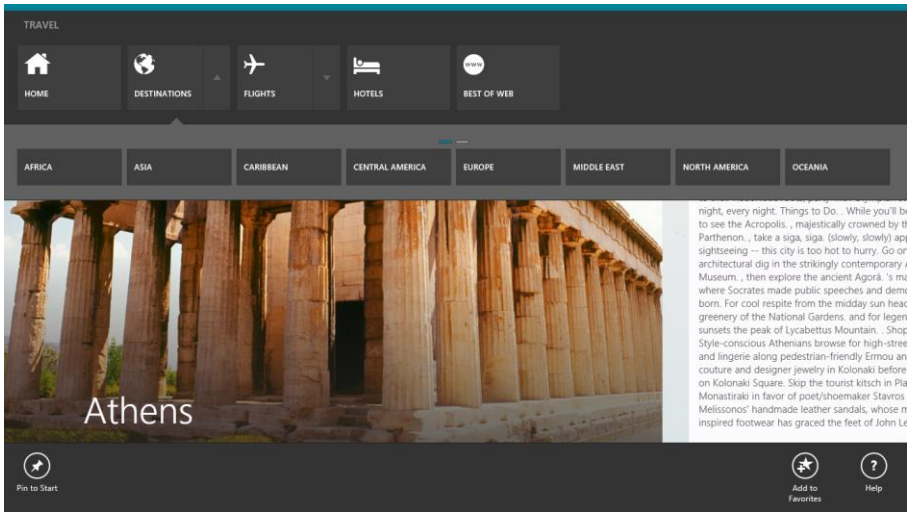


- Use *Charms* for common app commands where possible. That is, instead of supplying individual commands to share with specific targets such as email apps, your contacts, social network apps, and the like, use the Share charm. Instead of supplying your own Print commands, rely on the Device charm. And 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.

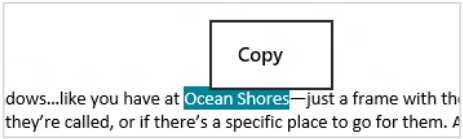
An exception to this guidance for charms is Search (not shown). Apps that wish to have their content participate in global search, which includes web results, can work with the charm. Apps that need only provide in-app search capabilities should provide appropriate controls on-canvas or in an app bar or nav bar.



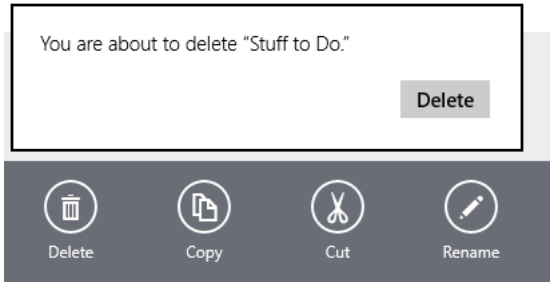
- Commands that can't be placed in Charms and don't need to be on the app canvas are then placed within the *nav bar* and *app bar*, as shown below in the Travel app. Both bars are invoked together by swiping in from either the top or bottom edges and are the closest analogies to traditional menus:
    - 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.



- *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.



- 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 SkyDrive app when using the Delete button:



- For blocking events that are not related to a user command but that affect the whole app, use a *message dialog*. A message dialog effectively 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," typically apply only to background apps.

- 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 and checklist for 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, begin to populate 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 can 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 and checklist 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 first appears when the charm is invoked. These commands then 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 nab 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 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. Since 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.

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 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 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'` you want 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

[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 much 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 since the `icon` property ultimately resolves to a string.

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.

- `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 usual 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 like this from anywhere in your app, and you can certainly listen to any other events you want to, especially when doing custom 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 a time 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 if necessary.

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, then—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` value 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 sample on the Windows Developer Center 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. Since 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 return 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:'home2', extraClass: '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);
    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 2013, 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 directly; instead, you should 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:

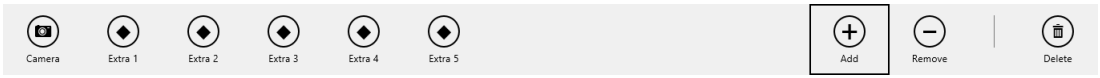
```
<div id="customColorAppBar" data-win-control="WinJS.UI.AppBar" class="customColor" ...>
```

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 {
```

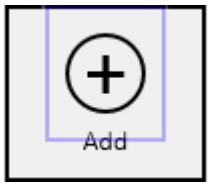
```
    background-color: rgba(90, 200, 90, 0.13);
    border-color: rgb(90, 200, 90);
}
```

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";
}
```

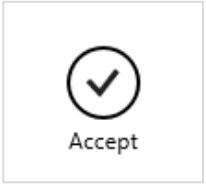
To provide graphics of your own, do the following for a 100% resolution scale:

- 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 demonstrates 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`; it would be more efficient to 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!):

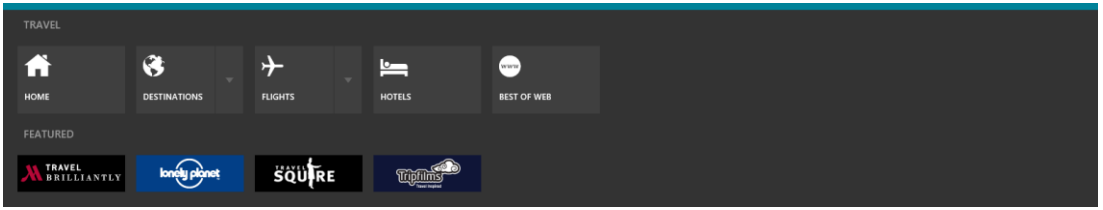
The point, though, is that a custom layout app bar is essentially nothing more than a container for whatever arbitrary content you want to place there, and that content is entirely under your control. The app bar in these cases is just taking care of showing and hiding that content at appropriate times 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.

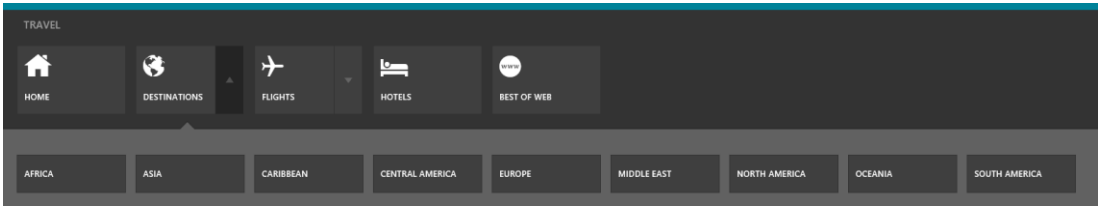
Nav Bar Features

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:

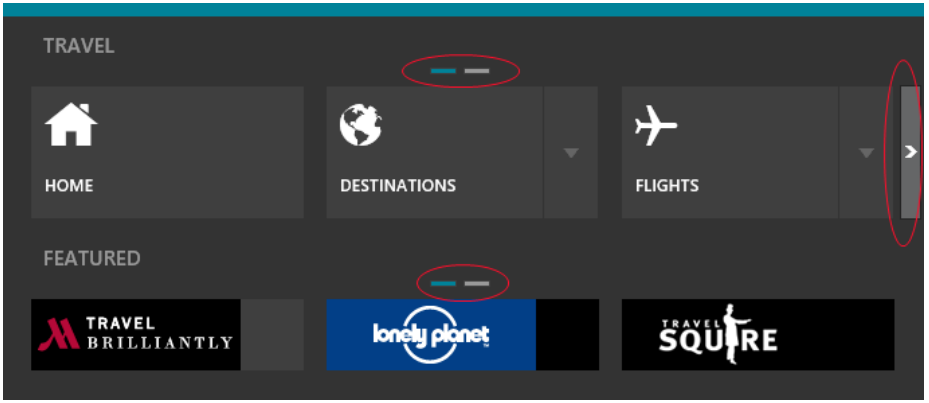


Tapping the down arrow next to Destinations or Flights opens up the second level in the navigation hierarchy, where again we see all the options when the screen is wide enough:

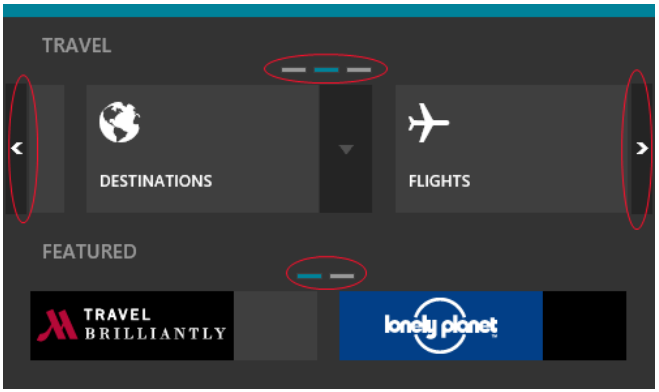


Notice how the down arrow by Destinations changed to an up arrow to indicate that this second level can be collapsed back to its previous state.

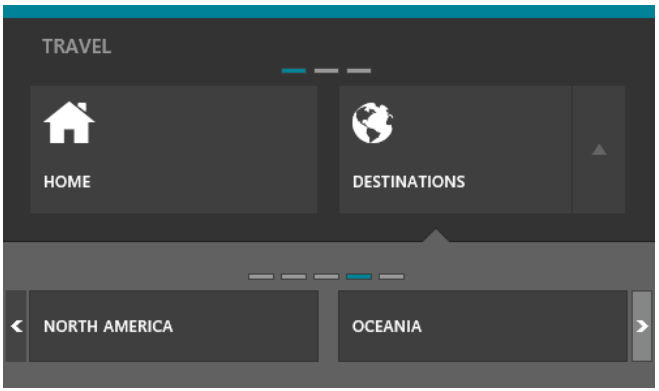
So far so good. Now if we resize the app to a narrower view, we'll clearly need a way to pan the commands left and right. In this case the NavBar provides panning arrows a'la FlipView (circled on the right side), as well as page indicators to show where you are in the list (circled in the middle):



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>
```

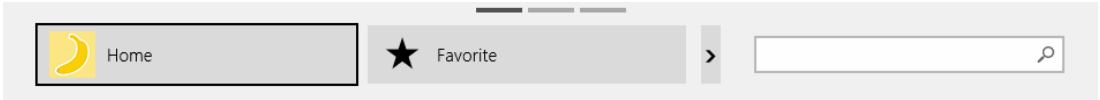
The result of this, in a view that's narrow enough to see the page indicators, is as follows:

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.

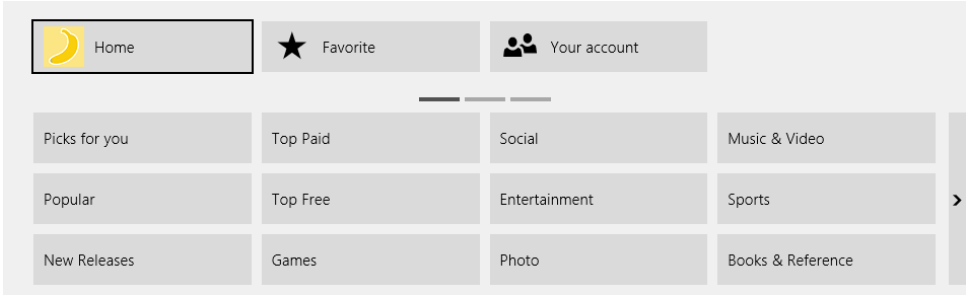
Besides `label`, `icon`, and `location`, the `NavBarCommand` has these additional properties:

- `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, which we'll cover shortly.

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.

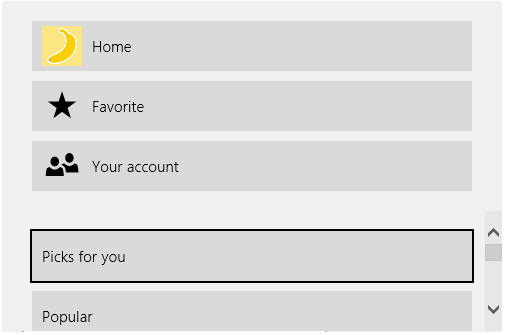
The `NavBarContainer` also supports these additional members:

- `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 will 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();
        }
```

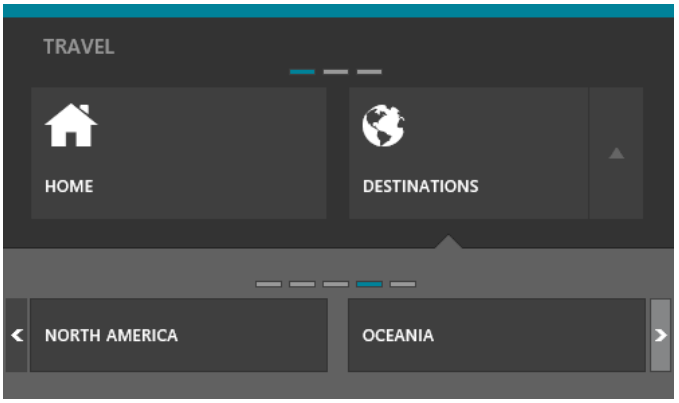
```
    function go() {
        flyout.removeEventListener('beforehide', go);
        navbarCommand.splitOpened = false;
    }
});
```

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:



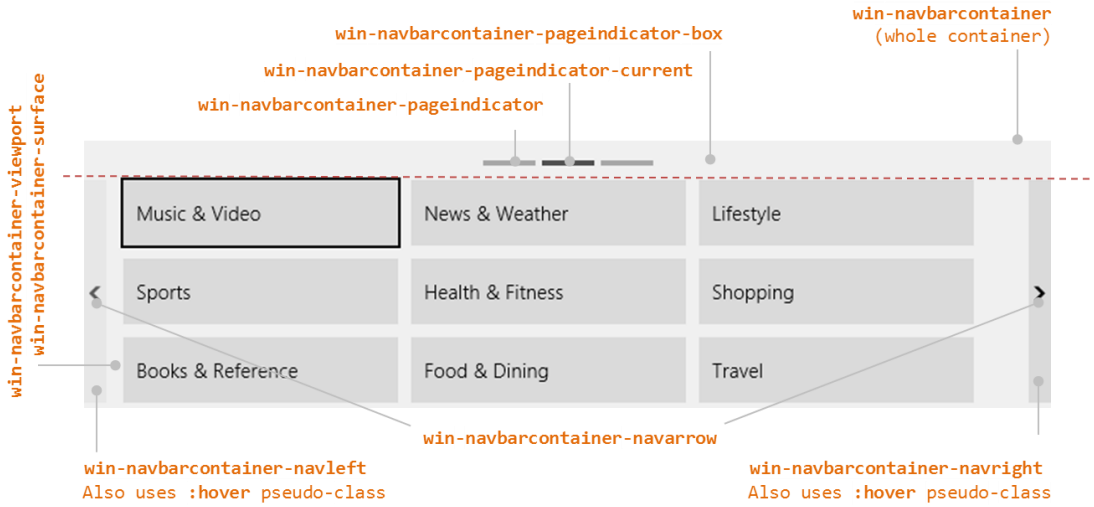
These appearances are easy to control, as the `NavBar`, `NavBarContainer`, and `NavBarCommand` classes all provide the usual `win-*` style hooks for their different parts.

> **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:



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 16-5 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 (on the next page). 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.

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.

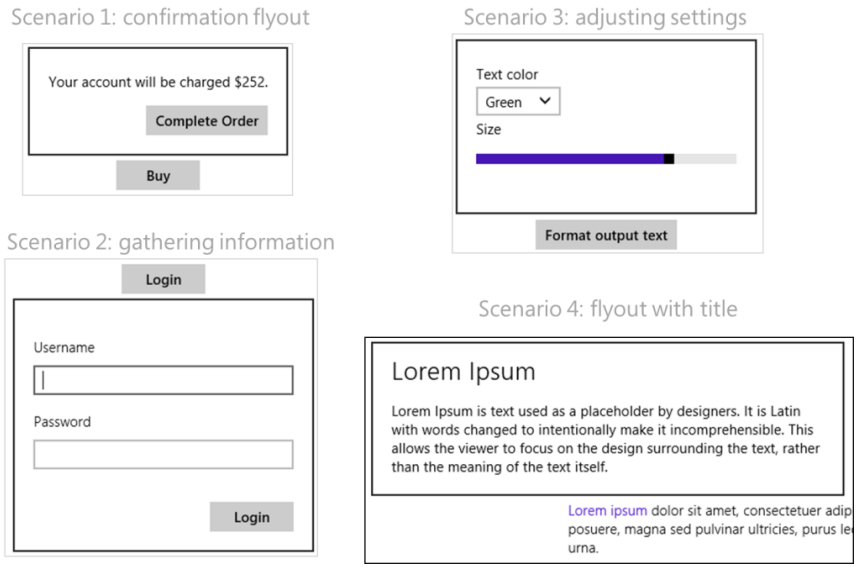


Figure 9-1 Examples of flyout controls from the HTML flyout control sample.

