Microsoft

Programming Windows® 8 Apps

with HTML, CSS, and JavaScript

FIRST PREVIEW

Kraig Brockschmidt

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

ISBN: 978-0-7356-7261-1

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

Introduction 6
Who This Book Is For 7
What You'll Need 8
Acknowledgements 8
Errata & Book Support 9
We Want to Hear from You 9
Stay in Touch 9
Chapter 1: The Life Story of a Metro Style App: Platform Characteristics of Windows 8 10
Leaving Home: Onboarding to the Store 11
Discovery, Acquisition, and Installation 14
Playing in the Sandbox: The App Container 17
Different Views of Life: View States and Resolution Scaling 21
Those Capabilities Again: Getting to Data and Devices 24
Taking a Break, Getting Some Rest: Process Lifecycle Management 26
Remembering Yourself: App State and Roaming 28
Coming Back Home: Updates and New Opportunities 32
And, Oh Yes, Then There's Design 34
Chapter 2: Quickstart 35
A Really Quick Quickstart: The Blank App Template 35
Blank App Project Structure 38
QuickStart #1: Here My Am! and an Introduction to Blend for Visual Studio 42
Design Wireframes 43
Create the Markup 45
Styling in Blend 47
Adding the Code 53
Extra Credit: Receiving Messages from the iframe 65
The Other Templates 66
Fixed Layout Template 67
Navigation Template 67
Grid Template 68

Split Template 68
What We've Just Learned 69
Chapter 3: App Anatomy and Page Navigation 70
Local and Web Contexts within the App Host 71
Referencing Content from App Data: ms-appdata 75
Sequential Async Operations: Chaining Promises 78
Debug Output, Error Reports, and the Event Viewer 81
App Activation 83
Branding Your App 101: The Splash Screen and Other Visuals 83
Activation Event Sequence 86
Activation Code Paths 87
WinJS.Application Events 90
Extended Splash Screens 91
App Lifecycle Transition Events and Session State 93
Suspend, Resume, and Terminate 94
Basic Session State in Here My Am! 98
Data from Services and WinJS.xhr 100
Handling Network Connectivity (in Brief) 103
Tips and Tricks for WinJS.xhr 104
Page Controls and Navigation 105
WinJS Tools for Pages and Page Navigation 105
The Navigation App Template, PageControl Structure, and PageControlNavigator 106
The Navigation Process and Navigation Styles 112
Optimizing Page Switching: Show-and-Hide 113
Completing the Promises Story 114
What We've Just Learned 116
Chapter 4: Controls, Control Styling, and Data Binding 117
The Control Model for HTML, CSS, and JavaScript 119
HTML Controls 120
WinJS stylesheets: ui-light.css, ui-dark.css, and win-* styles 123

Extensions to HTML Elements 124
WinJS Controls 124
WinJS Control Instantiation 126
Strict Processing and processAll Functions 127
Example: WinJS.UI.Rating Control 128
Example: WinJS.UI.Tooltip Control 129
Working with Controls in Blend 131
Control Styling 133
Styling Gallery: HTML Controls 135
Styling Gallery: WinJS Controls 138
Some Tips and Tricks 141
Custom Controls 142
Custom Control Examples 144
Custom Controls in Blend 146
Data Binding 147
Data Binding in WinJS 149
Additional Binding Features 155
What We've Just Learned 158
About the Author 159

Introduction

Welcome, my friends, to Windows 8! On behalf of the thousands of designers, program managers, developers, test engineers, and writers who have brought the product to life, I'm delighted to welcome you into a world of **Windows Reimagined**.

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

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

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

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

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

So when I bumped into my old friend Kyle Marsh in the fall of 2009 just after Windows 7 had been released and heard from him that Microsoft was planning to reinvigorate native app development for Windows 8, my ears were keen to listen. In the months that followed I learned that Microsoft was introducing a completely new API called the Windows Runtime (or WinRT). This wasn't meant to replace Win32, mind you; desktop apps would still be supported. No, this was a programming model built from the ground up for a new breed of touchcentric, immersive apps that could compete with those emerging on various mobile platforms. It would be designed from the app developer's point of view, rather than the system's, so that key features would take only a few lines of code to implement

rather than hundreds or thousands. It would also enable direct native app development in multiple programming languages, which meant that new operating system capabilities would surface to those developers without having to wait for an update to some intermediate framework.

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

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

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

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

Who This Book Is For

This book is about writing Metro style apps for Windows 8 using HTML5, CSS3, and JavaScript. Our primary focus will be on applying these web technologies within the Windows 8 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 everything else.

I'm also assuming that your interest in Windows 8 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 along with "Quickstart" sections to give you immediate experience with the tools, the API, and 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 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 core Windows engineering teams. 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

To work through this book, you should download and install the Windows 8 Release Preview along with the Windows SDK and tools. These, along with a number of other resources, are listed on http://msdn.microsoft.com/en-us/windows/apps/br229516. I also recommend you visit http://code.msdn.microsoft.com/windowsapps/Windows-8-Modern-Style-App-Samples and download the entire set of JavaScript samples; we'll be using many of them throughout this book.

Acknowledgements

In many ways, this isn't *my* book—that is, it's not an account of my own experiences and opinions about Metro style apps on Windows 8. I'm serving more as a storyteller, where the story itself has been written by the thousands of people in the Windows team whose passion and dedication have been a constant source of inspiration. Writing a book like this wouldn't be possible without all the work that's gone into customer research, writing specs, implementing, testing, and documenting all the details, managing daily builds and public releases, and writing perhaps the best set of samples I've ever seen for a platform. We'll be drawing on many of those samples, in fact, and even the words in some sections come directly from conversations I've had with the people who designed and developed a particular feature. I'm grateful for their time, and I'm delighted to give them a voice through which they can share their passion for excellence with you.

A number of individuals deserve special mention for their long-standing support of this project. First to Chris Sells, with whom I co-authored the earliest versions of this book; to Mahesh Prakriya, Ian LeGrow, Anantha Kancherla, Keith Boyd and their respective teams, with whom I've worked closely; and to Keith Rowe, Dennis Flanagan, and Ulf Schoo, under whom I've had the pleasure of serving. Thanks also to Devon Musgrave at Microsoft Press, and to all those who have reviewed chapters and provided answers to my endless streams of questions: Chris Tavares, Jesse McGatha, Josh Williams, Feras Moussa,

Jake Sabulsky, Henry Tappen, David Tepper, Mathias Jourdain, Ben Betz, Ben Srour, Adam Barrus, Ryan Demopoulos, Sam Spencer, Bill Ticehurst, Tarek Anya, Scott Graham, Scott Dickens, Jerome Holman, Kenichiro Tanaka, Sean Hume, Patrick Dengler, David Washington, Scott Hoogerwerf, Harry Pierson, Jason Olson, Justin Cooperman, Rohit Pagariya, Nathan Kuchta, Kevin Woley, Markus Mielke, Paul Gusmorino, as well as those I've forgotten and those still to come as additional chapters are added to this first preview. My direct teammates, Kyle Marsh, Todd Landstad, Shai Hinitz, and Lora Heiny have also been invaluable in sharing what they've learned in working with real-world partners.

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

Errata & Book Support

We've made every effort to ensure the accuracy of this preview ebook and its companion content. When the final version of this book is available (in fall 2012), any errors that are reported after the book's publication will be listed on our Microsoft Press site at oreilly.com. At that point, you can search for the book at http://microsoftpress.oreilly.com and then click the "View/Submit Errata" link. If you find an error that is not already listed, you can report it to us through the same page.

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

Please note that product support for Microsoft software is not offered through the addresses above.

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://www.microsoft.com/learning/booksurvey

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

Chapter 1

The Life Story of a Metro Style 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, of course, 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 provides what I personally think is a brilliant solution for Metro style 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.

WinRT APIs are implemented according to a certain low-level structure and then "projected" into different languages 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 names on methods, properties, and events.

The Windows team also made it possible to write native client 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 Metro style client apps. Those apps will, of course, be specific to the Windows 8 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!

Throughout this book we'll explore how to leverage what you know of standards-based web technologies to build great Metro style apps for Windows. 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, controls, collections, layout, input, and state management, followed by chapters on media, platform UI, networking, devices, WinRT components, the Windows Store, localization, accessibility, and more. So, 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 rest, renewal, and rebirth. 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!

Leaving Home: Onboarding to the Store

For Metro style 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 the Store alone.

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 Package command in Visual Studio.[1] The *package* itself is an *appx* file (.appx)—see Figure 1-1—that contains your app's code, resources, libraries, and a *manifest*. 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!

[1] To do this you'll need to create a developer account with the Store by using the Store/Open Developer Account command in Visual Studio Express. Visual Studio Express and Expression Blend, which we'll 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.

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.

The upload process will walk you through setting your app's name, 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 essentially goes through a series of job interviews, if you will: background checks (malware scans and GeoTrust certification) and manual testing by a human being who will read the notes you provide (so be courteous and kind!). Along the way, you can check your app's progress through the Windows Store Dashboard.[2]

The overarching goal with these job interviews (or maybe it's more like getting through airport security!) is to help users feel confident and secure in trying new apps, a level of confidence that isn't generally found with apps acquired from the open web. As all apps in the Store are certified, signed, and subject to ratings and reviews, customers can trust all apps from the Store as they would trust those recommended by a reliable friend. Truly, this is wonderful news for most developers, especially those just getting started—it gives you the same access to the worldwide Windows market that has

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

been previously enjoyed only by those companies with an established brand or reputation.

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

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 read the first parts of Chapter 17, "Apps for Everyone," before you start writing your app in earnest.)

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 Certification requirements for Windows apps. Otherwise, congratulations—your app is ready for customers!

Sidebar: The Store API and Product Simulator

The `Windows.ApplicationModel.Store.CurrentProduct` 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 `Windows.ApplicationModel.Store.CurrentProduct`**`Simulator`** class instead. This is entirely identical to `CurrentProduct` 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 `CurrentProductSimulator` to `CurrentProduct` 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 your 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 the platform 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 explains itself. Similarly, a communications app would generally ask for access to the camera and microphone, but a news reader app probably wouldn't. On the other hand, an ebook reader might declare access to the microphone *if* it had a feature to attach audio notes to specific bookmarks.

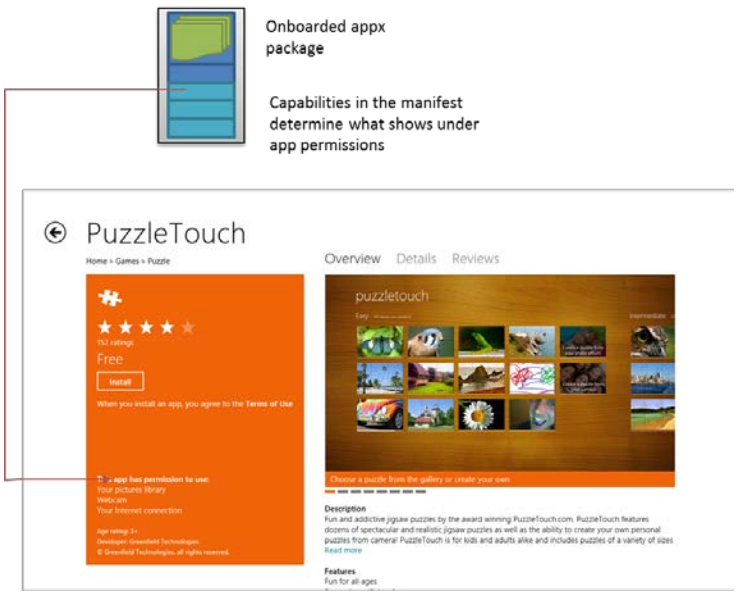


FIGURE 1-2 A typical app page in the Windows Store, where the manifest in the app package determines what appears in the app permissions. Here, for example, PuzzleTouch's manifest declares the Pictures Library, Webcam, and Internet (Client) capabilities.

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.[3]

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

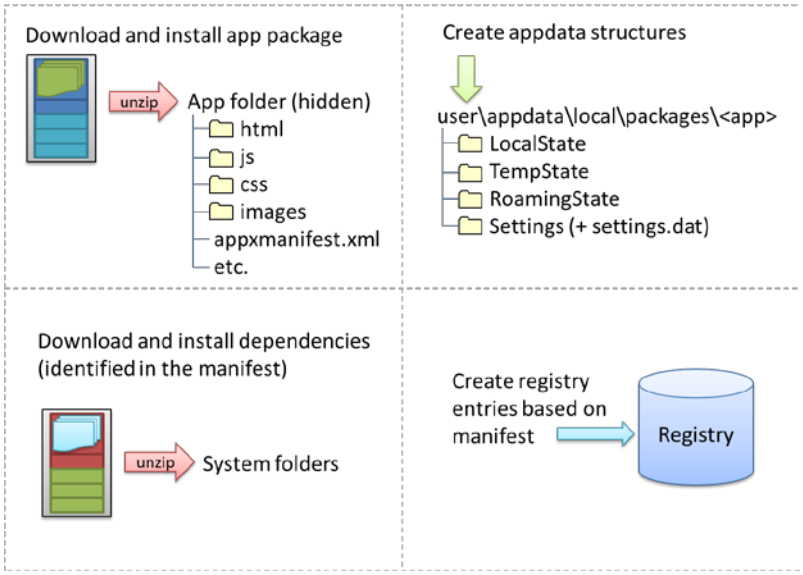


FIGURE 1-3 The Metro style app installation process; the exact sequence is unimportant.

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 all pretend to read? Almost all of them basically say that you can install the software on only one machine. Well, that changes with Metro style apps: instead of being licensed to a machine, they are

[3] 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.

licensed *to the user*, giving that user the right to install the app on up to five different devices.

In this way Metro style 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 the pets leftovers, 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 Metro style app is only parsed, compiled, and rendered at run time. (See the "Playing in the Sandbox: The App Container" section below.) As a result, a number of system-level features for Metro style 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 like any other intrinsic HTML controls. This is much more natural for developers to work with than having to create an instance of some WinRT class, bind it to an HTML element, and style it through code or some other markup scheme rather than CSS. Similarly, WinJS provides an animations library built on CSS, rather than forcing developers to learn some other structure to accomplish the same end. In both cases, WinJS provides a core implementation of the Metro style 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.

All in all, WinJS is essential for and shared between every Metro style 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.

Sidebar: Third-Party Libraries

WinJS is an example of a special shared library package that is automatically downloaded from the Windows Store for dependent apps. 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 Metro style apps. For example, Metro style apps written in JavaScript can certainly use jQuery, Modernizer, Dojo, prototype.js, Box2D, and others, with the caveat that some functionality, especially UI, might not be supported for Metro style apps. Apps can also use third-party binaries—known as WinRT components—that are again included in the app package. Also see the "Hybrid Apps" sidebar later in this chapter.

Playing in the Sandbox: The App Container

Now just as the needs of each day may be different when we wake up from our night's rest, Metro style 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 Metro style 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 manifest a running app with them. When your app is activated, then, what actually gets launched is that something: a special "app

host" process called wwahost.exe[4], as shown in Figure 1-4.

