

# Programming Windows Store Apps with HTML, CSS, and JavaScript

Second Edition

**FIRST  
PREVIEW**



Kraig Brockschmidt

## FIRST PREVIEW

This excerpt provides early content from a book currently in development, and is still in draft, unedited format. See additional notice below.

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

Copyright © 2013 Microsoft Corporation

All rights reserved. No part of the contents of this book may be reproduced or transmitted in any form or by any means without the written permission of the publisher.

This document supports a preliminary release of a software product that may be changed substantially prior to final commercial release. This document is provided for informational purposes only and Microsoft makes no warranties, either express or implied, in this document. Information in this document, including URL and other Internet website references, is subject to change without notice. The entire risk of the use or the results from the use of this document remains with the user.

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

Microsoft and the trademarks listed at <http://www.microsoft.com/about/legal/en/us/IntellectualProperty/Trademarks/EN-US.aspx> are trademarks of the Microsoft group of companies. All other marks are property of their respective owners.

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

**Acquisitions, Developmental, and Project Editor:** Devon Musgrave

**Cover:** Twist Creative • Seattle and Joel Panchot

# Introduction

It seems like it was only a few months ago that I was writing the introduction for the first edition of this book, *Programming Windows 8 Apps in HTML, CSS, and JavaScript*. Of course, it was only 8 months ago! It's been a remarkably short time between the release of Windows 8 and the Windows 8.1 Preview that we've made available as of June 26<sup>th</sup> 2013 for the //build conference. And yet much has been improved in the Windows platform during that time.

First of all, however, let me thank the hundreds of thousands of readers who downloaded the first edition of this ebook, both directly from Microsoft Press and from the Amazon Kindle store where the book has maintained a high ranking among programming titles as well as within the broader computer & technology category. I'm delighted that this work has been serving you well, and I was certainly inspired to start in on this second edition as soon as I began reading the specifications for Windows 8.1 Preview. My notes on what to add, what to change, and what to expand are quite lengthy!

In this First Preview of the second edition, which constitutes only those chapters that I and my editors have worked through so far, I will not cover the whole Windows 8.1 Preview story, of course. For that I can still recommend 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 the Windows 8.1 Preview, so you can find much more there. I can specifically recommend [Windows 8.1 Preview: New APIs and features for developers](#).

This second edition is intended to stand alone for developers who are starting with Windows 8.1 Preview. It represents the *state* of Windows 8.1 Preview rather than trying to document the delta from Windows 8. For this reason I'm not going into much detail about migrating apps from Windows 8 nor trying to highlight all the changes to both APIs and behaviors. Check the Developer Center for such information.

Here's a quick summary of what's in this First Preview:

- Chapter 1, "The Life Story of a Windows Store App," is much the same as in the first edition, with some small additions. For the most part, the core characteristics of the Windows platform is the same as with Windows 8, with the biggest exception being the view model for apps where we now have a variable sizing model.
- Chapter 2, "Quickstart," is updated for Windows 8.1 Preview, and I've added some sections that cover extra improvements to the Here My Am! app that we'll be building over the course of the book.
- Chapter 3, "App Anatomy, Page Navigation, and Promises," is expanded from the first edition. Besides updating the text for small bits like new tile sizes in Windows 8.1 Preview, I've added a section on extended splash screens, tightened up the discussion of promises, written out some

details of the new task scheduler for the UI thread, and included a new section on debugging and profiling.

- Chapter 4, “Using 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 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.
- Appendix A, “Demystifying Promises,” completes the discussion of promises that starts in Chapter 3. That is, Chapter 3 covers the essentials about using promises, which are often returned from asynchronous Windows Runtime APIs. 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, “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.

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 perhaps not quite on a daily basis as work on this second edition takes priority!

Be mindful that the chapter organization of this second edition is still in flux, so references to later chapters are subject to change. As you can expect from the length of the change list I mentioned earlier, I’m going to be adding many pages to this second edition that all need to be appropriately organized!

## 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 and Windows 8.1 Preview 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 B in this First 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 Preview installed on your development machine, along with the Windows SDK for Windows 8.1 Preview and the associated tools. All the tools, along with a number of other resources, are listed on the [Windows 8.1 Preview page](#). You'll specifically need Microsoft Visual Studio Express 2013 for Windows 8.1 Preview. 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 [Samples pack](#) listed on this page, or visit [Windows app samples](#) and specifically download the SDK's JavaScript samples. We'll be drawing from many—if not most—of these samples in the chapters ahead, pulling in bits of their source code to illustrate how many different tasks are accomplished.

One of my secondary goals in this book, in fact, is to help you understand where and when to use the tremendous resources in what is clearly the best set of samples I've ever seen for any release of Windows. You'll often be able to find a piece of code in one of the samples that does exactly what you need in your app or that is easily modified to suit your purpose. For this reason I've made it a point to personally look through every one of the JavaScript samples, understand what they demonstrate, and

then refer to them in their proper context. This, I hope, will save you the trouble of having to do that level of research yourself and thus make you more productive in your development efforts.

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

<http://aka.ms/FirstPreview/CompContent>

That companion content also contains a few additional examples of my own, which I always refer to as “examples” to make it clear that they aren't official SDK content. (I've also rebranded the modified samples to make it clear that they're part of this book.) I've written these to fill gaps that the SDK samples don't address, or to provide a simpler demonstration of a feature that a related sample shows in a more complex manner. You'll also find many revisions of an app I call “Here My Am!” that we'll start building in Chapter 2 and again refine throughout the course of this book. This includes localizing it into a number of different languages by the time we reach the end. (By the way, you might find that with Windows 8.1 Preview that the app runs better outside the debugger; in the debugger I've seen issues getting geolocation readings, which limits the app's functionality quite a bit.)

The companion content includes a few videos that explain how to use tools like Visual Studio and Blend much better than text. Note that with this First Preview, though, that I have not updated these videos to the Windows 8.1 Preview tools. They're still be useful, but keep an eye out for changes (such as the Device tab in Blend).

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 8 App Developer blog](#), which has recently become the [Windows App Builder Blog](#). This is a great resource to follow, and you might also refer to the [Windows Store for Developers blog](#), which has also merged into the App Builder site. 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 First

Preview we've elected not to hyphenate these, so with something like `Windows.Security.Cryptography.CryptographicBuffer.convertStringToBinary` you'll see a gap on the previous line. We'll take care of these for the final version of this second edition.

For simplicity's sake (and because such hyphens produced some headaches in the first edition of this book), I've often omitted the namespace 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: Platform Characteristics of Windows 8

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 subsequent versions, of course) provides 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 gradually 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.)

Throughout this book we’ll explore how to leverage what you know of standards-based web technologies to build great Windows Store apps. 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, using web content, controls, collections, layout, commanding, state management, and input, followed by chapters on media, animations, contracts through which apps work together, devices, WinRT components (through which you can use other programming languages and the APIs they can access), and the Windows Store (a topic that includes localization and accessibility). There is much to learn.

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!

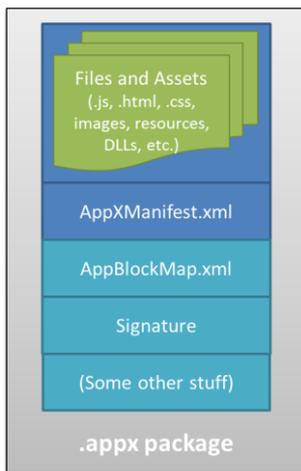
**Terminology note** What we refer to as *Windows Store apps*, or sometimes just *Store apps*, are those that are acquired from the Windows Store and for which all the platform characteristics in this chapter (and book) apply. These are distinctly different from traditional *desktop applications* that are acquired through regular retail channels and installed through their own installer 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.<sup>1</sup> The *package* itself is an *appx* file (.appx)—see Figure 1-1—that contains your app's code, resources, libraries, and a *manifest*. The manifest describes the app (names, logos, etc.), the *capabilities* it wants to access (such as areas of the file system 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. The blockmap, for its part, 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 the Windows Store only needs to download 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.

---

<sup>1</sup> 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 be using as well, are free tools that you can obtain from <http://dev.windows.com>. This also works in Visual Studio Ultimate, the fuller, paid version of this flagship development environment.

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 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](#).<sup>2</sup>

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 onboarding 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.

As a developer, 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 if you want to use a third-party commerce solution for in-app purchases (bypassing revenue sharing with the Store), these choices also affect your design from the get-go. And even if you're just going to give the app away to promote a cause or to just share your joy, understanding the relationship between the Store and your app is still important. For all these reasons, you might want to skip ahead and read the "Your App, Your Business" section of Chapter 18, "Apps for Everyone," 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!

---

<sup>2</sup> 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.

## Sidebar: The Store API and Product Simulator

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

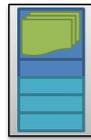
Of course, this begs a question: how can an app test such features before it's even in the Store? The answer is that during development, you use these APIs through the `CurrentAppSimulator` class instead. This is entirely identical to `CurrentApp` (and in the same namespace) except that it works against 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, you 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.)

## Discovery, Acquisition, and Installation

---

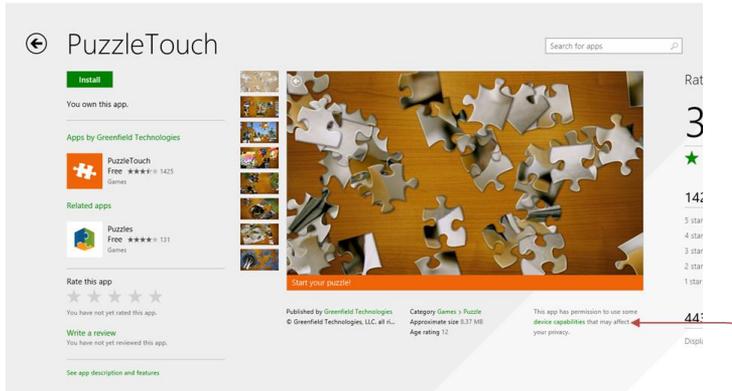
Now that your app is out in the world, its next job is to make itself known and attractive to potential customers. Simply said, while consumers can find your app in the Windows Store through browsing or search, you'll still need to market your product as always. That's one reality of publishing software that certainly hasn't changed. That aside, even when your app is found in the Store it still 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.



Onboarded app package

Capabilities in the manifest determine what shows under app permissions

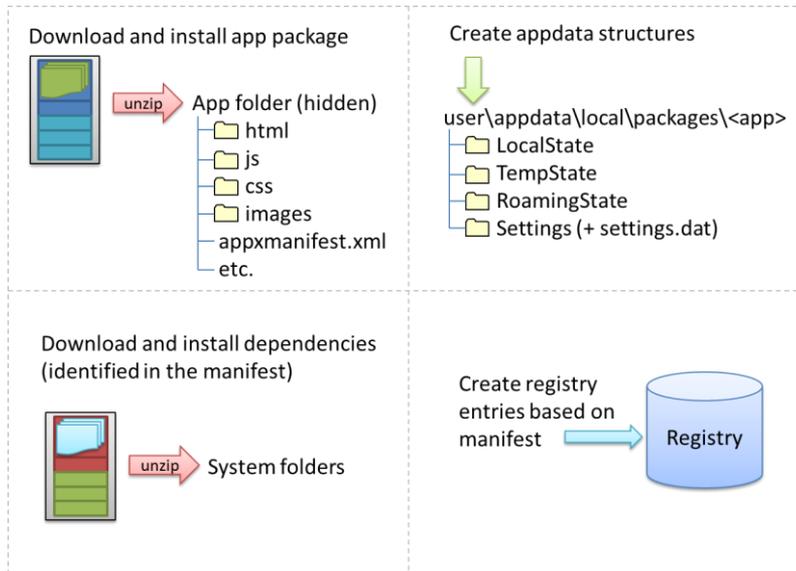


**FIGURE 1-2** A typical app page in the Windows Store, where the manifest in the app package determines what appears in the app permissions. PuzzleTouch, for example, declares the *Pictures Library*, *Webcam*, and *Internet (Client)* capabilities, which are shown by clicking Device Capabilities by the arrow.

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. (Note how Puzzle Touch does that for the camera.) 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.<sup>3</sup>

The user will also see your app pricing, of course, and whether you offer a trial period. Whatever the case, if they choose to install the app (getting it for free, paying for it, or accepting a trial), your app now becomes fully incarnate on a real user's device. The appx package is downloaded to the device and installed automatically along with any dependencies, such as the *Windows Library for JavaScript* (see "Sidebar: What is the Windows Library for JavaScript?"). As shown in Figure 1-3, the Windows deployment manager creates a folder for the app, extracts the package contents to that location, creates *appdata* folders (local, roaming, and temp, which the app can freely access, along with settings files for key-value pairs and some other system-managed folders), 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 the licensing agreement!

<sup>3</sup> The user always has the ability to disallow access to sensitive resources at run time for those apps that have declared the intent, as we'll see later. However, as those capabilities surface directly in the Windows Store, you want to be careful to not declare those that you don't really need.



**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 five 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. We like to describe this 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 an HTML implementation of a number of controls such that they appear as part of the DOM and can be styled with CSS like other intrinsic HTML controls. This is much more natural for developers than having to create an instance of some WinRT class, bind it to an 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. So WinJS 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 those 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 from 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**

WinJS is an example of a special shared library package that is automatically downloaded from the Windows Store for apps that depend on it. Microsoft maintains a few of these in the Store so that the package need be downloaded only once and then shared between apps. Shared third-party libraries are not currently supported.

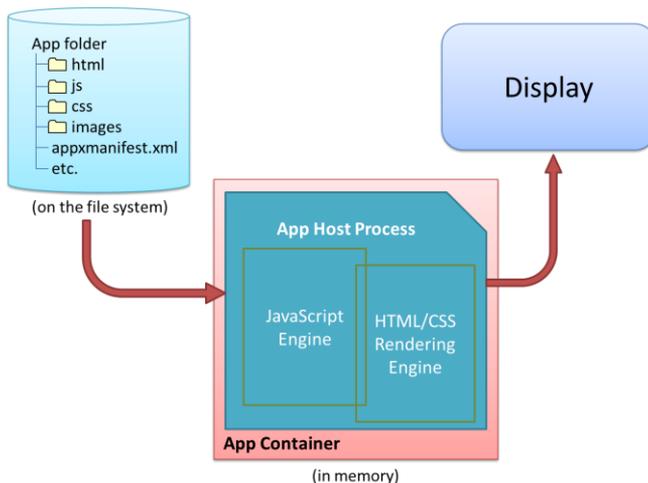
However, apps can freely use third-party libraries by bringing them into their own app package, provided of course that the libraries use only the APIs available to Windows Store apps. For example, apps written in JavaScript can certainly use jQuery, Modernizer, Dojo, prototype.js, Box2D, and others, with the caveat that some functionality, especially UI and script injection, might not be supported. 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."

## 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 hidden 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`<sup>4</sup>, 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.

---

<sup>4</sup> "wva" is an old acronym for Windows Store apps written in JavaScript; some things just stick....

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

- A number of methods in the DOM API are either modified or not available, depending on their design and system impact. For example, functions that display modal UI and block the UI thread are not available, like `window.alert`, `window.open`, and `window.prompt`. (Try `Windows.UI.Popups.MessageDialog` instead for some of these needs.)
- The engines support additional methods, properties, and even CSS media queries 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 get WinRT access nor cross-domain HTTP requests (though you can 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.

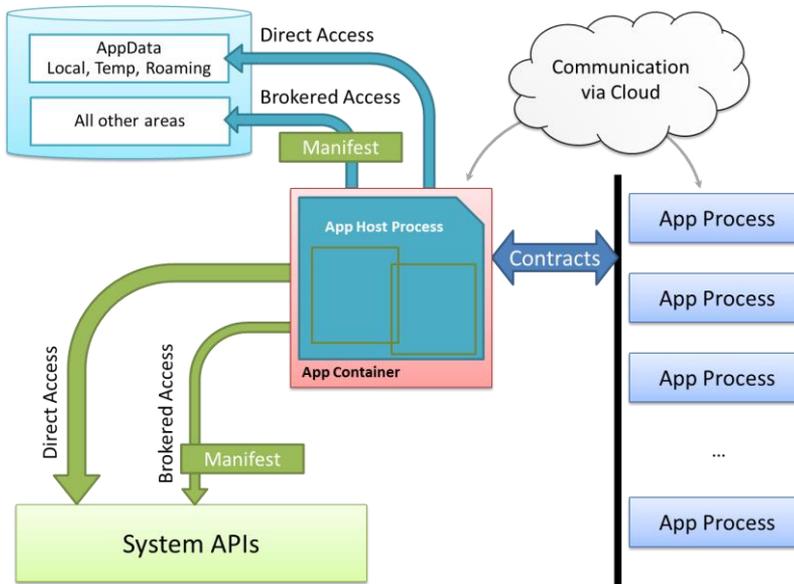
For full details on all these behaviors, see [HTML and DOM API changes list](#) and [HTML, CSS, and JavaScript features and differences](#) on the Windows Developer Center, <http://dev.windows.com>. As with the app manifest, you should become good friends with the Developer Center.

Now 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 with 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 don't need to 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 only access 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).

Such *mixed language apps* thus use HTML/CSS for their presentation layer and 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 17, "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 will work but incurs a significant memory overhead.

---

## 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, some apps will need more time to load, in which case you create an *extended splash screen*. This just means making the initial view of your main window look the same as the splash screen so that you can then overlay progress indicators or other helpful messages like "Go get a snack, friend, 'cause yer gonna be here a while!" Better yet, why not entertain your users so that they have fun with your app even during such a process? We'll see the details of extended splash screens in Chapter 3, "App Anatomy and Page Navigation."

Now, when a normally launched app comes up, 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!

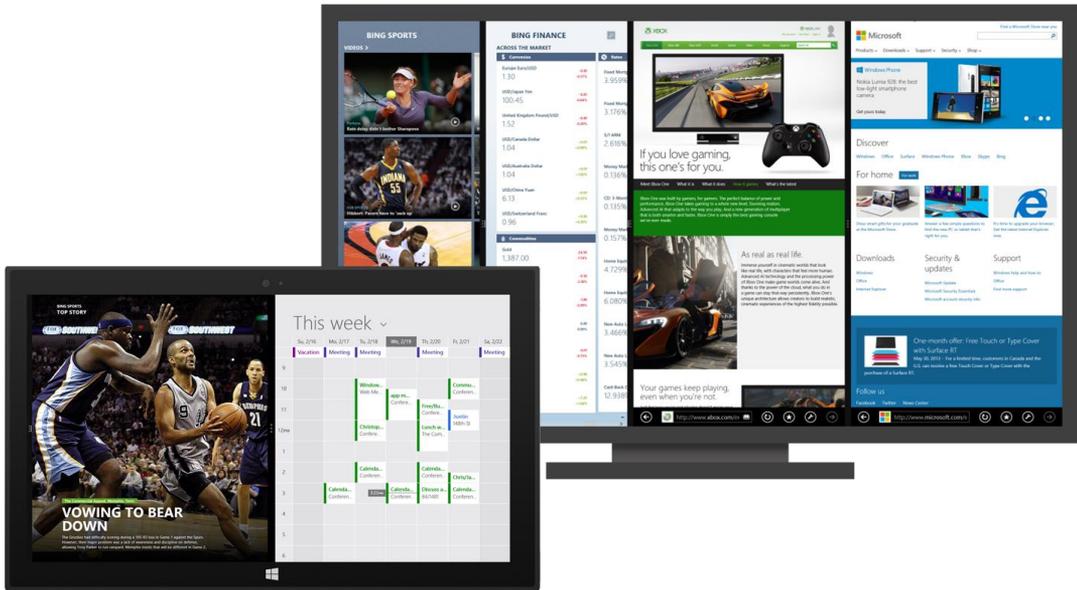
The purpose of those *edge gestures*—swipes from the edge of the screen toward the center—is to keep both system chrome and app commands (like menus and other commanding UI) out of the way until needed—an aspect of the design principle we call “content before chrome.” This helps the user fully stay immersed in the app experience. To be more specific, 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.

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 specify a preferred *startup orientation* in the manifest and can also *lock* the orientation when appropriate. For example, a movie player will generally want to lock into landscape mode such that rotating the device doesn't change the display. We'll see these layout details in Chapter 7, "Layout."

What's also true is that your app might not always be running full-screen. In landscape mode, you app can share the screen real estate with perhaps as many as four other apps, depending on the screen size.<sup>5</sup> (See Figure 1-6.) By default, Windows allows the user to resize an app down to 500 pixels wide, and you can indicate in your manifest that you support the narrower 320px width. In practical terms, it means that your app layout must be *responsive*, as it's called on the web, where you're able to 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 to remain visible. This makes it possible to really have two apps working together side-by-side. 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.

---

<sup>5</sup> For developers familiar with Windows 8, the distinct view states of filled, snapped, fullscreen-portrait, and fullscreen-landscape are replaced in Windows 8.1 Preview and beyond with this variable sizing.



**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 that can often be handled entirely within CSS media queries.

In narrow widths, especially the optional 320px minimum, apps will often change the view of their content or its level of detail. For instance, in portrait aspect ratios (height > width), horizontally oriented lists are typically switched to a vertical orientation, perhaps with fewer details. 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 its narrow views, the more likely it is that users will 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 an app 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 its view; it should just present itself appropriately for that width or orientation.

Beyond these partial views, an app should also expect to make good use of many different screen sizes. The app will be run on many different displays, anywhere from 1024x768 (the minimum hardware requirement), all the way up to resolutions like 2560x1440. The guidance here is that apps with fixed content (like a game board) will generally scale in size across different resolutions, whereas apps 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](#) topic.

It might also be true that you're running on a high-resolution device that also has a very small screen (high *pixel density*), such as the 10.6" Surface Pro that has a 1920x1200 resolution. 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, which need to accommodate those scales, as we'll see in Chapter 7.

As a final note, when an app is activated in response to a contract like Search or Share, its initial view might not be the full window at all but rather its specific landing page for that contract that overlays the current foreground app. We'll see these details in Chapter 13, "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 by 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 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 connectivity to its heart's content, either directly hosting content in its layout with the webview control or obtaining data through HTTP requests (Chapter 4). An app has many different controls at its disposal, as we'll see in Chapters 5 and 6, 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 7). An app can work with commanding UI like the app bar

(Chapter 8), manage state and user data (Chapters 9 and 10), and receive and process *pointer events*, which unify touch, mouse, and stylus (Chapter 11—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 11), rich media (Chapter 12), animations (Chapter 13), contracts (Chapter 14), *tiles and notifications* (Chapter 15), , and various devices and printing (Chapter 16). They can optimize performance and extend their capabilities through WinRT components (Chapter 17), and they can adapt themselves to different markets, provide accessibility, and work with various monetization options like advertising, trial versions, and in-app purchases (Chapter 18).