I'll again encourage you to read the Guidelines and checklist 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):

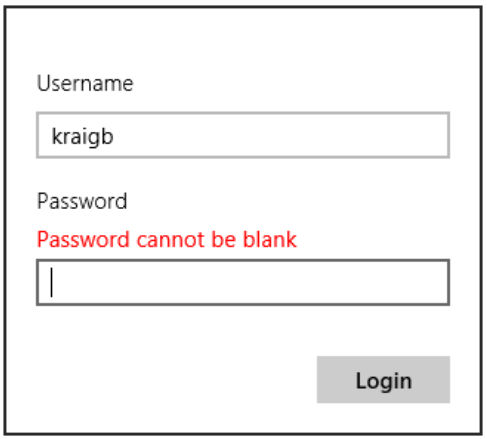
```
<div id="loginFlyout" data-win-control="WinJS.UI.Flyout" aria-label="{Login flyout}">
    <form onsubmit="return false;">
        <p>
            <label for="username">Username <br /></label>
            <span id="usernameError" class="error"></span>
            <input type="text" id="username" />
        </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.

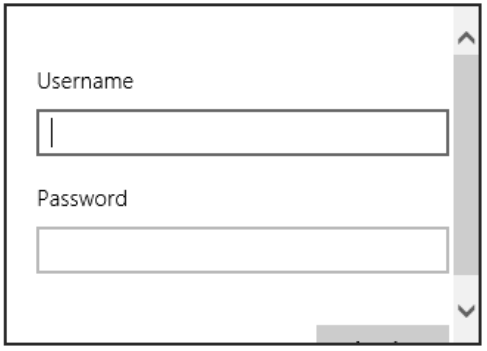
Handling the login controls in scenario 2 is pretty much the same, with some added code to make sure that both a username and password have been given. If not, the Login button handler displays an inline error and sets the focus to the appropriate input field:



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:



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!

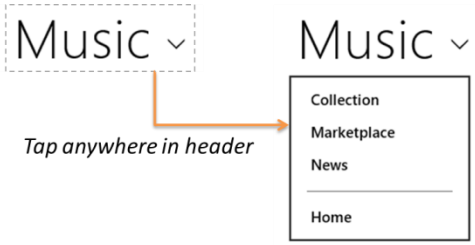
We've already seen how to use a flyout menu from an app bar command, which is covered in scenario 6 of the HTML flyout control sample (see the earlier "Command Menus" section). Another primary use case is to provide what looks like drop-down menu from a header element, covered in scenario 5. Here (see 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):

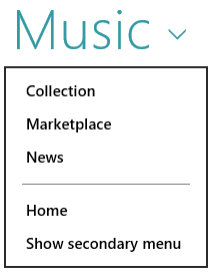


The individual menu commands are just `button` elements themselves, so you can attach `click` handlers to them as you need. As with the app bar, it's best to use the menu control's `getCommandById` to locate these elements because it's more direct than `document.getElementById` (as the SDK sample uses…sigh).

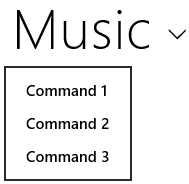
To see a secondary menu in action, try adding the following *secondaryMenu* element in html/header-menu.html before the *headerMenu* element and adding a `button` within *headerMenu* whose `flyout` property refers to *secondaryMenu*:

```
<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 and checklist 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 yourself 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. See 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 and checklist 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).

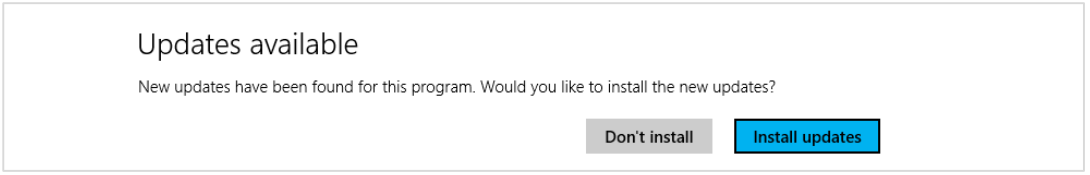
The `options` come from the `MessageDialogOptions` enumeration where there are only two members: `none` (the default, for no special behavior) and `acceptUserInputAfterDelay` (which causes the message dialog to ignore user input for a short time to prevent possible clickjacking; this exists primarily 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.

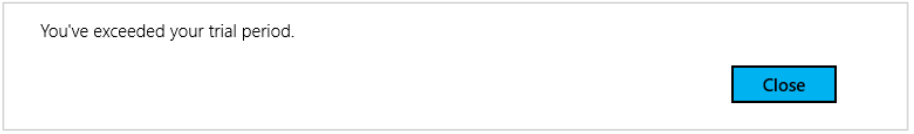
Finally, once the dialog is configured, you display it with a call to its `showAsync` method. Like the context menu, the result is the selected `UICommand` object that's given to the completed handler you provide to the promise's `then`/`done` method. Typically, you don't need to obtain that result because 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:

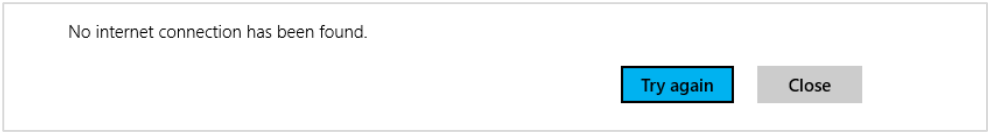


Scenario 2 shows the default Close command with a message and no title:



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:



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

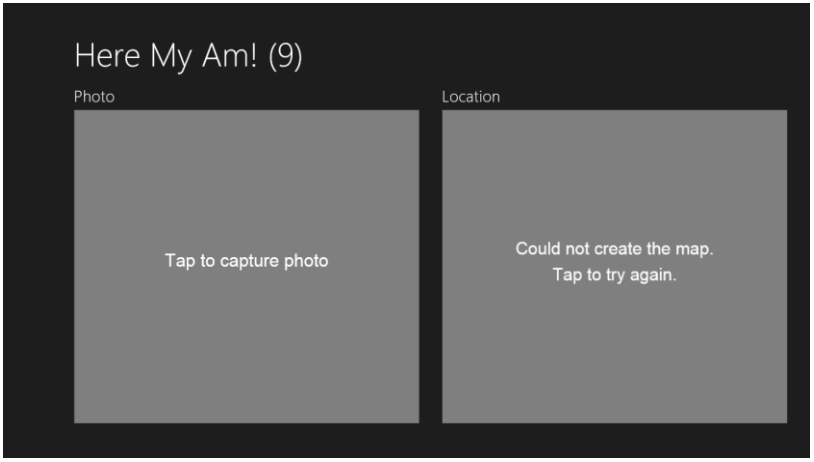
refresh the location on the map other than dragging the pin; that is, the geolocation API is used only on startup.

- 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. Since 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.)

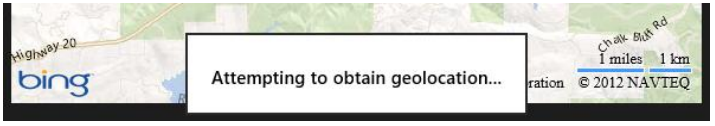
For the third problem—letting the user know that geolocation is trying to happen—we can show a small flyout message just before attempting to call the WinRT geolocator's `getGeopositionAsync` call. The flyout is defined in pages/home/home.html (our page control) to be centered along the bottom of the 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):



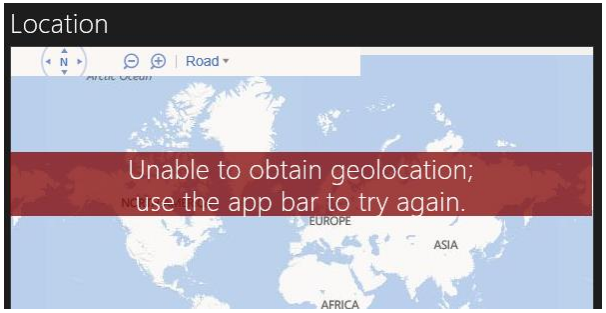
The next piece is to notify the user when obtaining geolocation fails. We could do this with another flyout with a Retry button, or with an inline message as below. We would *not* use a message dialog in this case, however, because the message could appear in response to a user-initiated refresh action. 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%;
}
```



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 completely 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";
    //...
},
```

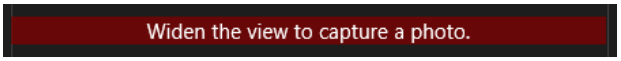
We can reuse the `.errorOverlay` class to add a similar message on top of the Photo area when the view width falls below 500px. In this case we need another element in the *photoSection* `div`:

```
<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:



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

It would be very interesting 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, your 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 > SkyDrive > 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 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, then, 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, providing 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, then, 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 Store-acquired apps. Thus they included Settings on the Charms bar alongside the other near-ubiquitous Search, Share, and Devices commands. For one thing, the Settings charm eliminates the need for users to remember where a particular app's settings are located, and apps don't need to wonder how, exactly, to integrate settings into their overall content flow and navigation hierarchy. That is, 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, then, an app's state and its Settings UI are intimately connected, as we will 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 delve into the subject more deeply in Chapter 11, "The Story of State, Part 2: User Data, Files, and SkyDrive."

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. Then we can 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 such data, 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 aren't part of 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` event.

- **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.

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 > SkyDrive > Sync Settings > Other Settings > Passwords, in which case the credential locker still maintain 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, there are a two other events that can affect app state that are not under an app's control (that is, outside of the running app or a background task):

- **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. The remainder of the chapter, starting with the "Settings Pane and UI" section, is concerned with how to surface those parts of your state that are user-configurable.

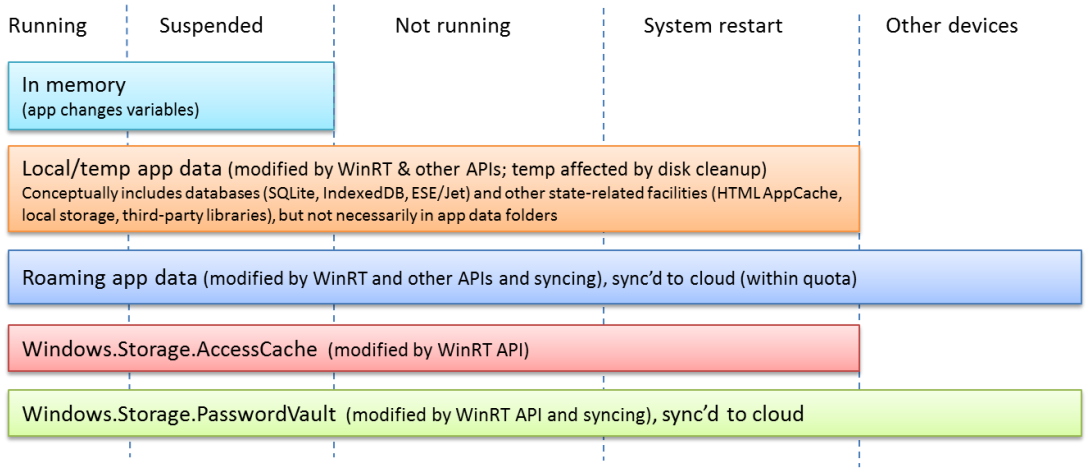


FIGURE 10-1 Different forms and locations of app data, how they persist across app lifecycle events, and the APIs that modify them.

App Data Locations

Now that we understand what kinds of information make up app state, let's delve deeper into the question of, "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. Since 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, as is shown in the figure, you'll see a value like *{"lastSuspendTime":1340057531501}*. The Sports app (along with News, Weather, etc.) 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.

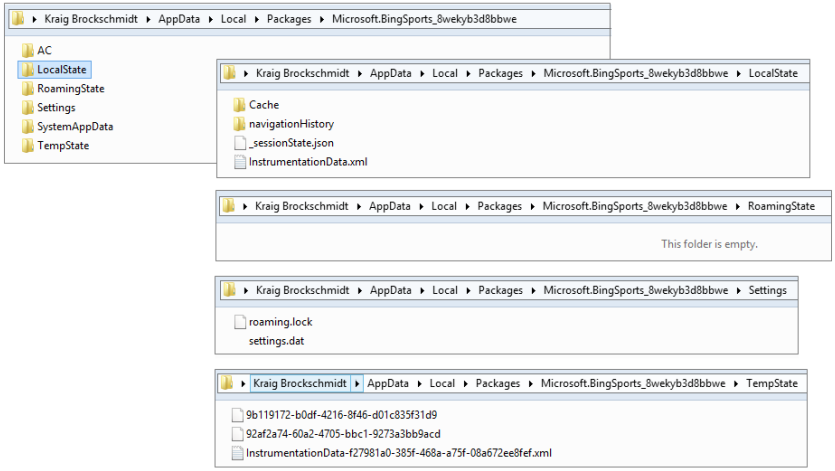


FIGURE 10-2 The Sports app's AppData folders and their contents.

> **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 can 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 for which you need to use `removeEventListener` as described in Chapter 3 in the "WinRT Events and removeEventListener" section.)

- `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, and roaming). The locality is identified with 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 the "Debug Output, Error Reports, and the Event Viewer" section. 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 settings 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 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 `IMapView` object through which you can enumerate its contents. The `values` collection, on the other hand, is a WinRT `IPropertySet` 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 the section "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, just that they 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 there is some built-in batching to prevent excessive disk activity when you change a number of values all in a row. In any case, you really 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

[79] If you like, you can maintain your own versioning system within particular files or settings. I would recommend, however, that you avoid 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.

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 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 can use 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 the "Background Tasks and Lock Screen Apps" section toward the end.

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`—that 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 is encapsulated within `createObjectURL`. There are other ways to use a blob 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 the "Q&A on Files, Buffers, Streams, and Blobs" section later in this chapter.

For working directly with files, now, 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 can be 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 can 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 There are some file extensions that 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.

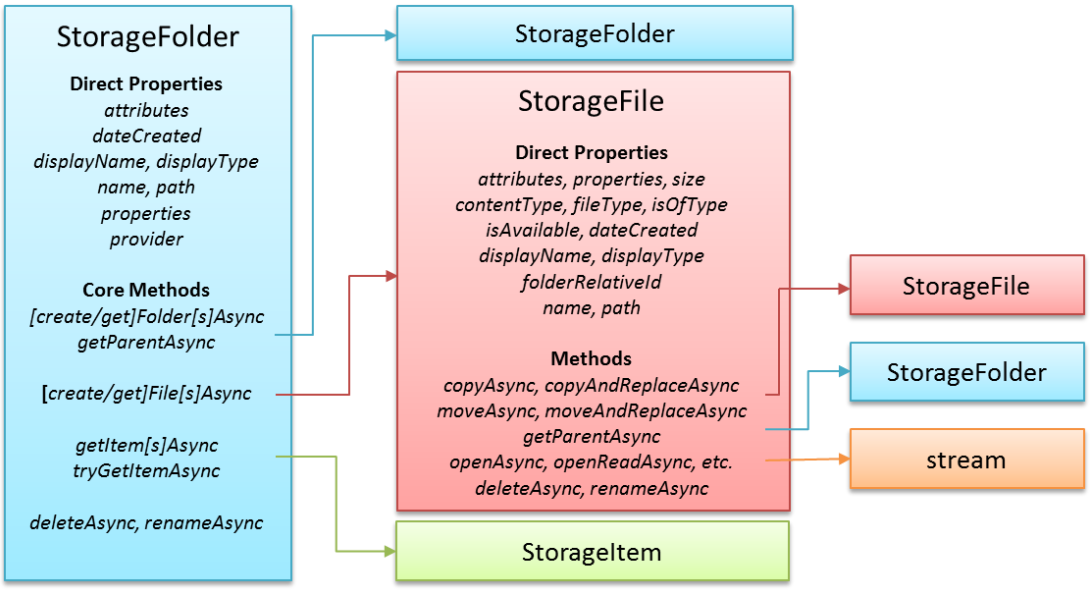


FIGURE 10-3 Core properties and methods of `StorageFolder` and `StorageFile` classes and the objects they produce.

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 whatever location, and whatever app is associated with that file type will receive 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. Pathnames also will not work for files provided by another app through the file picker, because the `StorageFile` object might not, in fact, refer to anything that actually exists on the file system.

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 that 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 the transactioning. 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 the static methods, createStreamedFileAsync, createStreamedFileFromUriAsync, replaceWithStreamedFileAsync, and replaceWithStreamed-FileFromUriAsync. These provide a StorageFile that you typically pass to other apps through contracts as we'll see more of 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 now, 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 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 could 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, 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 (see the "Tip" below).

> **Note** If you use a transacted output stream, you also need to call its `flushAsync` method in the chain after `DataWriter.storeAsync`.

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 very 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 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 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 available for *only* your AppData folders and not for the file system more broadly. For areas outside app

[80] If you're curious as to why async methods like `readText` and `writeText` don't have *Async* in their names, this was a conscious 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.

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 very 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 actual 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 you'll need to employ either a third-party JavaScript library or you can 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 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 wanted 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, then, 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 results later on. So technically, the file isn't actually "open" until those results are delivered.

Those results, 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" *ala'* `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 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 a memory allocation, it means feeing 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 that 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`; there are also the `IInputStream` and `IOutputStream` interfaces that 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` that 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, then, you call `FileRandomAccessStream.getInputStreamAt` on the source (for reading) and `FileRandomAccessStream.getOutputStreamAt` on the destination (for writing), then 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 then has just one method, `openReadAsync`, whose result is an `IRandomAccessStreamWithContentType` (the same thing as an `IRandomAccessStream` with an extra string property `contentType` to identify its data format).

To sum up, then, remember the difference between sequential streams (input or output) and random access streams, and that streams are just ways to talk to the bits (or bytes) that exist in some

backing entity like a file or memory. 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 section 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!), "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 in the first place (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()`,then 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 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 in the "FileIO, PathIO, and WinJS Helpers" section earlier 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, then, 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) {
        //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);
```

[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.

```
        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, then 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` at all 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 to not burden you when you don't.

Question #5: What are `MSStream` and `Blob` objects in HTML5 and how to 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 very 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 then 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 really remember where they left off. That's a decision you need to make for your own app and 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 its 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 write save your session variables whenever you like (including within `checkpoint`). You then restore it 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 really fouled up, but just uninstalling your app will clear all that out 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 reset `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`, `sessionStorage` can still be useful. You can also encode information into URIs as 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 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 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 the section "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 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 there are some app and systemwide limits 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.)
- 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 given 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" />
```

Beyond IndexedDB there are a few other database options for Store apps. 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 the subject provides many details for Windows Store apps.