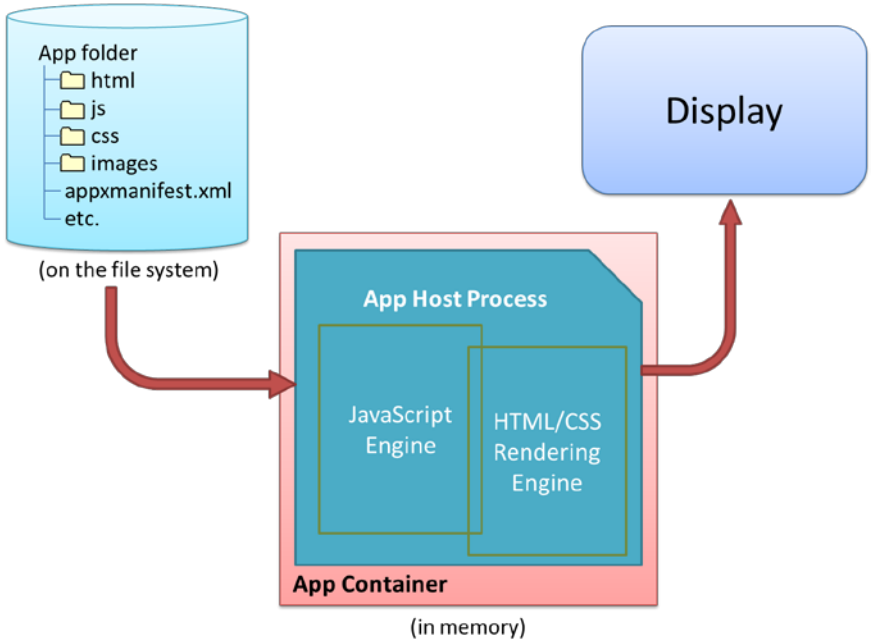


FIGURE 1-4 The app host is an executable (wwahost.exe) that loads, renders, and executes HTML, CSS, and JavaScript, in much the same way that a browser runs a web app.

The app host is more or less Internet Explorer 10 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` and `window.prompt`. (Try `Windows.UI.Popups.MessageDialog` instead.)
- The engines support additional methods, properties, and even CSS media queries that are specific to being a Metro style app as opposed to a website. For example, special media queries apply to the different Windows 8 *view states*; see the next section. 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 Metro style apps.)
- The default page of a Metro style app written in JavaScript runs in what's called the *local*

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

context wherein JavaScript code has access to WinRT, can make cross-domain XmlHttpRequests, and can access remote media (videos, images, etc.). However, you cannot load remote script (from http[s]:// sources, for example),[5] 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 individual `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 XHR. Web context `iframes` are generally used to host web controls on a locally packaged page (like a map), as we'll see in Chapter 2, "Quickstart," or to load pages that are directly hosted on the web.

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

Now all Metro style apps, whether hosted or not, run inside a security boundary called the *app container*. This is a sandbox that blocks local interprocess communication and either blocks or *brokers* access to system resources. The key characteristics of the app container are described next and then illustrated in Figure 1-5:

- All Metro style apps (other than those that are built into Windows) run at "base trust," which means that apps are treated like toddlers and not allowed to play with things like knives and chainsaws with which they could inflict serious damage on themselves and others.
- Metro style 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, if you will, 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.)
- Metro style apps cannot directly launch other apps by name or file path; they can programmatically launch other apps through file or protocol associations. As these are ultimately under the user's control, there's no guarantee that such an operation will start a specific app.
- Access to sensitive devices (like the camera, microphone, and GPS) is similarly controlled—the WinRT APIs that work with those devices will simply fail if the broker blocks those calls. And access to critical system resources, such as the registry, simply isn't allowed at all.
- Metro style apps are isolated from one another to protect from various forms of attack like defacement, intercepting input (click-jacking), and impersonation. This also means that some

[5] Note that it is allowable in the local context to eval JavaScript code obtained from remote sources through other means, such as XHR. The restriction on directly loaded remote script is to specifically prevent cross-site scripting attacks.

legitimate uses (like a snipping tool to copy a region of the screen to the clipboard) cannot be written as a Metro style app.

- Direct interprocess communication between Metro style apps, between Metro style and desktop apps, and between Metro style apps and local services, is blocked. Apps can still communicate through the cloud (web services, sockets, etc.), and many common tasks that require cooperation between Metro style apps—such as Search and Share—are handled through *contracts* in which those apps don't need to know any details about each other.

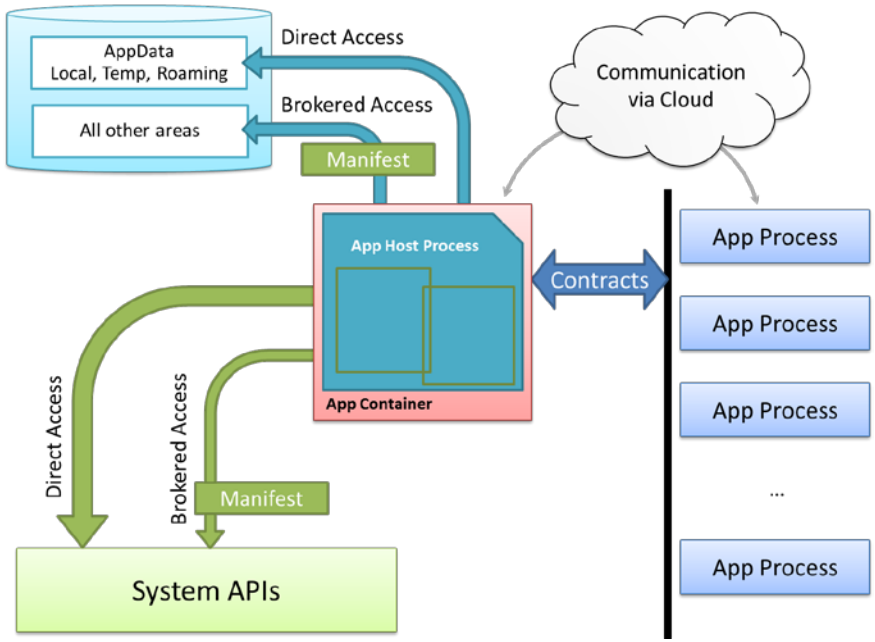


FIGURE 1-5 Process isolation for Metro style apps.

The upshot of all this is that certain types of apps just won't work as Metro style apps, such as file system utilities, antivirus, many kinds of development tools, registry cleaners, and anything else that can't be written with the WinRT APIs (or the available subset of Win32 and .NET APIs; see the next sidebar). In short, if there isn't an available API for the functionality in question, that functionality isn't supported in the app container. Such apps must presently be written as desktop apps.

Sidebar: Hybrid Apps

Metro style 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 small subset of Win32 and .NET APIs. (See Win32 and COM for Metro style apps.) Unfair? Not entirely, because you can write a *WinRT component* in those other languages that can the surface functionality built with those other APIs to the JavaScript environment (through the same projection mechanism that WinRT itself uses).

Because these components are also 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 *hybrid apps*, as they're called, thus use HTML/CSS for their presentation layer and some app logic, and they place the most performance critical or sensitive code in compiled DLLs. For details, see Chapter 16, "Extensions."

Different Views of Life: View States 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). 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—even though the Windows 8 Task Manager is in fact much more user-friendly.) 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?

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 keep the user fully 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 bottom edge, as you've probably seen, brings up the *app bar* where an app places most of its commands.

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 all these layout details in Chapter 6, "Layout."

What's also true is that your app might not always be running full-screen. In landscape mode, there are actually three distinct view states that you need to be ready for with every page in the app: *full-screen*, *snapped*, and *filled*. (See Figure 1-6.) These view states allow the user to split the screen into two regions, one that's 320 pixels wide along either the left or right side of the screen—the *snap region*—and a second that occupies the rest—the *fill region*. In response to user actions, then, your app might be placed in either region and must suck in its gut, so to speak, and adjust its layout appropriately. Most of the time, running in "fill" is almost the same as running in full-screen, except that the display area has slightly different dimensions and a different aspect ratio. Many apps will simply adjust their layout for those dimensions; in some cases, like movies, they'll just add a letterbox region to preserve the aspect ratio of the content. Both approaches are just fine.

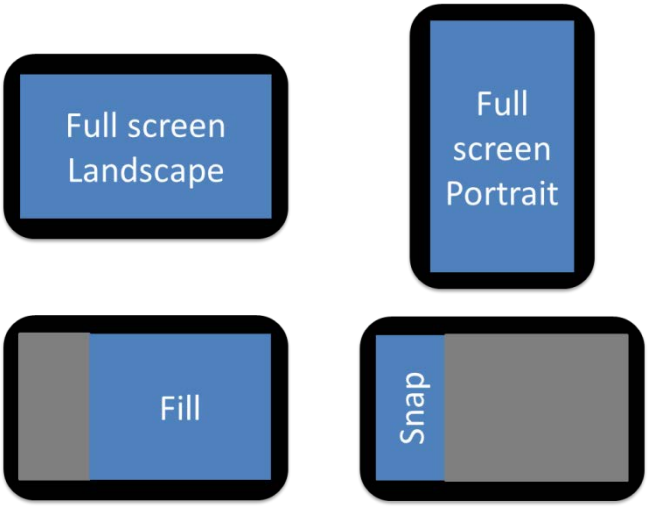


FIGURE 1-6 The four view states for Metro style apps; all pages within the app need to be prepared to show properly in all four view states, a process that generally just involves visibility of elements and layout that can often be handled entirely within CSS media queries.

When snapped, on the other hand, apps will often change the view of their content or its level of detail. Horizontally oriented lists, for instance, are typically switched into a vertical orientation, with fewer details. But don't be nonchalant about this: you really want to consciously design snap views for every page in your app and to 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 snap views, the more likely it is that users will keep that app visible even while they're working in another.

Another key point for snapping—and all the view states including portrait—is that they aren't mode changes. The system is just saying something like, "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 it's snapped; it should just present itself appropriately for that position. For snap

view in particular, if an app can't really continue to run effectively in snap, it should present a message to that effect with an option to un-snap back to full screen. (There's an API for that.)

Beyond the view states, an app should also expect to show itself in many sizes. It will be run on many different displays, anywhere from 1024x768 (the minimum hardware requirement for Windows 8, which also happens to be fill view size on 1366x768), 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 Designing flexible layouts and Designing UX for apps.

It might also be true that you're running on a high-resolution device that also has a very small screen (high *pixel density*), like 10" screens with a 2560x1440 resolution. Fortunately, Windows does automatic *scaling* such that the app still sees a 1366x768 display 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 also see in Chapter 6).

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 12, "Contracts."

Sidebar: Single-Page vs. Multipage Navigation

When you write a web app 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 Metro style 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 implication, 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 URL.

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 8 personality—it's the antithesis of "fast and fluid" and guaranteed to make designers cringe.

To avoid these concerns, Metro style 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 runtime (similar to AJAX). 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 basics of page loading and navigation in Chapter 3, "App Anatomy and Page Navigation."

Those Capabilities Again: Getting to Data and Devices

At run time, now, even inside the app container, the app has plenty of room to play and to delight your customers. It can utilize many different controls, as we'll see in Chapters 4 and 5, styling them however it likes from the prosaic to the outrageous and laying them out on a page according to your designer's fancies (Chapter 6). It can work with commanding UI like the app bar (Chapter 7) and receive and process *pointer events*, which unify touch, mouse, and stylus as shown in Chapter 9. (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 9), rich media (Chapter 10), animations (Chapter 11), contracts (Chapter 12), *tiles and notifications* (Chapter 13), network communication (Chapter 14), and various devices and printing (Chapter 15). And they can adapt themselves to different regional markets and provide accessibility (Chapter 17); work with various monetization options like advertising, trial versions, and in-app purchases (also Chapter 17); and draw on additional services like Windows Live (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 feature, then, are again controlled by a broker layer that will deny access if (a) the capability is not declared in the manifest, *or* (b) the user specifically disallows that access at run time. Those capabilities are listed in the following table:

Capability	**Description**	**Prompts for user consent at run time**
Internet (Client)	Outbound access to the Internet and public networks (which includes making requests to servers and receiving information in response).[6]	No
Internet (Client & Server) (superset of Internet Client; only one needs to be declared)	Outbound and inbound access to the Internet and public networks (inbound access to critical ports is always blocked).	No
Private Networks (Client & Server)	Outbound and inbound access to home or work intranets (inbound access to critical ports is always blocked).	No
Document Library	Read/write access to the user's Documents area on the file system for specifically declared file types.	No
Music Library Pictures Library Video Library	Read/write access to the user's Music/Pictures/Videos area on the file system (all files).	No
Removable Storage	Read/write access to files on removable storage devices for specifically declared file types.	No

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

Microphone	Access to microphone audio feeds (includes microphones on cameras).	Yes
Webcam	Access to camera audio/video/image feeds.	Yes
Location (GPS)	Access to the user's location.	Yes
Proximity	The ability to connect to other devices through near-field communication (NFC).	No
Text Messaging	The ability to send SMS (text) messages.	Yes
Enterprise Authentication	Access to intranet resources that require domain credentials; not typically needed for most apps.	No
Shared User Certificates	Access to software and hardware (smart card) certificates.	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 (from the app we'll create in Chapter 2). 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 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 8 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. So if you hit a problem 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, the documents library and removable storage capabilities both require 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 URIs to which your app can navigate within its own context (that is, without launching a browser). 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. Like I said earlier, you and your app become real 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 you're app is given this access, you can use certain APIs to record that permission so that you can get to those files and folders the next time your app is launched.

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

FIGURE 1-8 Using the file picker UI to access other parts of the file system from within a Metro style app, such as folders on a drive root (but not protected system folders). This is done by tapping the down arrow next to "Files."

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 *source* 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 or *view*—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 Metro style apps that helps conserve power and optimize battery life. One reality of life in 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 Metro style apps, Windows 8 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 state). So when most apps are no longer visible, there is really little need to keep their engines running on idle. 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 Metro style app in the background is completely obscured, it doesn't need to do such small bits of work and can be effectively frozen.

If the user then switches back to the app (in whatever view state, through whatever gesture), it will be scheduled for CPU time again and *resume* where it left off (adjusting its layout for the view state, of course). The app is also notified of this event in case it needs to re-sync with online services or refresh a view of a file system library, 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.

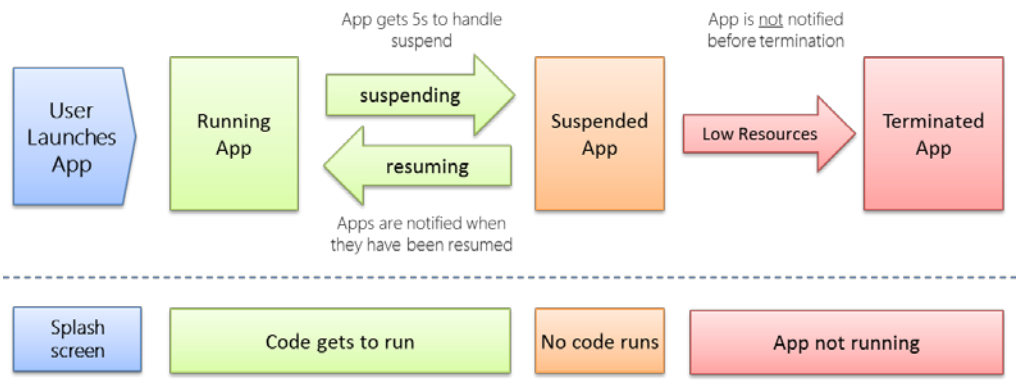


FIGURE 1-9 Process lifetime states for Metro style apps.