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
<i>Internet (Client)</i>	Outbound access to the Internet and public networks (which includes making requests to servers and receiving information in response). <sup>6</sup>	No
<i>Internet (Client &amp; Server)</i> (superset of <i>Internet (Client)</i> ; 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
<i>Private Networks (Client &amp; Server)</i>	Outbound and inbound access to home or work intranets (inbound access to critical ports is always blocked).	No
<i>Music Library</i> <i>Pictures Library</i> <i>Video Library</i> <sup>7</sup>	Read/write access to the user's Music/Pictures/Videos area on the file system (all files).	No
<i>Removable Storage</i>	Read/write access to files on removable storage devices for specifically declared file types.	No
<i>Microphone</i>	Access to microphone audio feeds (includes microphones on cameras).	Yes
<i>Webcam</i>	Access to camera audio/video/image feeds.	Yes
<i>Location</i>	Access to the user's location via GPS.	Yes
<i>Proximity</i>	The ability to connect to other devices through near-field communication (NFC).	No

<sup>6</sup> Note that network capabilities are not necessary to receive push notifications because those are received by the system and not the app.

<sup>7</sup> The *Documents Library* capability that was present in Windows 8 was removed as of Windows 8.1 Preview because the scenarios that actually needed it could be handled through file pickers.

<i>Enterprise Authentication</i>	Access to intranet resources that require domain credentials; not typically needed for most apps. Requires a corporate account in the Windows Store.	No
<i>Shared User Certificates</i>	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.



**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.

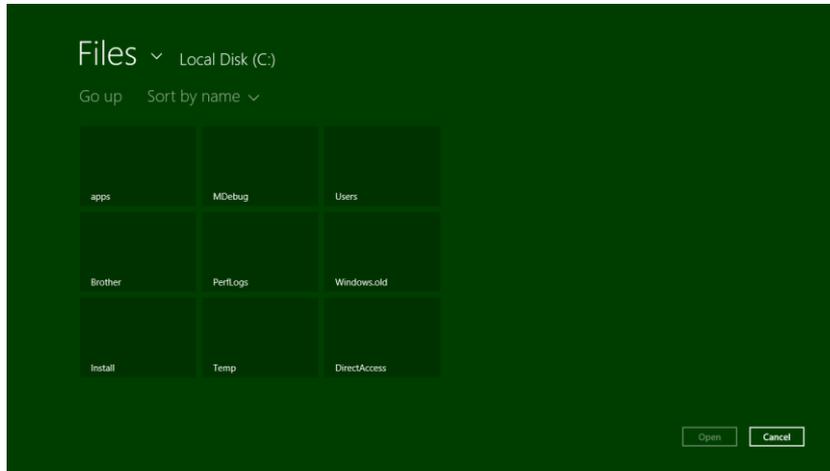
When you first start writing apps, really keep the manifest and these capabilities in mind—if you forget one, you’ll see APIs failing even though all your code is written perfectly (or was copied from a working sample). In the early days of building the first Windows Store apps at Microsoft, we routinely 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. The manifest is also where you declare things like your preferred orientation, *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 nonsystem areas of the file system—and any type of file—from within the file picker UI. (See Figure 1-8.) This explicit user action, in other words, is taken as consent for your app to access that particular file or folder

(depending on what you're asking for). Once your 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.

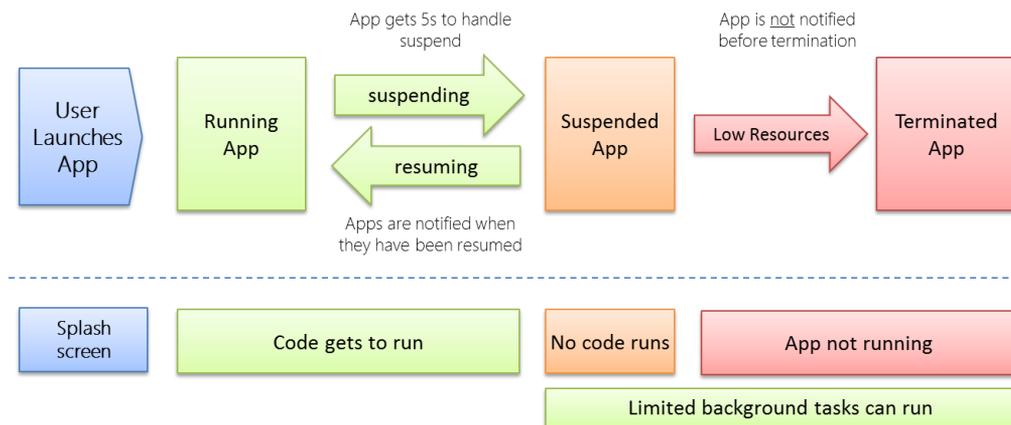
This automatic shutdown or sending the app to the background are examples of automatic *lifecycle management* for Windows Store apps that helps conserve power and optimize battery life. One reality of traditional multitasking operating systems is that users typically leave a bunch of apps running, all of

which consume power. This made sense with desktop apps because many of them can be at least partially visible at once. But for Store apps, the system is boldly taking on the job itself and using the full-screen nature of those apps 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—to offload downloads and uploads from app code, which means apps don't have to be running for such transfers to happen. 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) so that they need not even be running for this purpose—see Chapter 15, “Tiles, Notifications, the Lock Screen, and Background Tasks.” 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 12, “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 15, an app declares background tasks in its manifest that are tied to specific functions in their code. In this case, Windows will wake the app from the suspended state 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, because Windows Store policy specifically disallows apps with their own close commands or gestures—she still rightly thinks that the app is running. If the user 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

---

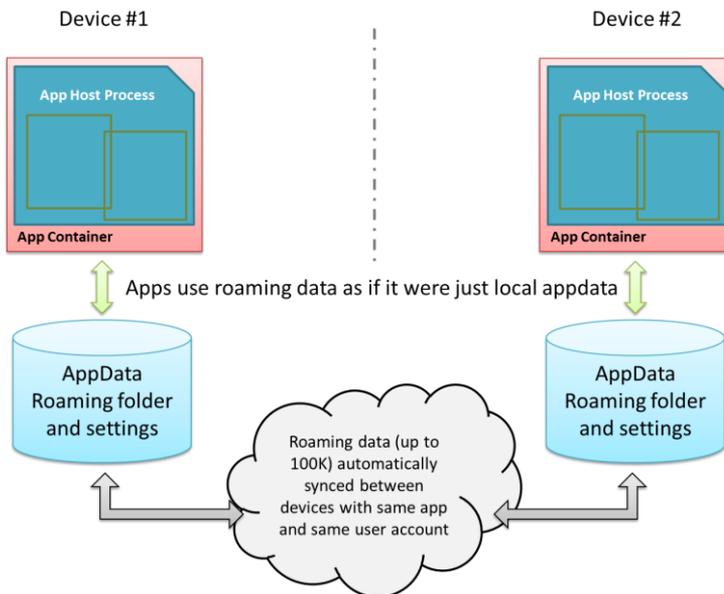
To step back for a moment, one of the key differences between traditional desktop applications and Windows Store apps is that the latter are inherently stateful. That is, once they’ve run the first time, they remember their state across invocations (unless explicitly closed by the user or unless they 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?”<sup>8</sup> 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 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 too. 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 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.

<sup>8</sup> 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. So in the vignette he’s shown in a straitjacket on the cover of his newest book, *Who am I?*

The 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" appended.) 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 9, "The Story of State.")

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 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 > Sync Your 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, I'm told that 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, app cache, and so forth. All of these will be stored within the app's LocalState folder.

## 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 you 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 too.)

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 often 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. 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, 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.

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 8—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. But let's be honest: as a developer, do you really want to ponder what "fast and fluid" means (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 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

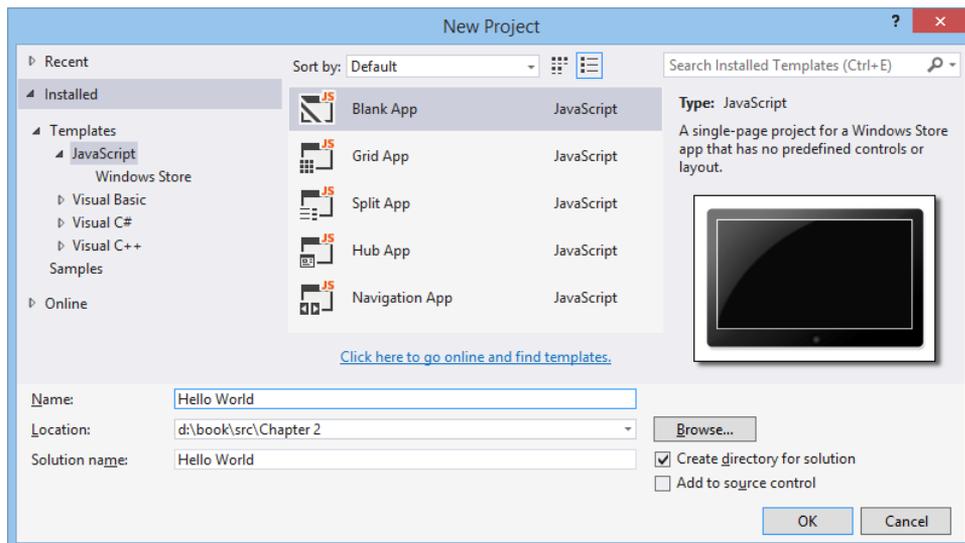
This is a book about developing apps. So, to quote Paul Bettany’s portrayal of Geoffrey Chaucer in *A Knight’s Tale*, “without further gilding the lily, and with no more ado,” let’s create some!

## A Really Quick Quickstart: The Blank App Template

---

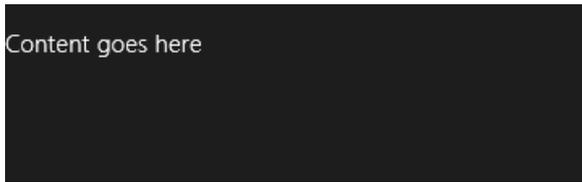
We must begin, of course, by paying due homage to the quintessential “Hello World” app, which we can achieve without actually writing any code at all. We simply need to create a new app from a template in Visual Studio:

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 Application 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.

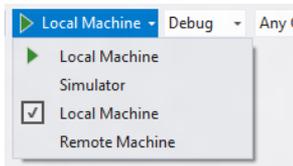
4. After Visual Studio churns for a bit to create the project, click the Start Debugging button (or press F5, or select the Debug > Start Debugging menu command). Assuming your installation is good, you should see something like Figure 2-2 on your screen.



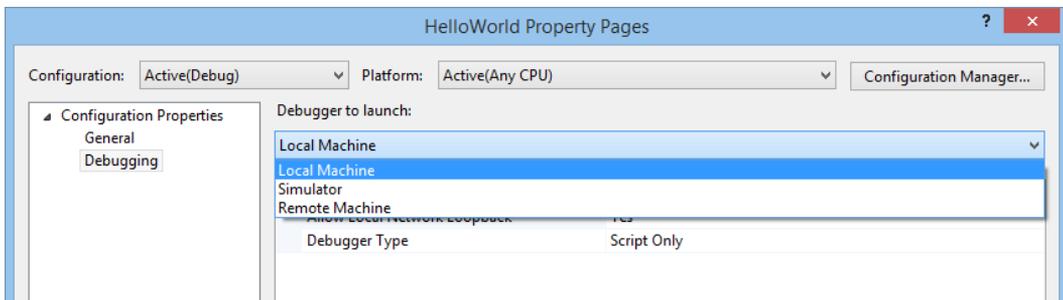
**FIGURE 2-2** The only vaguely interesting portion of the Hello World app's display. The message is at least a better invitation to write more code than the standard first-app greeting!

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.<sup>1</sup>

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.

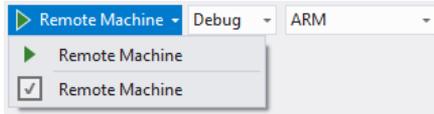
The Remote Machine option allows you to run the app on a separate device, which is absolutely essential for working with devices that can't run desktop apps at all, such as the Microsoft Surface and

---

<sup>1</sup> 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.

other ARM devices. Setting this up is a straightforward process: 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 the Attach To Process command, which allows you to debug an already-running app. See [How to start a debugging session \(JavaScript\)](#).

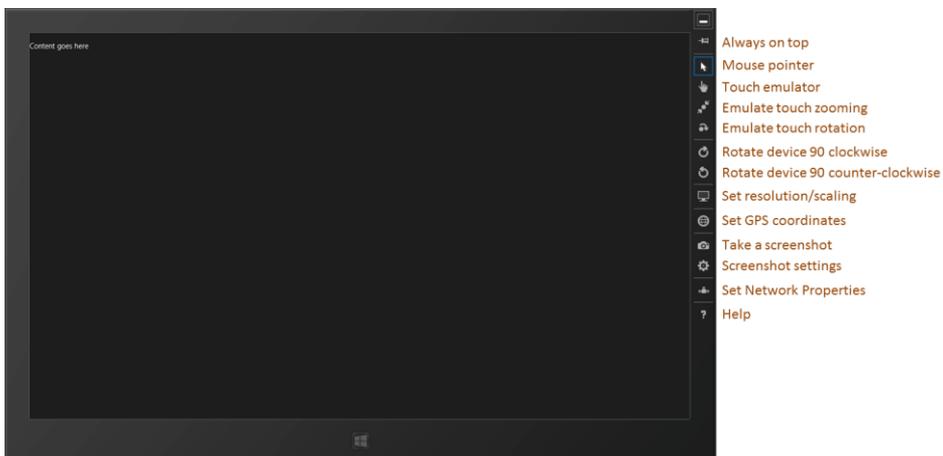
**Tip** If you ever load a Windows SDK sample into Visual Studio and Remote Machine is the only debugging option that's available, the build target is probably set to ARM (the rightmost drop-down):



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 17, "Windows Runtime 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 an ARM device (such as the Microsoft Surface), 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, 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 really 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 any project. 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 **css** folder contains a default.css file where you'll see media query structures for the four view states that all apps should honor. We'll see this in action in the next section, and I'll discuss all the details in Chapter 7, "Layout."

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

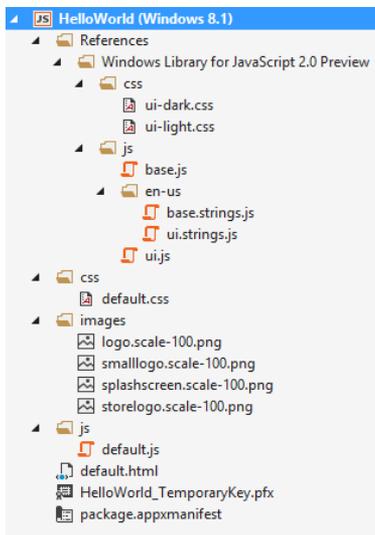
scaled versions too, as we'll see in Chapter 3, "App Anatomy, Page Navigation, and Promises"):

- **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.1.0/css/ui-dark.css" rel="stylesheet">
  <script src="//Microsoft.WinJS.1.0/js/base.js"></script>
  <script src="//Microsoft.WinJS.1.0/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 and see the effects.

**Tip** When referring to in-package resources, always use a leading `/` on URIs. This is especially important when using page controls (see Chapter 3) because those pages are loaded into a different location in the DOM than where they exist in the project structure.

Where the 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 comment block):

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

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

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

```

    }
    args.setPromise(WinJS.UI.processAll());
  }
};

app.oncheckpoint = function (args) {
};

app.start();
})();

```

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

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

Now, what is that `args.setPromise(WinJS.UI.processAll())` for? As we'll see many times, `WinJS.UI.processAll` instantiates any WinJS controls that are declared in HTML—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.<sup>2</sup> 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 were 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.<sup>3</sup>

---

<sup>2</sup> There is a similar function `WinJS.Binding.processAll` that processes `data-win-bind` attributes (Chapter 5), and `WinJS.Resources.processAll` that does resource lookup on `data-win-res` attributes (Chapter 18).

<sup>3</sup> 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

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, Control Styling, and Data Binding."

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

**Get familiar with Visual Studio** If you're new to Visual Studio, the tool can be somewhat daunting at first because it supports many features, even in the Express edition. For a quick, roughly 10-minute introduction, Video 2-1 in this chapter's companion content 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 particular 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 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. For example, the code I use below to implement camera capture and sharing data came directly from a couple of samples. (Again, if a sample seems to support only the Remote Machine debugging option, check the build target and set it to Any CPU.)

I also *strongly* encourage you to spend a half-day, even a full day, getting familiar with Visual Studio and Blend for Visual Studio, and just running samples so that you know what tremendous resources are available to you. 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 year or two and 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, 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 8, the *dynamic* aspects of an app—animations 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 12, "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.

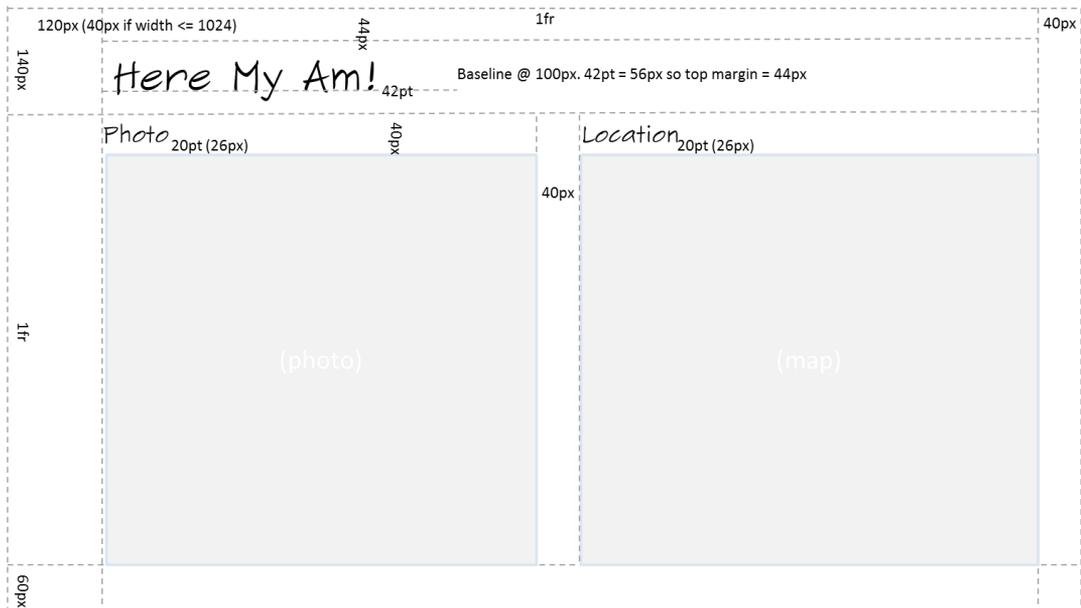
Generally speaking, layout is based on a basic 20 pixel unit, with 5 pixel sub-units. In the full landscape view of Figure 2-7, you can see the recommended left margin of 120px, the recommended top margin of 140px above the content region, and placement of the header's baseline at 100px, which

for a 42pt font translates to a 44px top margin. For partial landscape views with width  $\leq 1024$ px, 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.

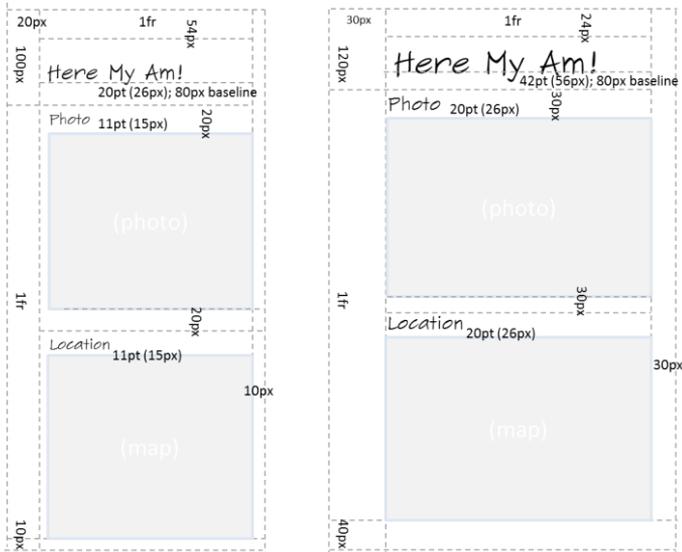
**What happened to snapped and filled views?** In the first release of Windows 8, app design focused on four view states known as landscape, portrait, filled, and snapped. With Windows 8.1 Preview, distinct names for these states are deprecated in favor of simply handling different display sizes and 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 narrow view (formerly known as snapped) in the 320-499px range. 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.

With this change, you can use the standard CSS3 media queries of landscape and portrait to differentiate aspect ratios; the view state media queries from Windows 8 still work as well, but 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 stylesheets in Windows 8 but were removed 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 ( $<1024$ px) widths. The “1fr” labels denote proportional parts of the CSS grid (see Chapter 7) 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  $\leq 1$ ).

## 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 320px 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 7, 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 how to then 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) or development (coding and debugging). 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>
      
    </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.<sup>4</sup>

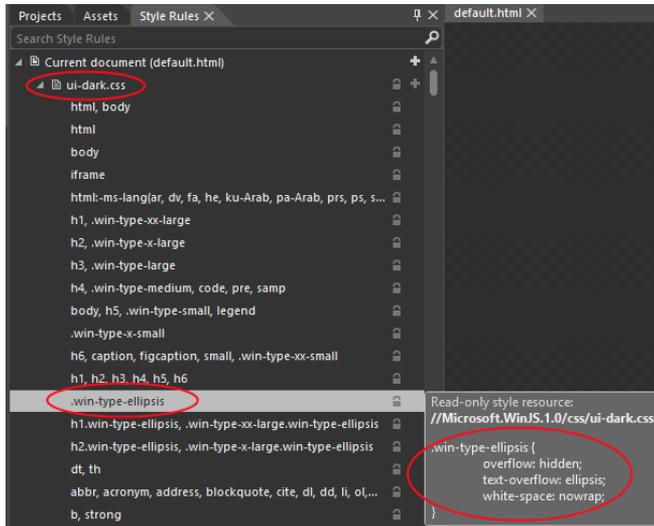
You'll see that some elements have style classes assigned to them. Those that start with `win-` come from the WinJS stylesheet.<sup>5</sup> You can browse these in Blend by using the Style Rules tab, shown in

---

<sup>4</sup> 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.