[82] It's a very robust solution—it's apparently used to operate commercial airliners like the massive Airbus A380.

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 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 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 APIs are explained in the "Custom Queries" section of Chapter 11.) 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 as well, 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 to do all your work, because the library handles the details of making the necessary HTTP requests.

Finally, SkyDrive is an option for working with cloud-based files. We'll talk more about SkyDrive 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 SkyDrive, FaceBook, 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 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 really 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 SkyDrive 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 actually 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 only aware of 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 really 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.

- 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 there are multiple *state* versions 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 there are multiple versions in use by the same Microsoft account. Only when all instances of the app have been updated 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.

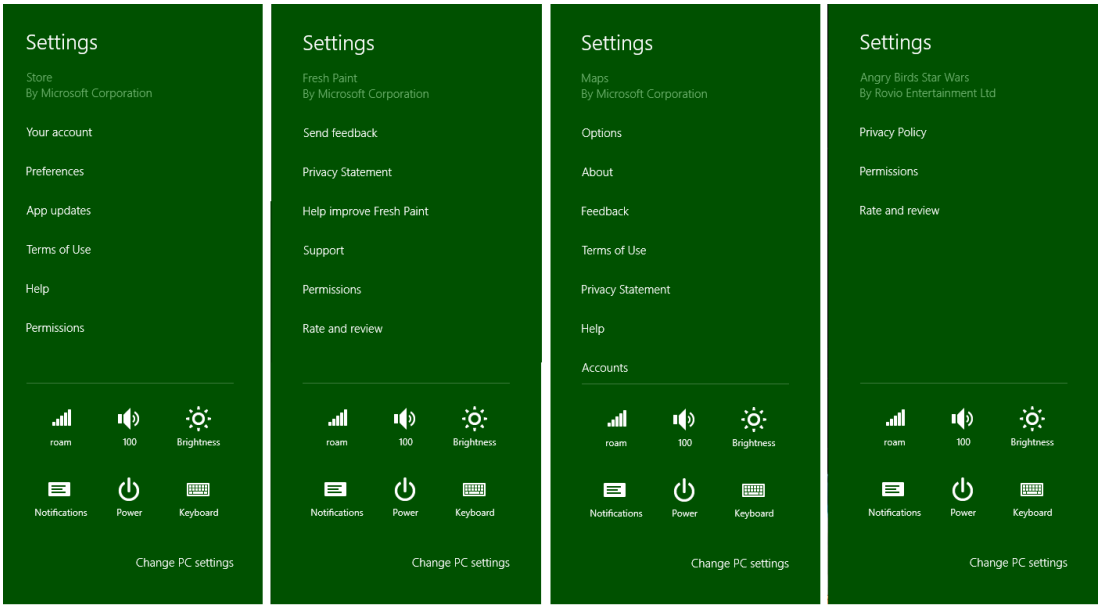


FIGURE 10-4 Examples of commands on the top-level settings pane. Notice that the lower section of the pane always has system settings and the app name and publisher always appear at the top. Permissions and Rate And Review are added automatically for apps acquired from the Store; Rate And Review is not 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!

Things 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.
- 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 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 on the 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 Windows Store 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.

A common group of settings are those that allow the user to configure their roaming experience—that is, a group of settings that determine what state is roamed (you see this on PC Settings > SkyDrive > 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 and checklist for login controls for more information on this subject. Guidelines for a Help command can also be found on Quickstart: add app help.

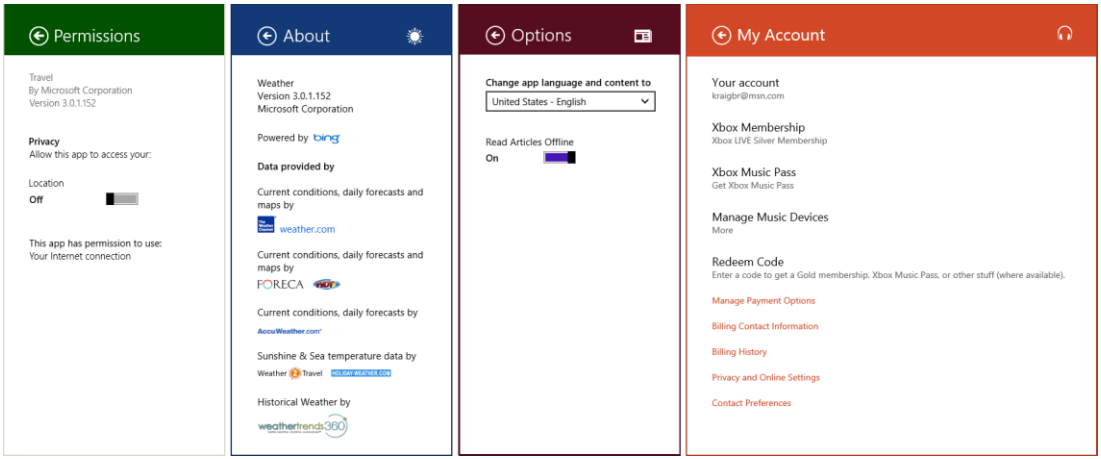


FIGURE 10-5 Examples of secondary settings panes in the Travel, Weather, News, and Music apps. The first three are the narrow size; the fourth is wide. Notice that each app-provided pane is appropriately branded and provides a back button to return to the main Settings pane. The Permissions pane is provided by the system and thus reflects the system theme (it cannot be customized).

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). That is, 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.

There are two ways to implement this process in an app written in HTML and JavaScript: using WinRT directly, or using the helpers in WinJS. Let's look at these in turn 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 as shown below:

```
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 in the next section.

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 settings commands, as we'll see in a moment.

> **Tip** `populateSettings` 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 have it appear at 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.

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, there are recommendations 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 you can place any HTML you want within the control, including other controls, and the flyout will automatically handle vertical scrolling, there's really no reason *not* to use it.

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

[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.

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">
        <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. The styles that apply here are `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).

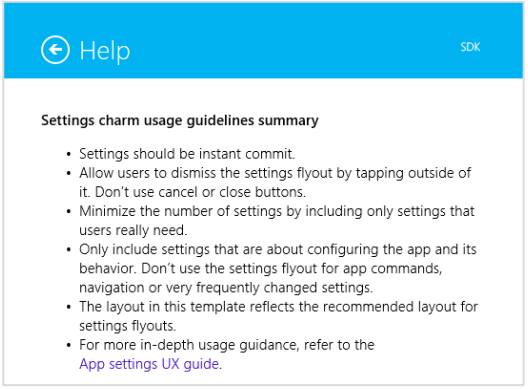


FIGURE 10-6 The Help settings flyout (truncated vertically) from scenario 2 of the App settings sample. Notice the hyperlink on the bottom.

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.

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 flyouts 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 system-provided commands like Permissions and Rate And Review. 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 do so. The Here My Am! app for this chapter provides a demonstration.

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");
```

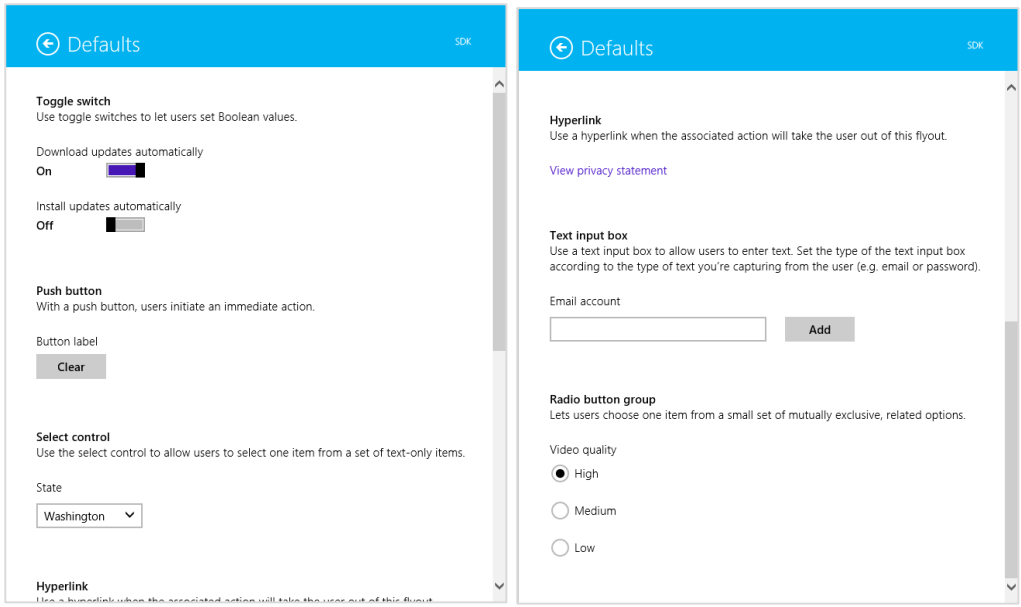


FIGURE 10-7 The settings flyout from scenario 4 of the App settings sample, showing how a flyout supports vertical scrolling; note the scrollbar positions for the top portion (left) and the bottom portion (right).

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.ApplicationSettings.SettingsPane.getForCurrentView().hide`.

You might use `showSettings` and `hide` together, in fact, if you need to navigate to a third-level settings pane. That is, 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 SkyDrive 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 SkyDrive

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 much 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 would buy 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.) In short, 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 really need great apps to efficiently manage it and present it in meaningful ways. There are many creative ways, for example, 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), as 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. In fact, 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 SkyDrive much more in this chapter. Users with a Microsoft account get cloud storage on SkyDrive for free, and the service is deeply integrated into the fiber of Windows as a whole. SkyDrive is what hosts all the user's roaming data (from both apps and the system). Users also automatically see a SkyDrive 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 question 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 on 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. 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.

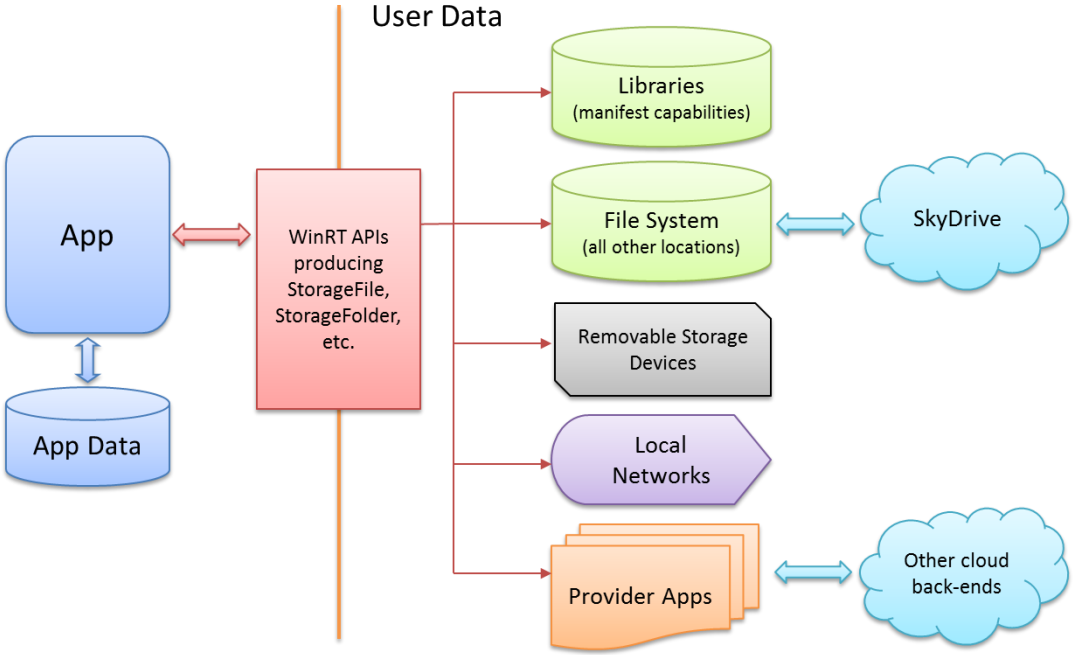


FIGURE 11-1 User data lives anywhere outside the app's package and app data folders. Access to user data locations happens through various APIs that produce `StorageFile`, `StorageFolder`, and related objects, which represent files and folders regardless of location. Access to some locations like libraries, local networks, and removable storage are determined through manifest capabilities; the rest are typically accessed through the File Picker API. SkyDrive is integrated as part of the local file system, and other apps can also 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 SkyDrive.

The right way to think about SkyDrive 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 SkyDrive folders and get file metadata like thumbnails through the `StorageFile` object. All this works because Windows automatically maintains local placeholder 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 to visually distinguish 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 SkyDrive 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 SkyDrive directly** Although SkyDrive 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 SkyDrive reference, on the Skydrive 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, there are really only three ways this happens:

- 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 at all. 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, then, which we'll see visually in "The File Picker UI" section later on, 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 will never preserve 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 provided the app has programmatic access through manifest capabilities.)

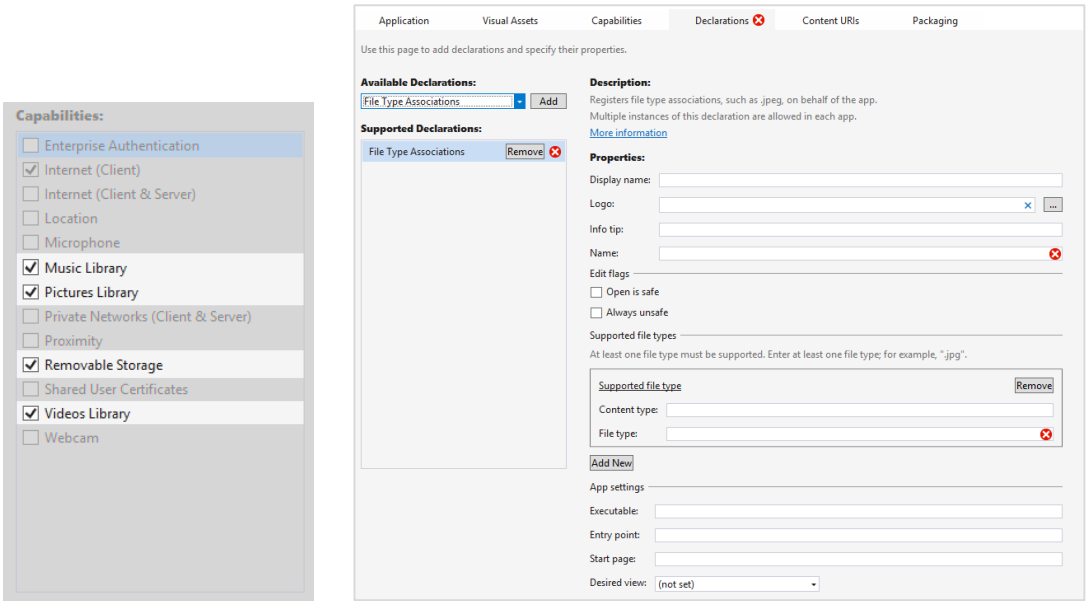


FIGURE 11-2 Capabilities related to user data in the manifest editor (left) and the file type association editor (right). The red X on the Declarations tab indicates the minimal required fields for an association.

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 also required one or more file type associations as with Removable Storage. The capability does exist in Windows 8.1 but must be added manually by editing the XML. Declaring it will trigger a more extensive (and time-consuming) process when you submit the app to the Store, which includes verification that you have 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 SkyDrive 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 really only useful 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 as user data isn't tied to any particular app, it can 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* on 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. It is through these that 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 how to get to the data in Chapter 10 through methods like `StorageFile.openAsync` and the kinds of things we can do with the resulting streams, buffers, and blobs.