There are a couple of exceptions to this. First, Windows provides a *background transfer* API—see Chapter 14, "Networking"—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 Notification Service, WNS) so that they need not even be running for this purpose—see Chapter 13, "Live Tiles and Notifications." Second, certain kinds of apps that 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 10, "Media," an app specifies background audio in its manifest (where else!) and sets certain properties on the appropriate audio elements. With system events, as we'll also see in Chapter 13, an app declares background tasks in its manifest that are tied to specific functions in their code. In both cases, then, Windows will not suspend the app when it's in the background, or it will wake the app from the suspended state when an appropriate trigger occurs.

Over time, of course, the user might have many Metro style 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; Windows Store policy specifically disallows apps with their own close commands or gestures—he or she still rightly thinks that the app is running. So if the user activates it again (as from its tile), the user will expect to return to the same place he or 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

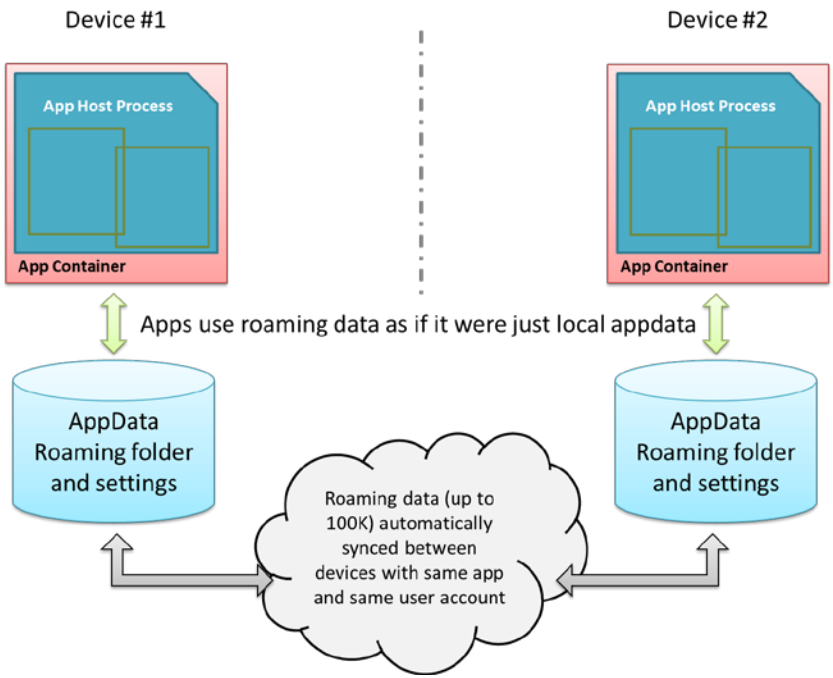
To step back for a moment, one of the key differences between traditional desktop apps and Metro style apps is that the latter are inherently stateful. That is, once they've run the first time, they remember their state across invocations (unless explicitly closed by the user or unless they provide an affordance to reset the state explicitly). Some desktop applications like Microsoft OneNote work like this, but most suffer from a kind of identity crisis when they're launched. Like Gilderoy Lockhart in

Harry Potter and the Chamber of Secrets, they often start up asking themselves, "Who am I?"[7] with no sense of where they've been or what they were doing before.

Clearly this isn't a good idea with Metro style 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 save their state when being suspended, in case they get terminated, and that they reload that state if they're launched again after being terminated. (An app receives a flag on startup to indicate its previous execution state, which determines what it should do with saved state. Details are in Chapter 3 and Chapter 9, "Input and Sensors.")

There's another dimension to statefulness too. Remember from earlier in this chapter that a user can install the same Metro style 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 8 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 LiveID, as shown in Figure 1-10.



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

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

They key here is understanding how and where an app saves its state. (We already know when.) If you recall, there's one place on the file system where an app has unrestricted access: its appdata folder. Within that folder, Windows automatically creates subfolders named LocalFolder, RoamingFolder, and TempFolder when the app is installed (I typically refer to them without the "Folder" 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 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 8, "Purposeful Animations.")

Now, although the app can write as much as it wants to the app data 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. (See Chapter 8 and Chapter 18, "Services.")

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) 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).

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 some "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 certain 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 at least.

Sidebar: Local vs. Temp Data

For local caching purposes, an app can use either local or temp storage. The difference is that local data will always be under the explicit control of the app. 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.

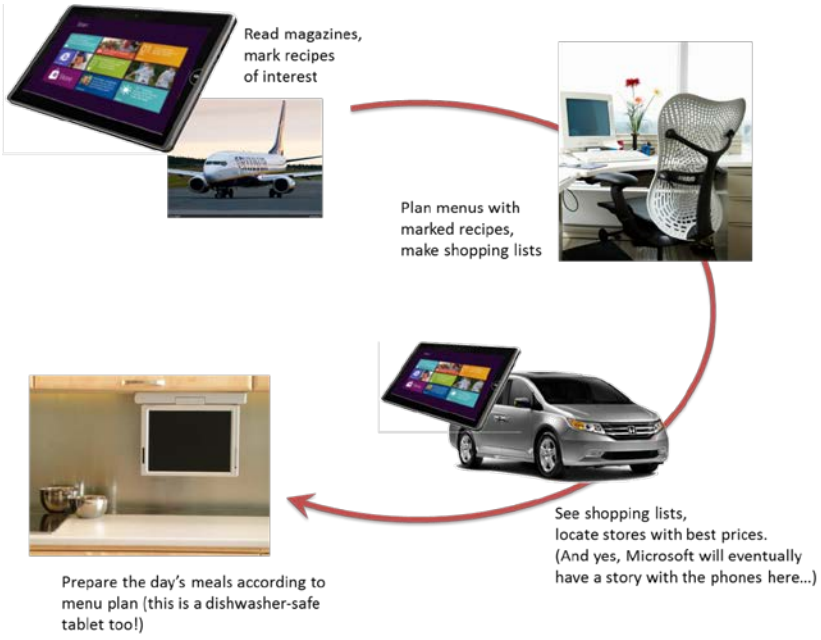
For Metro style apps written in HTML and JavaScript, you can also use existing caching mechanism like HTML5 local storage, IndexDB, appcache, and so forth. All of these will be stored within the app's Local folder.

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 your move through and interact with the world at large.



Coming Back Home: Updates and New Opportunities

If you're one of those developers that writes perfect apps 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—if you're not 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. Those customers who already have your app will also be notified that there's an update, which they can choose to install or not. (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 apps.)

When a user installs an update that has the same package name as an existing app, note that all the settings and appdata for the prior version remain intact. You 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 that Windows will transparently maintain multiple versions of the roaming state so long as there are separate versions installed on the user's devices. Once all the devices have the updated app and have converted their state, Windows will delete the old version.

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 Metro style 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 Metro style design—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 Metro—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 Metro style. But let's be honest: as a developer, do you really want to ponder what "fast and fluid" means? 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 four view states? 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.

So on that note, let's get into 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. 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.
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.

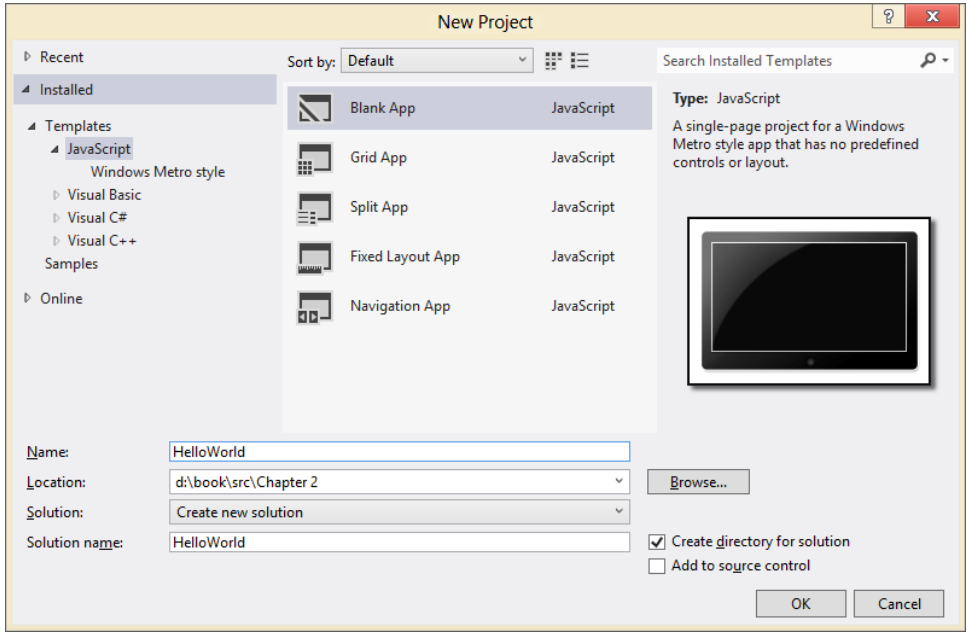


FIGURE 2-1 Visual Studio's New Project dialog.

4. After Visual Studio churns for a bit to create the project, click the Start Debugging button (or press F5, or select the same command from the Debug menu). 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 Metro style app on the other. Very handy. See Running Windows Metro style apps on the local machine for more on this.

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

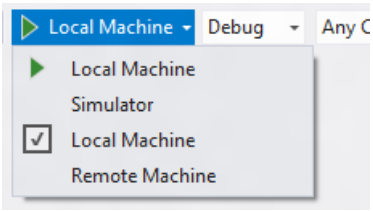


FIGURE 2-3 Visual Studio's debugging options on the toolbar.

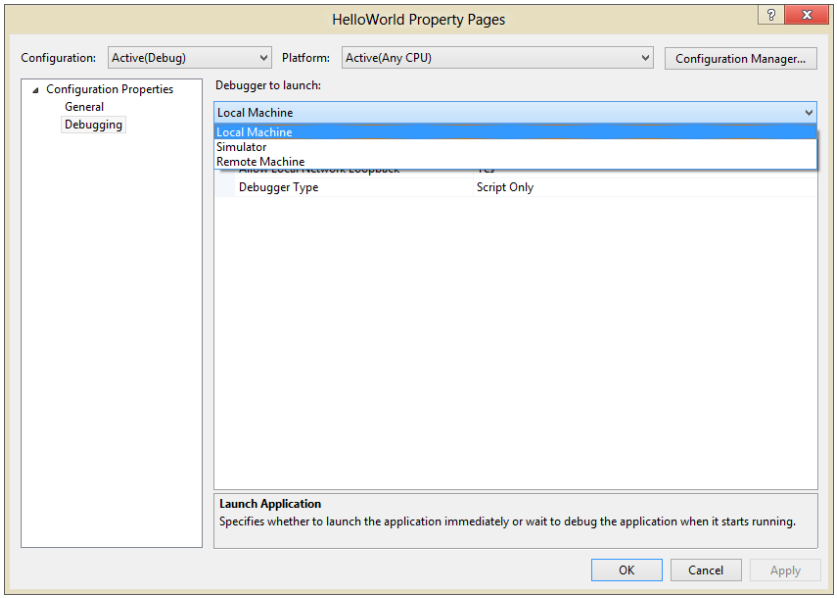


FIGURE 2-4 Visual Studio's debugging options in the app properties dialog.

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. We won't cover this topic in this book, so see Running Metro style apps on a remote machine for details. 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. For this I defer once more to the documentation: How to start a debugging session (JavaScript).

The Simulator is also very interesting, really the most interesting option in my mind and a place I imagine you'll be spending plenty of time. It duplicates your 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. We'll see more of the simulator as we go along, though you may also want to peruse the Running Metro style apps in the simulator topic.

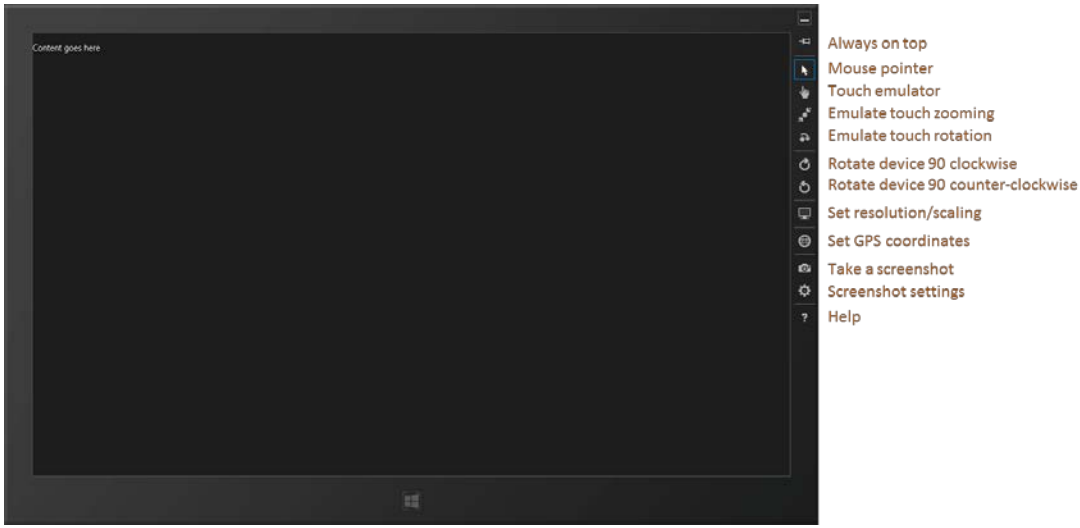


FIGURE 2-5 Hello World running in the simulator, with labels on the right for the simulator controls. Truly, the "Blank App" template lives up to its name!

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 page, where you can also uninstall it (a helpful step if you want to reset the appdata folders and other state).

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 wherever you want. 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 to side-load your app without having to give them your source project.

Blank App Project Structure

While an app created with the Blank template doesn't have much in the visual department, it provides much more where project structure is concerned. Here's what you'll find coming from the template, which 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.

- **package.appmanifest** The manifest. Opening this file will show Visual Studio's manifest editor (shown later in this chapter). I encourage you to browse around in this UI for a few minutes to familiarize yourself with what's all here. For example, you'll see references to the images noted below, a checkmark on the Internet Client capability checked, 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 Manifest designer topics. And if you want to explore the manifest XML directly, right-click this file and select View Code.
- **<Appname>_TemporaryKey.pfx** A temporary signature created on first run.

The css folder contains a core 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 6, "Layout."

The images folder contains four reference images, and unless you want to look like a real doofus developer, you'll *always* want to customize these before your app is complete (along with providing scaled versions as we'll see in Chapter 3, "App Anatomy and Page Navigation"):

- **logo.png** A default 150x150 (100% scale) image for the Start page.
- **smalllogo.png** A 30x30 image for the zoomed-out Start page.
- **splashscreen.png** A 620x300 image that will be shown while the app is loading.
- **storelogo.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.