<sup>5</sup> 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

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. This is here so you don't waste your time 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`), so we don't need to create an absolute URI for in-package content. (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.

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

---

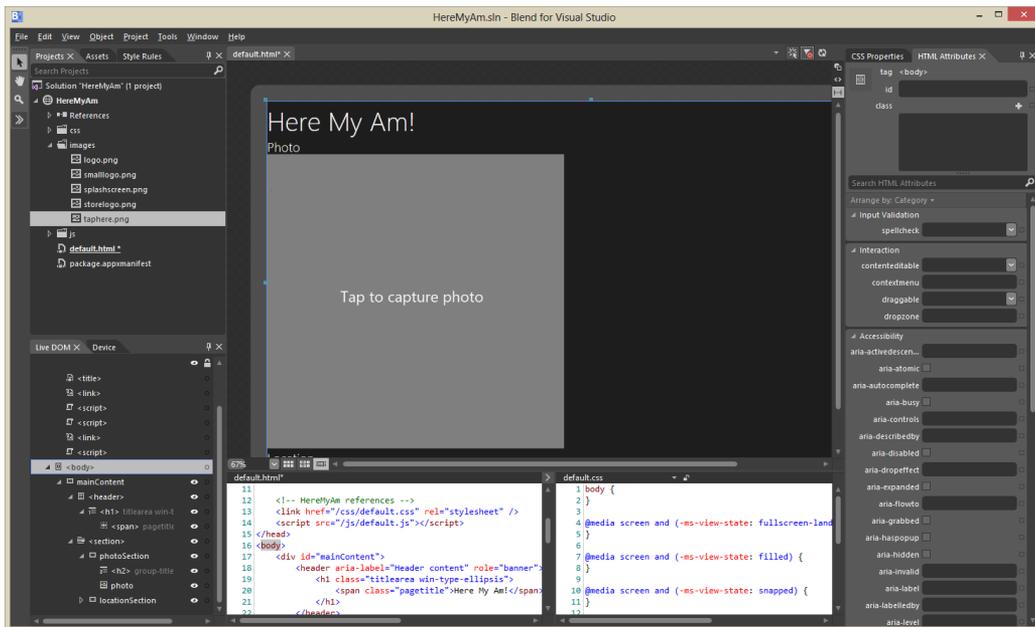
photo capture. The light stylesheet is recommended for apps that work more with textual content.

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 18, "Apps for Everyone," 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 18 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.

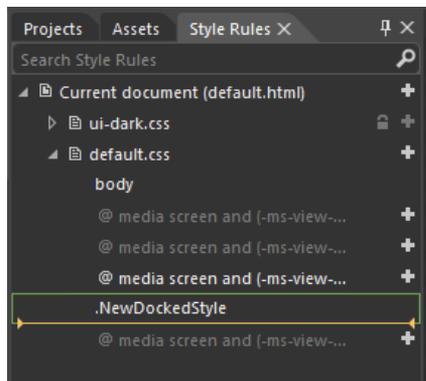
The tabs along the upper left give you access to your Project files, Assets like all the controls you can add to your UI, and a browser for all the Style Rules defined in the environment. On the lower left side, the Live DOM tab lets you browse your element hierarchy and the Device tab lets you set orientation,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

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 will also contain an item called “Computed Values” which will show you the exact values applied through the styles, such as the actual sizes of variable parts of a CSS grid.

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 a couple 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` after `.NewDockedStyle`, which is itself inside a media query:

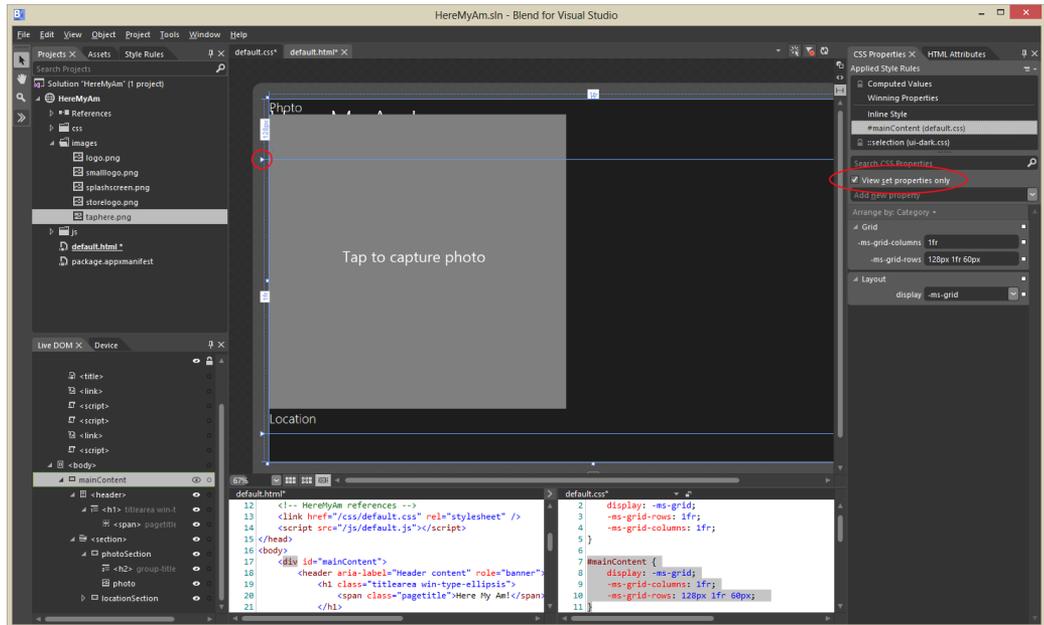


- In the Live DOM pane (the lower left side in Blend), right-click the element you want to style and select Create Style Rule From Element Id or Create Style Rule From Element Class. This will create a new style rule (at the insertion point indicated in Style Rules above). Then in the CSS properties 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 that will mean lots of hand-editing later on. So you might as well save yourself the trouble!

- 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; and -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.

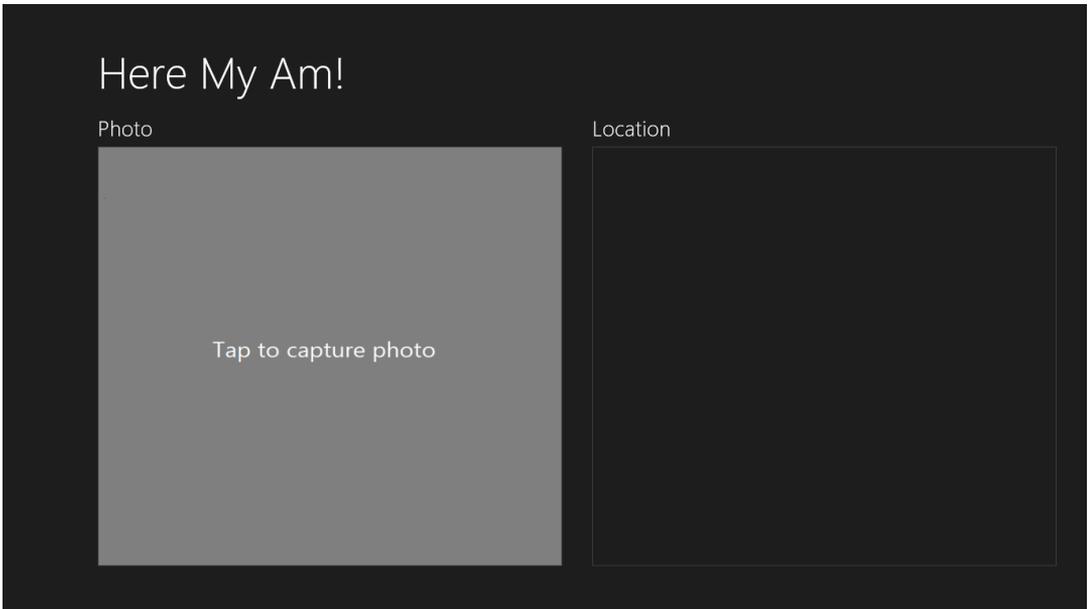


**FIGURE 2-11** Setting the grid properties for the `mainContent` `div`. Notice how the View Set Properties Only checkbox (upper right) makes it easy to see what styles are set for the current rule. Also notice how the grid rows and columns appear on the artboard, including sliders (circled) to manipulate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

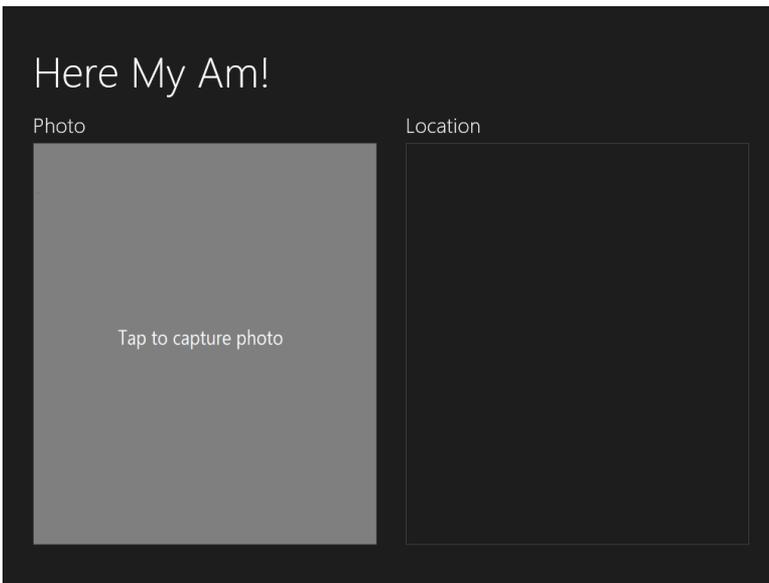
Showing this and the rest of the styling—going down into each level of the markup and creating appropriate styles in the appropriate media queries for the view states—is best done in video. Video 2-2 (available with this book's downloadable companion content) shows this process starting with the creation of the project, styling the different views, and switching to Visual Studio (right-click the project name in Blend and select Edit In Visual Studio) to run the app in the simulator for verification. It also demonstrates the approximate time it takes to style such an app once you're familiar with the tools.

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

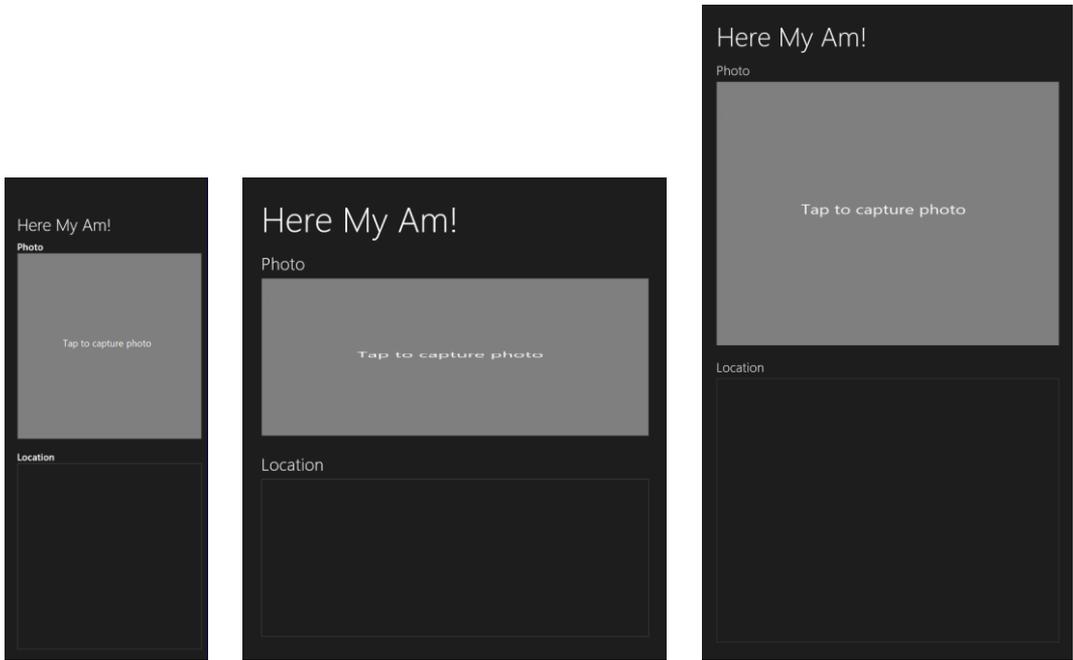
with! (And the time savings are even greater with Interactive Mode; see Video 5-1 in the companion content created for Chapter 5.)



**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 the 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.<sup>6</sup> 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

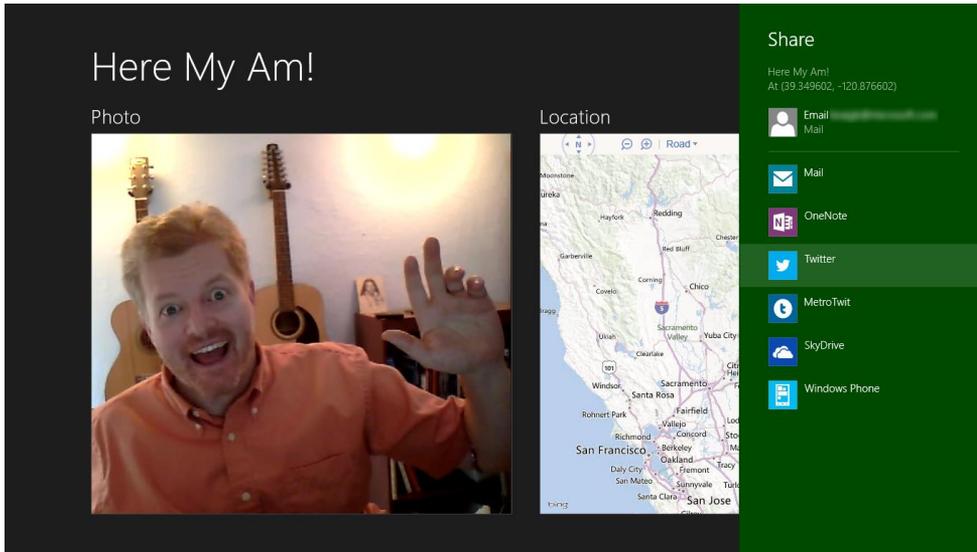
---

<sup>6</sup> 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.

`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 (and if you think that the coordinates shown here pinpoint my house, the truth is they'll send you out into the bushes a few miles away).

## 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 9, "The Story of State."

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:<sup>7</sup>

---

<sup>7</sup> Note that you should replace the credentials inside the `init` function with your own key obtained from <https://www.bingmapsportal.com/>.

```

<!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
        });
      }
    </script>
  </head>
</html>

```

```

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.

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 `onready` handler (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;

//Add this after the app.onactivated handler
app.onready = function () {
    var gl = new Windows.Devices.Geolocation.Geolocator();

    gl.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. To obtain coordinates, you can use either the WinRT or HTML5 geolocation APIs. The two are almost equivalent, with slight differences described in Chapter 10, “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.

Next, in the second parameter to `postMessage` (both in `default.js` and `map.html`) you see a combination of `ms-appx://` or `ms-appx-web://` with `document.location.host`. This essentially means “the current app from the [local or web] context,” which is the appropriate origin of the message. Notice that 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](#).

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, such 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). It 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 17).

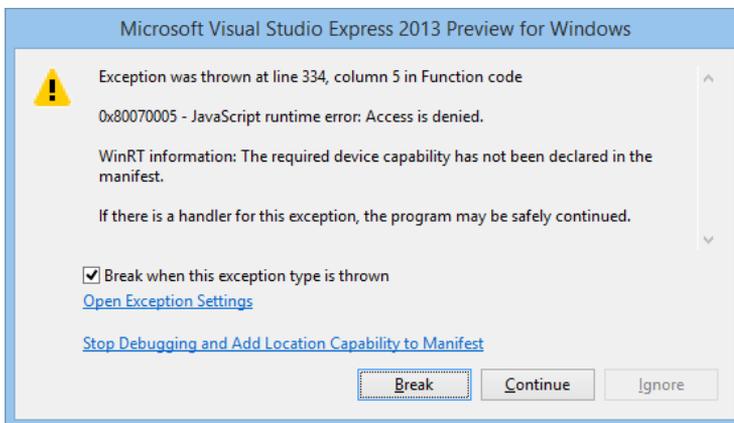
The argument passed to the completed handler contains the results of the async call, which in our example above is a `Windows.Geolocation.Geoposition` object containing the last reading. (When reading the docs for an async function, you'll see that the return type is listed like `IAsyncOperation<Geoposition>`; the name within `<>` indicates the actual data type of the results, so refer to the docs on that type for its details.) 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.<sup>8</sup>

**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:



---

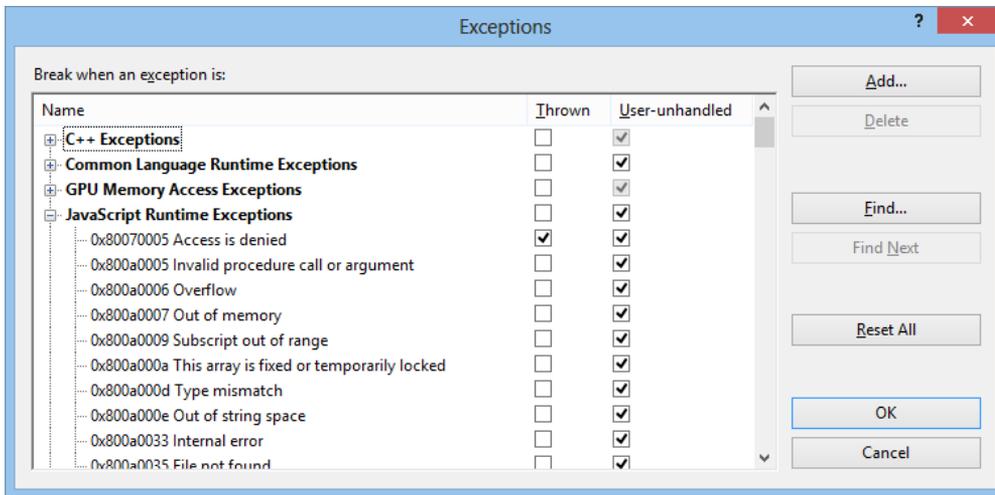
<sup>8</sup> Later, in the section "Receiving Messages from the `iframe`," we'll make the pushpin draggable and show how the app can pick up location changes from the map.

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:

```
g1.getGeopositionAsync().done(function (geocoord) {  
    //...  
}, function(error) {  
    console.log("Unable to get location.");  
});
```

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 8, “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 (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.

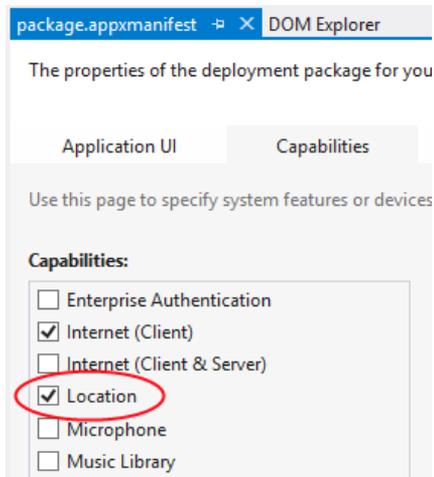


**FIGURE 2-16** JavaScript run-time exceptions in the Debug/Exceptions dialog of Visual Studio.

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.



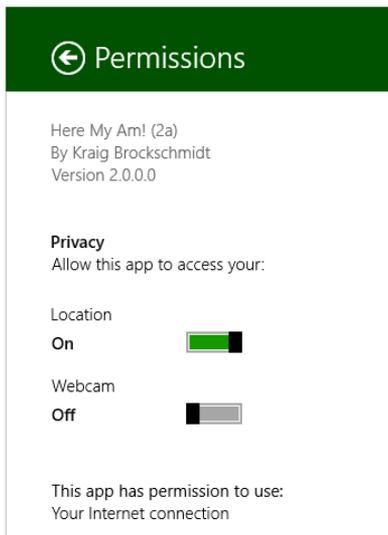
**FIGURE 2-17** Setting the *Location* capability in Visual Studio's manifest editor. (Note that Blend supports editing the manifest only as XML.)

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 again be invoked as the API will throw an *Access denied* exception.

Keep in mind that this consent dialog will only appear once for any given app, even across debugging sessions. 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 8. 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.

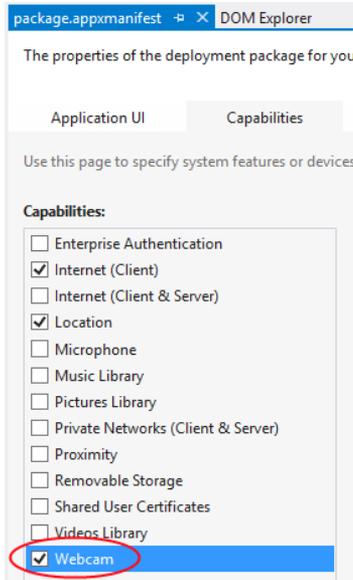
## Sidebar: Writing Code in Debug Mode

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 that new code in place.)

## 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; see Chapter 11.)

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 10:

```
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 function (see the function inside `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 other possibilities as discussed in Chapter 11), and then call `captureFileAsync`. This will check the manifest capability and prompt the user for consent, if necessary.

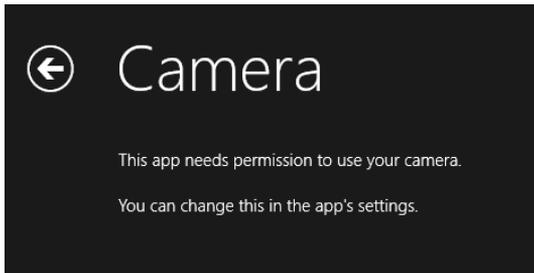
This is an async call, so we hook a `.done` on the end with a completed handler, which in this case will receive a `Windows.Storage.StorageFile` object. Through this object you can get to all the raw image data you want, but for our purpose we simply want to display it in the `img` element. That's easy 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 `StorageFile` object to the `URL.createObjectURL` method and get back an URI that can be directly assigned to the `img.src` attribute. The captured photo appears!<sup>9</sup>

---

<sup>9</sup> The `{oneTimeOnly: true}` parameter 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 would

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 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 view state of the app is too small for the capture UI to be usable, in which case `error.message` will say "A method was called at an unexpected time." As we'll see in Chapter 7, you can check the app's view state and take other action under such conditions, but for now we'll just fail silently.