In this chapter, then, we'll review a few of those basics and then focus on completing 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.

---

[84] There is a documents option for this API as well that has the same requirements as `KnownFolder.documentsLibrary`.

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

There is an additional set of capabilities with the file system that are not covered in this book, namely the `Windows.Security.EnterpriseData.FileRevocationManager` APIs. These help you manage copy protection for any `StorageItem` with what is called *selective wipe*, which is described in the Security documentation. This accommodates enterprise users that 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, then, 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 makes 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? 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 black 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.

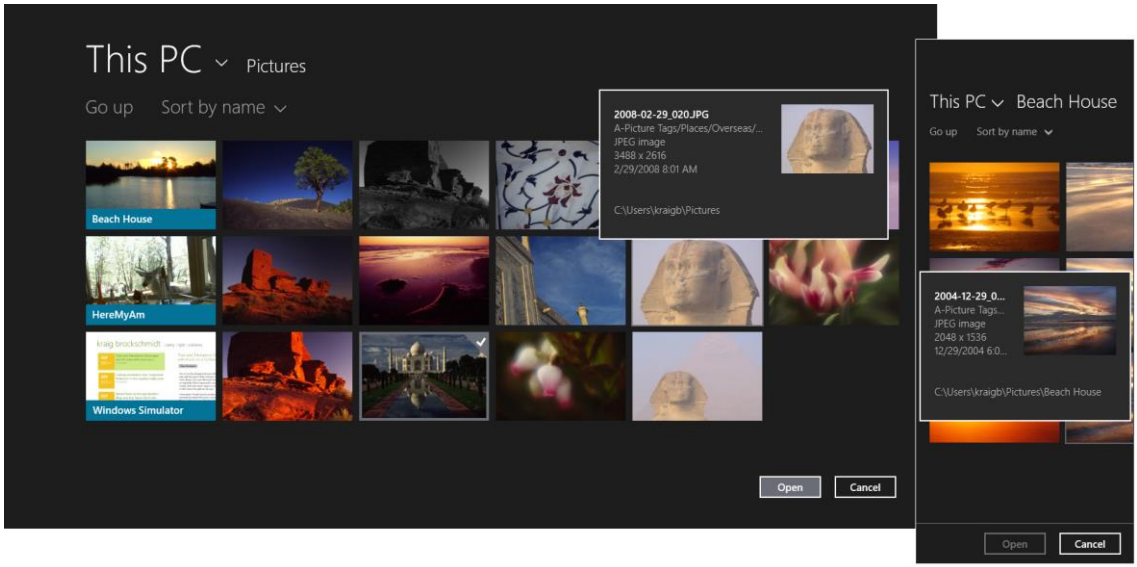


FIGURE 11-3 A single-selection file picker on the Pictures library in thumbnail view mode, with a hover tooltip showing for one of the items (the head of the Sphinx) and the selection frame showing on another (the Taj Mahal). The overlay on the right shows the file picker in a narrow 320px view. Bonus 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 choosing general files or other types like music (right side), additional apps like the Sound Recorder can appear.

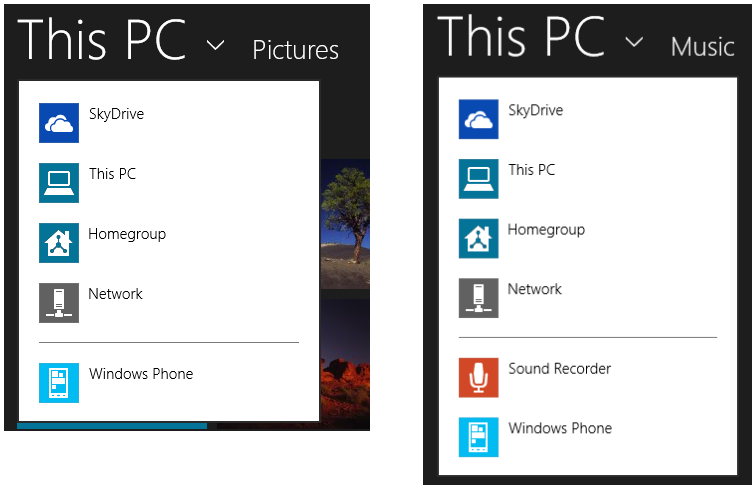


FIGURE 11-4 Selecting other picker locations; notice that SkyDrive and apps are listed along with file system locations. The picker on the left is invoked to select pictures, so only picture-providing apps appear; the picker on the right is invoked to select any type of file, so additional providers appear.

Choosing another file system or network location navigates there, of course, from which you can browse into other folders. As SkyDrive is built into Windows, it's effectively 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 > SkyDrive > File Storage > Save Documents to SkyDrive by Default).

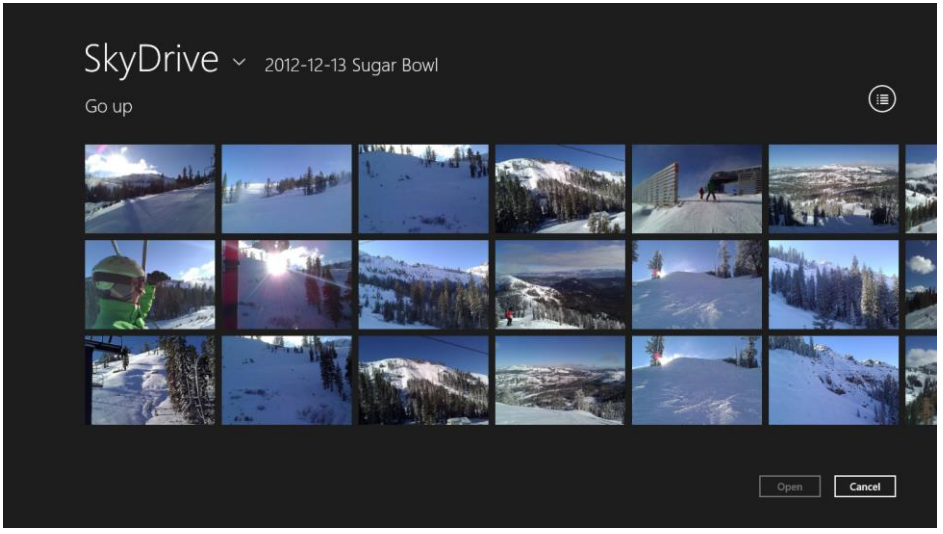


FIGURE 11-5 When picking files from SkyDrive, your cloud storage appears like any other local or network location.

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 really 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!

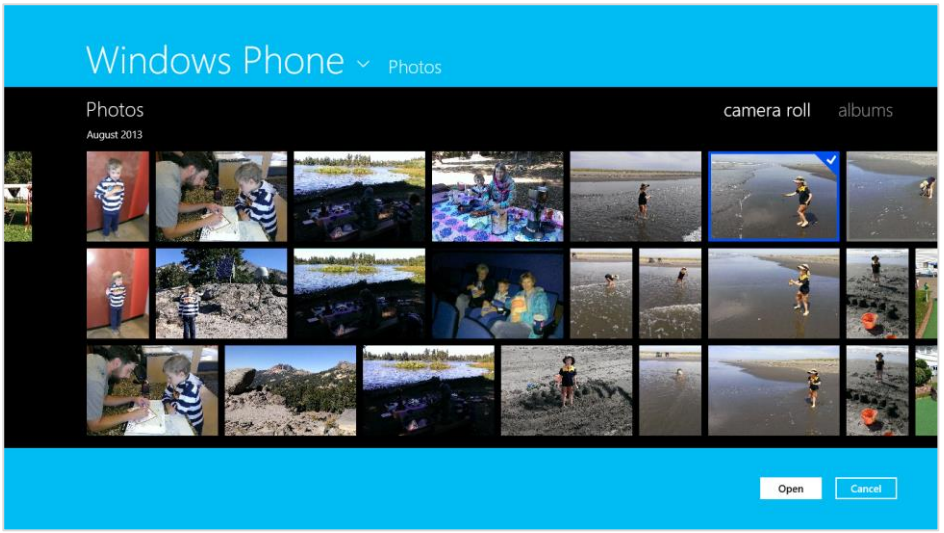


FIGURE 11-6 The Windows Phone app invoked through the file picker provider contract to select a single image.

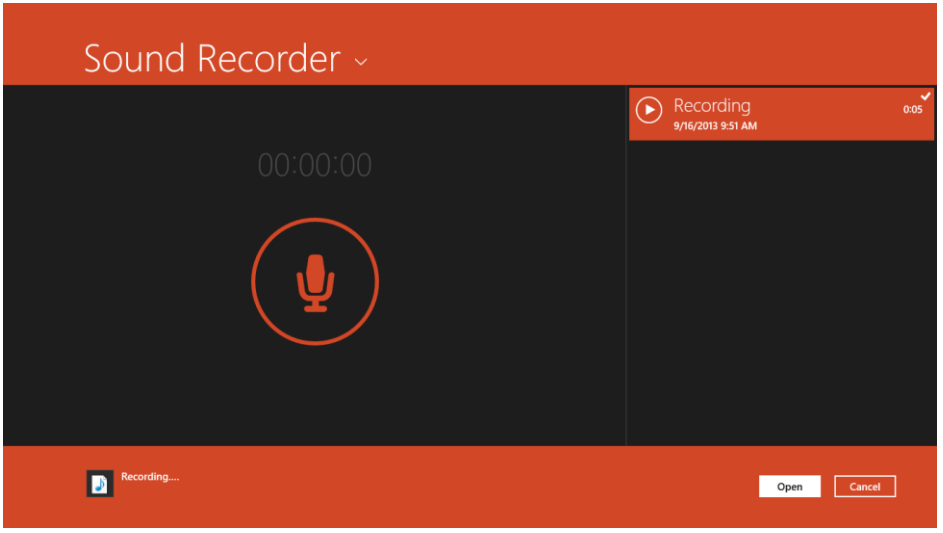


FIGURE 11-7 The Sound Recorder app invoked through the file picker provider contract. This is what appears after a sound has been recorded, and because the picker is invoked to select multiple files, the user can create and return multiple recordings at one time.

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 SkyDrive, an MP3 file from my Music folder, and a couple videos from the current folder.

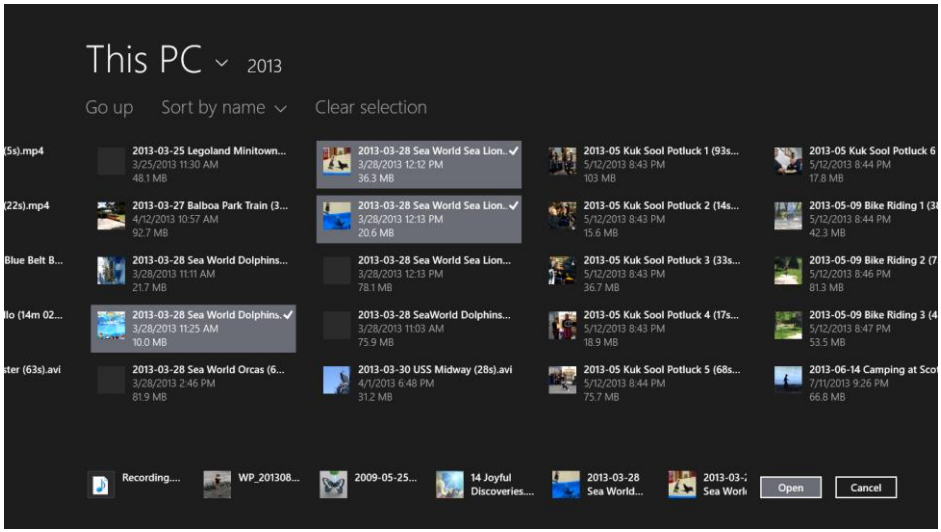


FIGURE 11-8 The file picker in multiselect mode with the selection basket at the bottom. The picker's layout here is a "list" view mode (not thumbnails) that's set independently from the selection mode.

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.

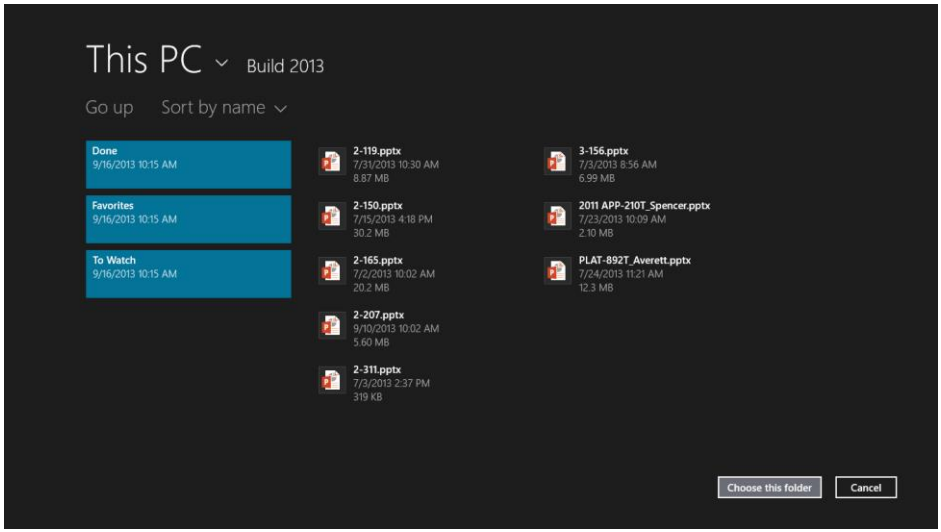


FIGURE 11-9 The file picker used to select a folder—notice that the button text changed and the picker shows the contents of the folder.

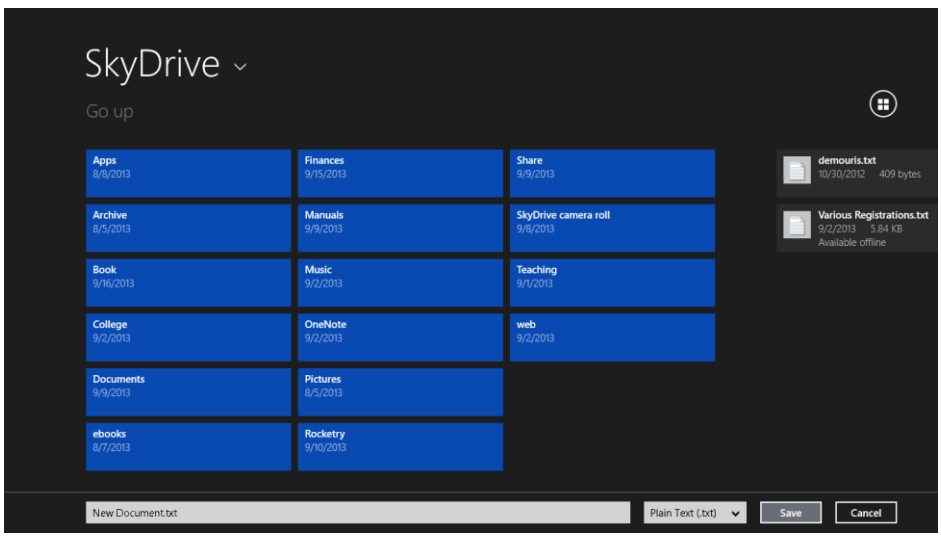


FIGURE 11-10 The file picker used to select a save location (defaulting to SkyDrive) and filename (at the bottom). Files that match the specified save type are also shown alongside folders.

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 also 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 then 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` (SkyDrive 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 the user 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.

> **SkyDrive or Documents folder?** If the user has PC Settings > SkyDrive > File Storage > Save to SkyDrive by Default turned on, the file picker will show SkyDrive 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 SkyDrive 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 SkyDrive 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 as we see here. 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. If, however, 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 then restarted later on)? Furthermore, how does an app remember such files and folders in its saved 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, whose purpose is to save `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 can then 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 through the `Windows.Storage.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 designers of the API 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 then 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 the section "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 and we won't cover them more 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 "SkyDrive" 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 in the next section, "Availability")
- The getThumbnailAsync and getScaledImageAsThumbnailAsync methods (described in the "Thumbnails" section)
- The StorageFile.properties property and the getBasicPropertiesAsync method (described in the "File Properties" section)

Availability

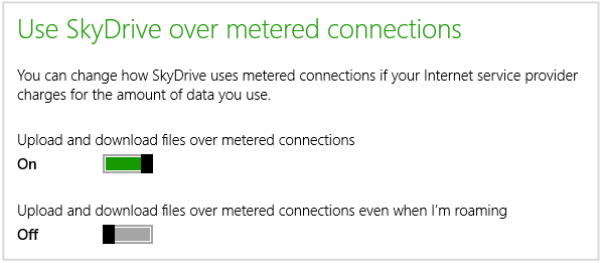
First is the isAvailable property that I mentioned earlier. When working with files that might originate in the cloud, it's generally unimportant for an app to know where it came from, how it might be downloaded, and so forth, because the provider that's behind the StorageFile can take care of all that transparently like Windows does for SkyDrive. 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.