The js folder contains a simple default.js.

The References folder points to CSS and JS files for the WinJS library. You can open any of these to see how WinJS itself is implemented. (Note: if you want to search within these files, you must open and search only within the specific file. These are not included in solutionwide or projectwide searches.)

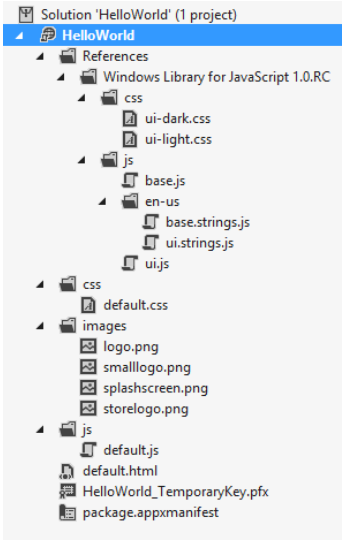


FIGURE 2-6 A Blank app project fully expanded in Solution Explorer.

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.RC/css/ui-dark.css" rel="stylesheet">
    <script src="//Microsoft.WinJS.1.0.RC/js/base.js"></script>
    <script src="//Microsoft.WinJS.1.0.RC/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 (perhaps using `ui-light.css` instead) in every HTML file of your project. The //'s in the WinJS paths refer to shared libraries rather than files in you 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 effect.

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`:

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

    var app = WinJS.Application;
    var activation = Windows.ApplicationModel.Activation;
    WinJS.strictProcessing();

    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 and `WinJS.strictProcessing` in Chapter 4, "Controls, Control Styling, and Basic Data Binding." For now, remember from Chapter 1, The Life Story of a Metro Style App," that an app can be activated in many ways. These are indicated in the `args.detail.kind` property, whose values come from the `Windows.ApplicationModel.Activation.ActivationKind` enumeration.

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

Now, what is that `args.setPromise(WinJS.UI.processAll())` for? As we'll see many times, `WinJS.UI.processAll` instantiates any WinJS controls that are declared in HTML—any element (commonly a `div` or `span`) that contains a `data-win-control` attribute whose value is the name of a

constructor function. Of course, 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.[8] As for `args.setPromise`, that's employing something called a deferral that we'll fittingly defer to Chapter 3.

That little `app.start();` at the bottom is also a very important piece, even for as short as it is. It makes sure that various events that were queued during startup get processed. We'll again see the details in Chapter 3.

Finally, you may be asking, "What on earth is all that ceremonial `(function () { … })();` business about?" It's just a conventional way in JavaScript (called the *module pattern*) to keep 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 modulewide but don't appear in the global namespace.[9]

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.

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.

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 whose fathers were also engineers—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, this next app can capture an image from a camera, locate your position on a map, and share that information through the Windows 8 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. But more importantly, it took a short amount of time because I learned how to use my tools—especially

[8] There is a similar function `WinJS.Binding.processAll` that processes `data-win-bind` attributes (Chapter 4), and `WinJS.Resources.processAll` that does resource lookup on `data-win-res` attributes (Chapter 17).

[9] See Chapter 2 of Nicolas Zakas's *High Performance JavaScript* (O'Reilly, 2010) for the performance implications scoping.

Blend—and I knew where I could find code that already did most of what I wanted, namely all the Windows SDK samples that you can download from http://code.msdn.microsoft.com/windowsapps/.

As we'll be drawing from some of these most excellent samples in this book, I encourage you to go download the whole set—go to the URL above, and the first download below the featured ones will take you to a page where 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 sourcing data via share came directly from a couple of samples.

I also *strongly* encourage you to spend a day, even a half-day, getting familiar with Visual Studio and Blend for Visual Studio and just perusing through the samples so that you know what's there. Trust me: 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*—Metro style—to apply to apps. 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 whatever 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 certain kinds of 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 couple years and invest in that training yourself. Simply said, *design matters* for Metro style apps, and it will make the difference between apps that merely exist in the Windows Store and are largely ignored and apps that succeed. 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 Abode Illustrator, be sure to visit http://design.windows.com for the philosophy and details of Metro style design, plus design resources.)

When I had the idea for this app, I drew up a simple wireframe, let a few designers laugh at me behind my back, and landed on layouts for the full screen, portrait, snap, and fill view states as shown in Figure 2-7 and Figure 2-8.

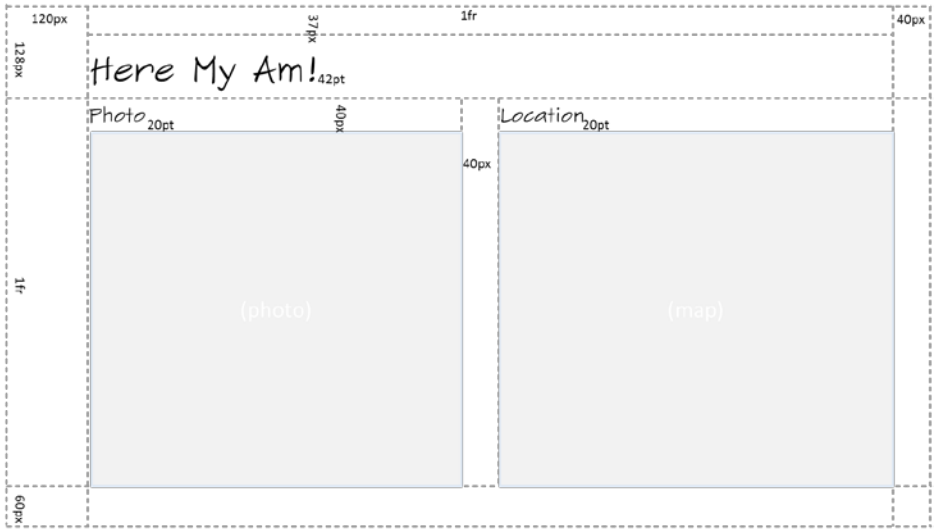


FIGURE 2-7 Full-screen landscape and filled (landscape) wireframe. These states typically use the same wireframe (the same margins), with the proportional parts of the grid simply becoming smaller with the reduced width.

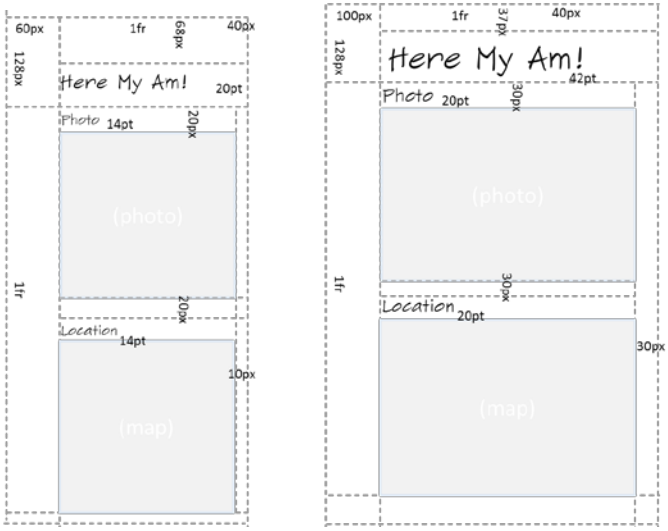


FIGURE 2-8 Snapped wireframe (left; landscape only) and full-screen portrait wireframe (right).

Sidebar: Design for All Four View States!

Just as I thought about all four view states together for Here My Am!, I encourage you to do the same for one simple reason: *your app will be put into every view state whether you design for it or not*. Users, not the app, control the view states, 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 6, 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 four view states.

This might sound like a burden, but view states don't affect function: they are simply different views of the same information. Remember that changing the view state never changes the *mode* of the app. Handling the view states, 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.

One of the important aspects of Metro style design is following what's called the layout *silhouette*: the size of the header fonts, their placement, the specific margins, and all that (as marked in the previous figures). It might seem restrictive, but the purpose of this recommendation is to encourage a high degree of consistency between apps so that users' eyes literally develop muscle memory for common elements of the UI. Some of this can be found in Understanding the Windows 8 silhouette and is otherwise incorporated into the templates along with many other design aspects. It's one reason why Microsoft generally recommends starting new apps with a template and going from there. What I show in the wireframes above reflects the layouts provided by one of the more complex templates.

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. As you may know, Blend has been available (at a high price) to designers and developers working with XAML (the presentation framework that is used by Metro style apps written in C#, Visual Basic, and C++). Now Blend is free and also supports HTML, CSS, *and* JavaScript. I emphasize that latter point because it 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 share the same project file formats and have commands to easily switch between them, depending on whether you're focusing on design or development. 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 below. (Note: Video 2-1, a snapshot of which is shown later in this chapter, shows all these steps together.)

Following the practice of writing pure markup in HTML—with no styling and no code—let's drop the following markup into the `body` element of default.html (replacing the one line of `<p>Content goes here</p>`):

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

Here we see the five elements in the wireframe: a main header, two subheaders, a space for a photo (defaulting to an image with "tap here" instructions), and an `iframe` that specifically houses a page in which we'll instantiate a Bing maps web control.[10]

You'll see that some elements have style classes assigned to them. Those that start with `win-` come from the WinJS stylesheet.[11] You can browse these in Blend by using the Style Rules tab, shown in Figure 2-9. Other styles like `titlearea`, `pagetitle`, and `group-title` are meant for you to define in your own stylesheet, thereby overriding the WinJS styles for particular elements.

[10] 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.

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

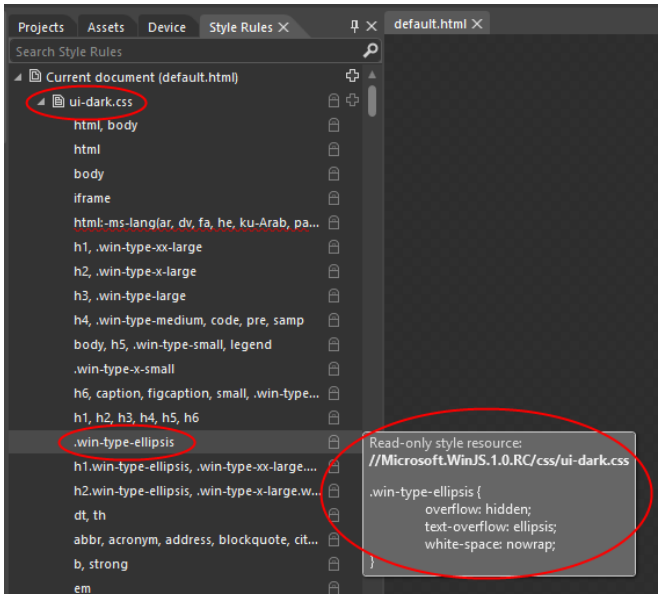


FIGURE 2-9 In Blend, the Style Rules tab lets you look into the WinJS stylesheet and see what each particular style contains. Take special notice of the search bar under the tabs. This is here so you don't waste your time visually scanning for a particular style—just start typing in the box, and let 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 everything inside it 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, 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 URL.

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 (including variables and functions) is shared between these contexts; communication between the two goes through the HTML5 `postMessage` function, as we'll see later.

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 17, "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 use 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.

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.

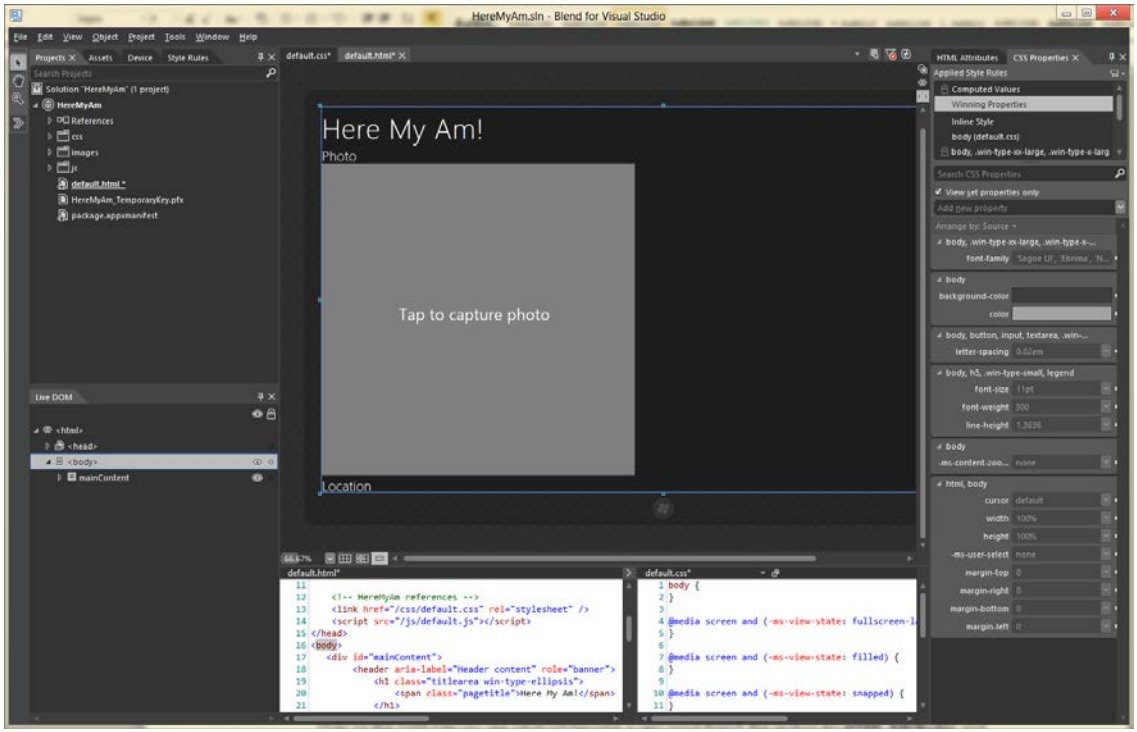


FIGURE 2-10 The app in Blend without styling, showing a view that is much like the Visual Studio simulator. If the taphere.png image doesn't show after adding it, use the View/Refresh menu command.

The tabs along the upper left in Blend give you access to your Project files; Assets like all the controls you can add to your UI; Device features like setting orientation, screen resolution, and view state; and a browser for all the Style Rules defined in the environment. On the lower left side, the Live DOM area lets you browse your element hierarchy. Clicking an element here will highlight it in the designer, just like 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. In the latter case, 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). What's selected in that box, mind you, will determine where changes in the properties pane below will be written, so be very conscious of your selection!

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 makes Blend's display in the artboard work better at present. So add `display: -ms-grid; -ms-grid-rows: 1fr; -ms-grid-columns: 1fr` to default.css for that element.

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 within them, following the general pattern of styling that works

best in Blend:

- Right-click the element you want to style in the Live DOM, and select Create Style Rule From Element Id or Create Style Rule From Element Class.

  Note If both of these items are disabled, go to the HTML Attributes pane (upper right) and add an id, class, or both. Otherwise you'll be hand-editing the stylesheets later on to move styles around, so you might as well save yourself the trouble.

  This will create a new style rule in the app's stylesheet (e.g., default.css). In the CSS properties pane on the right, then, find the rule that was created and add the necessary style properties in the pane below.