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

---

leak memory with each new picture. 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.

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.<sup>10</sup> First we just need to set up an appropriate listener—place this code 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 has nothing to do with particular view states like landscape or portrait; it's really just a way of saying, "get the singular instance of this system object that's related to the current app." You use `getForCurrentView` instead of creating an instance with `new` because you only ever need one instance of such objects for any given app. `getForCurrentView` will instantiate the object if necessary, or return one that's already available.

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);
```

---

<sup>10</sup> 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.

```

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.

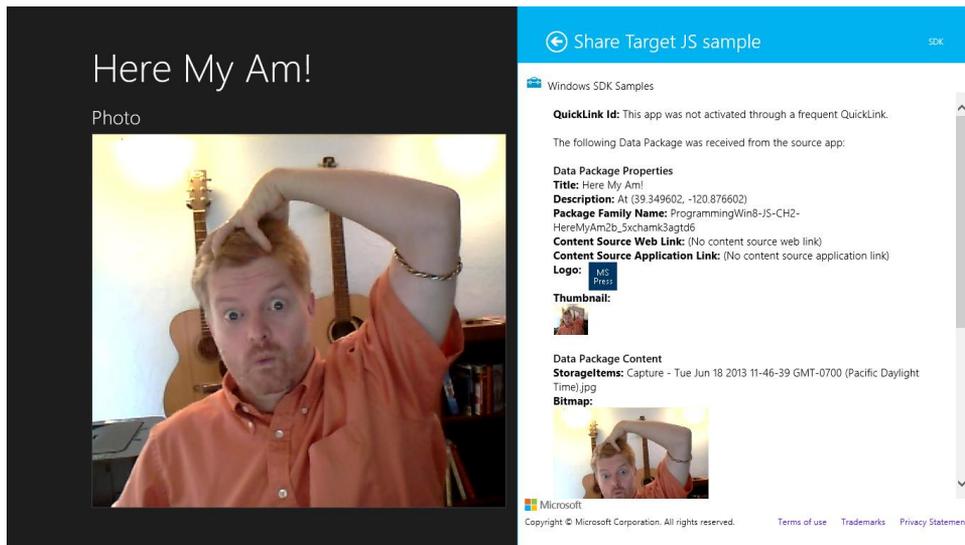
//Put the image file in an array and pass it to setStorageItems
data.setStorageItems([lastCapture]);

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

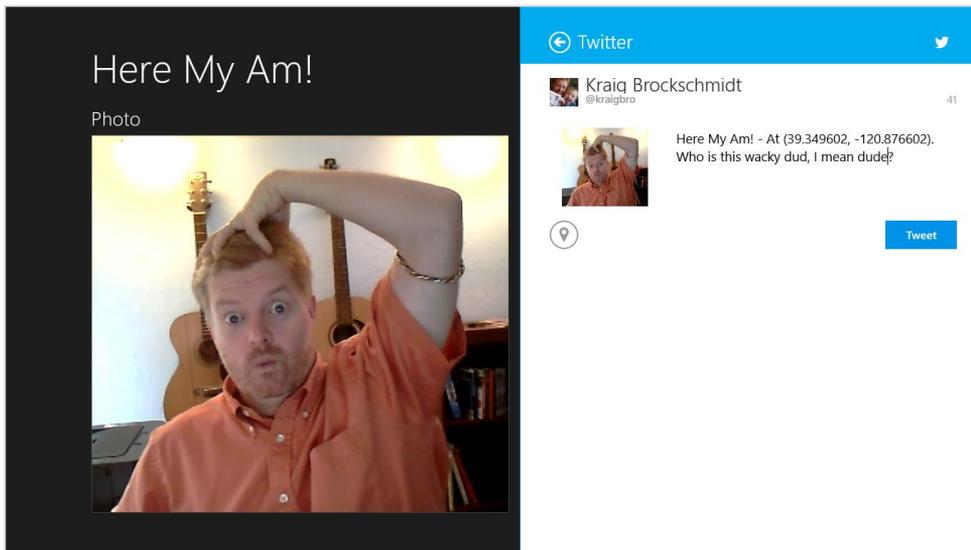
```

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 13, "Contracts." We'll also talk more about files and streams in Chapter 9.

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.



**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 upon:

- 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.
- Although automatically cropping the captured image to the size of the photo display area, this 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 view states.
- 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 8 when we implement an app bar command for this purpose).

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 will also give us the opportunity in Chapter 18 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 18 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 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 an event of our choosing:

```
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 18). 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 (a deterrent to users downloading it).

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 use a localized string later on) on a `canvas` element and then use the HTML blob APIs 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 });
}

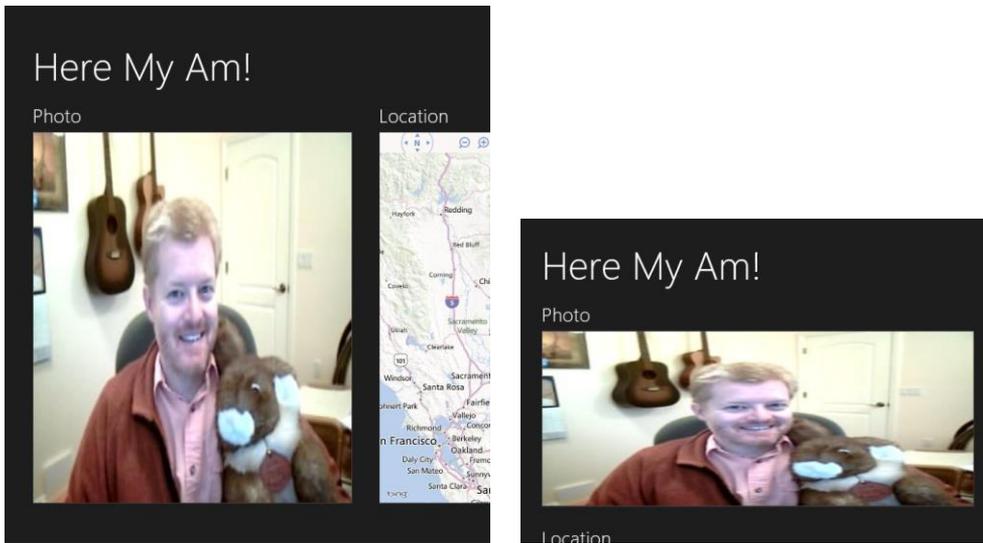
```

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, 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">
  
</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;
  }

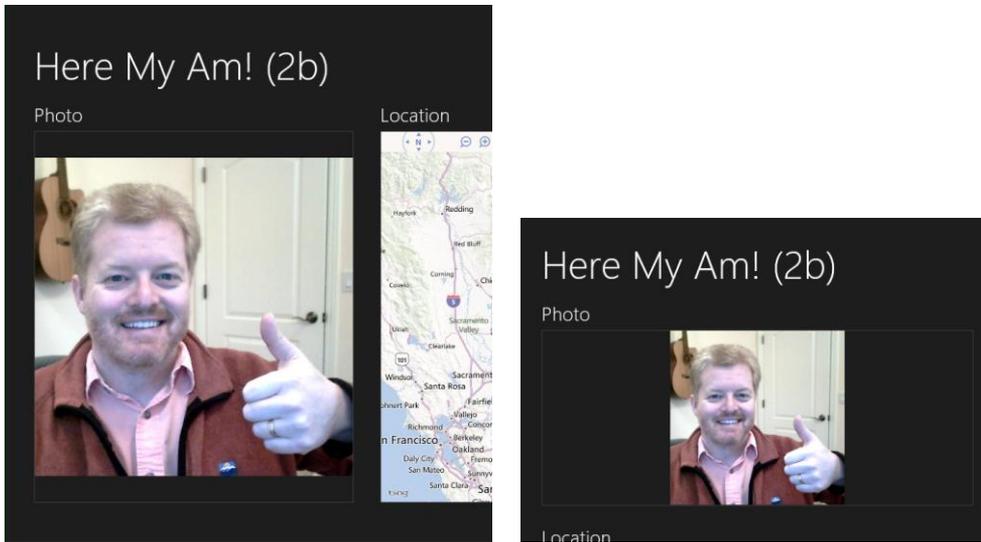
  //If we have an image, scale it, otherwise regenerate the placeholder
  if (lastCapture != null) {
    scaleImageToFit(photoImg, document.getElementById("photo"), lastCapture);
  }
}
```

```

    } 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\ProgrammingWin8-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 9), 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).

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 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, as we'll discuss in Chapter 4, and the `ms-resource` protocol for resources, as described in Chapter 18.)

To move the file, we can use its built-in `StorageFile.copyAsync` method, which requires a target `StorageFolder` object and a new name. We can 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.
- Create or open the HereMyAm folder in local appdata, resulting in a folder object.
- Copy the captured file to that folder, resulting in a new file.
- Delete the temp file, which has no results.

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 other's completed handlers—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). We'll talk more about all this in Chapter 3.

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 picture 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.

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 only with 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. We'll close this chapter, then, with a short introduction to these very handy tools to get you started on your own projects.

### Navigation 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 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 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 6, "Collections and 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 Template

*“A three-page project for a Windows Store app that implements the hub navigation patterns 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 7.

## 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 view states 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.

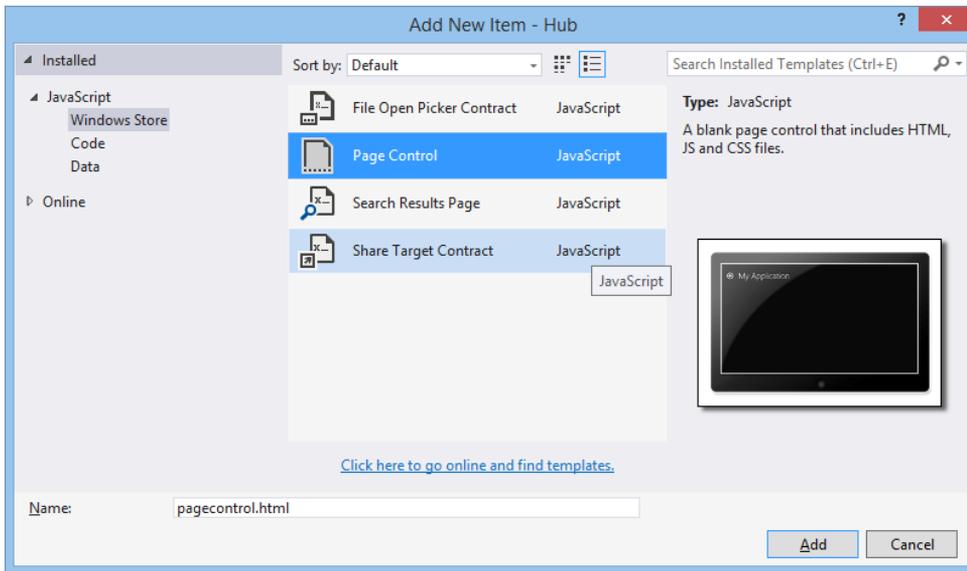


FIGURE 2-24 Available item templates for a Windows Store app written in JavaScript.

## 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 view states, 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 view states, 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, Page Navigation, and Promises

During the early stages of writing this book (the first edition, at least), I was also 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!

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 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 14, "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, “Using 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. A few other fundamentals, like input, we’ll pick up 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.

**Note** This system-provided splash screen is composed of only your splash screen image and your background color and does not allow any customization. However, through an *extended* splash screen, described later, you can control the entire display.

Additional graphics and settings in the manifest also affect your branding and overall presence in the system, as shown in the table below. 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.)

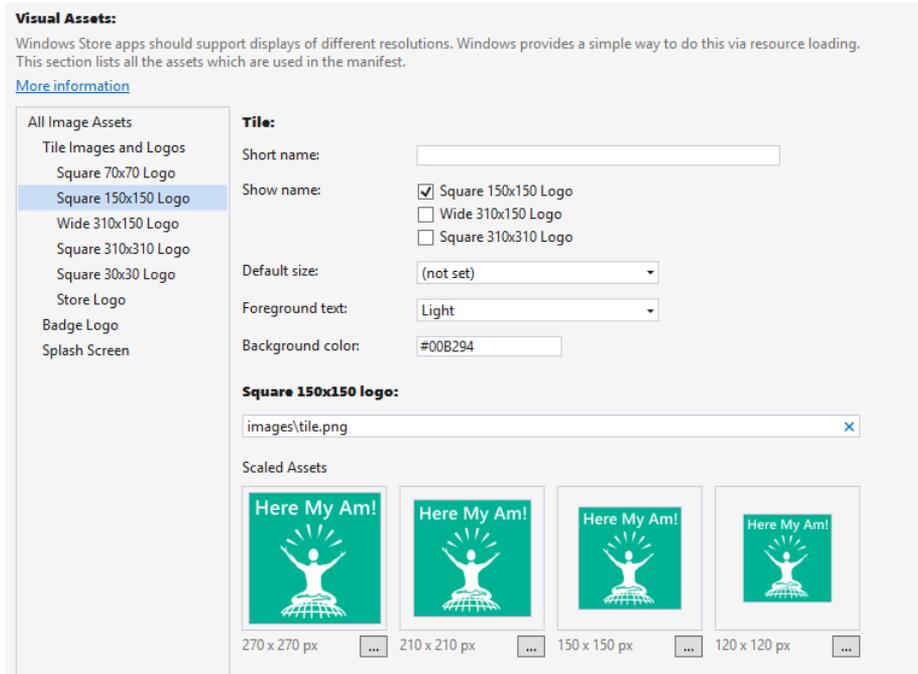
You can see that the table lists multiple sizes for various images specified in the manifest (on the Application UI tab) to accommodate varying pixel densities: 100%, 140%, and 180% scale factors, and even a few at 80% (don't neglect the latter: they are typically used for most desktop monitors). Although 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.

All items in this table are found in the Application UI > Visual Assets section of the manifest editor (except for the first, which is found at the top). 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.

Item	Use	Image Sizes 100%	140%	180%
Display Name (at the top of the Application UI tab)	Appears in the "all apps" view on the Start screen, search results, the Settings charm, and in the Store.	n/a	n/a	n/a
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.	n/a	n/a	n/a

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.	n/a	n/a	n/a
Tile > Default size	Indicates whether to show the square or wide tile on the Start screen after installation.	n/a	n/a	n/a
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 <a href="#">Paciello Group's Contrast Analyzer</a> for more.	n/a	n/a	n/a
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.	n/a	n/a	n/a
Splash Screen > Background color	Color that will fill the majority of the splash screen; if not set, the App UI Background color is used.	n/a	n/a	n/a
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.)	70x70 (+ 80% scale at 56x56)	98x98	126x126
Square 150x150 logo	Square tile image for the Start screen.	150x150 (+ 80% scale at 120x120)	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.	310x150 (+80% scale at 248x120)	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.	310x310 (+80% scale at 248x248)	434x434	558x558
Square 30x30 logo	Tile used in zoomed-out and "all apps" views of the Start screen, and in the Search and Share panes if the app supports those contracts as targets. Also used on the app tile if you elect to show a logo instead of the app name in the lower left corner of the tile. Note that there are also four "Target size" icons that are specifically used in the desktop file explorer when file type associations exist for the app. We'll cover this in Chapter 13, "Contracts."	30x30 (+80% scale at 24x24)	42x42	54x54
Store logo	Tile/logo image used for the app on its product description page in the Windows Store. This image appears only in the Windows Store and is not used by the app or system at runtime.	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;	24x24	33x33	43x43

	see Chapter 14, “Alive with Activity”).			
Splash screen	When the app is launched, this image is shown in the center of the screen against the Background color. The image can utilize transparency if desired.	620x300	868x420	1116x540



**FIGURE 3-1** Visual Studio’s Visual Assets section in the Application UI tab of the manifest editor. It automatically detects whether a scaled asset exists for the base filename (such as images\logo.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 18, "Apps for Everyone."

**Tip** Two other branding-related resources you might be interested in are the [Branding your Windows Store app](#) topic in the documentation (covering design aspects) and the [CSS styling and branding your app sample](#) (covering CSS variations and dynamically changing the active stylesheet).

## 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 UI tab of the manifest editor.<sup>1</sup>
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 below.)
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. `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.
6. 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.
7. `window.onload` fires. At this point you trust that document layout is complete, as when performing initialization that's sensitive to element size.

What's also very different 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.

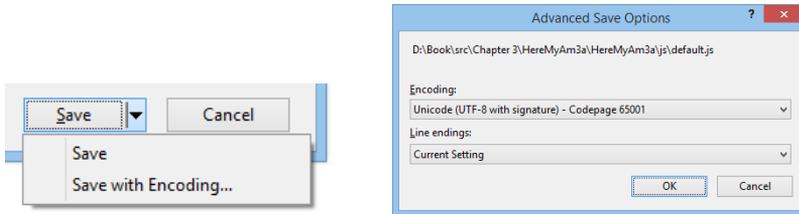
For the time being, though, let's concentrate on how we work with this core launch process, and because you'll generally do your initialization work within the `activated` event, let's examine that structure more closely.

---

<sup>1</sup> 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.

## 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.

## 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, Control Styling, and Data Binding."
  - `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:

<b>args.details Property</b>	<b>Type (in Windows.Application-Model.Activation)</b>	<b>Description</b>
<code>kind</code>	<a href="#">ActivationKind</a>	The reason for the activation. The possibilities are <code>launch</code> (most common); <code>restrictedLaunch</code> (specifically for app to app launching); <code>search</code> , <code>shareTarget</code> , <code>file</code> , <code>protocol</code> , <code>fileOpenPicker</code> , <code>fileSavePicker</code> , <code>contactPicker</code> , and <code>cachedFileUpdater</code> (for servicing contracts); and <code>device</code> , <code>printTask</code> , <code>settings</code> , and <code>cameraSettings</code> (generally used with device apps). For each supported activation kind, the app will have an appropriate initialization path.
<code>previousExecutionState</code>	<a href="#">ApplicationExecutionState</a>	The state of the app prior to this activation. Values are <code>notRunning</code> , <code>running</code> , <code>suspended</code> , <code>terminated</code> , and <code>closedByUser</code> . Handling the <code>terminated</code> case is most common because that's the one where you want to restore previously saved session state (see "App Lifecycle Transition Events").
<code>splashScreen</code>	<a href="#">SplashScreen</a>	Contains an <code>ondismissed</code> event for when the system splash screen is dismissed. This also contains an <code>imageLocation</code> property ( <code>Windows.Foundation.Rect</code> ) with coordinates where the splash screen image was displayed, as noted later in "Extended Splash Screens."

Additional properties provide relevant data for the activation. For example, `launch` provides the `tileId` and `arguments` from secondary tiles (see Chapter 14), as well as a `preLaunchActivated` flag (a Boolean) that indicates the app can skip visual aspects of startup like extended splash screens. 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.<sup>2</sup> This is fired before `activated`.
- `activated` Queued in the local context for `Windows.UI.WebUI.WebUIApplication.onactivated` (and again 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` 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 9, "The Story of State.")

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 9.

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

---

<sup>2</sup> 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`.

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. Clearly, your handling of the event won't really be complete until those operations are finished. But because they're 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!

For this reason, you need a way to tell Windows and WinJS to defer their default behaviors until the 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.

Let's see first how this works 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 “Extended Splash Screens” 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 “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`.

## Extended Splash Screens

Though the default splash screen helps keep the user engaged when they launch an app, the user won’t stay engaged if that same splash screen stays up for a really long time. In fact, “a really long time” for a typical consumer amounts to all of 15 seconds (if that), at which point he or she will pretty much start to 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, Windows has an automatic 15-second timeout for launching an app. If the 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.)

**Note** The 15-second timeout is entirely independent from the deferral mechanism as described in the previous section. Using a deferral applies only to handling async operations during activation and does not lengthen the timeout.

Shortening the amount of time a user has to wait is the whole reason why Windows will pre-launch frequently used apps without making them visible. Nevertheless, it’s still important to get going quickly.

The first consideration, of course, is to optimize your startup process to be as fast as possible. (For example, you can use the `defer="defer"` attribute on `script` attributes to defer-load JavaScript files that aren’t needed during initialization.) Still, sometimes an app really needs more than 15 seconds to get going, especially the first time it’s run after being installed. For example, an app might need to expand a bunch of compressed data from its app package into local appdata such that subsequent launches are much faster. Many games do this with graphics and other resources, optimizing the local storage for device characteristics; other apps might populate a local IndexedDB from data in a JSON

file or download and cache a bunch of data from an online service. (In the latter scenario you can pre-cache online content, as explained in Chapter 4.)

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, you want to show the user that something is actually happening so that she doesn't think to switch away 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* is what allows you to fully customize the splash screen experience. In truth, an extended splash screen is not a system object or such—it's just an implementation strategy for the first page of your app in which you'll do the majority of your startup work before displaying your real home page. In fact, a typical approach is to just overlay a full-sized `div` on top of your home page for this purpose and then remove that `div` from the DOM (or animate it out of view) when initialization is complete.

The trick (as recommended 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).

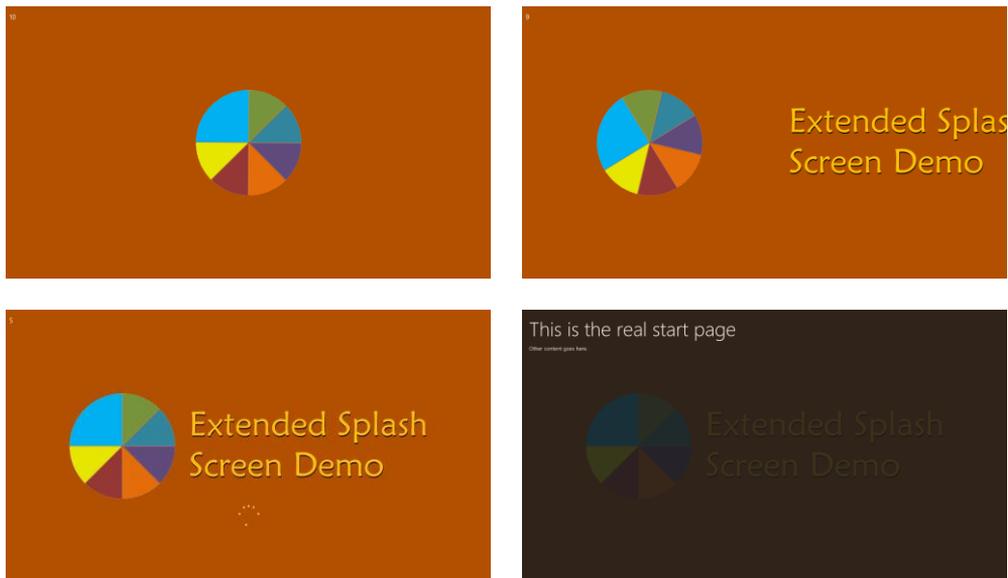
I recommend installing and running various apps from the Store to see different effects in action. A few popular titles include Skype, Netflix, Jetpack Joyride, and some of the Bing apps that are included with Windows (like News and Travel). Netflix, for example, gracefully slides the default splash screen logo up to make space for a shadow and a progress ring. Now compare the experience of a good extended splash screen to the static default experience that other apps provide. Which do you prefer? And which do you think users of your app will prefer?

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 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 state at any time. So, as with every other page in your app, make sure your extended splash screen can handle different sizes and orientations. We'll talk about this more in Chapter 7, "Layout," when we discuss sizing strategies.

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 is 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 `ExtendedSplashScreen` example in this chapter's companion content. The four stages of this splash screen (for a full landscape view) are shown in Figure 3-2. Note that this example does *not* accommodate different views at present; we'll fix that in Chapter 7.



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

The first stage, in the upper left of Figure 3-2, 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>
  
  
  <progress id="progress" class="win-ring win-large"></progress>
</div>
```

In the second `div`, which overlays the first because it's declared last, the `counter` element shows a countdown for debug purposes, but you can imagine such a counter turning into a determinate progress bar or a similar control. The rest of the elements provide the images and a progress ring. But we can't position any of these elements in CSS until we know more about the size of the screen. The best we can do is set the `splashScreen` element to fill the screen with the background color and set the `position` style of the other elements to `absolute` so that we can set their exact location from code. This is done in `default.css`:

```
#splashScreen {
  background: #B25000; /* Matches the splash screen color in the manifest */
  width: 100%;        /* Cover the whole display area */
  height: 100%;
}

#splashScreen #counter {
  margin: 10px;
  font-size: 20px;
}

#splashScreen #logo {
  position: absolute;
}

#splashScreen #title {
  position: absolute;
}

#splashScreen #progress {
  position: absolute;
  color: #fc2; /* Use a gold ring instead of default purple */
}
```

In `default.js` now, we declare some 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 3-2, only it's our extended splash screen page and not the default one. Thus we can return from the `activated` handler at this point 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 3-2.