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
SkyDrive file marked "available offline"	true	true	true
SkyDrive 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 > SkyDrive > Metered Connections, as shown below. These are simple on/off settings regardless of file size.



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 really ever be loading up a full image file is when you truly 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 and checklist 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 then 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.

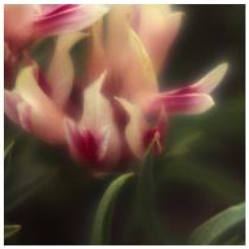
- `resizeThumbnail` Scale the thumbnail to match your *requestedSize*.
- `useCurrentScale` Increases *requestedSize* based on the pixel density of the display (that is, the scaling factor; see Chapter 8, "Layout and Views").
- `returnOnlyIfCached` Forces the API to fail if a thumbnail does not exist 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):

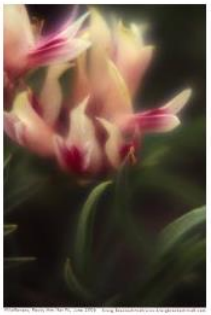
```
var requestedSize = 200,
    thumbnailMode = modes[modeSelected],
    thumbnailOptions = Windows.Storage.FileProperties.ThumbnailOptions.useCurrentScale;

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 in this section 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.

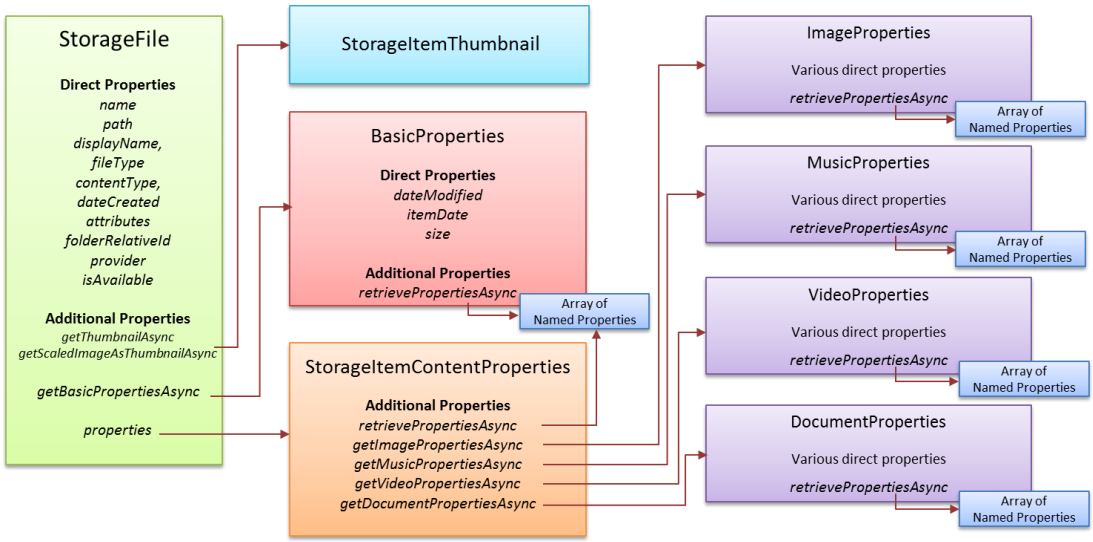


FIGURE 11-1 Relationships between the `StorageFile` object and metadata objects.

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, as discussed in the next section, which are especially helpful when creating gallery experiences over the user's media libraries or some other arbitrary folder. The other two methods then 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 the section "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, tthose 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 then 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");
});
```

We'll see a fuller 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.

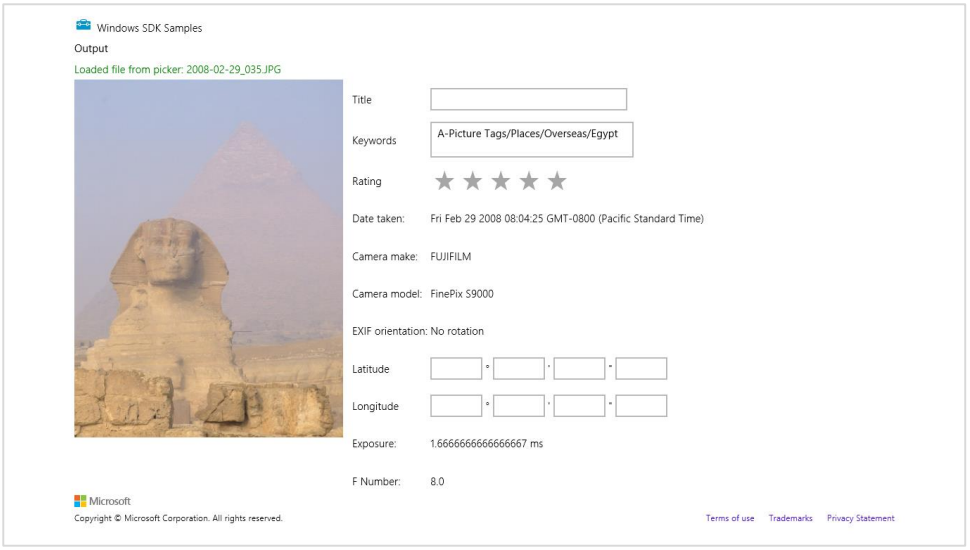


FIGURE 11-12 Image file properties in the Simple imaging sample, which is panned down to show the whole image and more of the property fields.

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 good friend, `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` property 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 the "StorageFile Properties and Metadata" section 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.retrieve-PropertiesAsync` for the *System.FreeSpace* property and you'll actually get the free space on the drive where the folder lives. Pretty cool, eh?

Then there 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 the "Simple Enumeration and Common Queries" section below.

- Any method or property with `Query` in the name All of these members deal with file queries, which is how you do deep enumerations of folder contents based on various criteria (namely, values for 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. (There is again the special `Windows.Storage.DownloadsFolder` mentioned at the beginning of this chapter in "The Big Picture of User Data" that doesn't need any more explanation.)

KnownFolders and the StorageLibrary Object

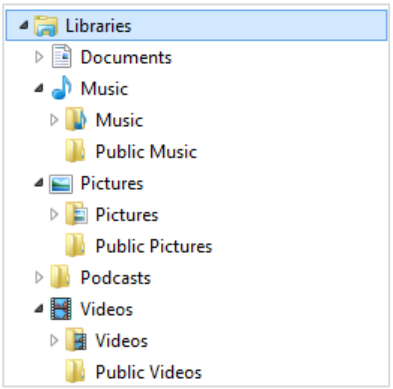
As you already know, `Windows.Storage.KnownFolders` gives you direct access to the `StorageFolder` objects for various user data locations. The `picturesLibrary`, `musicLibrary`, and `videosLibrary` are the obvious ones, but there are a number of others we haven't yet mentioned, some of which depend 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 are, for their part, 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 Podcast artifacts):



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 a 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 for the folders 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 SkyDrive 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, completing with a Boolean to indicate 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. An app uses this to update its own 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.

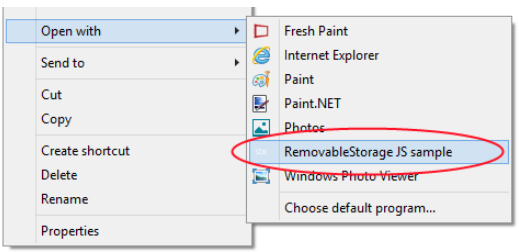
> **Hint** Remember that `definitionChanged` is a WinRT event, so be sure to remove its listeners appropriately. I will note that the Library management sample does *not* do this properly.

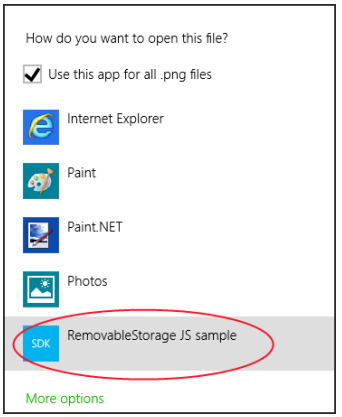
Removable Storage

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 really don't support, because users will attempt to launch your app through those associations and will reflect their disappointment in your Store ratings!

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 you create the appropriate declarations and handle file activation is a subject we'll return to later in the "File Activation and Association" section. 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 then 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 then 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

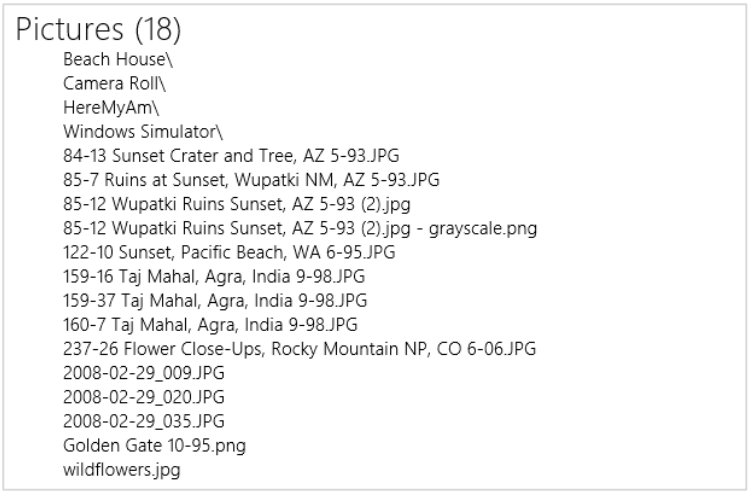
A simple or *shallow* enumeration of folder contents happens through the variants of `getFilesAsync`, `getFoldersAsync`, and `getItemsAsync` in the `StorageFolder` object, which take no arguments. Each method results in a vector of the appropriate item types: `getFilesAsync` enumerates files only and 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 really 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 the documents library and removable storage, 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):



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 SkyDrive 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 the "Availability" section earlier, because those files are marked online-only:

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). There are two parallel methods, `createFolderQuery` and `createItemQuery`, that 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 the results of the query. This is the signal to 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 (there is not a variant 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 the 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 music properties (e.g., *System.Music.AlbumTitle*, *System.Music.Artist*, etc.).

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 much more 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! (This 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 like so, 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 then 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!):

As for scenario 3 of the sample, that brings us into the matter of query options, which are discussed in the next section.

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.

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 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 the three `StorageFolder` methods that accepts them: `createFileQueryWithOptions`, `createFolderQueryWithOptions`, and `createItemQueryWithOptions`. These result in `StorageFileQueryResult`, `StorageFolderQueryResult`, and `StorageItemQueryResult` objects, as already discussed in the previous section. 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 the full name of Windows properties 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 the section "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`.

`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

[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`.

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:

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 return to in Chapter 15, "Contracts.") It runs a query against the Music library, as we've been doing, using the `CommonFileQuery.orderBySearch-Rank` 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) {

        // [Some output code omitted]
        files.forEach(function (file) {
            var searchHits = fileQuery.getMatchingPropertiesWithRanges(file);
```

```
            // [More 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 the "Thumbnails" section 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.)

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

[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.

a QueryOptions from scratch, setting up its own sorting by *System.IsFolder* and *System.ItemName*. 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 there is a third option: 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 file was launched.
    }
});
```

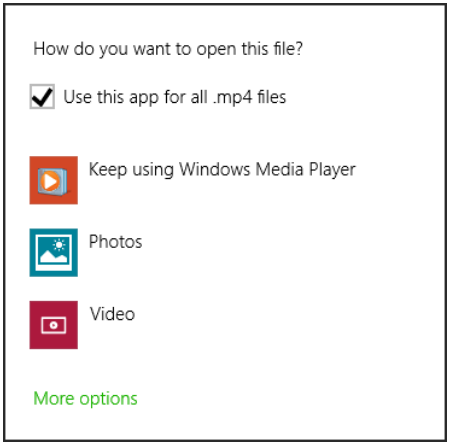
Note `launchUriAsync` does not recognize `ms-appx`, `ms-appx-web`, or `ms-appdata` URIs, because these already map the current app. The app should just display such pages directly.

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.

`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 a `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`:



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. 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 different types of devices, and you should be sure to provide them all.

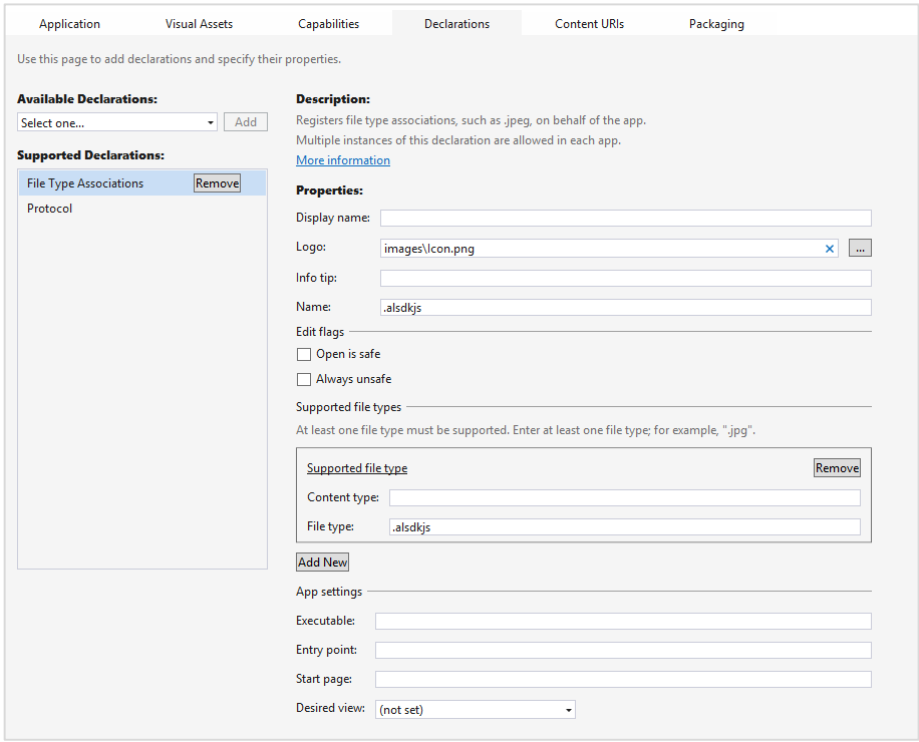


FIGURE 11-13 The Declarations > File Type Associations UI in the Visual Studio manifest designer.

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:

---

[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.

```
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.

Then there are three members that 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).
- SkyDrive 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 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!

Fortunately, touch screens have progressed tremendously in recent years. They are responsive enough for general purpose use (that is, you don't have to stab them to register a point), are built into high-resolution displays, are relatively inexpensive, and are capable of doing something more than replicating the mouse—namely, supporting multitouch and sophisticated gestures.

Great touch interaction is thus now a fundamental feature of great apps, and 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. Similarly, designing for touch means thinking through how gestures might enrich the user experience—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 was the *only* means of interaction that your users might have. At the same time, it's very 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 obsolete keyboards. The availability of speech recognition or handwriting has obsoleted neither mouse nor keyboard. I think the same is true for touch: it's really 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, we'll see ourselves 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 requires this for apps as well.) For this reason, Windows provides a unified pointer-based input model wherein you can differentiate the different inputs if you really need to but can otherwise treat them equally. You can also focus more on higher-level gestures as well, which can arise from any input source, and not worry about raw pointer events at all. Indeed, the very fact that we haven't even brought this subject up until now, midway through this book, gives testimony to just how natural it is to work with all kinds of pointer input without having to think about it: the controls and

other UI elements we've been using have done all that work for us. Handling such events ourselves thus arises primarily when creating your own controls or otherwise doing 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 just get for free. And when an app wants to work more closely with raw handwriting input—known as ink—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 more 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 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 we've also found that a phrase like "touch-first design" that sounds great to a consumer can be a terrifying proposition for developers! With all the attention around touch, consumer expectations are often very 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 really *is* a design issue more than an implementation issue.

We'll talk more about designing for touch in the next section. What I wanted to discuss first is how you as a developer should approach implementing those designs once you have them so that you don't make any distinctions between the types of pointer input until its necessary:

- First, *use templates and standard controls* and you get lots of touch support for free, along with mouse, pen, stylus, and keyboard support. If you build up 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). Note that if you want to capture and save pointer input directly without concern for gestures, there is also built-in support for *inking*, as we'll see later on.

- 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 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. (Note: The vendor-specific events `MSPointerDown`, `MSPointerMove`, and so on still work but are deprecated.)

- Finally, an app can *work directly with the gesture recognizer* to provide its own interpretations of pointer events into gestures.