- Repeat with every other element.

If you look in the default.css file, you'll notice that the body element is styled with a 1x1 grid—leave this in place, because it makes sure the rest of your styling adapts to the screen size.

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: 128px 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 our case we could use one grid, but instead we'll add those margins in a nested grid within the header and section elements.

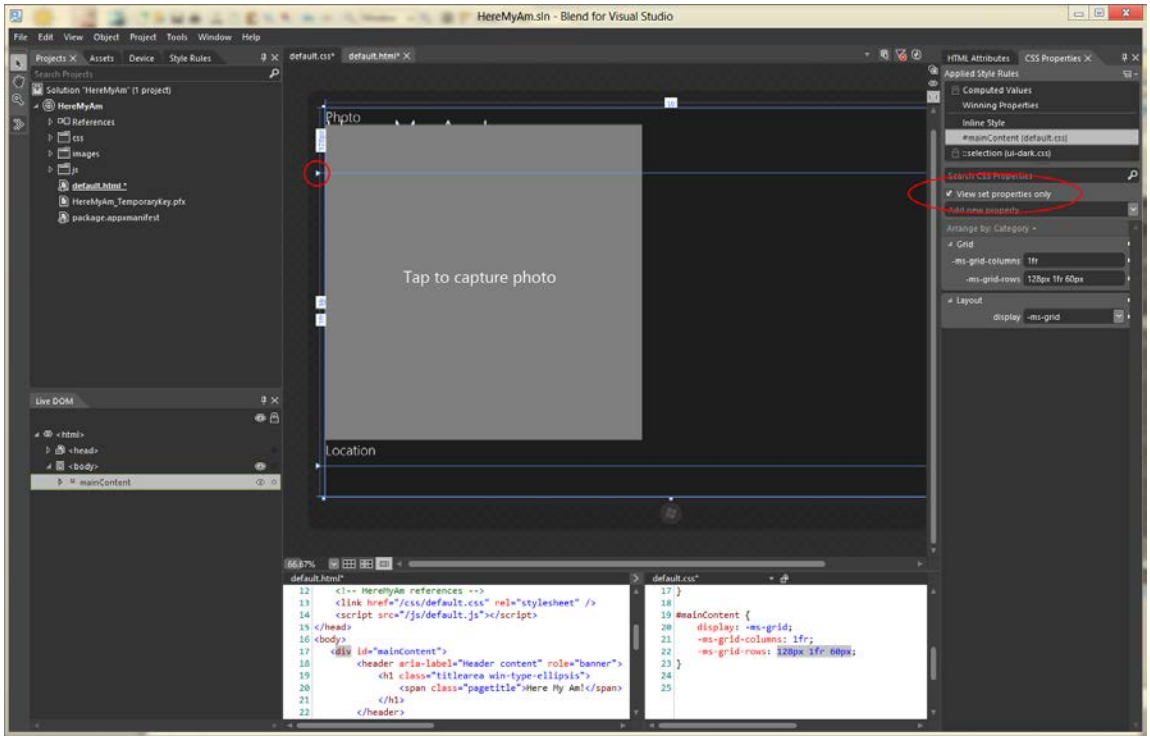
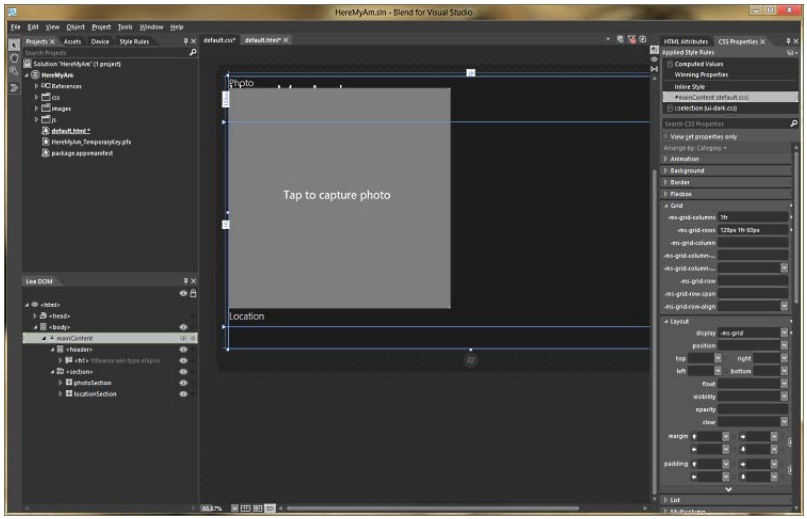


FIGURE 2-11 Setting the grid properties for the *mainContent* `div`. Notice how the View Set Properties Only checkbox (upper right) makes it easy to see what styles are set for the current rule. Also notice in the main "Artboard" how the grid rows and columns are indicated, including sliders (circled)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

Showing this and the rest of the styling—going down into each level of the markup and creating appropriate styles—is best done in video. Video 2-1, which is available as part of this book's downloadable companion content, shows this whole process starting with the creation of the project, styling the different view states, 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 as a verification.



VIDEO 2-1 Styling the Here My Am! app in Blend, demonstrating the approximate amount of time it takes to style an app like this 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. Note that I did need to do a little textual editing of the CSS for those view states, but all in all the process is quite straightforward. 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 all over again, a painful process that I'm sure you're familiar with! (And the time savings are even greater with Interactive Mode.)

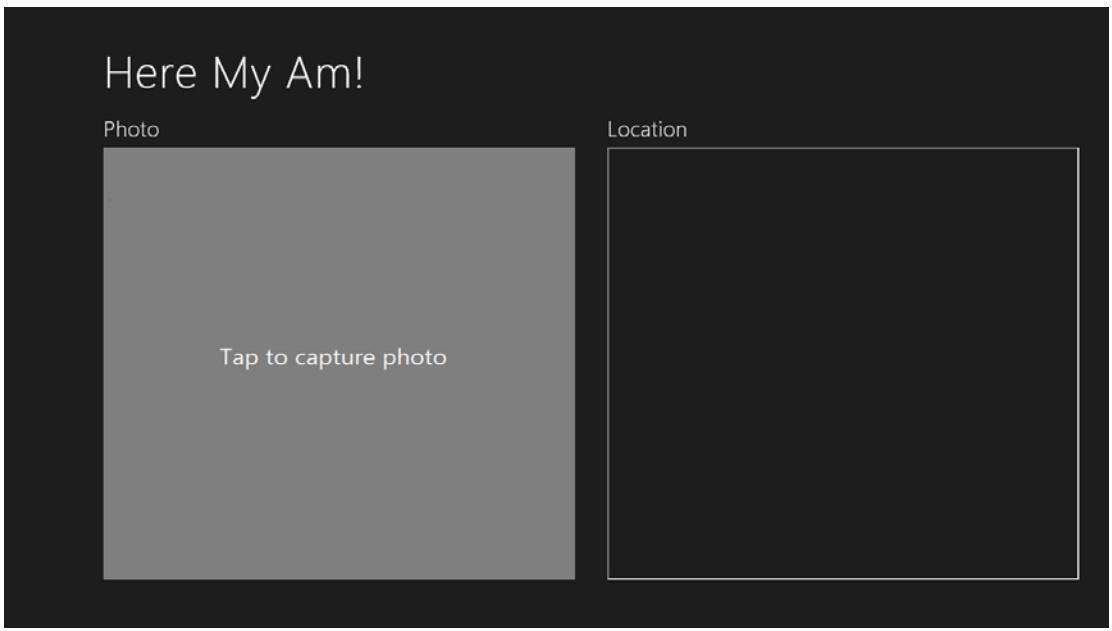


FIGURE 2-12 Full-screen landscape view.

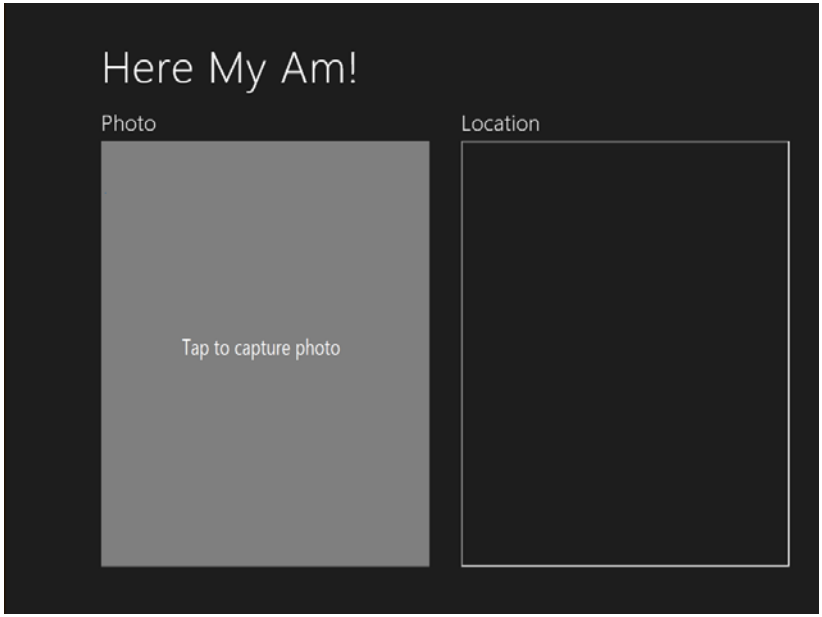


FIGURE 2-13 Filled view (landscape only).

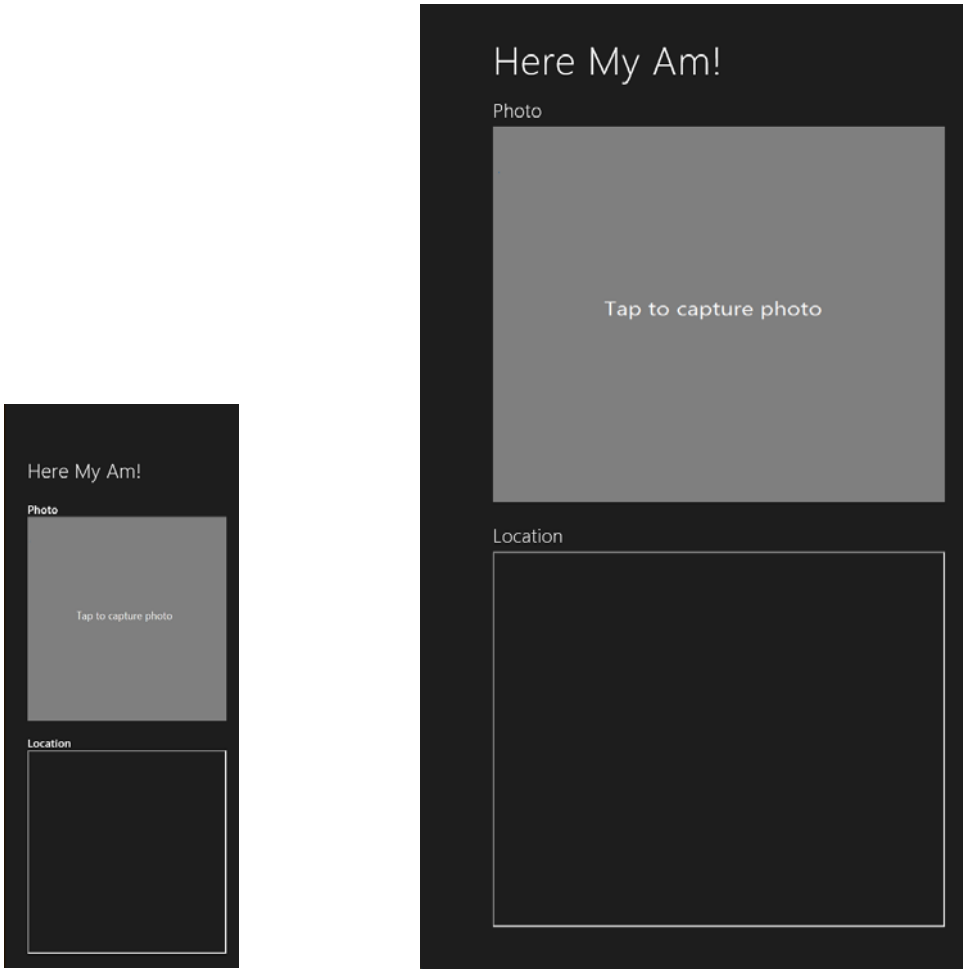


FIGURE 2-14 Snapped view (landscape only) and full-screen portrait view.

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.[12] At this point, we have the layout and styles for all the necessary view states, and our code doesn't need to care about any of it except to make some minor refinements, as we'll see in a moment.

What this means is that, for the most part, we can just write our app's code against the markup and

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

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. In this case we'll want to adjust the zoom level of the map in snapped view to account for the smaller display area. We'll just show this map automatically, so there's no control to start this process.
- Use the WinRT APIs for camera capture to get a photograph in response to a tap on the Photo `img` element.
- Provide the photograph and the location data to the Share charm when the user invokes it.

Figure 2-15 shows what the app will look like when we're done.

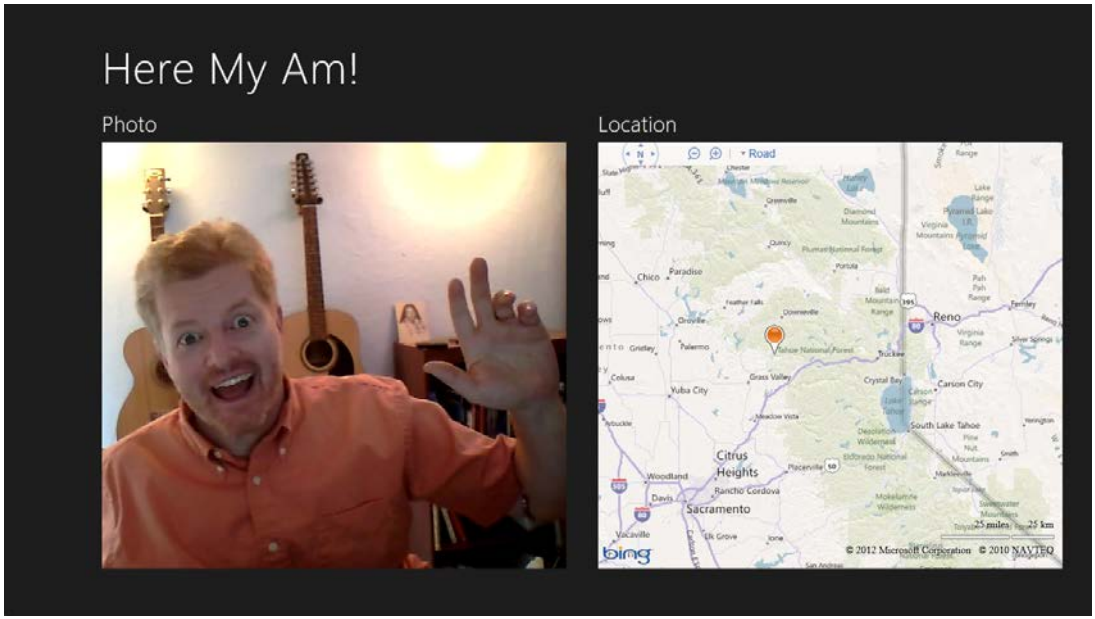


FIGURE 2-15 The Here My Am! app in its completed state.