To fade from our extended splash screen to the main page (a partial fade is shown on the lower right of Figure 3-2), 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
  });
}

```

We'll explore the animations library in Chapter 12, "Purposeful Animations," so for now, you can refer to the documentation if you want to know more.

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 3-2 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!

## 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* in the code above.

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 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 *always* listens 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 the previous section. (In that case, however, you might notice that I still cleaned up the listener explicitly, because there's no reason to keep any closures in memory once the extended splash screen completes.)

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 what are called *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 for 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 necessarily 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 7.<sup>3</sup>

When your app is *closed* (the user swipes top to bottom or 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). 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.](#))
- **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 10), 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 18). There are also times when you might want to refresh your layout

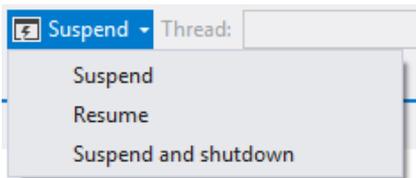
---

<sup>3</sup> 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.

(as we'll see in Chapter 7), because it's possible for your app to resume 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 8), refreshing any tile updates (see Chapter 14), and checking any saved state that might have been modified by background tasks or roaming (Chapter 9).

- **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 3-3. 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!)

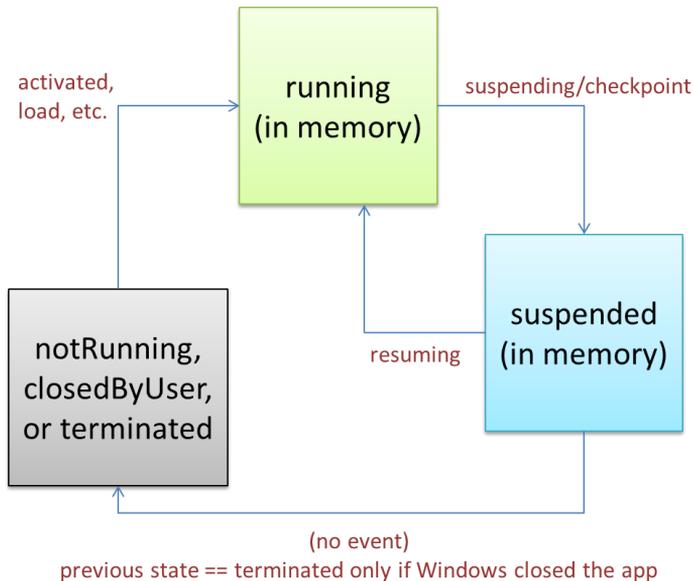


**FIGURE 3-3** The Visual Studio toolbar drop-down to simulate suspend, resume, and terminate.

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 3-4 and the table below describes how the `previousExecutionState` values are determined.

Value of <code>previousExecutionState</code>	Scenarios
<code>notrunning</code>	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 relaunched quickly, Windows has to immediately terminate it without finishing the suspend operation). App was terminated in Task Manager while running or closed itself with <code>MApp.terminateApp</code> .
<code>running</code>	App is <i>currently running</i> 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 <code>focus</code> and <code>visibilitychange</code> will both be raised).
<code>suspended</code>	App is <i>suspended</i> and then invoked in a way other than the app tile (as above for <code>running</code> ). In addition to focus/visibility events, the app will also receive

	the <code>resuming</code> event.
<code>terminated</code>	App was previously suspended and then terminated by Windows due to resource pressure. Note that this does not apply to <code>MSApp.terminateApp</code> because an app would have to be running to call that function.
<code>closedByUser</code>	App was closed by an uninterrupted close gesture (swipe down 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 <code>notrunning</code> instead.



**FIGURE 3-4** 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 9. 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 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 9. 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.

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.

Third, 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](#).

A final aid ties into a deferral mechanism like the one for activation. Windows will normally suspend your app as soon as you return from the `suspending` event (regardless of whether five seconds have elapsed). If you start async operations within your handler, then, you need to defer suspension until those operations are complete.

On the WinRT level, the event args for `suspending` contains an instance of `Windows.UI.WebUI.WebUIApplication.SuspendingOperation`. This provides a `getDeferral` method that returns a deferral object with a `complete` method. As with activation, you call `complete` when your async operations are finished.

WinJS again provides a deferral mechanism oriented around promises. The event args for `WinJS.Application.oncheckpoint` provides a `setPromise` method that ties into the underlying WinRT deferral. You pass a promise for an async operation (or combined operations) to `setPromise`, which in turn calls the deferral's `complete` method once the promise is fulfilled.

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.

A basic demonstration of using the suspending deferral, by the way, can be found in the [App activated, resume, and suspend sample](#). This also provides an example of activation through a custom URI scheme, a subject that we'll be covering later in Chapter 13, "Contracts." 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) {
    var gl = new Windows.Devices.Geolocation.Geolocator();

    gl.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 pathname: 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;
img.src = app.sessionState.imageURI;
```

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, so 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");
        img.src = app.sessionState.imageURI;
        scaleImageToFit(img, document.getElementById("photo"), file);
    });
}

updatePosition();
```

I've placed the code to set `img.src` 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 obtained a `StorageFile` from the file picker, we'd need to save that in the `Windows.Storage.AccessCache` to preserve access permissions across sessions. We'll revisit the access cache in Chapter 9.

## 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. In many web applications, page-to-page navigation uses `<a href>` hyperlinks or 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.

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.<sup>4</sup> 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[A11]`, 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. For some examples, 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 `state`, `location`, `history`, `canGoBack`, and `canGoForward` properties. The stack 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`).<sup>5</sup>

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

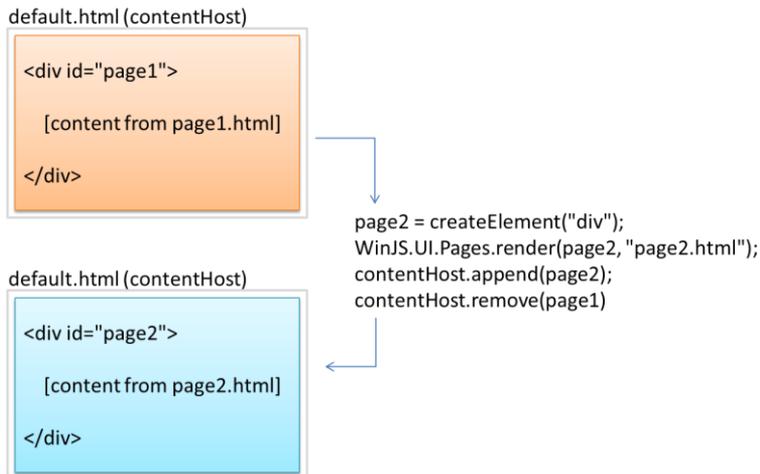
---

<sup>4</sup> If you are at all familiar with user controls in XAML, this is the same idea.

<sup>5</sup> 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`.

The basic process is as follows, and it's also shown in Figure 3-5:

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 3-5** 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.<sup>6</sup> In any

---

<sup>6</sup> 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.

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.

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
<code>init</code>	Called before elements from the page control have been created.
<code>processed</code>	Called after <code>WinJS.UI.processAll</code> 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, WinJS animates the new page into view with <code>WinJS.UI.Animation.enterPage</code> , so all initialization of properties and data-binding should occur within this method.
<code>ready</code>	Called after the page have been added to the DOM.
<code>error</code>	Called if an error occurs in loading or rendering the page.
<code>unload</code>	Called when navigation has left the page.
<code>updateLayout</code>	Called in response to the <code>window.onresize</code> 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.<sup>7</sup>

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;
        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 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();