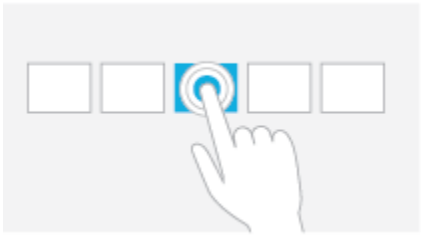
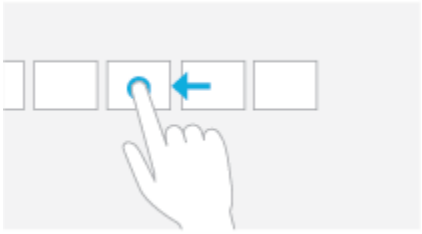
So, what about legacy DOM events that we already know and love, beyond `click`? Can you still work with the likes of `mousedown`, `mouseup`, `mouseover`, `mousemove`, `mouseout`, and `mousewheel`? The answer is yes, because pointer events from all input sources will be automatically translated into these legacy events. This can be useful when you're porting code from a web app into a Windows Store app, for example. 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.

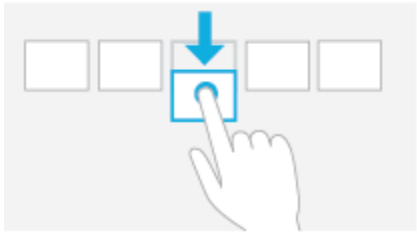
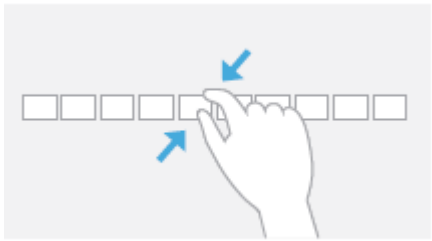
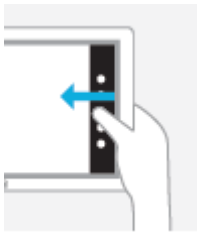
Note Visual feedback for touch input is one of the App certification requirements (section 3.5) and applies to everything in your app as well as any web content you might display in a webview or `iframe` element. Providing feedback means executing small animations that acknowledge the touch. For this you can use the WinJS animations library or straight CSS animations and transitions, as discussed in Chapter 14, "Purposeful Animations."

The Touch Language, Its Translations, 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 (see the previously linked topics for videos of these gestures).

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.

Additional details and guidelines for designing around this touch language can be found on the Gestures, manipulations, and interactions topic.

You might notice in the table above that many of the gestures in the touch language don't actually have a single event associated with them (like pinch or rotate) 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 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. The "Standard Keystrokes" section 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, conflicted with the zoom gesture, and 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 simply 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. We also recommend 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 while a slow flick highlights 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 that of other apps and might become confused (or upset) when those apps don't behave in the same way, especially if those apps use a similar gesture for a completely different purpose! 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 probably 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:[89]

```
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

[89] As WinRT object events, these are subject to the considerations in "WinRT Events and removeEventListener" in Chapter 3.

`contextmenu` event specifically, you usually want to prevent the edge gesture.

CSS Styles That Affect Input

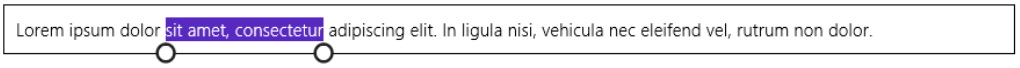
While we're on the subject of input, it's a good time to mention a number of CSS styles that affect the input an app might receive.

One style is `-ms-user-select`, which we've encountered a few times already in Chapter 3, "App Anatomy and Performance Fundamentals," 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` will enable 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 an element that is not a text control and 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 second 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:



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!

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 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:[90]

- `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.
- `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), which does not indicate the presence of the on-screen keyboard, which is always available; `keyboardPresent` specifically indicates a physical keyboard device.
- `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.UI-`

[90] `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`.

Settings.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;
}
```

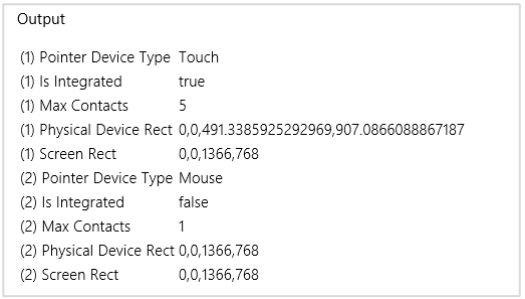
Note In the web context where WinRT is not available, some information about capabilities can be obtained through the msPointerEnabled, msManipulationViewsEnabled, and msMaxTouchPoints properties that are hanging off DOM elements. These also work in the local context. The msPointerEnabled flag, in particular, 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.



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. So 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 directly work 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 in the next section. 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 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:

- `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.

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.

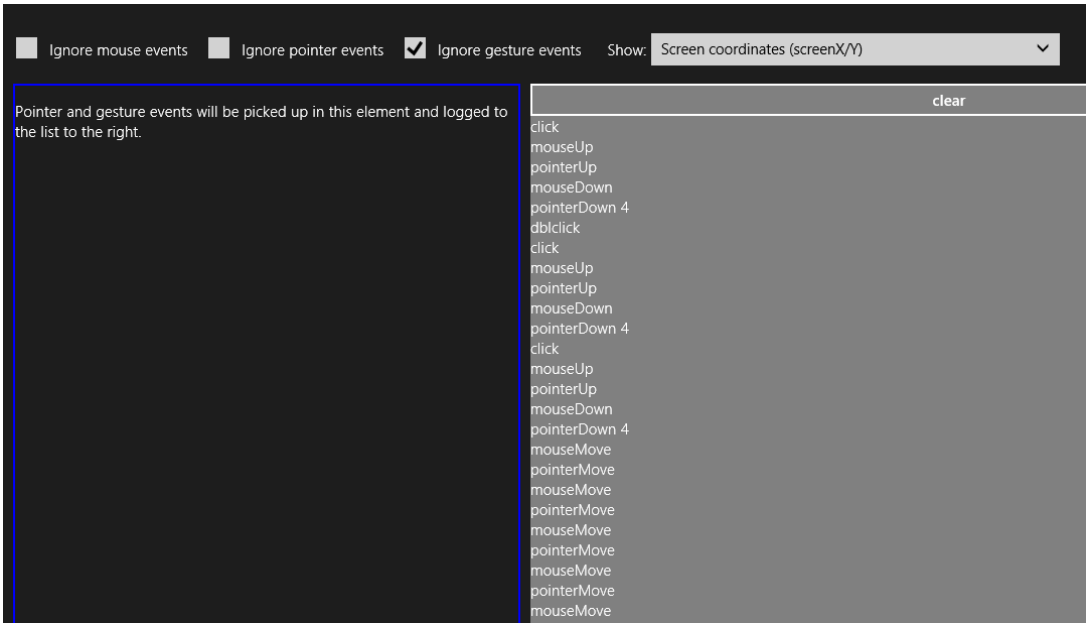


FIGURE 12-1 The PointerEvents example display (screen shot cropped a bit to show detail).

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 in the next section, 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 on the following page 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).
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 just fire automatically like `click` and `pointer*` events, and 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 first 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 (as presented in a small way in one of the samples in "The Gesture Samples" below) that allows multiple people sitting around a table-size touch screen to work with individual pieces as they will. 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 in this section has no dependency on WinRT APIs. As a result, the gesture events described here 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.

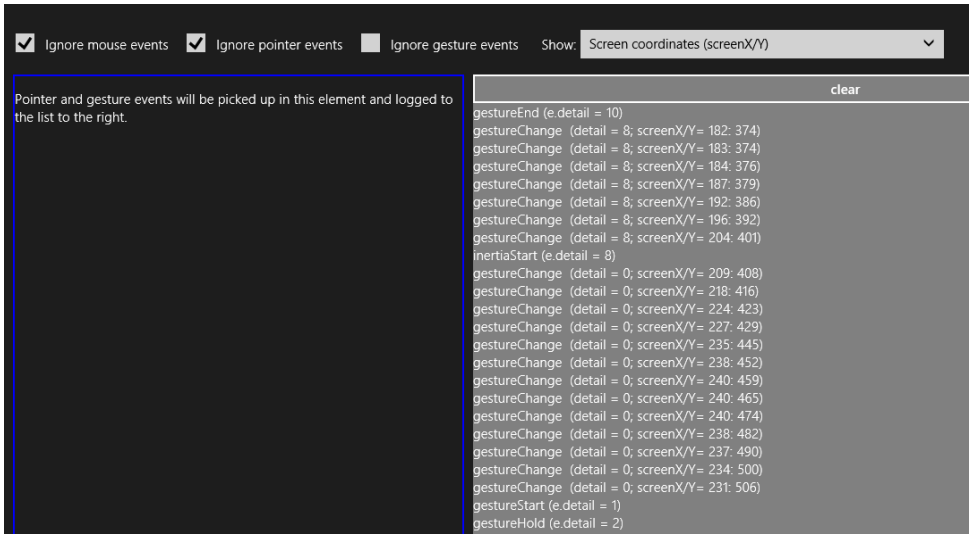
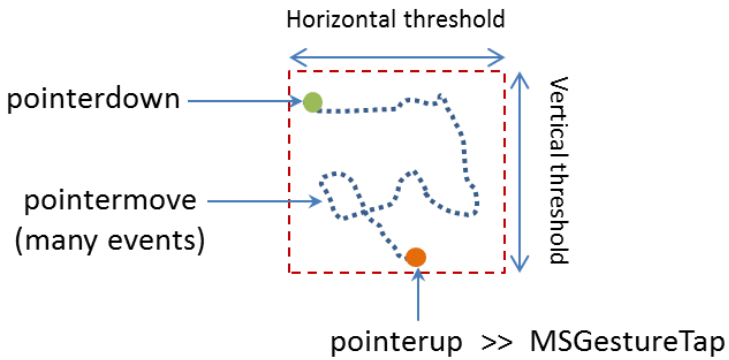


FIGURE 12-2 The PointerEvents example output for gesture events (screen shot cropped a bit to emphasize detail).

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`:



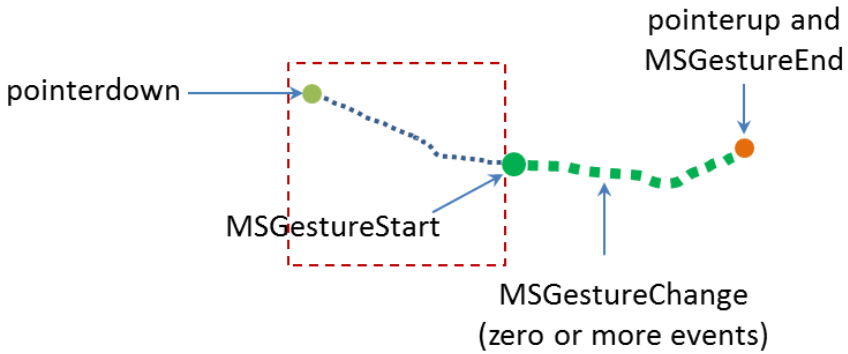
What then differentiates `MSGestureTap` and `MSGestureHold` is a *time threshold*:

- `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 as shown in the following table:

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.

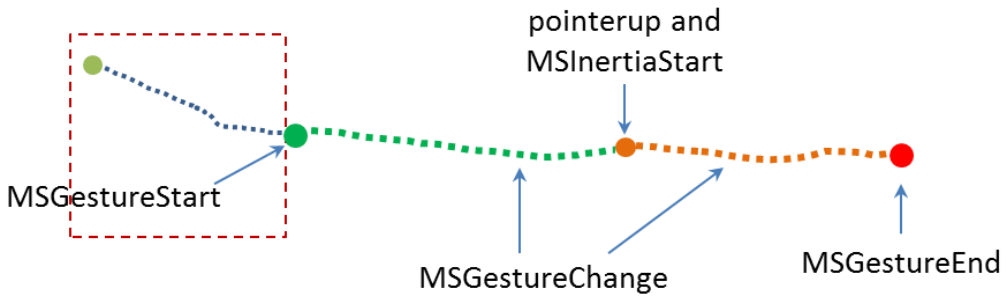
Many of these properties become much more interesting when a pointer moves *outside* the movement threshold, after which time you'll no longer see the tap or hold events. Instead, as soon as the pointer leaves the threshold area, `MSGestureStart` is fired, followed by zero or more `MSGestureChange` events (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.

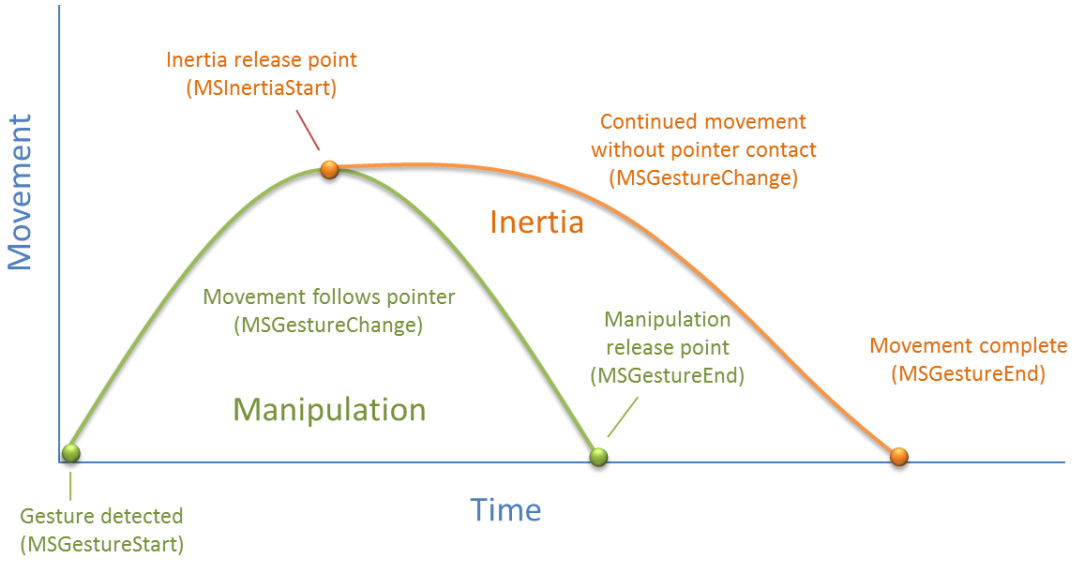


FIGURE 12-3 A conceptual representation of manipulation (green) and inertial (orange) motions.

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 multi-point 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. One thing 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.

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, as 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 onto 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 Windows look and feel 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 then set its `gestureSettings` properties for the kinds of manipulations and gestures you're interested in. The documentation for `Windows.UI.Input.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 `Windows.UI.Input.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 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 earlier section on the touch language. It addresses a few of the questions about when and if custom gestures are really 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 of course the keyboard remains utterly important for textual input, whether it's a physical key-board or the on-screen "soft" keyboard. It is especially important for accessibility as well, as some users are physically unable to use a mouse or other devices. In fact, the App certification requirements (section 3.5) make keyboard input mandatory.

Fortunately, there is nothing special about handling keyboard input in a Windows Store app, but 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 imple-menting special key combinations. See "Standard Keystrokes" later in this section 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 as this will interfere with screen readers.
- Have `accesskey` attributes on those elements that should have keyboard shortcuts.
- Call the DOM focus API on whatever element should be the default.
- Take advantage of the keyboard support that already exists in built-in controls, such as the App Bar.