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` of the main page. This is, at present, the recommended way to incorporate Bing Maps into a Metro style app written in JavaScript. Doing so also 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.

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:[13]

```
<!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);

            //Generic 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(message) {
                var call = JSON.parse(message.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);
            }


            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);
            }

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

[13] Note that you should replace the credentials inside the init function with your own key obtained from https://www.bingmapsportal.com/.

```
                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:
"AhTTNOioICXvPRPUdrO_NAYWj64MuGK2msfRendz_fL9B1U6LGDymy2OhbGj7vhA",
                    //zoom: 12,
                    mapTypeId: Microsoft.Maps.MapTypeId.road
                });
            }

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

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

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

                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 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 our app code. We instead use the HTML5 `postMessage` function, which raises a message event within the `iframe`.

In the code above, you can see that we pick up such messages and pass them to the `processMessage` function, a little generic function that turns a JSON string into a local function call, complete with arguments.

To see how this works, let's look at how we call `pinLocation` from within default.js. To make this call, we need some coordinates, which we can get from the WinRT Geolocation APIs. We'll do this within the `onactivated` handler, so the user's location is just set on startup (and saved in the `lastPosition` variable sharing later on):

```
//Drop this after the line: WinJS.strictProcessing();
var lastPosition = null;


//Place this after args.setPromise(WinJS.UI.processAll());
var gl = new Windows.Devices.Geolocation.Geolocator();

gl.getGeopositionAsync().done(function (position) {
    //Save for share
    lastPosition = { latitude: position.coordinate.latitude,
        longitude: position.coordinate.longitude };

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

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

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

A few key points about this code. First, to obtain coordinates, you can use the WinRT geolocation API or the HTML5 geolocation API. The two are almost equivalent, with slight differences described in Appendix B, "Comparing Overlapping WinRT and HTML5 APIs." The API exists in WinRT because other supported languages (like C# and C++) don't have access to the HTML5 geolocation APIs, and because we're primarily focused on the WinRT APIs in this book, we'll just use functions in the `Windows.Devices.Geolocation` namespace.

Next, in the second parameter to `postMessage` you see a combination of `ms-appx-web://` with `document.location.host`. This essentially means "the current app," which is the appropriate origin of the message.

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 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 led 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. It can also take two optional functions to wire up progress and error handlers as well. We'll see more about promises as we progress through this book, such as the `then` function that's just like `done` but allows further chaining (which we'll see in Chapter 3).

The argument passed to the *completed handler* (a function pass as the first argument to `done`) 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 the <> indicates the actual data type of the results, so you'll normally follow the link to that topic for the 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.[14]

One final note about async APIs. Within the WinRT API, all async functions have "Async" in their names. Because this isn't common practice within *JavaScript* toolkits or the DOM API, async functions within WinJS don't use that suffix. In other words, WinRT is designed to be language-neutral, but WinJS is designed to follow 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 tells us: we neglected to set the Geolocation capability in the manifest. Without that capability set, calls like this that depend on that capability will throw an exception.

We were running in the debugger, so that exception was kindly shown to us. If you run the app outside of the debugger—try it from the tile that should be on your Start page—you'll see that it 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:

```
gl.getGeopositionAsync().then(function (position) {
    //...
}, function(error) {
    console.log("Unable to get location.");
});
```

[14] The pushpin itself is draggable, but to no effect at present. See the section "Extra Credit: Receiving Messages from the iframe" later in this chapter for how we can pick up location changes from the map.

The `console.log` function writes a string to the *JavaScript Console window* in Visual Studio, which is obviously a good idea. Now run the app outside the debugger and you'll see that it comes up, because the exception is now considered "handled." In the debugger, set a breakpoint on the `console.log` line inside and you'll hit that breakpoint after the exception appears (and you press Continue).

If the exception dialog gets annoying, you can control which exceptions pop up like this in the Debug --> Exceptions dialog box (shown in Figure 2-16) within JavaScript Runtime Exceptions. If you uncheck User-unhandled, you won't get a dialog when the exception occurs. (That dialog also has a checkbox for this as well.)

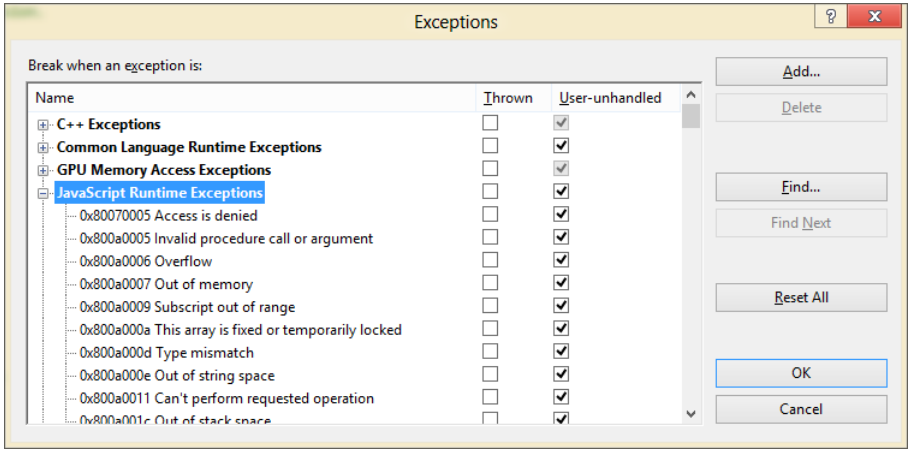


FIGURE 2-16 JavaScript run-time exceptions in the Debug/Exceptions dialog of Visual Studio.

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

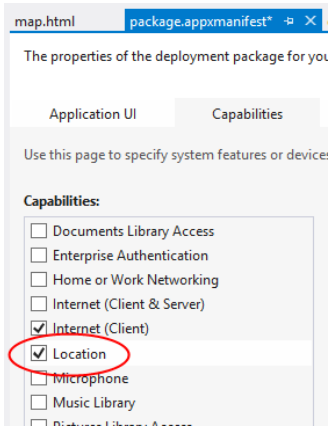


FIGURE 2-17 Setting the Location capability in Visual Studio's manifest editor. (Note that Blend supports editing the manifest only as XML.)

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 Figure 2-18. If the user blocks access here, the error handler will again be invoked as the API will throw an Access denied exception.

FIGURE 2-18 A typical consent popup, reflecting the user's color scheme, that appears when an app first tries to call a brokered API (geolocation in this case). If the user blocks access, the API will fail, but the user can later change consent in the Settings/Permissions panel.

Sidebar: How Do I Reset User Consent for Testing?

While debugging, you might notice that this popup appears only once, even across subsequent debugging sessions. To clear this state, invoke the Settings charm in the running app and click Permissions, and you'll see toggle switches for all the relevant capabilities. If for some reason you can't run the app at all, go to the Start screen and uninstall the app from its tile. You'll then see the popup when you next run the app.

Note that there isn't a notification when the user changes these Permission settings. The app can detect a change only by attempting to use the API again. We'll revisit this subject in Chapter 8, "State, Settings, Files, and Documents."

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 it doesn't on Windows 8. All the complexities of camera capture have been nicely encapsulated within the `Windows.Media.Capture` API to such an extent that we can add this feature with only a few lines of code.

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

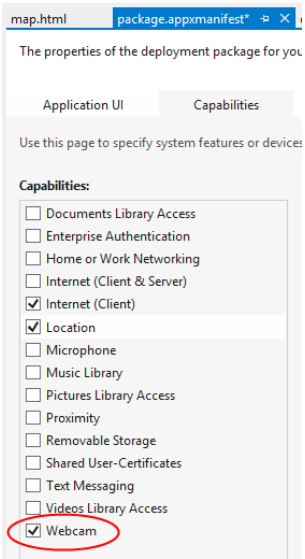


FIGURE 2-19 The camera capability in Visual Studio's manifest editor.

On first use of the camera at run time, you'll see a consent dialog, as with geolocation, like the one shown in Figure 2-20.

FIGURE 2-20 Popup for obtaining the user's consent to use the camera. You can control these through the Settings/Permissions panel at any time.

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 9, "Input and Sensors":

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

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;
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 that = this;

    var captureUI = new Windows.Media.Capture.CameraCaptureUI();

    //Indicate that we want to capture a PNG 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.png;
    captureUI.photoSettings.croppedSizeInPixels =
        { width: this.clientWidth, height: this.clientHeight };

    captureUI.captureFileAsync(Windows.Media.Capture.CameraCaptureUIMode.photo)
        .done(function (capturedFile) {
            //Be sure to check validity of the item returned; could be null if the user canceled.
            if (capturedFile) {
                lastCapture = capturedFile;  //Save for Share
                that.src = URL.createObjectURL(capturedFile, {oneTimeOnly: true});
            }
        }, function (error) {
            console.log("Unable to invoke capture UI.");
        });
}
```

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?

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 may other possibilities as discussed in Chapter 10, "Media"), 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.

Fortunately, that's super-easy as well! You can hand a `StorageFile` object directly to the HTML `URL.createObjectURL` method and get back an URL that can be directly assigned to the `img.src` attribute. Voila! The captured photo appears![15]

Note that `captureFileAsync` will call the completed handler if the UI was successfully invoked but

[15] The `{oneTimeOnly: true}` parameter indicates that the URL is not reusable and should be revoked via `URL.revokeObjectURL` when it's no longer used, as when we replace the `img src` with a new picture. Without this, we would 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.

the user hit the back button and didn't actually capture anything. This is why the extra check is there for 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, but note that a denial of consent will show a message in the capture UI directly (see Figure 2-21), so it's unnecessary to have an error handler for that purpose with this particular API. In most cases, however, you'll want to have an error handler in place for async calls.

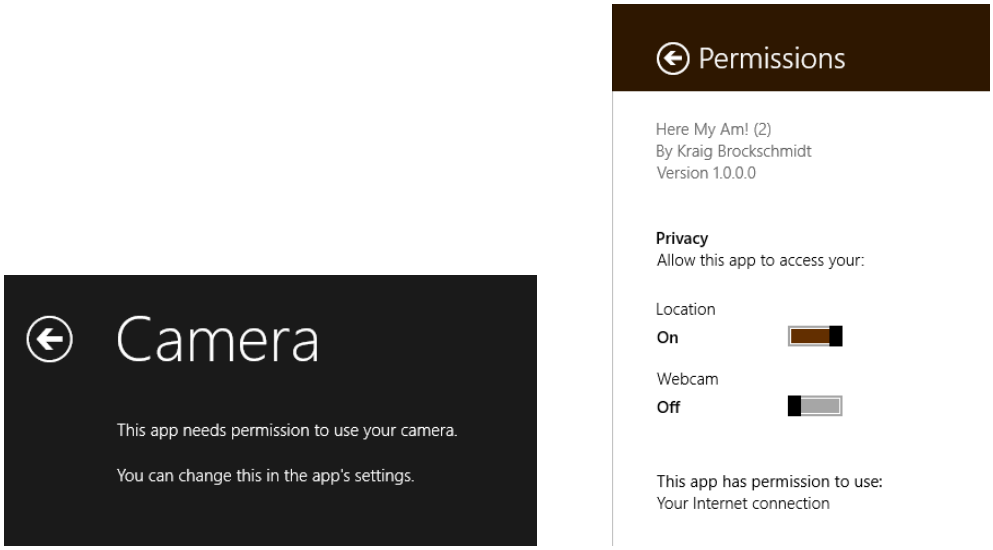


FIGURE 2-21 The camera capture UI's message when consent is denied (left); you can change permissions through the Settings/Permission pane (right).

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 has 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 sends whatever data that *source* app makes available (if any) to whatever other *target* app the user selects (from a list of those whose manifests identify then as targets). 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 as 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. This way, the user immediately returns to that source app when the sharing is completed, rather than having to switch back to that app manually.

So instead of adding code to our app to share the photo and location to a particular target, like Facebook, we only need to package the data appropriately when Windows asks for it.

That asking comes through the `datarequested` event sent to the `Windows.ApplicationModel.DataTransfer.DataTransferManager` object. First we just need to set up an appropriate listener—place this code is in the `onactivated` 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);
```

The idea of a *current view* is something that we'll see pop up now and then. It reflects that an app can be launched for different reasons—such as servicing a contract—and thus presents different underlying pages or views to the user at those times. These views (unrelated to the snap/fill/etc. view *states*) can be active simultaneously. To thus make sure that your code is sensitive to these scenarios, certain APIs return objects appropriate for the current view of the app as we see here.

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. (We've saved these in `lastPosition` and `lastCapture`.) In our case, we make sure we have position and a photo, then fill in text and image properties:

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

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

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

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

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

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

    //The setBitmap method requires a RandomAccessStream. A deferred share is set up so that the
```

```
    //RandomAccessStream can be created from the file asynchronously.
    var deferral = request.getDeferral();
    data.setBitmap(streamReference);
    deferral.complete();
}
```

The latter part of this code is pretty standard stuff for sharing a file-based image (which we have in `lastCapture`). The deferral business is a way to not deliver the whole image until it's really asked for—again, standard stuff. I got most of this code, in fact, directly from the SDK's Share content source app sample.

With this last addition of code, and a suitable sharing target installed (such as the SDK's Share content target app sample, as shown in Figure 2-22), 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!

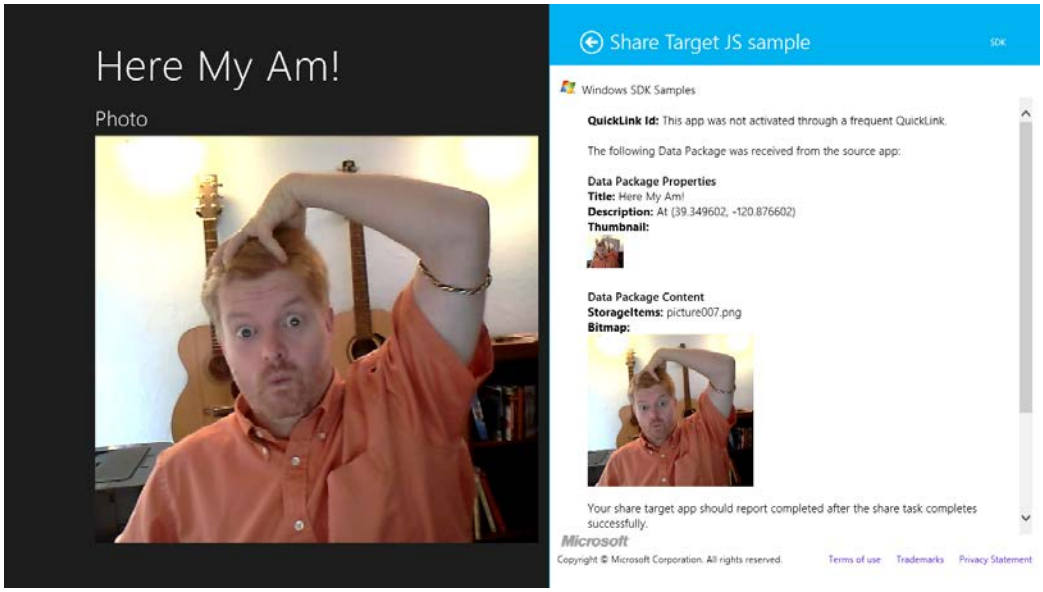


FIGURE 2-22 Sharing (monkey-see, monkey-do!) to the Share target SDK sample. Share targets appear as a partial overlay on top of the current app, so the user never leaves the app context.