```

---

<sup>7</sup> If the use of `.bind(this)` is unfamiliar to you, please see my blog post, [The purpose of this<event>.bind\(this\)](#).

```

    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.innerHTML = "";
        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 3-5. 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 *dispose* 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 remove any event handlers it added.

**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`.

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 the series of posts on my blog starting with [Exploring WinJS.Class Patterns, Part 1: Defining Classes and Object Construction](#).

## 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) 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 7.

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 6, 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 8, “Commanding UI”). When using page controls and `PageControlNavigator`, navigation commands or buttons can just invoke `WinJS.Navigation.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.

Showing progress indicators can help alleviate the user's anxiety, and the recommendation is to show such indicators after two seconds and provide a means to cancel the operation after ten seconds. Even so, users are notoriously impatient and will likely want to quickly switch between a list of items and item details. In this case, page controls might not be the best design. Forcing your customers to stare at a spinning progress control time and time again will probably not keep them as customers!

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 did with the 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/page3` (`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`.

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. Because there isn't a means to specifically unload or reload CSS files, it boils down to avoiding collision between selectors. You can do this with unique selectors for each page, or you can scope your styles to each page specifically. For the latter, wrap each page's contents in top-level `div` with a unique class, as in `<div class="page1">`. This allows you to scope every style 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.

## Be True to Your Promises: Creating, Joining, and Error Handling

---

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, as mentioned with extended splash screens earlier, 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.

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," a draft version of which is included in this First Preview PDF. Here I want to focus on the most essential aspects of promises 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.

**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](#).

## 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," clearly I don't have those donuts right now, but I assume that I'll have them some time in the future, and when I do, I'll deliver them.

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 within the promise 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.<sup>8</sup>

---

<sup>8</sup> If you want to impress your friends while reading the WinRT API documentation, know that if an async function shows it

Finally, two static methods on the [WinJS.Promise](#) object might come in handy when using promises:

- [is](#) determines whether an arbitrary value is a promise, returning a Boolean. It basically makes sure it's an object with a function named "then"; it does not test for "done".
- [theneach](#) takes an array of promises and subscribes completed, error, and progress handlers to each promise by calling its [then](#) method. Any of the handlers can be `null`. The return value of [theneach](#) is itself a promise that's fulfilled when all the promises in the array are fulfilled. We call this a *join*, as described in the next section.

**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	<a href="#">any</a>	<a href="#">join</a>
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 <a href="#">key</a> property identifies the promise that was fulfilled and whose <a href="#">value</a> property is an object containing that promise's state. Within that state is a <a href="#">_value</a> 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 <code>null</code> or <code>undefined</code> ). 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 <a href="#">any</a> cancels all promises in the array, even if the first has already been fulfilled.	Cancels all other promises that are still pending.

---

returns a value of type `IAsync[Action | Operation]WithProgress`, it will invoke progress handlers. If it lists only `IAsync[Action | Operation]`, progress is not supported.

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.<sup>9</sup>

**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.

## Debugging and Profiling

---

As we've been exploring the core anatomy of an app in this chapter, now's a good time to talk about debugging and profiling, or, as I like to put it, becoming a doctor of internal medicine for your app. After all, we've been learning about app anatomy, so it's appropriate to explore how we diagnose how well that anatomy is working.

---

<sup>9</sup> 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.

## Debug Output, Error Reports, and the Event Viewer

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. 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 in a moment.

For readers who are seriously into logging, beyond the kind you do with chainsaws, WinJS provides a more flexible method called `WinJS.log`. This 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`.

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 do yourself with the above syntax. 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.
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".

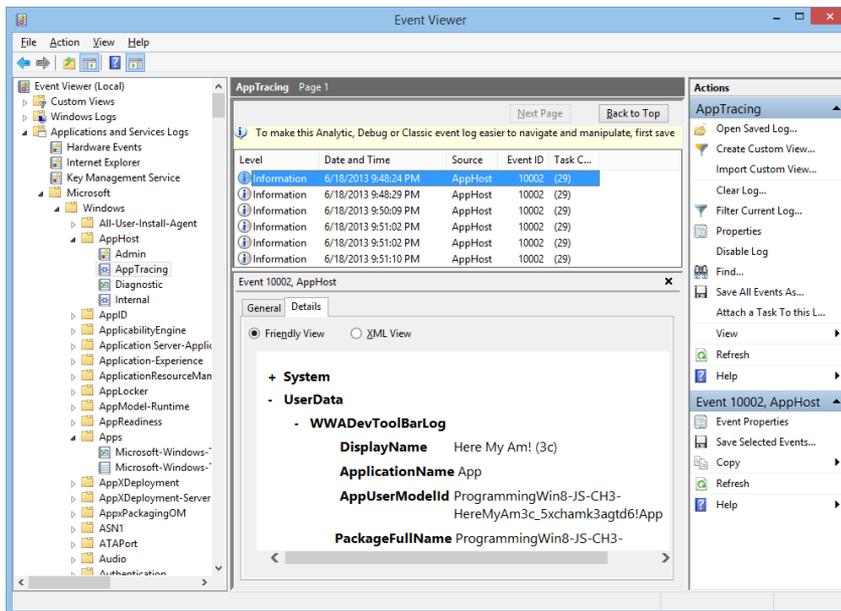
Conveniently, WinJS offers a basic implementation of this which you set up by calling `WinJS.Utilities.startLog()`. This assigns a function to `WinJS.log` that uses `WinJS.Utilities.formatLog` to produce decent-looking output to the console. What's very useful is that you can pass a list of tags (in 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** Before creating an app package to upload to the store, be sure to comment out your call to `startLog`. That way your app won't be making any unnecessary calls in its released version.

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 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.<sup>10</sup> This is where error reports, console logging, and unhandled exceptions (which again terminate the app without warning) can be recorded. To enable this, navigate to Application And Services Logs (after waiting for a minute while the tool initializes itself) and 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 3-6. Then right-click AppTracing (also under AppHost) and select Enable Log. This will trace any calls to `console.log` as well as other diagnostic information coming from the app host.



**FIGURE 3-6** App host events, such as unhandled exceptions, load errors, and logging can be found in Event Viewer.

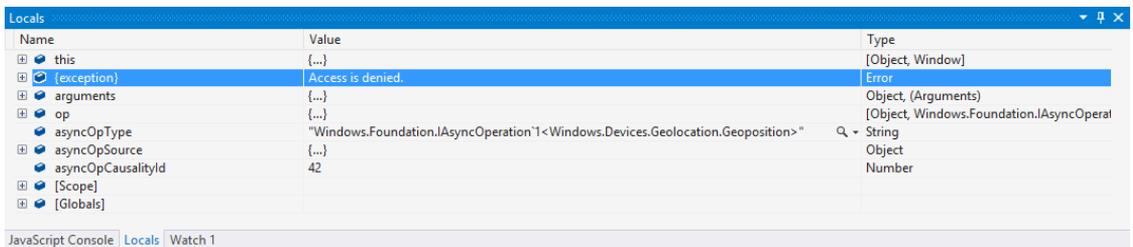
<sup>10</sup> If you can't find Event Viewer, press the Windows key to go to the Start screen and then invoke the Settings charm. Select Tiles, and turn on Show Administrative Tools. You'll then see a tile for Event Viewer on your Start screen.

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 3-7) whenever an exception is thrown, regardless of whether it's handled and before reaching any of your error handlers.



**FIGURE 3-7** 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 3-7 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 3-8.



**FIGURE 3-8** 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.

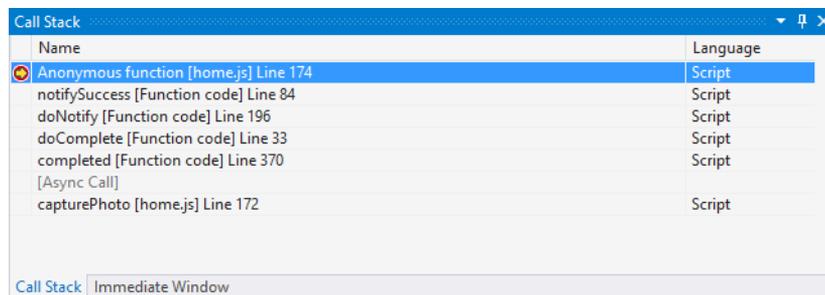
Do leave User-unhandled checked for everything else. In fact, unless you have a specific reason not to, make sure that User-unhandled is checked next to JavaScript Runtime Exceptions here because this will include those exceptions not otherwise listed. This way you can catch (and fix) any exceptions that might abruptly terminate the app, which is something your customers should never experience.

## 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 within each completed handler and on the first line of each error function. As each breakpoint is hit, you can step through that completed handler. When you reach the next async call, 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 (in the lower right of Visual Studio), as shown here:

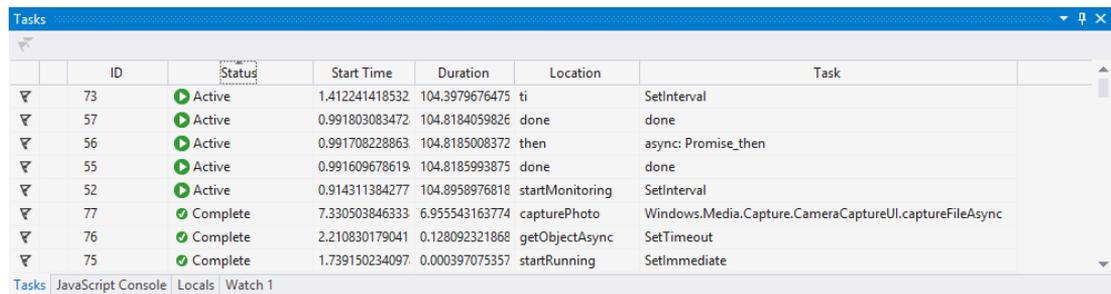


Generally speaking, the Call Stack shows you the sequence of functions that lead up to the point where the debugger stopped, at which point you can double-click any of the lines and examine that function context. With async calls, however, what you primarily see is the chain of generic handlers within WinJS or the JavaScript projection layer, none of which give you much useful information about the original calling context. Fortunately, that latter context is shown separately below the [Async Call] line near the bottom. In this example I set a breakpoint in the first completed handler after starting

camera capture in `HereMyAm3c`. That completed handler is an anonymous function, as the first line of the Call Stack indicates, but you can see at the bottom that the real context is the `capturePhoto` function within `home.js`.

The real utility of this comes when an exception occurs somewhere other than within your own handlers, because the lines after [Async Call] tell you the exact code that started the operation.

The other feature for async debugging is the Tasks pane, as shown below. You turn this on through the `Debug > Windows > Tasks` menu command. You'll see a full list of active and completed async operations that are part of the current call stack.



	ID	Status	Start Time	Duration	Location	Task
▼	73	Active	1.412241418532	104.3979676475	ti	SetInterval
▼	57	Active	0.991803083472	104.8184059826	done	done
▼	56	Active	0.991708228863	104.8185008372	then	async: Promise_then
▼	55	Active	0.991609678619	104.8185993875	done	done
▼	52	Active	0.914311384277	104.8958976818	startMonitoring	SetInterval
▼	77	Complete	7.330503846333	6.955543163774	capturePhoto	Windows.Media.Capture.CameraCaptureUI.captureFileAsync
▼	76	Complete	2.210830179041	0.128092321868	getObjectAsync	SetTimeout
▼	75	Complete	1.739150234097	0.000397075357	startRunning	SetImmediate

## Performance and Memory Analysis

Alongside its excellent debugging tools, Visual Studio also offers additional 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.

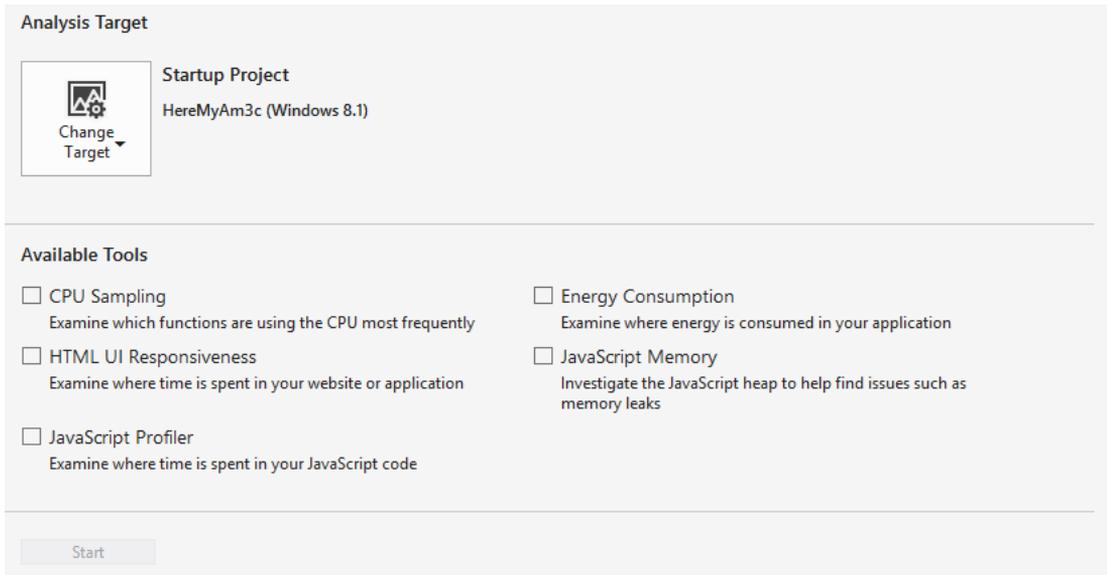
When running analysis tools, it's important that you *exercise the app like a user would*. That way you get results that are meaningful to the real user experience—that is, the human experience!—rather than results that would be meaningful to a robot. In fact, as you think about performance, approach it as a means of optimizing your app's dynamic user experience. In the end, all the performance analysis in the world won't be worth anything unless it translates into two things: better ratings and reviews in the Windows Store, and greater app revenue. Otherwise all your work is a classic case of what Tom DeMarco, in his book *Why Does Software Cost So Much?* (Dorset House, 1995) calls "measurement dysfunction," which means focusing on details that ultimately deliver the wrong results.<sup>11</sup>

---

<sup>11</sup> 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 the other."

Remember also to *run performance analysis on a variety of hardware*, especially lower-end devices such as ARM tablets where performance improvements are going to matter much more than they will on 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!

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



What's shown here are the tools built into Visual Studio; the hub is extensible with third-party tools that will appear here as well. For the built-in tools, the table below explains what each one does and how to think about data they collect.

Tool	Description
CPU Sampling	<p>Starts a data collector and launches the app. Once you exercise the app through whatever scenarios you'd like to analyze, return to Visual Studio and click on the Stop Profiling link. After a few seconds Visual Studio will then display a report that shows when and where function calls are made, and how much time is being spent in which function. This is a much better view into what's happening with the app over time than a static report or watching the CPU % in Task Manager. It's helpful for finding bottlenecks which could certainly impact user experience, though if you're spending most of the app's time within functions that are doing the work the user really wants, then it's necessarily not a bad thing!</p> <p>See <a href="#">How to profile JavaScript code in Windows Store apps on a local machine</a> and <a href="#">How to profile JavaScript code in Windows Store apps on a remote device</a>.</p>

HTML UI Responsiveness	Launches the app and 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.
JavaScript Profiler	Overlaps somewhat with HTML UI Responsiveness, producing very similar results but with more detail. The HTML UI Responsiveness tool, for example, tells you when script is executing; the Profiler tool includes what was executing when and how long it too, as well as an aggregated report of calls over time. A walkthrough of working with this data can be found on <a href="#">How to profile a JavaScript App for performance problems</a> (MSDN blogs).
Energy Consumption	Launches the app and collects data about power usage (in milliwatts) over time, split up by CPU, display, and network.
<a href="#">JavaScript Memory</a>	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.

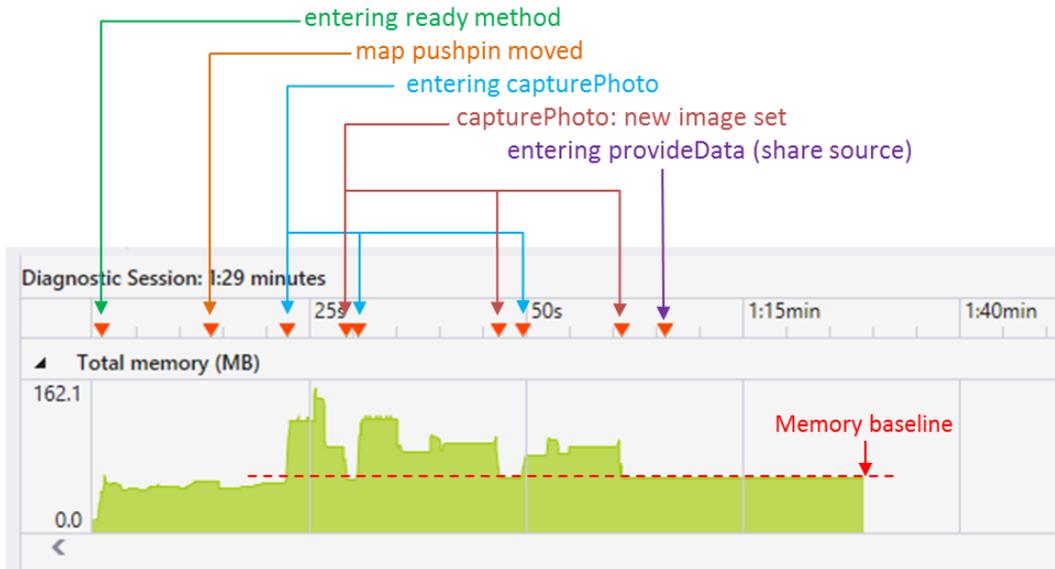
I very much encourage you to spend a few hours exercising these tools and getting familiar with the information they provide. Then make them a regular part of your coding/testing cycle, because the earlier you can catch performance and memory issues, the easier it will be to fix them. For more detailed information in these areas, see [Performance best practices for Windows Store apps using JavaScript](#) and [General best practices for performance](#).

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.<sup>12</sup> Events written with this function appear as User Marks in the timelines generated by the different tools, as shown in Figure 3-9. 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.

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 3-9 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`).

---

<sup>12</sup> 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](#).



**FIGURE 3-9** 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.

Referring to Figure 3-9, here’s what I did after starting up the app in the memory analyzer. Once the home page was up (first mark), I repositioned the map and its pushpin (second mark), and you can see that this increased memory usage a little within the Bing maps control. Next I invoked the camera capture UI (third mark), which clearly increased memory use as expected. After taking a picture and displaying it in the app (fourth mark), you can see that the allocations from the camera capture UI have been released, and that we land at a baseline footprint that now includes a rendered image. I then do into the capture UI two more times, and in each case you can see the memory increase during the capture, but it comes back to our baseline each time we return to the main app. There might be some small differences in memory usage here depending on the size of the image, but clearly we’re cleaning up the image that gets replaced each time. 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 going up over time, then I’d know I had a leak somewhere.

## 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 working with in 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 extended splash screens when an app needs more time to load, and deferrals when the app needs to use async operations on startup.
- APIs in the `MSApp` object for prioritizing work on the UI thread.
- The 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.
- Methods for getting debug output and error reports for an app, within the debugger and the Windows Event Viewer.
- How to debug asynchronous code.
- How to perform basic performance and memory analysis with Visual Studio tools.

## Chapter 4

# Using 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.<sup>1</sup>

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 like controls and other UI elements!

---

<sup>1</sup> 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.

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 16, "Devices and Printing." Similarly, setting up service connections for live tiles and push notifications are covered in Chapter 14, "Alive with Activity." 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 topics in Appendix B, "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 14 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 14) 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. Windows 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 always 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.

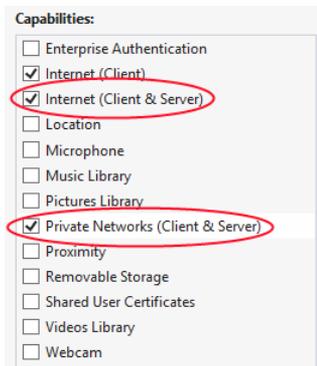
**Note** Network connectivity, by its nature, is an intricate subject, as you'll see in the sections that follow. But don't feel compelled to think about all these up front! If you want to take connectivity entirely for granted for a while and get right into playing with web content and making HTTP requests, 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 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. 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.



**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.

## 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](#):

- `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.
- `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](#).

- [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.
- [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 [proxyUri](#)s (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. A vector is also derived from an interface called [IIterable](#) whose single method [first](#) returns an [iterator](#) object that can also be used to traverse the collection. Overall, vectors and iterators are part of a group of classes in [Windows.Foundation.Collections](#) that also includes *key-value pairs*, *maps*, *observable maps*, and *property sets*. We'll encounter more of these throughout this book.

## 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, 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 properties describing authentication and encryption types.

More generally interesting is the `getNetworkConnectivityLevel`, 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.

To track the inbound and outbound traffic on a connection, the `getLocalUsage` method returns a [DataUsage](#) object that contains `bytesReceived` and `bytesSent`, either for the lifetime of the connection or for a specific time period. 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 [NetworkInformation.onnetworkstatuschanged](#) event, 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). For example, the event will be fired if the connectivity level of a profile changes. 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.navigator.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 14; 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 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.

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:

Name	CPU time	Network	Metered network	Tile updates
Mail, Calendar, People, a...	0:01:11	4.4 MB	0 MB	0.1 MB
Maps	0:00:03	0.6 MB	0 MB	0 MB
Mashup JS sample	0:00:09	1.1 MB	0 MB	0 MB
MessageDialog JS sample	0:00:06	0 MB	0 MB	0 MB
Music	0:00:00	0 MB	0 MB	0 MB
News	0:00:01	0.5 MB	0 MB	0.5 MB
Package JS sample	0:00:01	0 MB	0 MB	0 MB
Page controls JS sample	0:00:01	0 MB	0 MB	0 MB
Photos	0:00:31	8.1 MB	0 MB	0.1 MB

Programmatically, as noted before, the profile supplies usage information through its `getConnectionCost` and `getDataPlanStatus` methods. The first method returns a `ConnectionCost` object with four properties:

- `networkCostType` A `NetworkCostType` value, one of `unknown`, `unrestricted` (no extra charges), `fixed` (unrestricted up to a limit), and `variable` (charged on a per-byte or per-megabyte 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` unless the user consents to more data usage.
- `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`.

The second method, `getDataPlanStatus`, returns a `DataPlanStatus` object with these properties:

- `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 and/or warn the user about possible overage charges. 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`.

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 `getLocalUsage` method comes in handy to obtain a `DataUsage` object for a given period (see [How to retrieve connection usage information for a specific time period](#)). Call `getLocalUsage` with the time period between `lastSyncTime` and `DateTime.now()`. Then add that value to `megabytesUsed` and subtract the result from `dataPlanLimitInMegabytes`. This tells you how much more data you can transfer before incurring extra costs, thereby providing the basis for asking the user, "Downloading this file will exceed your data plan limit. Do you want to proceed?"

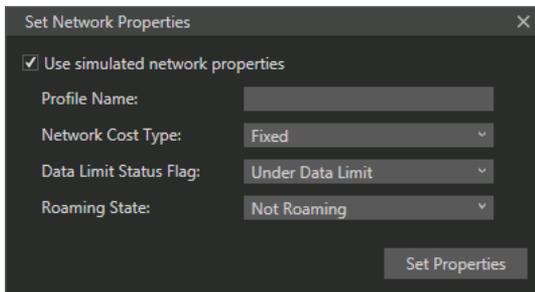
For simplicity's sake, you can think of cost awareness in terms of three behaviors: normal, conservative, and opt-in, which are described on [Managing connections on metered networks](#) and, more broadly, on [Developing connected apps](#). Both topics provide additional guidance on making the kinds 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.

**Tip** A very 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](#).

## Sidebar: Simulating Metered Networks

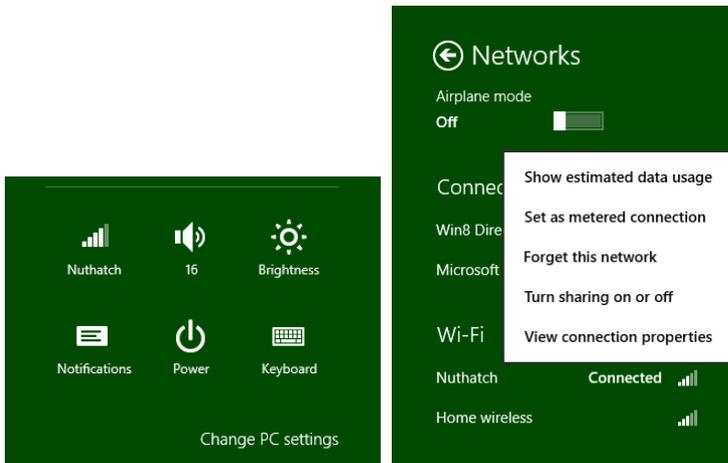
You may be thinking, “OK, so I get the need for my app to behave properly with metered networks, but how do I test such conditions without signing up with some provider and paying them a bunch of money (including roaming fees) while I’m doing my testing?” The simple answer is that you can simulate the behavior of metered networks with either the Visual Studio simulator or 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:

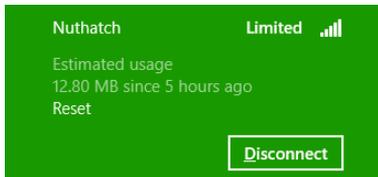


In this dialog you can create a profile with whatever name and options you'd like. The variations for cost type, data limit status, and roaming allow you to test all conditions that your app might encounter. As such, this is your first choice for working with cost awareness.

To simulate a metered network with a Wi-Fi connection, invoke the Settings charm and tap on your network connection near the bottom (see below left, specifically the upper left icon, shown here as “Nuthatch”). In the Networks pane that then opens up (below right), right-click a wireless connection to open the menu and then select Set As Metered Connection:



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 Estimated Data Usage menu item 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, you can 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 tied 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. 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.

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? 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 `iframe` or `webview` elements), are also cached automatically like typical temporary Internet files. Note that all caching mechanisms are subject to the storage limits defined by Internet Explorer (whose subsystems are shared with apps written with HTML and JavaScript). 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`. Avoid using the `RoamingFolder` to cache data acquired from online sources: besides running the risk of exceeding the roaming quota (see Chapter 9, “The Story of State, Part 1”), 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 Chapter 9.

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 websites typically aren't set up well for touch interaction (which violates requirement 3.5) and often won't work well in different view states (violating requirement 3.6). 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 UI > 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
<code>&lt;iframe src="ms-appx:///"&gt;</code>	<code>iframe</code> in local context	Not allowed
<code>&lt;iframe src="ms-appx-web:///"&gt;</code>	<code>iframe</code> in web context	<code>iframe</code> in web context
<code>&lt;iframe src="https:// "&gt;</code>	<code>iframe</code> in web context	<code>iframe</code> in web context
<code>&lt;a href="[uri]" target="myFrame"&gt;</code> <code>&lt;iframe name="myFrame"&gt;</code>	<code>iframe</code> in local or web context depending on [uri]	<code>iframe</code> in web context; [uri] cannot begin with <code>ms-appx</code> .
<code>&lt;a href="ms-appx:///"&gt;</code>	Links to page in local context	Not allowed unless explicitly specified (see below)
<code>&lt;a href="ms-appx-web:///"&gt;</code>	Not allowed	Links to page in web context
<code>&lt;a href="[uri]"&gt;</code> with any other protocol including <code>http[s]</code>	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). 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 Preview for use with a Windows 8.1 Preview 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.<sup>2</sup>

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 below. Now 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.)

---

<sup>2</sup> 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.<sup>3</sup> 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.

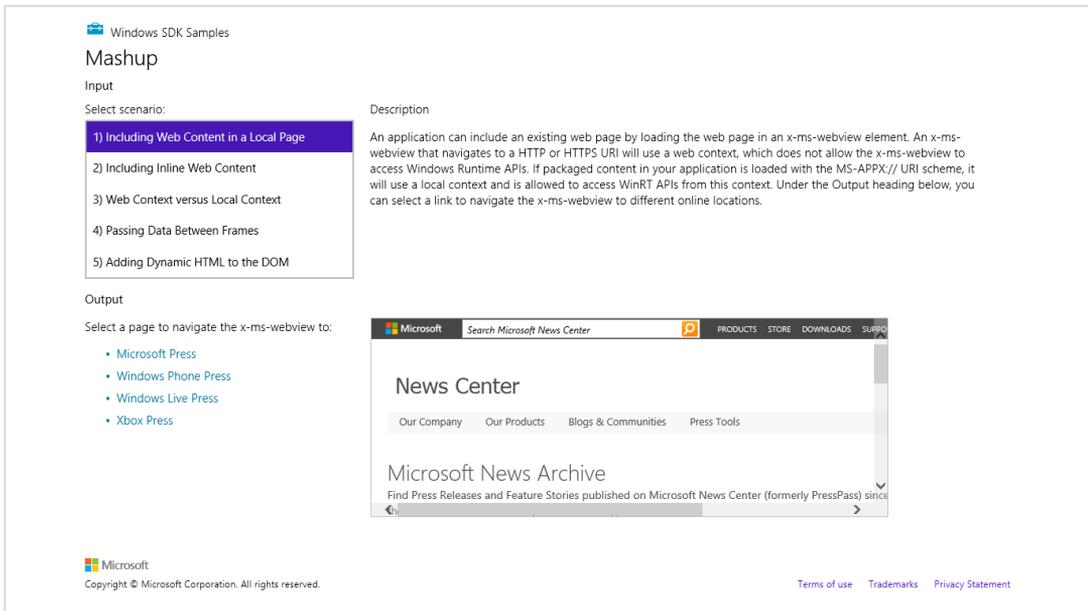
The simplest use case for the webview (and I call it this because it's tiresome to type out the funky element name every time) is to just point it to a URI through its `src` attribute. One example is in scenario 1 of the [Integrating content and controls from web services sample](#) (`html/webContent.html`), with the results shown in Figure 4-2:

```
<x-ms-webview id="webContentHolder"
src="http://www.microsoft.com/presspass/press/NewsArchive.aspx?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.

---

<sup>3</sup> 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 three 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`.
- `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 all three 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 webviewController = document.getElementById("webview");
webviewController.navigate("http://go.microsoft.com/fwlink/?LinkId=294155");
```

Scenario 2 shows `navigateToString` by loading an in-package HTML file into a string variable, which is equivalent to calling `navigate` with the same `ms-appx-web` URI. Of course, if you have the content in an HTML file already, you would just use `navigate`! It's more common, then, to use `navigateToString` with content that's being generated dynamically. 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`):

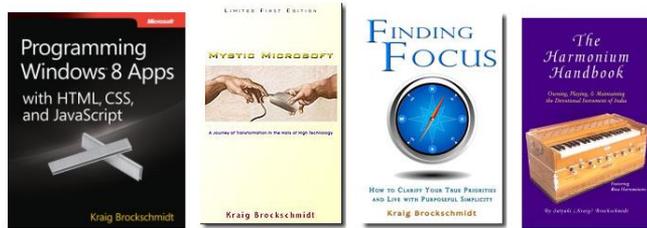
```
var local = Windows.Storage.ApplicationData.current.localFolder;

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:

## Dynamically-created page

This document contains its own styles as well as a remote image references.



Take careful note of the fact that I create this dynamic page in a subfolder within local appdata. The webview specifically disallows navigation to pages in a root local, roaming, or temp appdata folder to protect the security of other appdata files and folders. That is, because the webview runs in the web context and can contain any untrusted content you might have downloaded from the web, and because the webview allows that content to `exec` script and so forth, you don't want to risk exposing potentially sensitive information elsewhere within your appdata. By forcing you to place appdata content in a subfolder, you would have to consciously store other appdata in that same folder to allow the webview to access it. It's a small barrier, in other words, to give you pause to think clearly about exactly what you're doing!

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 17, "An Introduction to 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.sdk.samples.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 `MSWebViewUnviewableContentIdentified` event instead. It's also worth mentioning that the standard `change` event will also fire when navigation happens, but this also happens when setting other

properties, so it's not as useful for navigation purposes.

Scenario 1 of the HTML WebView control sample, which we saw earlier for [navigate](#), essentially gives you a simple web browser by wiring these methods and events to a couple of buttons. Note that any popups from websites you visit will open in the browser alongside the app.

**Tip** You'll find when working with the webview in JavaScript that the object does *not* provide equivalent [on\\*](#) properties for these events. This omission was a conscious choice to avoid potential naming conflicts with emerging standards. At present, then, you must use [addEventListener](#) to wire up these events.

## 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.

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** If you're loading a webview from an [http\[s\]](#) URI, you *must* add a content URI rule to your manifest to allow it to raise an event from script, otherwise that event will be blocked. This is not required for in-package or local content, or for a webview loaded with [navigateToString](#) or [navigateToLocalStreamUri](#).

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 a 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-ish object that has `complete` and `error` events. 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!