As an example, the Here My Am! we've been working with in this book has been updated in this chapter's companion content with 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. There are two special concerns with the soft keyboard: how to make it appear, and the effect of its appearance on app layout. At the end of this section 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" />
```

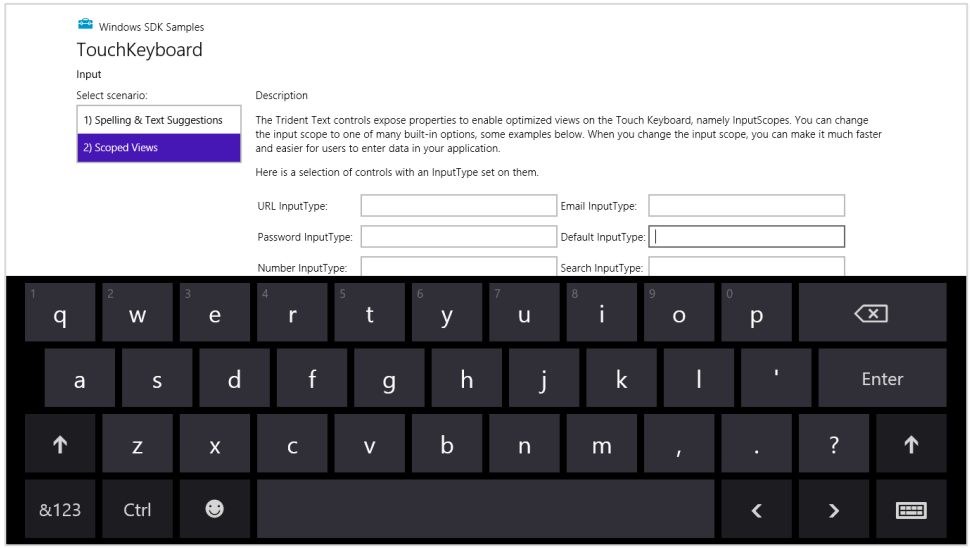


FIGURE 12-5 The soft keyboard appears when you touch an input field, as shown in the Input Touch keyboard text input sample (scenario 2). The exact layout of the keyboard changes with the type of input field.

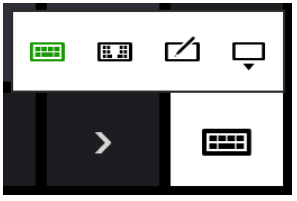
What's shown in Figure 12-5 is the default keyboard. If you tap in the Search field, you get pretty much the same view except the Enter key turns into Search. For the Email field, it's much like the default view except you get @ and .com keys to either side of the spacebar:

The URL keyboard is the same except a few keys change and Enter turns into Go:

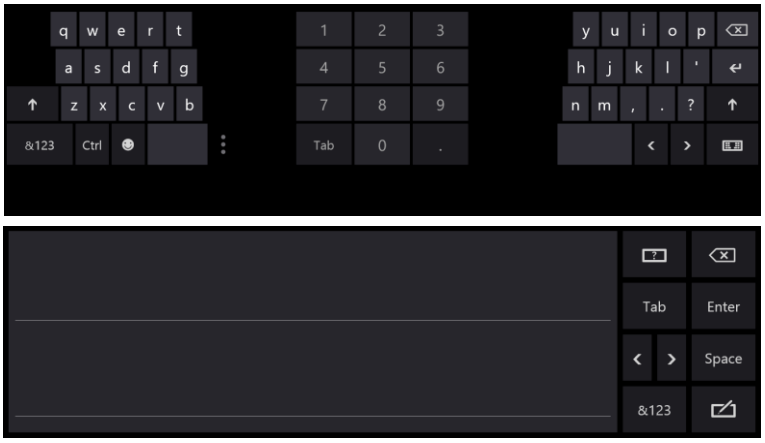
For passwords you get a key to hide keypresses (to the left of the spacebar), which prevents a visible animation from happening on the screen—a very important feature if you're recording videos!

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:

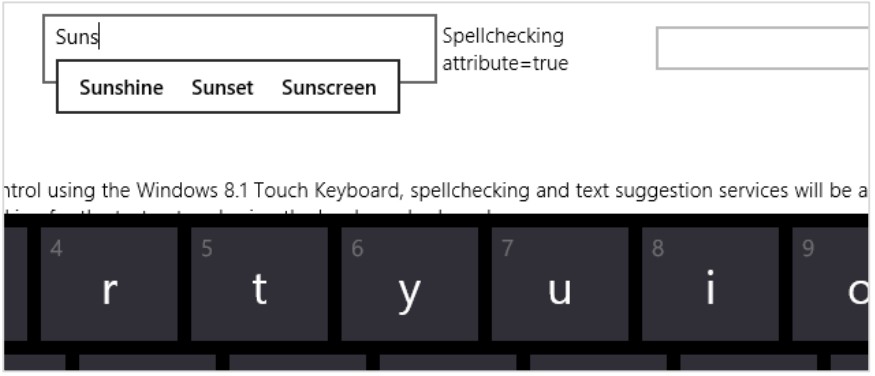


The options here are the normal (wide) keyboard, the split keyboard, a handwriting recognition panel, and a key to dismiss the soft keyboard entirely. Here's what the default split keyboard and handwriting panels look like:



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:



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 will try 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.

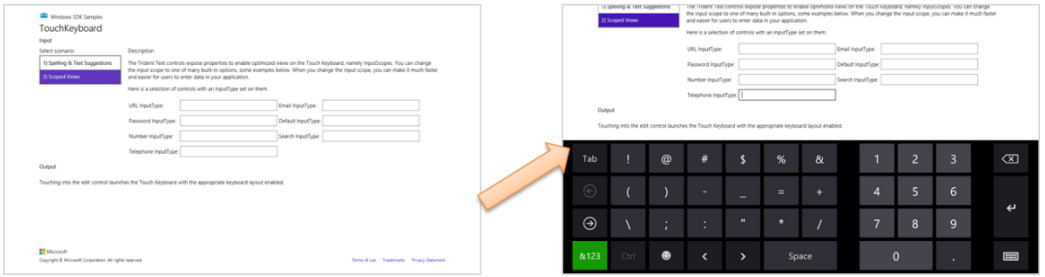


FIGURE 12-6 When the soft keyboard appears, Windows will automatically slide the app page up to make sure the input field isn't obscured.

Although this can be the easiest solution to 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 the second longest JavaScript sample name in the Windows SDK, it actually runs a mere fifth. In fourth place is the Input Gestures and manipulations with GestureRecognizer sample, which we'll see later on. The bronze goes to the Using requestAnimationFrame for power efficient animations sample, and the Unselectable content areas with -ms-user-select CSS attribute sample, which we've seen, gets the silver. And in first place? The Windows Runtime in-process component authoring with proxy\stub generation sample! I don't mind such long names, however—I'm delighted that there 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.

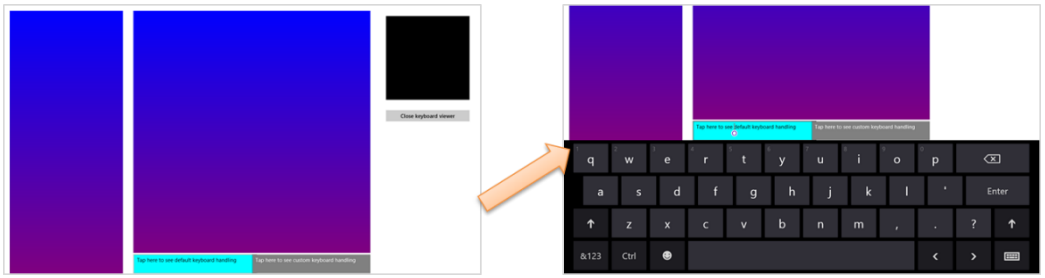


FIGURE 12-7 Tapping on the left *defaultHanding* element at the bottom shows the default behavior when the keyboard appears, which offsets other page content vertically.

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 short of it is that 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.

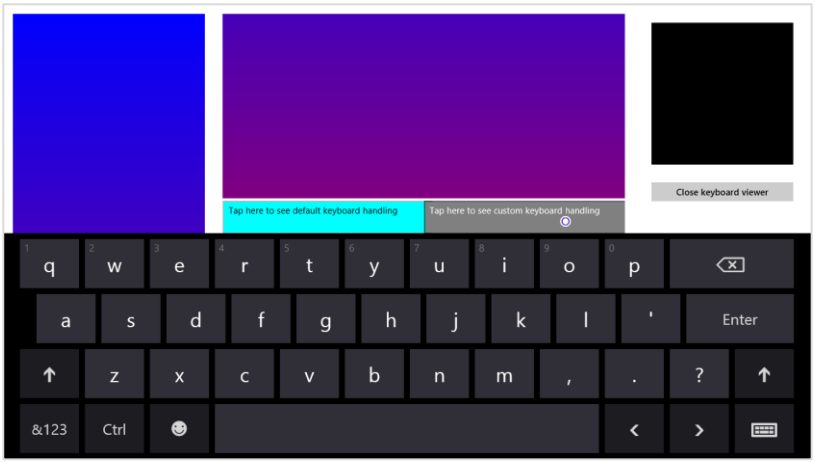


FIGURE 12-8 Tapping the gray *customHandling* element on the right shows custom handling for the keyboard's appearance.

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 actual input data (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. Ink, in other words, 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 then render itself to a drawing surface such as a `canvas`. Here's its basic function:

- 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 already. 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`, `process-PointerUpdate`, 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.

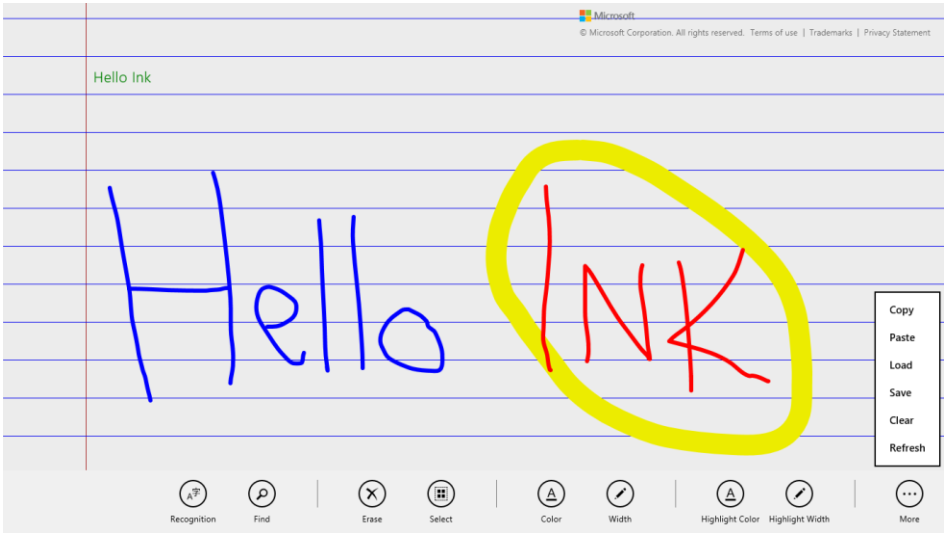


FIGURE 12-9 The Input Ink sample with many commands on its app bar. The green "Hello Ink" text in the upper left was generated by tapping the Recognition command.

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 made a valuable contribution.)

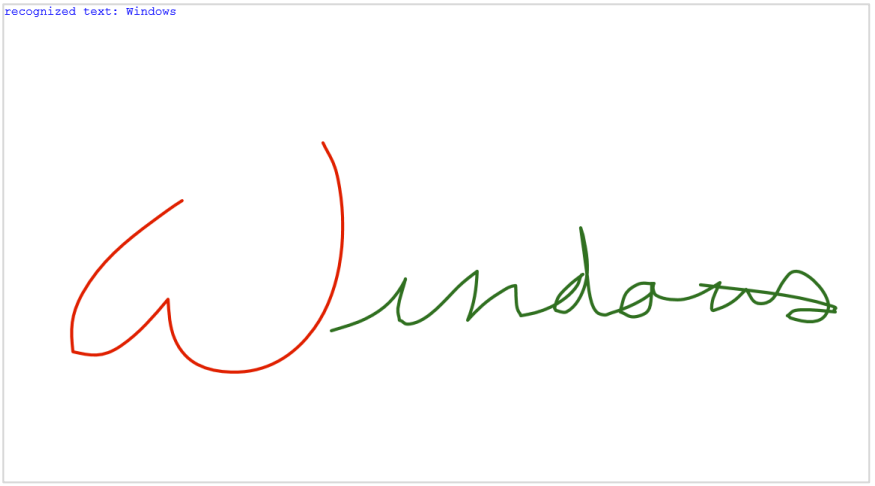


FIGURE 12-10 The Input Simplified Ink sample doing a great job recognizing my sloppy mouse-based handwriting.

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 your app must also provide a Privacy Statement in the Windows Store. 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 WiFi 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 results (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!:

```
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 };
}
```

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 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 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

[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.

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 often 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, because you again want 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/scneario2.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, 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 *gefencing* capabilities, which we'll talk about in the next section.

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 around those areas that are meaningful 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 such that you can take equally meaningful action.

The Geolocator's `movementThreshhold` and `reportInterval` properties can, of course, 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 save your from many of the 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 then 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 then 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. (Do make special note of the Location background task declaration in the manifest.) This does mean, of course, that the app has to be run at least once to establish the background task, as doing so will also prompt the user for consent (because it affects battery life).

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).

In that task you then obtain the `GeofenceMonitor` object (just as shown above) 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, for its part, just takes this information and outputs it to the display; see the `getGeofenceStateChangedReports` 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 then 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 on the Windows App Builder blog.

Now 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 copy can be found on Guidelines for geofencing apps, but be aware that there is not a JavaScript-specific version of this topic at present.

Sensors

As I wrote in the 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 surrounding the device.

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

[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.

its default position, rounding everything off, as it were, to the nearest 90-degree quadrant. The full Orientation sensor, on the other hand, 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 rightside-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!

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`, `rotated90DegreesCounter-clockwise`, `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 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 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!

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, then, 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 also want the fuller backstory on *async* APIs, read Keeping apps fast and fluid with asynchrony in the Windows Runtime on the Windows 8 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 one that is fascinating to take 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 future. 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.

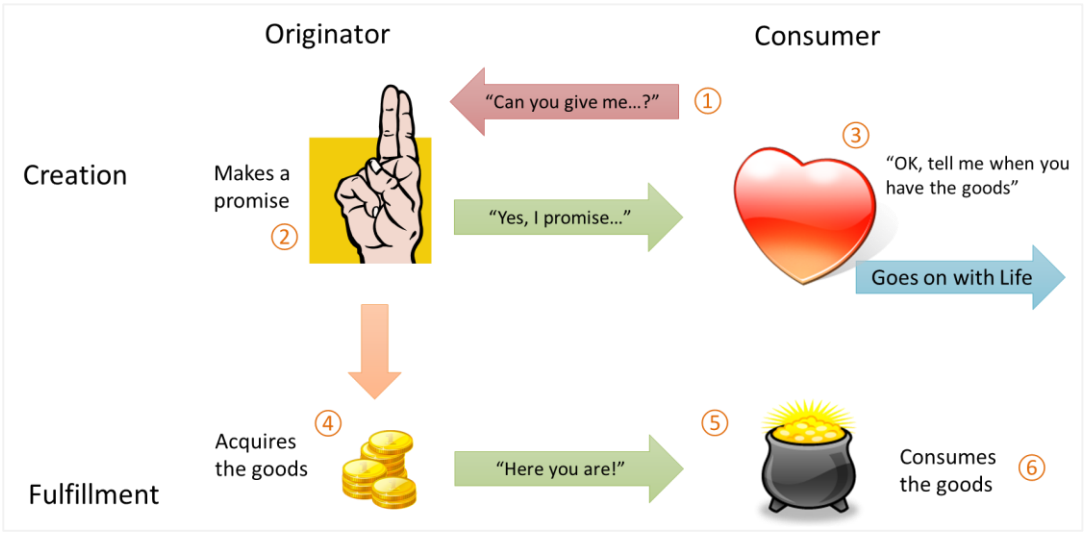


FIGURE A-1 The core relationship encapsulated in a promise.

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 it 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 somewhere else; thus, 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. This 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 in the first place, which is why we use promises for those APIs. 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 really 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.

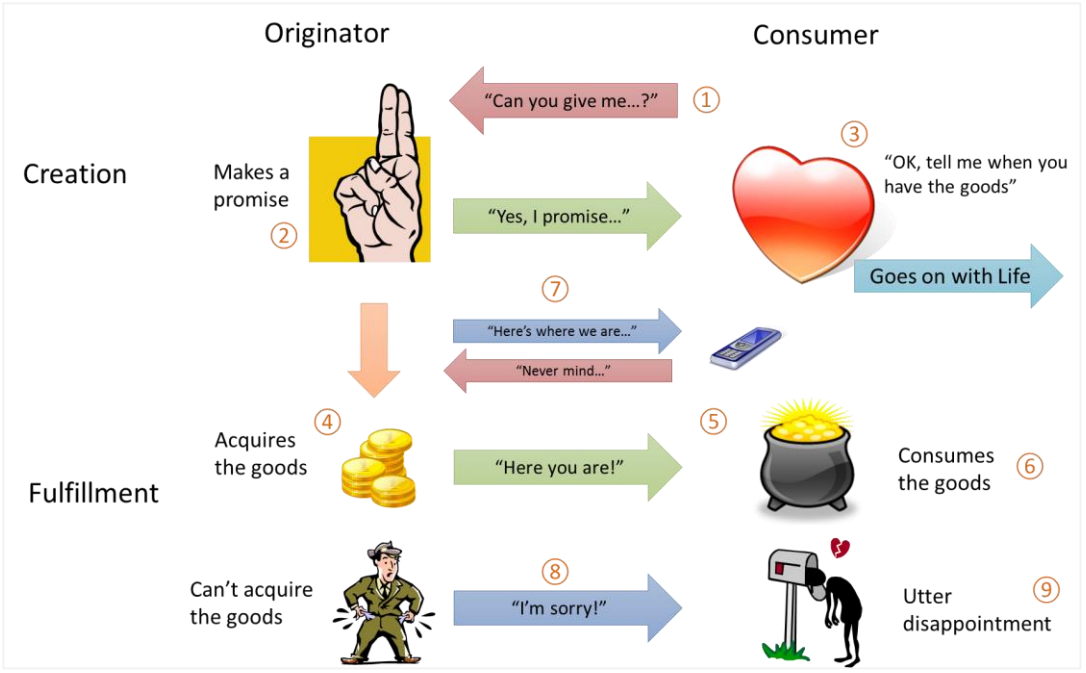


FIGURE A-2 The full promise relationship.

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 generally means 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, there are two approaches we can take. One way is to have a simple property on the promise object to which the consumer assigns the callback function. The other way 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.[94] 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 requirement 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 very 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

---

[94] 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!

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 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 to also illustrate an asynchronous operation, let's look at a few examples 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 as 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. Hre'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 you can name the argument passed to the completed handler (*results* above) whatever you want. It's your code, and you're in control.

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):

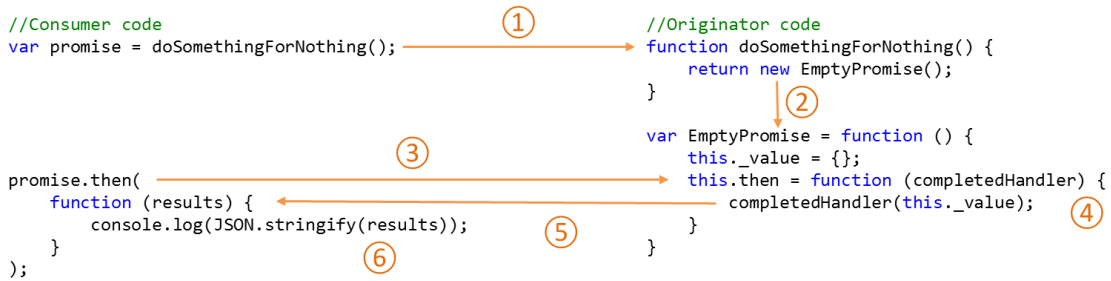
```
//Originator code
var EmptyPromise = function () {
   this._value = {};
   this.then = function (completedHandler) {
      completedHandler(this._value);
   }
}
```

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 passed that promise to another consumer.[95]

Here's how the code executes, identifying the steps in the core relationship:

---

[95] 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!



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

While using promises with known values seems like a way to waste memory and CPU cycles for the sheer joy of it, 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 = 1249999750000
{}
```

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

As 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?