Extra Credit: Receiving Messages from the iframe

There's one more piece I've put into Here My Am! to complete the basic interaction between app and `iframe` content: the ability to post messages from the `iframe` back to the main 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 whatever object is given containing parameters, and then stringifies the whole bit and posts it back to the parent.

When a pushpin is dragged, Bing maps raises a `dragend` event, which we'll wire up and handle in the `setLocation` function just after the pushpin is created (also in map.html):

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

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) {
    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.

The Other Templates

In this chapter we've worked only with the Blank App template so that we could understand the basics of writing a Metro style 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.

Fixed Layout Template

"A project for a Windows Metro style app that scales a fixed aspect ratio layout." (Blend/Visual Studio description)

What we've seen so far are examples of apps that adapt themselves to changes in display area by adjusting the layout. In Here My Am!, for instance, we used CSS grids with self-adjusting areas (those 1fr's in rows and columns). This works great for apps with content that is suitably resizable as well as apps that can show additional content when there's more room, such as more news headlines or items from a search.

Other kinds of apps are not so flexible, such as games where the aspect ratio of the playing area needs to stay constant. (It would not be fair if players on larger screens got to see more of the game!) So, when the display area changes—either from view states or a change in display resolution—they do better to scale themselves up or down rather than adjust their layout.

The Fixed Layout template provides the basic structure for such an app, just like the Blank template provides for a flexible app. The key piece is the `WinJS.UI.ViewBox` control, which automatically takes care of scaling its contents while maintaining the aspect ratio:

```
<body>
    <div data-win-control="WinJS.UI.ViewBox">
        <div class="fixedlayout">
            <p>Content goes here</p>
        </div>
    </div>
</body>
```

In default.css, you can see that the `body` element is styled as a CSS flexbox centered on the screen and the *fixedLayout* element is set to 1024x768 (the minimum size for the fullscreen-landscape and filled view states). Within the child `div` of the `ViewBox`, then, you can safely assume that you'll always be working with these fixed dimensions. The `ViewBox` will scale everything up and provide letterboxing as necessary.

Note that such apps might not be able to support an interactive snapped state; a game, for example, will not be playable when scaled down. In this case an app can simply pause the game and try to unsnap itself when the user taps it again. We'll revisit this in Chapter 6.

Navigation Template

"A project for a Windows Metro style 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. As discussed in Chapter 1, Metro style apps written in HTML/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.

This 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.

Grid Template

"A multi-page project for a Windows Metro style app that navigates among groups of items. 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 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 5, "Collections and Collection Controls." It also provides the structure for an app bar and shows how to simplify the app's behavior in snap view.

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

Split Template

"A two-page project for a Windows Metro style 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 then navigates to a group detail page that is split into two sides (hence the template name). The left side contains a vertically panning 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 both snap and portrait views intelligently. That is, because vertically oriented views don't lend well to splitting the display (contrary to the description above!), the template shows how to switch to a page navigation model within those view states to accomplish the same ends.

What We've Just Learned

- How to create a new Metro style app from the Blank app template.
- How to run an app inside the local debugger and within the simulator.
- The features of the simulator.
- The basic project structure for Metro style apps, including WinJS references.
- The core activation structure for an app.
- The role and utility of design wireframes in app development, including the importance of designing for all view states.
- How to quickly and efficiently add styling to an app's markup in Blend for Visual Studio.
- How to safely use web content (such as Bing maps) within an `iframe` and communicate between that page and the app.
- How to use the WinRT APIs, especially async methods involving promises but also geolocation and camera capture.
- The importance of manifest capabilities in being able to use certain WinRT APIs.
- How to share data through the Share contract.
- The kinds of apps supported through the other app templates: Fixed Layout, Navigation, Grid, and Split.

Chapter 3

App Anatomy and Page Navigation

During the early stages of writing this book, 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 most 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 becomes completely invisible to the eye once the wallboard is on and the finish work is in place.

But then, imagine what the house would be like without such careful attention to structural details. Imagine having some light switches that just didn't work or controlled the wrong fixtures. Imagine if the plumbing leaked. Imagine if cabinets and trim started falling off the walls after a week or two of living in the house. Even if the house managed 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! That is, an app might be visually beautiful, even stunning, but once you really start using it day to day, a lack of attention on the fundamentals will become painfully apparent.

This chapter, then, is about those fundamentals: the core foundational structure of an app upon which you can build something that can look beautiful *and* really work well. We'll first complete our understanding of the hosted environment and then look at activation (how apps get running) and lifecycle transitions. We'll then look at page navigation within an app, and we'll see a few other important considerations along the way, such as working with multiple async operations.

Let me offer you advance warning that this is an admittedly longer and more intricate chapter than many that follow, since it specifically deals with the software equivalents of framing, plumbing, and wiring. With our house, I can completely attest that installing the lovely light fixtures my wife picked out seemed, in the moment, much more satisfying than the framing 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!

Local and Web Contexts within the App Host

As described in Chapter 1, "The Life Story of a Metro Style App," Metro style apps written with HTML, CSS, and JavaScript are not directly executable apps 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 (plus resources, of course) 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 Metro style apps.

We've already covered most of the characteristics of the hosted environment in Chapter 1 and Chapter 2, "Quickstart":

- The app host (and the apps in it) run at base trust with brokered access to sensitive resources.
- 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.
- HTML content in the app package can be loaded into the *local* or *web context*, depending on the `ms-appx:///` and `ms-appx-web:///` scheme used to reference that content (`///` again means "in the app package"). Remote content (referred to with `http[s]://`) always runs in the web context.
- The local context has access to the WinRT API, among other things, whereas the web context is allowed to load and execute remote script but cannot access WinRT.
- Plug-ins are generally not allowed in either context.
- The HTML5 `postMessage` function can be used to communicate between an `iframe` and its containing parent across contexts. This 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. (Windows Store policy actually disallows this, and apps submitted to the Store will be 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, cannot be used on web *pages* outside of an app.)

Now what we're really after in this chapter is not so much these characteristics themselves but their

impact on the structure of an app. First and foremost is that an app's home page (the one you point to in the manifest in the Start page field of the Application UI tab[16]) *always* runs in the local context, and any page to which you navigate directly (`<a href>` or `document.location`) must also be in the local context. (You can try otherwise, but the app host will display an interesting "not supported" message right inside your app!) Those pages, however, can contain `iframe` elements in either context, depending on which scheme you use.

A local context page can contain an `iframe` in either local or web context, provided that the `src` attribute refers to content in the app package (and by the way, programmatic read-only access to your package contents is obtained via `Windows.ApplicationMode.Package.Current.InstalledLocation`). Referring to any other location (`http[s]://` or other protocols) will always place the `iframe` in the web context.

```
<!-- iframe in local context with source in the app package -->
<!-- this form is only allowed 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="http://www.bing.com"></iframe>
```

Also, if you use an `<a href="..." target="...">` tag with `target` pointing to an `iframe`, the scheme in `href` determines the context.

A web context page, for its part, can contain an `iframe` only in the web context; for example, the last two `iframe` elements above are allowed, whereas the first two are not. You can also use `ms-appx-web:///` within the web context to refer to other content within the app package, such as images.

Although not commonly done within Metro style apps for reasons we'll see later in this chapter, similar rules apply with page-to-page navigation using `<a href>` or `document.location`. Since the whole scene here can begin to resemble overcooked spaghetti, the exact behavior for these variations and for `iframes` is described in the following table:

Target	Result in Local Context Page	Result in Web Context Page
`<iframe src="ms-appx:///">`	`iframe` in local context	Not allowed
`<iframe src="ms-appx-web:///">`	`iframe` in web context	`iframe` in web context
`<iframe src="http[s]:///">` or other scheme	`iframe` in web context	`iframe` in web context
`<a href="[uri]" target="myFrame"> <iframe name="myFrame">`	`iframe` in local or web context depending in [uri]	`iframe` in web context; [uri] cannot begin with `ms-appx`.

[16] The manifest names this the "Start page," but I prefer "home page" to avoid confusion with the Windows Start screen.

`<a href="ms-appx:///">`	Navigates to page in local context	Not allowed unless explicitly specified (see below)
`<a href="ms-appx-web:///">`	Not allowed	Navigates to page in web context
`<a href="[uri]">` with any other protocol including `http[s]`	Opens default browser with [uri]	Opens default browser with [uri]

When an `iframe` is in the web context, note that its page can contains `ms-appx-web` references to in-package resources, even if the page is loaded from a remote source (`http[s]`). Such pages, of course, would not work in a browser.

The last two items in the table really mean that a Metro style app cannot navigate from its top-level page (in the local context) directly to a web context page of any kind (local or remote). That's just life in the app host! Such content must be placed in an `iframe`.

Similarly, navigating from a web context page to a local context page is not allowed by default, but 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:

```
//This must be called from the local context
MSApp.addPublicLocalApplicationUri("ms-appx:///frame-local.html");
```

The Direct Navigation example for this chapter gives a demonstration of this.

One other matter that arises here is the ability to grant a web context page access to specific functions like geolocation, writing to the clipboard, and file downloads—things that web pages typically assume they can use. By default, the web context in a Metro style app has no access to such operating system capabilities. For example, create a new Blank project in Visual Studio with this one line of HTML in the body of default.html:

```
<iframe src="http://maps.bing.com" style="width:1366px; height: 768px"></iframe>
```

Then set the Location capability in the manifest (something I forgot on my first experiment with this!), and run the app. You'll see the Bing page you expect.[17] However, attempting to use geolocation from within that page—clicking the locator control to the left of "World," for instance—will give you the kind of error shown in Figure 3-1.

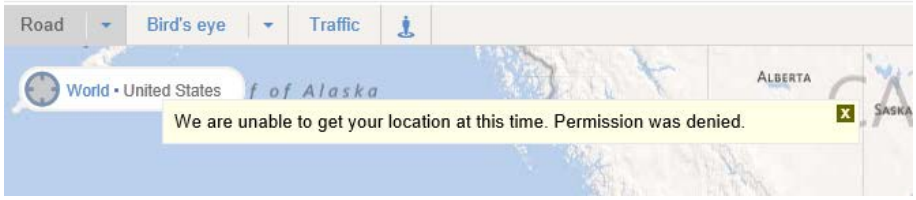


Figure 3-1 Use of brokered capabilities like geolocation from within a web context will generate an error.

Such capabilities are blocked because web content loaded into an `iframe` can easily provide the

[17] If the color scheme looks odd, it's because the `iframe` is picking up styles from the default ui-dark.css of WinJS. Try changing that stylesheet to ui-light.css for something that looks more typical.

means to navigate to other arbitrary pages. From the Bing maps page used above, for example, a user can go to the Bing home page, do a search, and end up on any number of untrusted and potentially malicious pages. Whatever the case, those pages might request access to sensitive resources, and if they just generated the same user consent prompts as an app, users could be tricked into granting such access.

Fortunately, if you ask nicely, Windows will let you enable those capabilities for 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. Each rule 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 included within another rule.)

Such rules are created in the Content Uri tab of Visual Studio's manifest editor, as shown in Figure 3-2. Each rule needs to be the exact URI that might be making a request, *http://www.bing.com/maps/*. Once we add that rule (as in the completed ContentUri example for this chapter), Bing maps is allowed to use geolocation. When it does so, the standard Metro style prompt will appear (Figure 3-3), just as if the app had made the request.

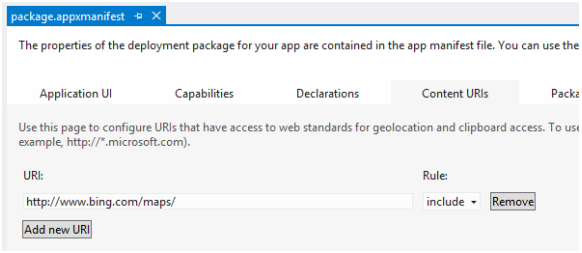


Figure 3-2 Adding a content URI to the app manifest; the contents of the text box is saved when the manifest is saved. Add New URI creates another set of controls in which to enter additional rules.

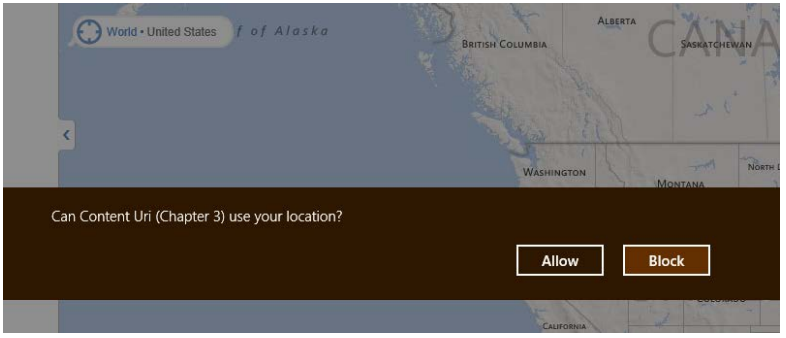


Figure 3-3 With an content URI rule in place, web content in an `iframe` acts like part of the app. This shows exactly why content URI rules are necessary to protect the user from pages unknown to the app that could otherwise trick the user into granting access to sensitive resources.

Referencing Content from App Data: ms-appdata

As we've seen, the `ms-appx[-web]:///` schema allow an app to navigate `iframe` elements to pages that exist inside the app package, or on the web. 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 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 off, the `file://` protocol is 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. Within the local context of an app, `ms-appdata` is a shortcut to the 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` (and similar forms with `roaming` and `temp`) wherever a URI can be used, including within a CSS style like `background`.

Unfortunately, the caveat—there always seems to be one with the app container!—is that `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.).

Can you do any kind of dynamic page generation, then? Well, yes: you need to load file contents and process them manually. You can get to your appdata folders through the `Windows.Storage.ApplicationData` API and go from there. To load and render a full HTML page would require that you patch up all external references and play some magic with script, but it can be done if you really want.

A similar question is whether you can generate and execute script on the fly. The answer is again qualified. Yes, you can take a JavaScript string and pass it to the `eval` or `execScript` functions. The inevitable caveat here is that automatic script filtering is applied to that code that prevents injection into the DOM via properties like `innerHTML` and `outerHTML`, and methods like `document.write` and `DOMParser.parseFromString`. Yet there are certainly situations where you, the developer, really know what you're doing and enjoy juggling flaming swords and running chainsaws and thus want to get around such restrictions, especially when using third-party libraries. (See the sidebar below.) Acknowledging that, Microsoft provides a mechanism to consciously circumvent all this: `MSApp.execUnsafeLocalFunction`. For all the details regarding this, refer to Developing secure apps, which covers this along with a few other obscure topics (like the `sandbox` attribute for `iframes`) that I'm not including here.