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 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();
});
```

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 it's `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();
    });

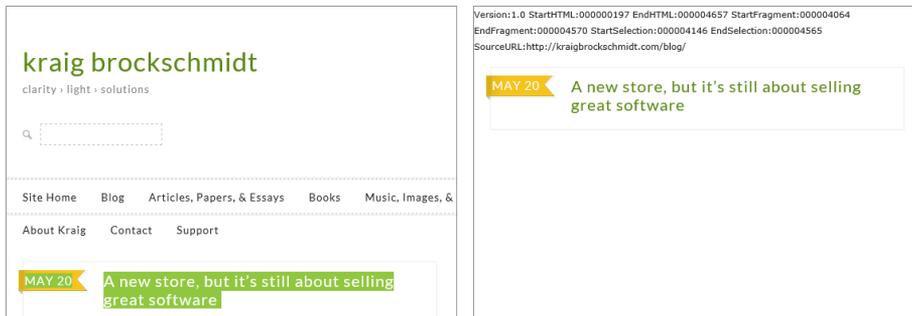
    //Navigate the output webview to the selection, or show an error
    var output = document.getElementById("webviewOutput");

    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.

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 very much 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.

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 B.

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 [readystatechange](#) 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 XHR 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, which 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.

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 8 developer blog, noting that the default HTTP verb is GET, so we don't have to specify it explicitly:

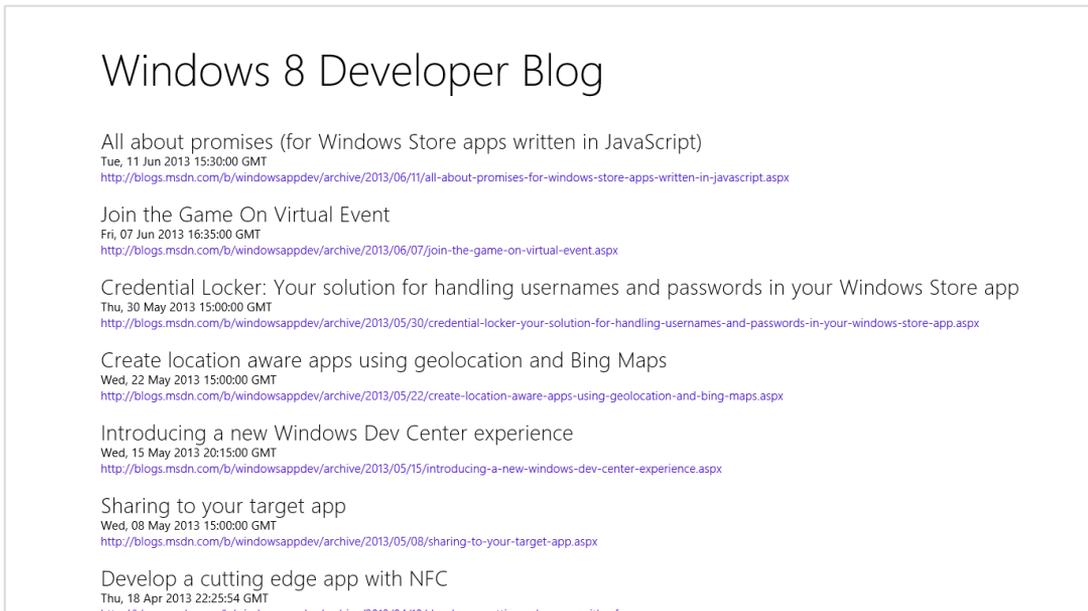
```
WinJS.xhr({ url: "http://blogs.msdn.com/b/windowsappdev/rss.aspx" })
    .done(processPosts, processError, showProgress);
```

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.<sup>4</sup>



**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](#) along with the tutorial called [How to create a](#)

<sup>4</sup> Again, WinRT has a specific API for dealing with RSS feeds in `Windows.Web.Syndication`, as described in Appendix B. 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.

[mashup](#) in the docs. Additional notes on [XMLHttpRequest](#) and [WinJS.xhr](#) can be found in Appendix B.

## Using Windows.Web.Http.HttpClient

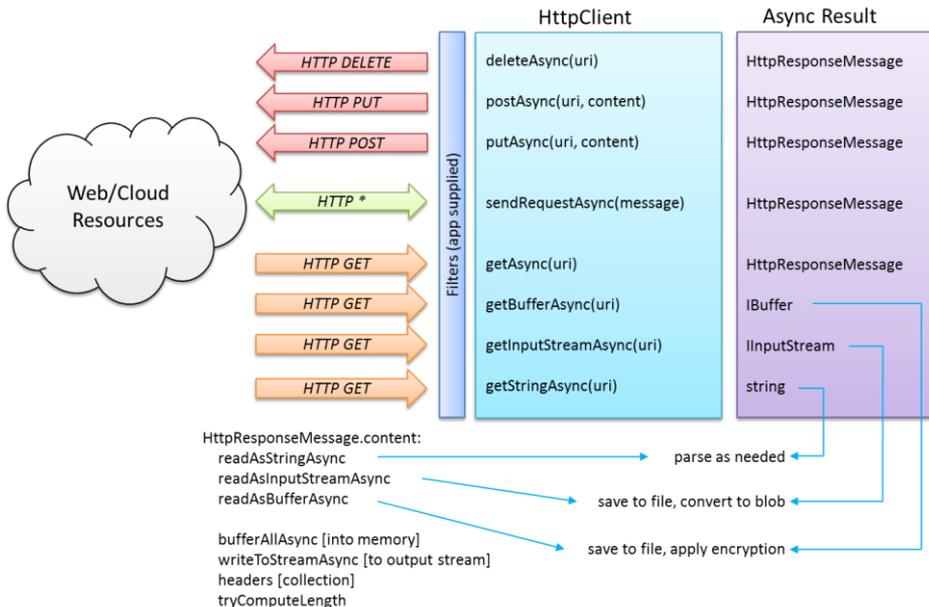
Let's now see the same app implemented with [Windows.Web.Http.HttpClient](#), which you'll find in SimpleXhr2 in the companion content. For our purposes, the [HttpClient.GetStringAsync](#) method is sufficient:

```
var htc = new Windows.Web.Http.HttpClient();
htc.GetStringAsync(new Windows.Foundation.Uri("http://blogs.msdn.com/b/windowsappdev/rss.aspx"))
    .done(processPosts, processError, showProgress);
```

This function delivers the response body text to our completed handler (`processPosts`), so we just need to create a [DOMParser](#) object to talk to the XML document. After that we have the same thing as we received from [WinJS.xhr](#):

```
var parser = new window.DOMParser();
var xml = parser.parseFromString(bodyText, "text/xml");
```

The [HttpClient](#) object provides a number of other methods to initiate various HTTP interactions with a web resource, as illustrated in Figure 4-5.



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

In all cases, the URI is represented by a [Windows.Foundation.Uri](#) object, as we saw in the earlier code snippet. All of the specific `get*` methods fire off an HTTP GET and deliver results in a particular

form: a string, a buffer, and an input stream. All of these methods (as well as [sendRequestAsync](#)) support progress, and the progress handler receives an instance of [Windows.Web.Http.HttpProgress](#) that contains various properties like [bytesReceived](#).

Working with strings are easy enough, but what are these buffer and input streams? These are specific WinRT constructs that can then be fed into other APIs such as file I/O (see [Windows.Storage.Streams](#) and [Windows.Storage.StorageFile](#)), encryption/decryption (see [Windows.Security.Cryptography](#)), and also the HTML blob APIs. For example, an [IInputStream](#) can be given to [MSApp.createStreamFromInputStream](#), which results in an HTML [MSStream](#) object. This can then be given to [URL.createObjectURL](#), the result of which can be assigned directly to an [img.src](#) attribute. This is how you can easily fire off an HTTP request for an image resource and show the results in your layout without having to create an intermediate file in your appdata.

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](#),<sup>5</sup> 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

---

<sup>5</sup> 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](#).

`isSuccessStatusCode` (to check for the range of 200–299).

- A collection of `headers`.
- The original `requestMessage` (an `HttpRequestMessage`).
- The `source`, a value from `HttpResponseMessageSource` that tells you whether the data was received over the network or loaded from the cache.
- The response `content`, an object with the `IHttpContent` interface as before. Through this you can obtain the response data as a string, buffer, input stream, and an in-memory array (`bufferAllAsync`).

It's clear, then, that the `HttpClient` object really gives you complete control over whatever kind of HTTP requests you need to make to a service, including additional capabilities like cache control and cookie management as described in the following two sections. It's also clear that `HttpClient` is still somewhat of a low-level API. For any given web service that you'll be working with, then, I very much recommend creating a layer or library that encapsulates requests to that API and the process of converting responses into the data that the rest of the app wants to work with. This way you can also isolate the rest of the app from the details of your backend, allowing that backend to change as necessary without breaking the app. It's also helpful if you want to incorporate additional features of the `Windows.Web.Http` API, such as filtering, cache control, and cookie management.

I'd love to talk about cookies first (it's always nice to eat dessert before the main meal!) but it's all part of *filtering*. Filtering is a mechanism through which you can control how the `HttpClient` manages its requests and responses. A filter is either an instance of the default `HttpBaseProtocolFilter` class (see the `Windows.Web.Http.Filters` namespace) or an instance of a derived class. You pass this filter object to the `HttpClient` constructor, which will use `HttpBaseProtocolFilter` as a default. To do things like cache control, though, you create an instance of `HttpBaseProtocolFilter` directly, set properties, and then create the `HttpClient` with it.

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`, `mostRecent`, 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.

For full demonstrations of the API, including filtering refer to the [HttpClient sample](#) in the Windows SDK. Here's a quick run-down of what its scenarios demonstrate:

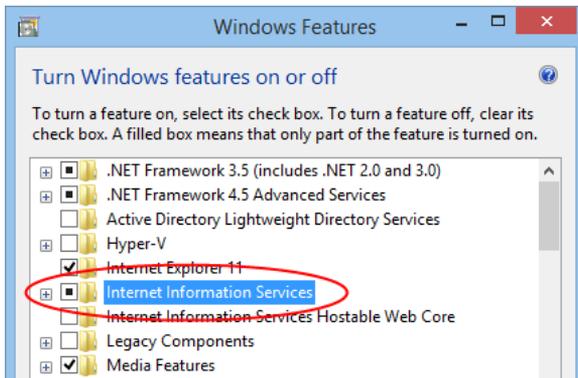
- **Scenarios 1–3** GET requests for text (with cache control), stream, and an XML list.
- **Scenarios 4–7** POST requests for text, stream, multipart MIME form, and a stream with progress.
- **Scenarios 8–10** Getting, setting, and deleting cookies.
- **Scenario 11** A metered connection filter that 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.<sup>6</sup>

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 14.

## 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. So really, I'll leave such details up to you because it's primarily a matter of retrieving and processing data that has little to do with the Windows platform (except for creating UI with collection controls, but that's for a later chapter).

Instead, what concerns us here are the implications of suspend and resume. In particular, an app cannot predict how long it will stay suspended before being resumed or before being terminated and restarted.

In the first case, an app that gets resumed will have all its previous data still in memory. It very much needs to decide, then, whether that data has become stale since the app was suspended and whether

---

<sup>6</sup> 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](#).

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 scenario, 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 8 developer blog, on the other hand, new posts don't show up more than once a 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 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 (such as the timestamp) helps you decide whether to use the cache or load data anew.

## Prefetching Content

HTTP requests made through the `XMLHttpRequest`, `WinJS.xhr`, and `HttpClient` APIs all interoperate with the internet cache, such that repeated requests for the same remote resource can be fulfilled from the cache. (Of these APIs, only `HttpClient` specifically gives you control over how the cache is used.) Caching works great, of course, for offline scenarios and improving performance generally. One of the first things that many connected apps do upon launch is to make HTTP requests for their home page content, which typically forces the user to stare at a progress indicator for some time. If that content is already in the cache, those apps will start up and navigate between their pages more quickly.

To improve this performance even further, apps can ask 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).
- 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 actually happening decreases.
- Windows limits the overall number of requests to 40.
- Resources are cached only for the length of time indicated in the response headers.

In other words, apps don't have control over *whether* their prefetching requests are fulfilled—Windows optimizes the process so that users see increased performance for the apps they use and the content they access most frequently. Apps, for their part, 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 URIs into the `Windows.Networking.BackgroundTransfer.ContentPrefetcher.contentUri` collection. This is a `vector` object, so you'd use methods like `append` to add URIs, each of which is an instance of `Windows.Foundation.Uri`. Note that you can modify this list both from the running app and from a background task; use the latter to periodically refresh the list without having the user run the app.

The second means is to give the prefetcher the URI of an XML file (local or remote) that contains

your list. This allows a service to maintain a dynamic list of URIs (like those of a news feed) such that your prefetching can stay very current. The XML in this case should be structured as follows, with as many URIs as are needed:

```
<?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.

## 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 background transfer API, [Windows.Networking.BackgroundTransfer](#) (which includes the prefetcher, as we just saw). 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 needing to worry about such concerns themselves. Transfers continue when an app is suspended and will be paused if the app is terminated. 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, and enumerating pending transfers, which will restart any that were paused or otherwise interrupted. (On the other hand, if the user directly closes the app through a gesture, Alt+F4, or Task Manager, all pending transfers for that app are canceled. This is also true if you stop debugging an app in Visual Studio.)

Generally speaking, then, it's highly recommended that you 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.

**Note** In both download and upload cases, the connection request will be aborted if a new connection is not established within five minutes. After that, any other HTTP request involved with the transfer times out after two minutes. Background transfer will retry an operation up to three times if there's 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<sup>7</sup> and you'll see that image appear on the display. (You can also copy an image file to `c:\inetpub\wwwroot` and point to it there.) Note that you can kick off multiple transfers to observe how they are all managed simultaneously; the cancel, pause, and resume buttons help with this.

Three flavors of download are supported in the WinRT API and reflected in the sample:

- A normal download at normal priority. Such a transfer continues to run when the app is suspended, but if it's a long transfer it could be slowed (throttled) or paused depending on system conditions like battery life and network type.
- 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.
- 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`. 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 for the operation. You attach your completed, error, and progress handlers with a call to the promise's `then` or `done`. Here's code from `js/downloadFile.js`:<sup>8</sup>

```
// 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);
```

<sup>7</sup> Might I suggest <http://kraigbrockschmidt.com/images/photos/kraigbrockschmidt-dot-com-122-10-S.jpg?>

<sup>8</sup> 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 imageUri = 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.

There is also 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.

## 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—that is, call the `cancel` method of the *promise* returned by `startAsync`. This means that you need to hold onto the promises for each transfer you initiate.

## 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 `isConstrained` 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 a discussion of breaking up large files and multipart uploads, see Appendix B.

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 B.

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:<sup>9</sup>

```
// 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 how 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

---

<sup>9</sup> As with downloads, the code in the sample has more structure than shown here and again defines its own `UploadOperation` class with the same name as the one in WinRT, so I'm omitting mention of it.

with messages like “Your new photo gallery of 108 images has finished uploading.”

The downloader and uploader objects each have four different notification objects you can set:

- `successToastNotification` and `failureToastNotification` Instances of the [Windows.UI.Notifications.ToastNotification](#) class.
- `successTileNotification` and `failureTileNotification` Instances of the [Windows.UI.Notification.TileNotification](#) class.

For details on using these classes, including all the different templates you can use, refer to Chapter 14. 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 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.<sup>10</sup>

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

---

<sup>10</sup> The optional `group` argument for the other methods is obsolete and replaced with these that work with a `transferGroup` argument.

operations in that group.

This brings up an interesting situation: if your app has been terminated and later restarted, how do you restart transfers that were paused? The answer is quite simple. By enumerating transfers through `getCurrentDownloads[ForTransferGroup]Async` and `getCurrentUploads[ForTransferGroup]Async`, incomplete transfers are automatically restarted. But then how do you retrieve the promises originally returned by the `startAsync` methods? Those are not values that you can save in your app state and reload on startup, and yet you need them to be able to cancel those operations, if necessary, and also to attach your completed, error, and progress handlers.

For this reason, both `DownloadOperation` and `UploadOperation` objects provide a method called `attachAsync`, which returns a promise for the operation just like `startAsync` did originally. You can then call the promise's `then` or `done` methods to provide your handlers:

```
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 to 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 B.

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, 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 else 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 (which can be turned off in PC Settings > Sync Your Settings > Passwords). This help to create a seamless experience with your app as the user moves between devices.<sup>11</sup>

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.retrievePassword` method.

For further demonstrations of the credential locker—the code is very straightforward—refer to the

---

<sup>11</sup> 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.

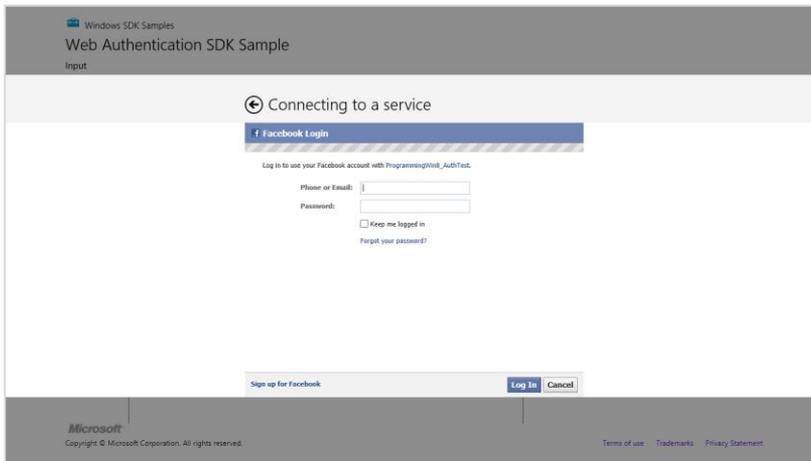
[Credential locker sample](#). This shows variations for single user/single resource (Scenario 1), single user/multiple resources (Scenario 2), multiple users/multiple resources (Scenario 3), and clearing out the locker entirely (Scenario 4).

## 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. 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.

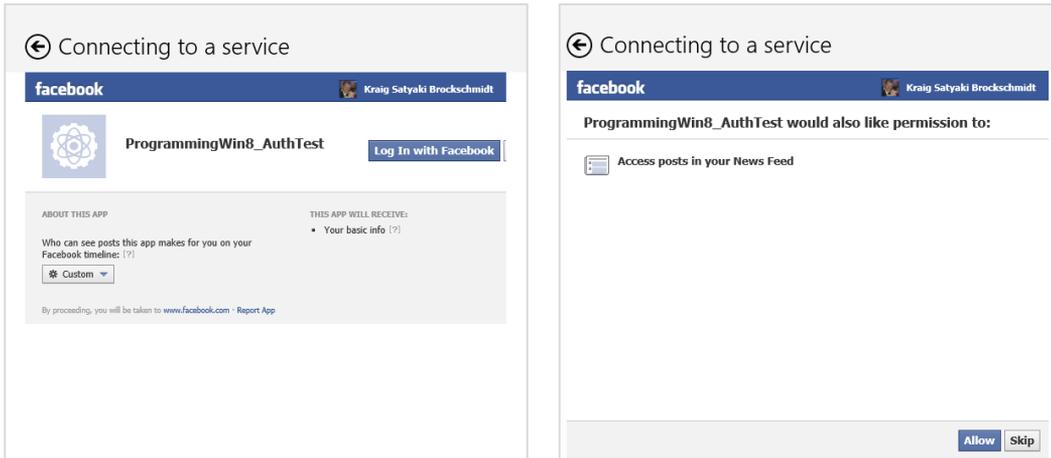


**FIGURE 4-6** The Web authentication broker sample using a Facebook login page.

**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.

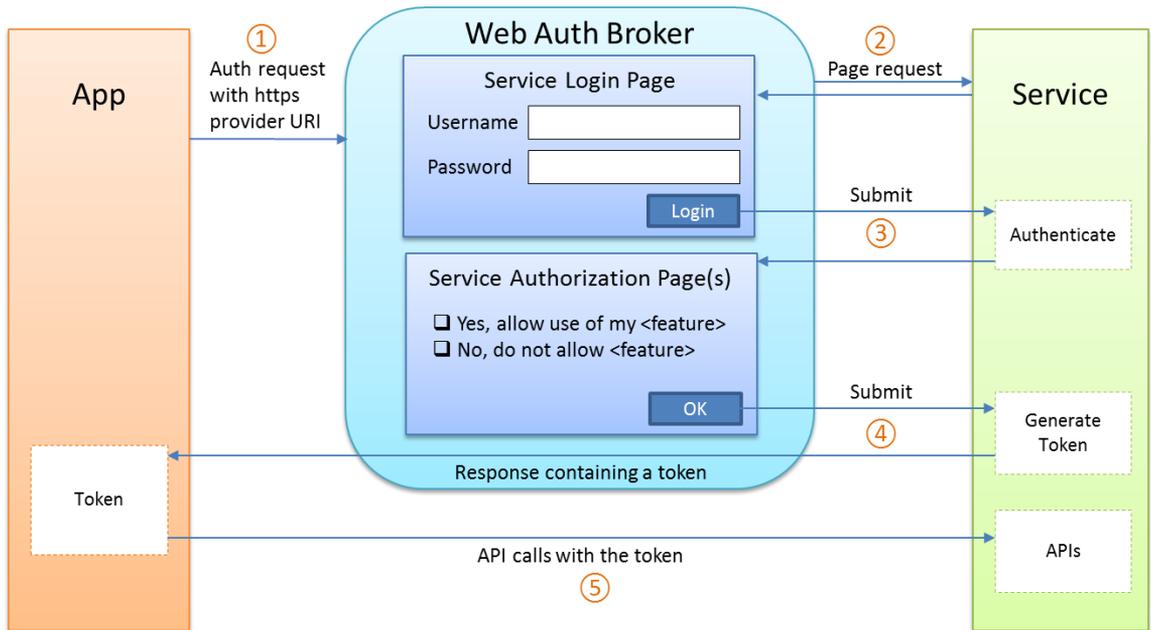
Within the broker UI—the branding of which is under the control of the provider—the user might be taken through multiple pages on the provider’s site (but note that the back button next to the “Connecting to a service” title dismisses the dialog entirely). But this begs a question: how does the broker 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` (returns the body of the page with the results), 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.<sup>12</sup>

The results given to the completed handler for `authenticateAsync` is a `WebAuthenticationResult` object. This contains properties named `responseStatus` (a `WebAuthenticationStatus` with either `success`, `userCancel`, or `errorHttp`), `responseData` (a string that will contain the page title and body if the `useTitle` and `useHttpPost` options are set, respectively), and `responseErrorDetail` (an HTTP response number).

**Tip** Web authentication events are visible in the Event Viewer under *Application and Services Logs > Microsoft > Windows > WebAuth > Operational*. This can be helpful for debugging because it brings out information that is otherwise hidden behind the opaque layer of the broker. The Fiddler tool is also very helpful for debugging. For more details, see [Troubleshooting web authentication broker](#).

Generally speaking, the app is most interested in the contents of `responseData`, because it will contain whatever tokens or other keys that might be necessary later on. Let's look at this again in the context of Scenario 1 of the [Web authentication broker sample](#). Set a breakpoint within the completed handler for `authenticateAsync` (line 59 or thereabouts), and then run the sample, enter an app ID you created earlier, and click Launch. (Note that the `callbackUri` parameter is set to `https://www.facebook.com/connect/login_success.html`, which is where the authentication process finishes up.)

In the case of Facebook, the `responseData` contains a string in this format:

```
https://www.facebook.com/connect/login_success.html#access_token=<token>&expires_in=<timeout>
```

where `<token>` is a bunch of alphanumeric gobbledygook and `<timeout>` is some period defined by Facebook. If you're calling any Facebook APIs—which is likely because that's why you're authenticating through Facebook in the first place—the `<token>` is the real treasure you're after because it's how you authenticate the user when making later calls to that API.

This 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

---

<sup>12</sup> As described on [How the web authentication broker works](#), `requestUri` and `callbackUri` “correspond to an Authorization Endpoint URI and Redirection URI in the OAuth 2.0 protocol. The OpenID protocol and earlier versions of OAuth have similar concepts.”