There are a number of reasons. 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):

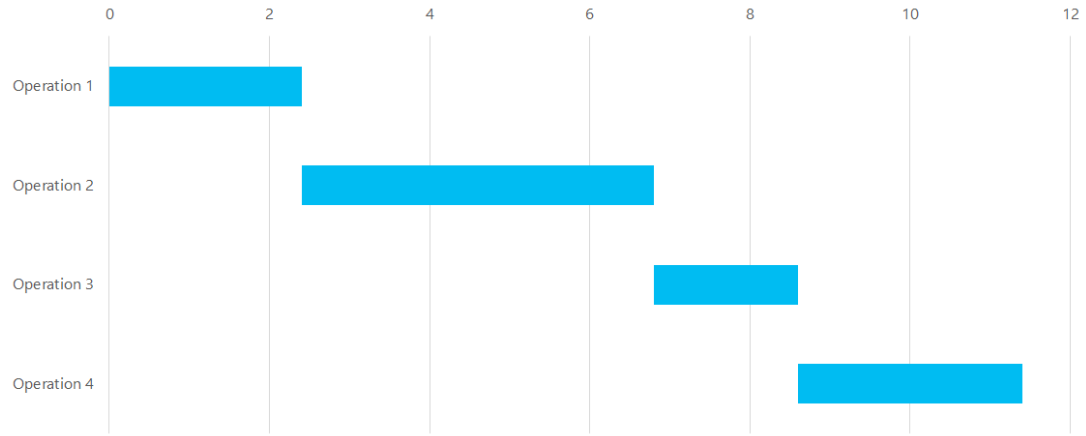
```
promise created
```

```
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 is as follows, 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 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 out, it means that any implementation of `then` must return a promise whose result is the return value of the completed handler given 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). So 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 `IntegerSummartionPromise`, so `InnerPromise1` attaches its own completed handler. When that handler is called, `InnerPromise1` can then 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, so the chain finally 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.

Now 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 there are still a number of unsolved problems, a few of which include:

- `InnerPromise.then` can handle 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.
- There are some repeated code structures, which beg for some kind of consolidation.
- 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! There are a number of subtle issues that start to arise when you put this kind of thing into practice.

Fortunately, there are software engineers who *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 already 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 anywhere an asynchronous API is involved and even at other times. Those promises all meet the necessary specifications, as their underlying classes are supplied by the operating system, which is to say, WinJS. From the consumer'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 handle any number of handlers, correctly process errors, and basically 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.

Now 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. 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, then, 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`, 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 error 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 as 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 (scenario 7)", "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 give 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 only pass its error to any error handler if you call `then`. Still, this `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` 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 as 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 again 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.

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` will now contain 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 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 such 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. Note that 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 stand-alone 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 `parentedComplete` 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 5, "Controls and Control Styling," called "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 5, 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. So 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 batching 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, though, we're returning a promise from `delayedPromise.then` which 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 then 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 a language like C++. Having myself done plenty of C++ in the past, much more than true functional programming, 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 idea of a "class"—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 really 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, this type of function becomes an object constructor and effectively defines a class as we know in object-oriented programming. We can use it as follows:

```
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
   }
);
```

As 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 instance. 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 really 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 really 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.

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.

WinJS.Class.derive

The next part of object-oriented programming that we typically need is the ability to create derived classes—that is, to basically 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, really 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` then adds the static and instance properties as did `define`.

Because `baseClass` already has its own instance and static members, they're already 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.

In object-oriented programming, there is 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 provides 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 article on Wikipedia. `WinJS.Class.mix` is basically a way to do something like multiple inheritance by simply mixing together all the *instance* members of any number of other mixin 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."

So 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 a case of true multiple inheritance in the strict object-oriented sense.)

WinJS itself uses the mixing 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`, `WinJS.Utilities.eventMixin`, `WinJS.Binding.observableMixin`, and `WinJS.Binding.dynamicObservableMixin`:

- `WinJS.UI.DOMEventMixin` contains standard implementations of `addEventListener`, `removeEventListener`, `dispatchEvent`, and `setOptions`, which are commonly used for custom controls. If you look in the WinJS file ui.js, you'll see that all the WinJS controls bring this into their mix.
- `WinJS.Utilities.eventMixin` is basically the same thing without `setOptions`, as it is meant for objects that don't have associated UI elements (it also can't just use the listener methods on the element in the DOM, so it has 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 idea 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 itself 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.

So 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, then, this section explores 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 basically 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`, `WinJS.Utilities.removeClass`, `WinJS.Utilities.hasClass`, and `WinJS.Utilities.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 actually 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 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. That is, in CSS you often use values suffixes like "px" and "em" and so on that, of course, aren't values that are meaningful 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 numerical value—what this function does is just assign the value to the element's left property (saving the prior value so that nothing gets altered) and then read 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 basically 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, it's helpful to 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 clearly 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. Simply said, 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 list 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 various 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 and checklist 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 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 want, 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 might animate the default logo off the screen and pull in other interesting content, perhaps even playing a video. 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 same location and then give the user some great entertainment 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. So, 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."

For one example of an extended splash screen, refer to the Splash screen sample in the Windows SDK. While 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 ExtendedSplash-Screen1 example in 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.

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>
```

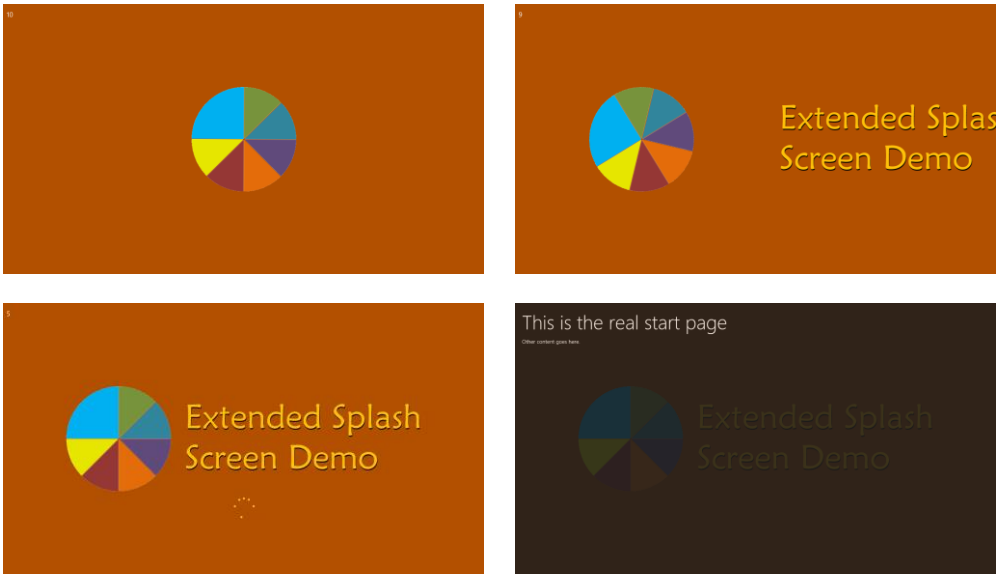


FIGURE B-1 Four stages of the ExtendedSplashScreen1 example: (1) the default splash screen, upper left, (2) animating the logo and the title, upper right, (3) showing the progress indicator, lower left, and (4) fading out the extended splash screen to reveal the main page, lower right. A 10-second countdown in the upper left corner of the screen simulates initialization work.

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 module-wide variables for the splash screen elements, plus two values 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, I 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. One detail to always keep in mind is that animations run in the GPU (graphics processing unit) when they affect only *transform* and *opacity* properties; animating anything else will run on the CPU and generally perform 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, just run the example!

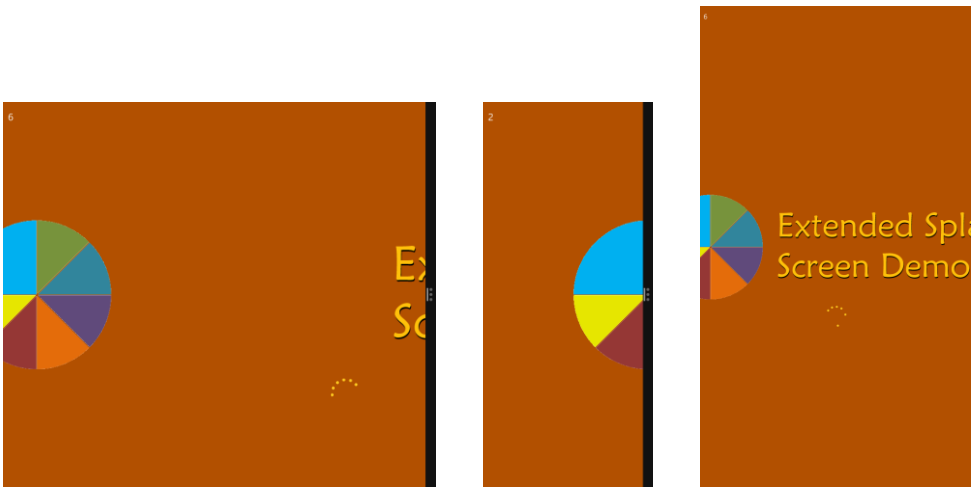
This code structure will likely be similar to what you need in your own apps, only use a single `setTimeout` call to delay showing a `progress` control, replace the `countDown` routine with your real async initialization work, and set up whatever elements and animations are specific to your splash

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 pretty much 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, and full screen portrait, for instance—we get dreadful results in just about every case other than full screen landscape:



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 instead 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 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. With your own HTML and CSS you can 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 feature and concern yourself merely with the positional relationships of the items within the ListView and a few layout-specific concerns, if needed, such as virtualization and animation.

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 see 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 owing to the ListView's use of absolute positioning and the fact that it was never designed for layout extensibility. The ListView was revamped for WinJS 2.0 and now uses 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.[96] 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:

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 is how the layout gets information from the ListView, through the members of the `WinJS.UI.ILayoutSite2` interface.
`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.

[96] 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.

`hitTest`, `dragOver`	If the ListView's `itemsReorderable` is true, these is called when an item is dragged over the ListView. `hitTest` specifically determines which item is at a particular (x,y) coordinate; `dragOver` (which receives the coordinates and a `dragInfo` object) is the signal for 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, as it will fill the rest in with no-ops. We call this a "pay for play" mode. 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	Above plus `layout`
virtualization support	Above plus `itemsFromRange`
keyboard support	Add `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 want to support additional features. Let's now 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 really 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 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 is building 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 just rendered vertically like any other markup.

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 ListView. 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.

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.

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 can, of course, 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 really 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. 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, so 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 this element has a width appropriate to all the items it contains. Of course, unless we know that there are 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 due to items being added and moved, which allow you to minimize modifications you need to make to other items and thus reduce rendering overhead. The return value of `layout` is a promise that is 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 object can provide other methods and properties through which it can communicate with the app. After all, the app has to 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, then, 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 };
}
```

Thus, 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 layouts we can do. 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. Custom layouts let you play around with positioning however you like without having to be concerned with all the rest!

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 what items are visible for a particular pixel range through `itemsFromRange`, because the layout understand how big items are and how 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 SDK, not the extras 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, dimensional properties, to be accurate, must make sure 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

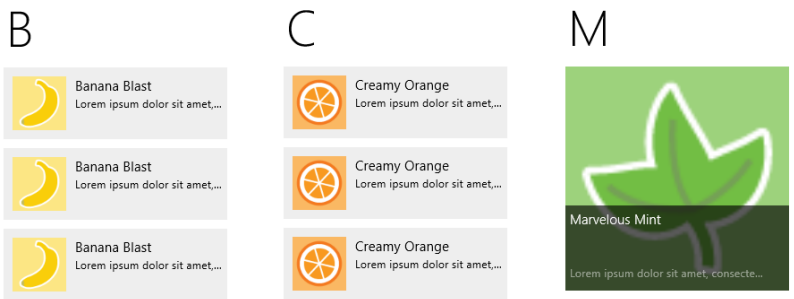
When a ListView is supplied with a group data source in addition to its item data source, what changes for a custom layout is that the *tree* argument to the `layout` method will contain more than one item. Each item in *tree*, of course, is a single group that contains whatever items belong to that 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):



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 is an extension of 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 type, but looks like this:

```
return { type: WinJS.UI.ObjectType.item, index: itemIndex };
return { type: WinJS.UI.ObjectType.groupHeader, index: headerIndex };
```

The `hitTest` method should also include an `insertAfterIndex`, which indicates 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 as expected.)

In this 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 (which contains `index` and `type` properties as in the return object) and a `WinJS.UtilitiesKey`, and 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, so 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 as a whole, 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 to the layout if it wants to take any specific 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>
```

In this case 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 couldn't rely on fixed values here and would 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 as all its members start with _ indicating, "don't touch"). 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 we'd be dropping on
    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
- 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 simple and 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," there are a few other points to make 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. These are 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:` URI's (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) that 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.

Now, 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 you can go to the MultipartUploadServer 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. Then 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 are 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 8 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, which you can use if you want (see the Windows Runtime XML data API sample), but this API is somewhat redundant with the built-in XML/DOM APIs already 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;
}
```

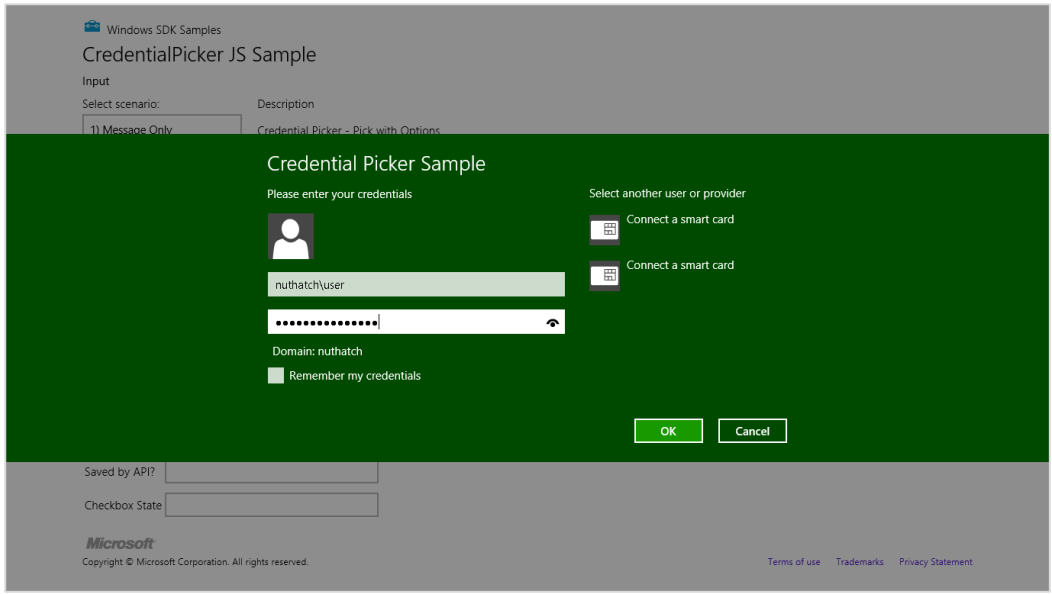
```
var resourceUri = new Windows.Foundation.Uri( /* service address */ );
createClient();

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!");
    }
}
```

The Credential Picker UI

For enterprise scenarios where the Web Authentication Broker won't suffice for authentication needs, 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.



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 screen shot:

```
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 screen shot:

```
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`.)

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.