OAuth 2.0 protocol, the `requestUri` for each is relatively simple (ignore the word wrapping):

```
https://www.facebook.com/dialog/oauth?client_id=<client_id>&redirect_uri=<redirectUri>&scope-read_stream&display=popup&response_type=token
```

```
https://accounts.google.com/o/oauth2/auth?client_id=<client_id>&redirect_uri=<redirectUri>&response_type=code&scope=http://picasaweb.google.com/data
```

where `<client_id>` and `<redirectUri>` are replaced with whatever is specific to the app. Twitter and Flickr, for their parts, use OAuth 1.0a protocol instead, so much more ceremony goes into creating the lengthy OAuth token to include with the `requestUri` argument to `authenticateAsync`. I'll leave it to the sample code to show those details.

## Single Sign-On

What we've seen so far with the credential locker and the web authentication broker works very well to minimize how often the app needs to pester the user for credentials. Where a single app is concerned, it would ideally only ask for credentials once until such time as the user explicitly logs out. But what about multiple apps? Imagine over time that you acquire some dozens, or even hundreds, of apps from the Windows Store that use services that all require authentication. Even if those services exclusively use well-known OAuth providers, it'd still mean that you'd have to enter your Facebook, Twitter, Google, LinkedIn, Tumblr, Yahoo, or Yammer credentials in each and every app. At that point, the fact that you only need to authenticate each app once gets lost in the overall tedium!

From the user's point of view, once they've authenticated through a given provider in one app, it makes sense that other apps should benefit from that authentication if possible. Yes, some apps might need to prompt for additional permissions and some providers may not support the process, but the ideal is again to minimize the fuss and bother where we can.

The concept of *single sign-on* is exactly this: authenticating the user in one app (or the system in the case of a Microsoft account) effectively logs the user in to other apps that use the same provider. To make this work, the web authentication broker keep persisted logon cookies for each service in a special app container that's completely isolated from apps but yet allows those cookies to be shared between apps (like cookies are shared between websites in a browser). At the same time, each app must often acquire its own access keys or tokens, because these should not be shared between apps. So the real trick is to effectively perform the same kind of authentication we've already seen, only to do it without showing any UI unless it's really necessary.

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](#).<sup>13</sup> 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.

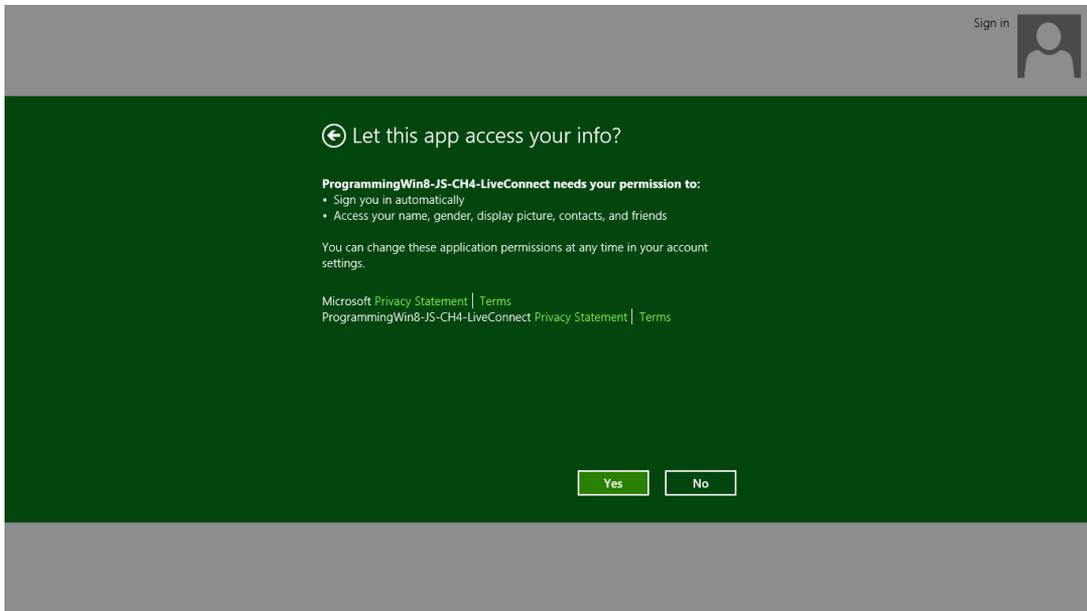
## The Live Connect User Experience

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](#)

---

<sup>13</sup> 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 8 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.

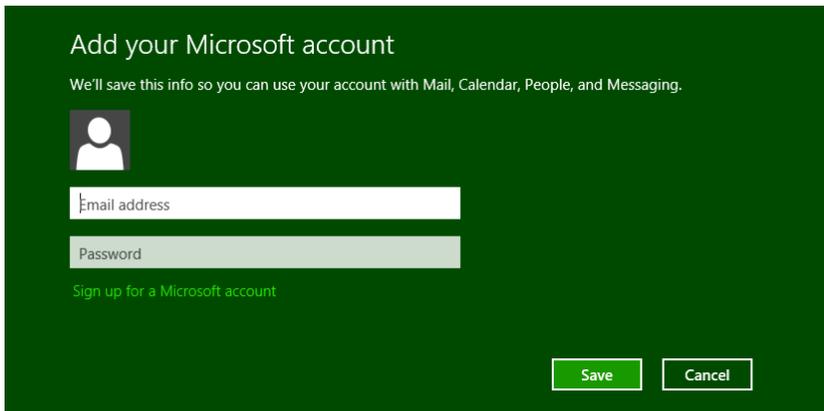
[Microsoft account sign-in experience](#) for additional requirements.)



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

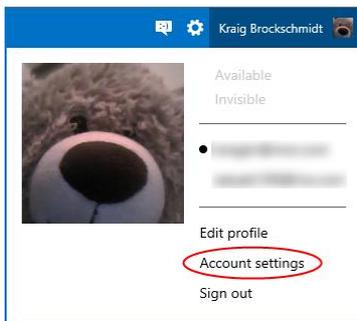
With this Live Connect login, the information that appears here (and in the other UI described below) comes through a configuration that's specific to Live Connect. You can do this in two ways, assuming you've created a profile for the app in the Windows Store dashboard. One way is to visit <https://account.live.com/Developers/Applications/> 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), 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.

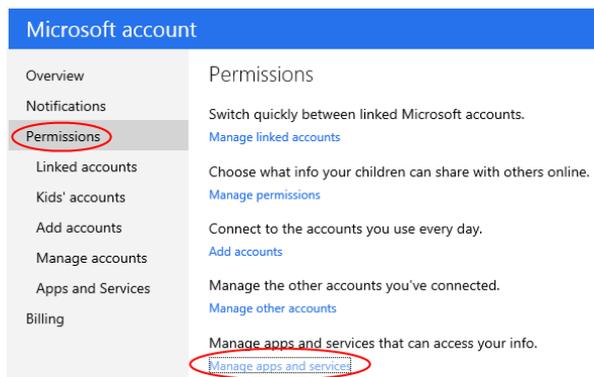


**FIGURE 4-10** The Microsoft account login dialog if the user logged in to Windows with a domain account.

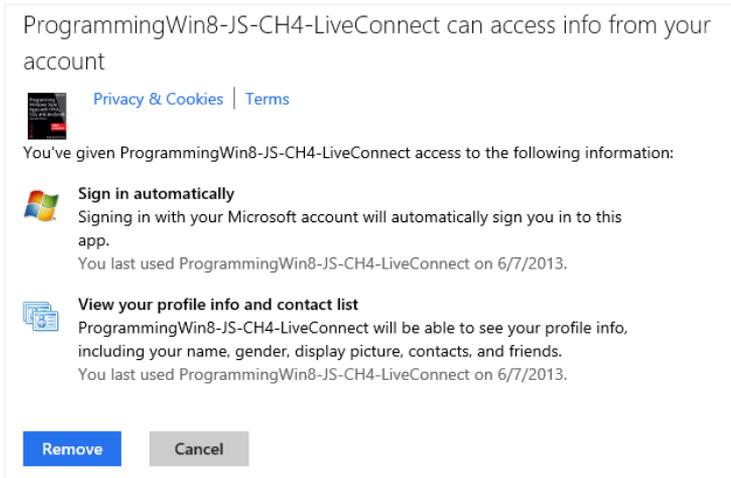
Once you've consented to any request from an app, those permissions can be managed through the Microsoft Account portal, <https://account.live.com>. You can also get there from <http://www.live.com> by clicking your name on the upper right. This will pop up some options (as shown below), where Account 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. 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. 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. 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 18, "Apps for Everyone."

Third is the [UserInformation](#) class, whose capabilities are clearly exercised within PC Settings > Account > Account picture:

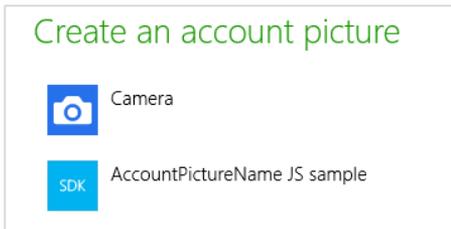
- **User name** If the `nameAccessAllowed` property is `true`, an app can then call `getDisplaynameAsync`, `getFirstNameAsync`, and `getLastNameAsync`, all of which provide a string to your completed handler. If `nameAccessAllowed` is false, these methods will complete but provide an empty result. Also note that the first and last names are available only from a Microsoft account.
- **User picture** Retrieved through `getAccountPicture`, which returns a [StorageFile](#) for the image. 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 > Personalize 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 13, "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 is 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 includes 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 and the lock screen image.

## Appendix A

# Demystifying Promises

In Chapter 3, “App Anatomy, Page Navigation, and Promises,” 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 6, “Collections and 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.

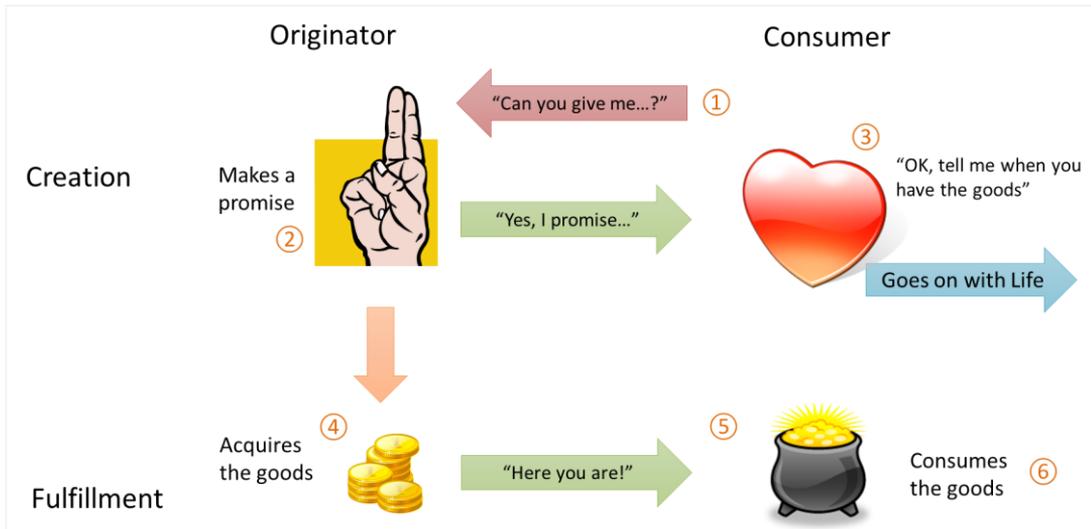
## What Is a Promise, Exactly? The Promise Relationships

---

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 brings 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.

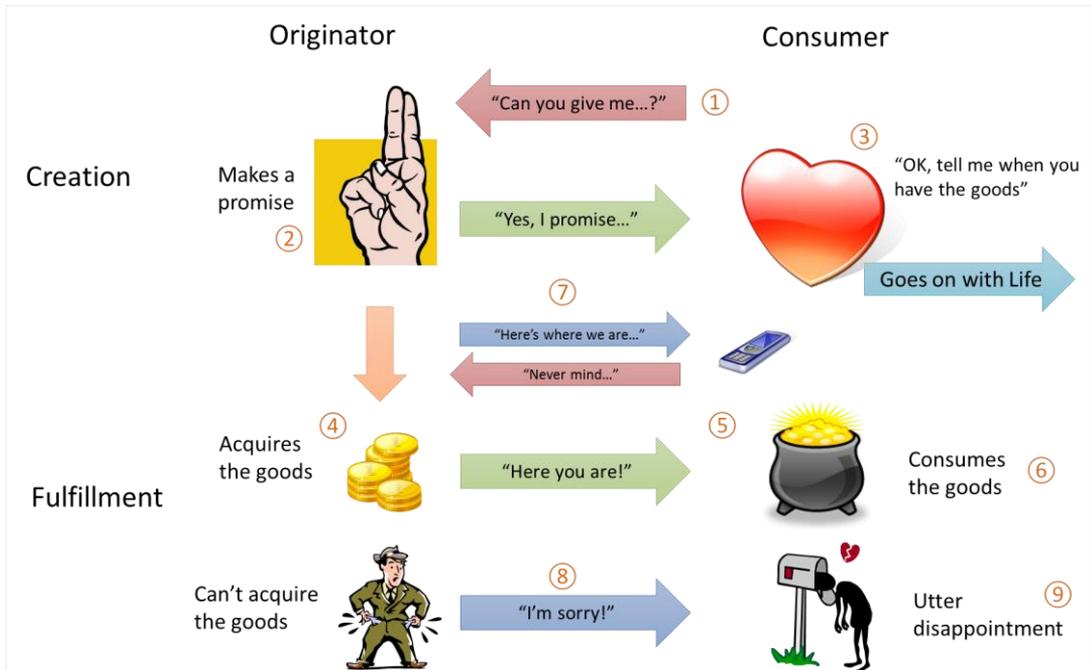


FIGURE A-2 The full promise relationship.

## 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.<sup>1</sup> 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 subscribe to the promise, passing a completed handler as the first argument. The promise must internally retain this function (unless the value is already

---

<sup>1</sup> 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!

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. Here's how we would use the API in the consumer:

```
//Consumer code
var promise = doSomethingForNothing();

promise.then(function (results) {
  console.log(JSON.stringify(results));
});
```

The output of this would be as follows:

```
{}
```

**App.log in the example** Although I'm showing `console.log` in the code snippets here, the Promises sample uses a function `App.log` that ties into the `WinJS.log` mechanism and directs output to the app canvas directly. This is just done so that there's meaningful output in the running example app.

The consumer code could be shortened to just `doSomethingForNothing().then(function (results) { ... });`; but we'll keep the promise explicitly visible for clarity. Also note that 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):

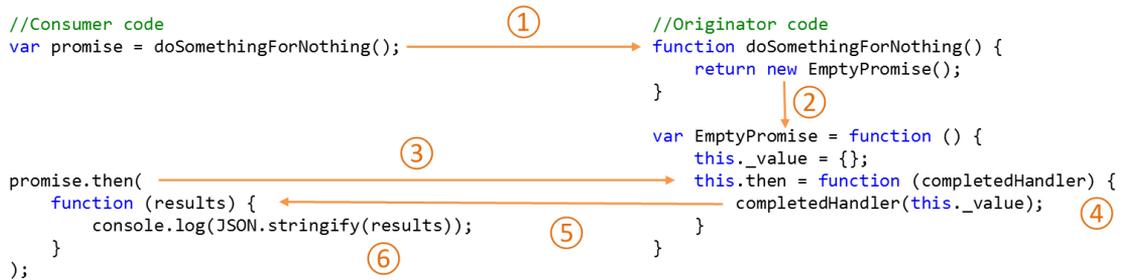
```
//Originator code
var EmptyPromise = function () {
  this._value = {};
  this.then = function (completedHandler) {
    completedHandler(this._value);
  }
}
```

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.<sup>2</sup>

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

---

<sup>2</sup> 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).

## 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.

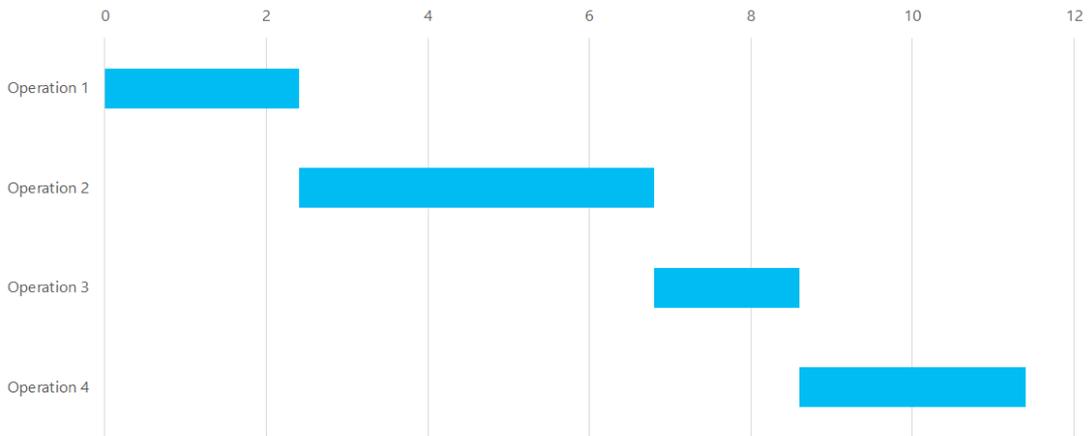
Running this code, you'll see that we pass through all the consumer code first and the first call to `then` starts the operation. The progress handler will be called a number of times and then the completed handlers. The resulting output is as follows (the numbers might change depending on the current 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, 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 handler), 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 `IntegerSummationPromise`, 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 `opPromise2` and `opPromise1.then`. As a result, this join will not complete until both `opPromise2` is fulfilled *and* the completed handler for `opPromise1` returns. Voila! The `completed2` handler we now attach to this join will not be called until after `completed1` 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 `completedn` 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."

## Appendix B

# 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 Chapter 4, in the section “Data from Services and HTTP Requests,” 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 <a href="#">XMLHttpRequest enhancements</a> .	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 section "Creating Promises" in Chapter 3 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 to debug 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 business than a large download. If an error occurs during the upload, it can invalidate the entire transfer—a very 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 using [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](#) 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 IE 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 [Cryptography.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.

And lastly it's worth at least listing the API under [Windows.Security.ExchangeActiveSyncProvisioning](#) 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 and AtomPub 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 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.

## 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 through 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.

One last note: if [retrieveFeedAsync](#) throws an exception, 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.");
    }
}
}

```

## 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);

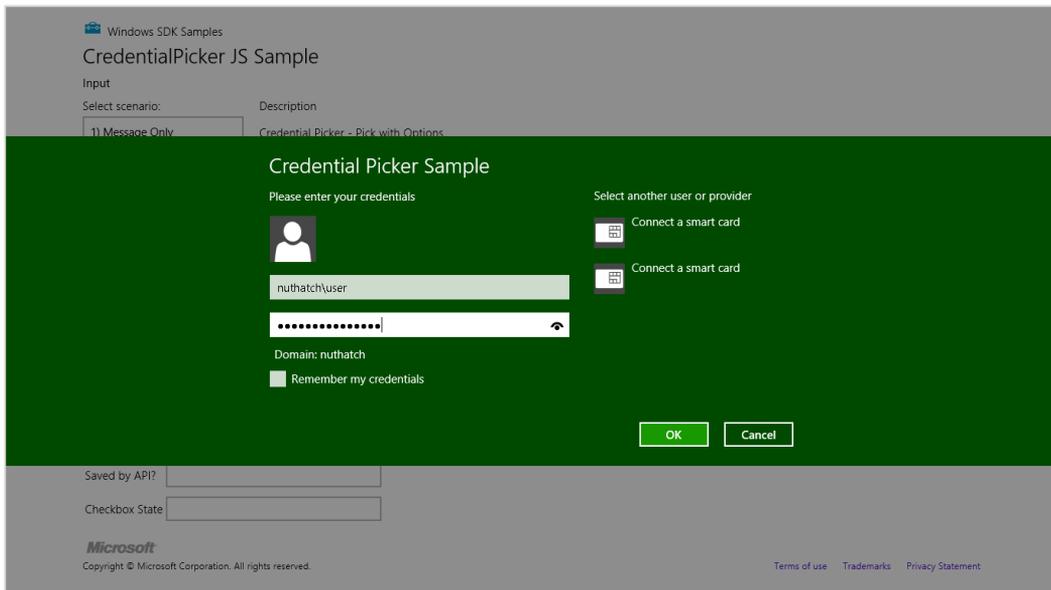
    // Match error number with a WebErrorStatus value, in order to deal with a specific error.
    var errorStatus = Windows.Web.WebError.getStatus(err.number);
    if (errorStatus === Windows.Web.WebErrorStatus.unauthorized) {
        displayLog("Wrong username or password!");
    }
}

```

## 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 `customAuthenticationProcoto1` 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 ( <code>credentialSaved</code> 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.previousCredential` 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 `CredentialPicker.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)
<a href="#">Check if current session is remote sample</a>	This sample demonstrates the use of <a href="#">Windows.System.RemoteDesktop</a> API. Specifically, this sample demonstrates how to use the <a href="#">InteractiveSession.IsRemote</a> property to determine if the current session is a remote session.
<a href="#">HomeGroup app sample</a>	Demonstrates how to use a HomeGroup to open, search, and share files. This sample uses some of the HomeGroup options. In particular, it uses <a href="#">Windows.Storage.Pickers.PickerLocationId</a> enumeration and the <a href="#">Windows.Storage.KnownFolders.homeGroup</a> property to select files contained in a HomeGroup.
<a href="#">Remote desktop app container client sample</a>	Demonstrates how to use the <a href="#">Remote Desktop app container client</a> objects in an app.
<a href="#">RemoteApp and desktop connections workspace API sample</a>	Demonstrates how to use the <a href="#">WorkspaceBrokerAx</a> object in a Windows Store app.
<a href="#">SMS message send, receive, and SIM management sample</a>	Demonstrates how to use the Mobile Broadband SMS API ( <a href="#">Windows.Devices.Sms</a> ). This API can be used only from mobile broadband device apps and is not available to apps generally.
<a href="#">SMS background task sample</a>	Demonstrates how to use the Mobile Broadband SMS API ( <a href="#">Windows.Devices.Sms</a> ) with the Background Task API ( <a href="#">Windows.ApplicationModel.Background</a> ) 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.
<a href="#">USSD message management sample</a>	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 <a href="#">Windows.Networking.NetworkOperators</a> .)



## 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](http://www.kraigbrockschmidt.com).