And curiously enough, WinJS actually makes it *easier* for you to juggle flaming swords and running chainsaws! `WinJS.Utilities.setInnerHTMLUnsafe`, `setOuterHTMLUnsafe`, and `insertAdjacentHTMLUnsafe` are wrappers for calling DOM methods that would otherwise strip out risky content.

All that said (don't you love being aware of the details?), let's look at an example of using

ms-appdata, which will probably be much more common in your app-building efforts.

Sidebar: Third-Party Libraries and the Hosted Environment

In general, Metro style 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 only being used from the web context. (WinJS, mind you, doesn't need bundling because it's provided by the Windows Store—such "framework packages" are not enabled for third parties in Windows 8.)

Second, DOM API changes and app container restrictions might affect the library. For example, library functions using `window.alert` won't work. One library also cannot load another library from a remote source in the local context. Most importantly, anything in the library that assumes a higher level of trust than the app container provides, such as assuming open file system access, will have issues.

The most common issue comes up when libraries inject elements or script into the DOM (as through `innerHTML`), a widespread practice for web apps that is not generally allowed within the app container. For example, trying to create a jQuery datepicker widget (`$("myCalendar").datepicker()`) will hurl out this kind of error. You can get around this on the app level by wrapping the code above with `MSApp.execUnsafeLocalFunction`, but that doesn't solve injections coming from deeper inside the library. In the jQuery example given here, the control can be created but clicking a date in that control generates another error.

In short, you're free to use third-party libraries so long as you're aware that they were generally written with assumptions that don't always apply within the app container. Over time, of course, fully Windows 8–compatible versions of such libraries will emerge.

Here My Am! with ms-appdata

OK! Having endured seven pages of esoterica, let's play with some real code and return to the Here My Am! app we wrote in Chapter 2. Here My Am! used the convenient `URL.createObjectURL` method to display a picture taken through the camera capture UI in an `img` element:

```
captureUI.captureFileAsync(Windows.Media.Capture.CameraCaptureUIMode.photo)
    .done(function (capturedFile) {
        if (capturedFile) {
            that.src = URL.createObjectURL(capturedFile);
        }
    });
```

This is all well and good, if we just take it on faith that the picture is stored somewhere—we don't really care so long as we get a URL. Truth is, pictures (and video) from the camera capture API are just stored in a temp file; 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-CH3-HereMyAm3a_5xchamk3agtd6\TempState\picture001.png*. Egads. Not the friendliest of locations, and definitely not one that we'd want a typical consumer to ever see!

With an app like this, let's copy that temp file to a more manageable location, which could, for example, allow the user to select from previously captured pictures. We'll make a copy in the app's local appdata folder and use `ms-appdata` to set the `img src` to that location. Let's start with the call to `captureUI.captureFileAsync` as before:

```
//For use across chained promises
var capturedFile = null;

captureUI.captureFileAsync(Windows.Media.Capture.CameraCaptureUIMode.photo)
    .then(function (capturedFileTemp) {
        //Be sure to check validity of the item returned; could be null if the user canceled.
        if (!capturedFileTemp) { throw ("no file captured"); }
```

Notice that instead of calling `done` to get the results of the promise, we're using `then` instead. This is because we need to chain a number of async operations together and `then` allows errors to propagate through the chain, as we'll see in the next section. In any case, once we get a result in `capturedFileTemp` (which is in a gnarly-looking folder), we then open or create a "HereMyAm" folder within our local appdata. This happens via `Windows.Storage.ApplicationData.current.localFolder`, which gives us a `Windows.Storage.StorageFolder` object that provides a `createFolderAsync` method:

```
        //As a demonstration of ms-appdata usage, copy the StorageFile to a folder called HereMyAm
        //in the appdata/local folder, and use ms-appdata to point to that.
        var local = Windows.Storage.ApplicationData.current.localFolder;
        capturedFile = capturedFileTemp;
        return local.createFolderAsync("HereMyAm",
            Windows.Storage.CreationCollisionOption.openIfExists);
    })
    .then(function (myFolder) {
        //Again, check validity of the result operations
        if (!myFolder) { throw ("could not create local appdata folder"); }
```

Assuming the folder is created successfully, `myFolder` will contain another `StorageFile` object. We then use this as a target parameter for the temp file's `copyAsync` method, which also takes a new filename as its second parameter. For that name we'll just use the original name with the date/time appended (replacing colons with hypens to make a valid filename):

```
        //Append file creation time (should avoid collisions, but need to convert colons)
        var newName = capturedFile.displayName + " - "
            + capturedFile.dateCreated.toString().replace(/:/g, "-") + capturedFile.fileType;
        return capturedFile.copyAsync(myFolder, newName);
    })
    .done(function (newFile) {
        if (!newFile) { throw ("could not copy file"); }
```

Because this was the last async operating in the chain, we use the promise's `done` method for

reasons we'll again see in a moment. In any case, if the copy succeeded, `newFile` contains a `StorageFile` object for the copy, and we can point to that using an `ms-appdata` URL:

```
        lastCapture = newFile;  //Save for Share
        that.src = "ms-appdata:///local/HereMyAm/" + newFile.name;
    },
    function (error) {
        console.log(error.message);
    });
```

The completed code is in the HereMyAm3a example.

Of course, we could still use `URL.createObjectURL` with `newFile` as before (making sure to provide the `{ oneTimeOnly=true }` parameter to avoid memory leaks). While that would defeat the purpose of this exercise, it works perfectly (and the memory overhead is essentially the same since the picture has to be loaded either way). In fact, we'd need to use it if we copy images to the user's pictures library instead. To do this, just replace `Windows.Storage.ApplicationData.current.localFolder` with `Windows.Storage.KnownFolders.picturesLibrary` and declare the Pictures library capability in the manifest. Both APIs give us a `StorageFolder`, so the rest of the code is the same except that we'd use `URL.createObjectURL` because we can neither use `ms-appdata://` nor `file://` to refer to the pictures library. The HereMyAm3a example contains this code in comments.

Sequential Async Operations: Chaining Promises

In the previous code example, you might have noticed how we throw exceptions whenever we don't get a good result back from any given async operation. Furthermore, we have only a single error handler at the end, and there's this odd construct of returning the result (a promise) from each subsequent async operation instead of just processing the promise then and there.

Though it may look odd at first, this is actually 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, with each promise fulfilled with `done`. Here's how the async calls in previous code would be placed with this approach (extraneous code removed for simplicity):

```
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 significantly more difficult. When promises are nested, error handling must be done at each level; 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.

Chaining promises solves all of this with the small tradeoff of needing to declare a few extra temp variables outside the chain. With chaining, you `return` the promise out of each completed handler (rather than calling the next async function and tagging on a `.done`). This allows you to indent all the async calls at the same level, and it also has the effect of propagating errors down the chain. When an error happens within a promise, you see, what comes back is still a promise object, and if you call its `then` method (but not `done`—see the next section), it will again return *another* promise object with an error. As a result, any error along the chain will quickly propagate through to the first available error handler, thereby allowing 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 a `done(null, errorHandler)` call, replacing the previous `done` with `then`:

```
captureUI.captureFileAsync(Windows.Media.Capture.CameraCaptureUIMode.photo)
    //...
    .then(function (newFile) {
    })
    .done(null, function (error) {
    })
})
```

Finally, a word about debugging chained promises (or nested ones, for that matter). Each step involves an async operation, so you can't just step through as you would with synchronous code (otherwise you'll end up deep inside WinJS). Instead, set a breakpoint on the first line within each completed handler and on the first line of the error function at the end. 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 which you'll hit the breakpoint in the next completed handler or you'll hit the breakpoint in the error handler.

Error Handling Within Promises: then vs. done

Although it's common to handle errors at the end of a chain of promises, as demonstrated in the code above, you can still provide an error handler at any point in the chain—`then` and `done` both take the same arguments. If an exception occurs at that level, it will surface in the innermost error handler.

This brings us to the difference between `then` and `done`. First, `then` returns another promise, thereby allowing chaining, whereas `done` returns `undefined` (so it's always at the end of the 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`. 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 the `WinJS.Application.onerror` or `window.onerror` events. (The latter will get the error if the former doesn't handle it.) If you don't, the app will be instantly terminated and an error report sent to the Windows Store dashboard. We actually recommend that you provide a `WinJS.Application.onerror` handler for this reason.

In practical terms, 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`, always use `done` at the end of a chain even for a single async operation.

There is much more you can do with promises, by the way, like combining them, canceling them, and so forth. We'll come back to all this at the end of this chapter.

Debug Output, Error Reports, and the Event Viewer

Speaking of exceptions and error handling, it's sometimes heartbreaking to developers that `window.prompt` and `window.alert` are not available to Metro style 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 to use `console.log`, as shown earlier, 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.[18]

Another DOM API function to which you might be accustomed is `window.close`. You can still use this as a development tool, but Windows interprets this call in released apps 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, Metro style apps should not provide their own close affordances.)

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 `MSApp.terminateApp` because it allows you to also include information as to the exact nature of the error that shows 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.[19] This is where error reports, console logging, and unhandled exceptions (which again terminate the app without warning) can be recorded.

To enable this, you need to do a couple steps. First navigate to and expand Microsoft/Windows/AppHost, select and right-click Admin, and then select View -> Show Analytic And Debug Logs for full output, as shown in Figure 3-4. This will enable tracing for errors and exceptions. Then right-click AppTracing (also under AppHost) and select *Enable Log*. This will trace your calls to `console.log` as well as other diagnostic information coming from the app host.

---

[18] For readers who are seriously into logging, beyond the kind you do with chainsaws, check out the `WinJS.Utilities` functions `startLog`, `stopLog`, and `formatLog`, which provide additional functionality on top of `console.log`. I'll leave you to commune with the documentation for these but wanted to bring them to your awareness.

[19] If you can't find Event Viewer, press the Windows key to go to the Start page, 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 page.

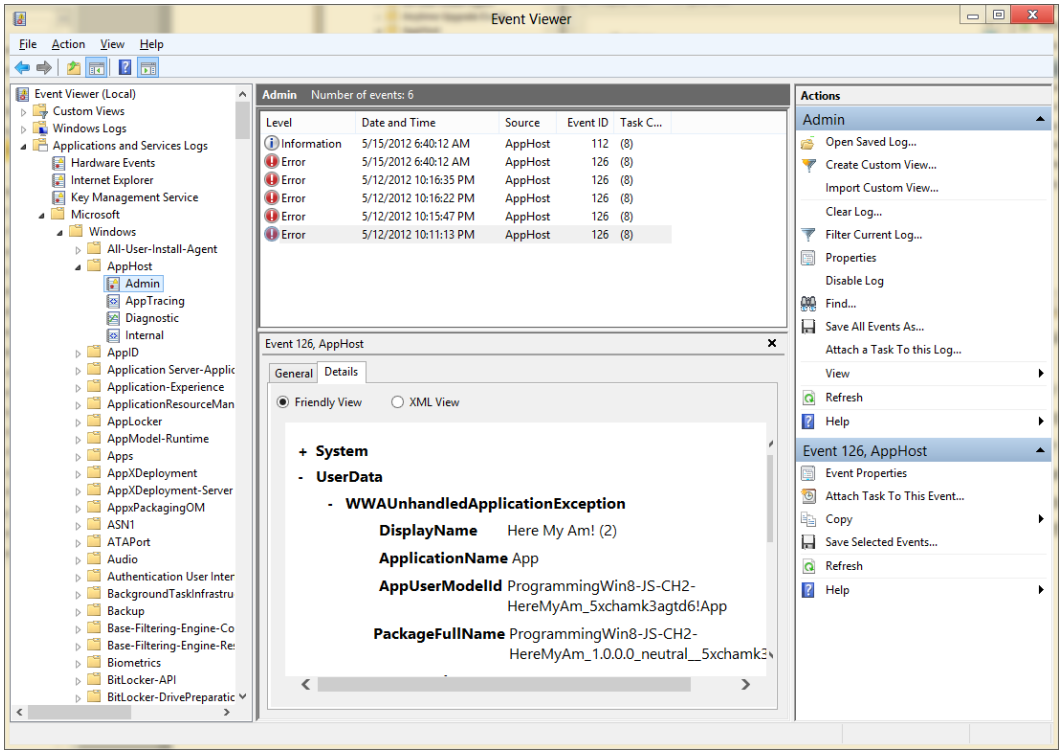


Figure 3-4 App host events, such as unhandled exceptions and load errors, can be found in Event Viewer.

We already introduced the 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 (Figure 3-5) whenever an exception is thrown, regardless of whether it's handled and before reaching any of your error handlers. If you have error handlers, you can safely click the Continue button in the dialog, and you'll eventually see the exception surface in those error handlers. (Otherwise the app will terminate.) If you click Break instead, you can find the exception details in the debugger's Locals pane, as shown in Figure 3-6.

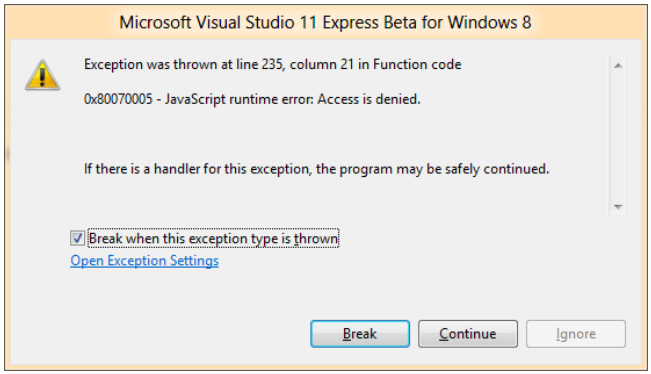


Figure 3-5 Visual Studio's exception dialog. As the dialog indicates, it's safe to press Continue if you have an error handler in the app; otherwise the app will terminate. Note that the checkbox in this dialog is a shortcut to toggle the Thrown checkbox for this exception type in the Exceptions dialog.

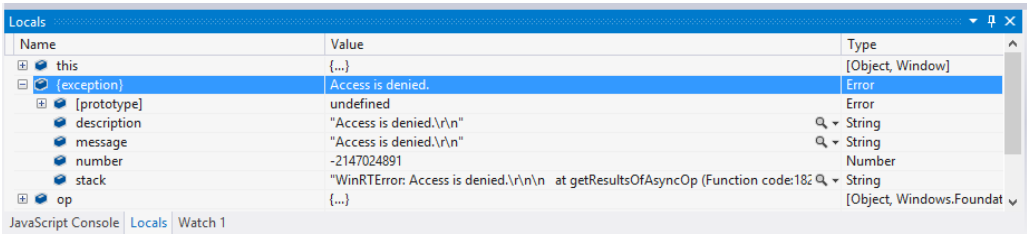


Figure 3-6 Information in Visual Studio's Locals pane when you Break on an exception.

The User un-handled 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! Still, you can try it as a test, and then leave checks only for 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 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.

App Activation

First, let me congratulate you for coming this far into a very detailed chapter! As a reward, let's talk about something much more tangible and engaging: the actual activation of an app and its startup sequence, something that can happen a variety of ways, such as via the Start screen tile, contracts, and file type and protocol associations. In all these activation cases, you'll be writing plenty of code to initialize your data structures, reload previously saved state, and do everything to establish a great experience for your users.

Branding Your App 101: The Splash Screen and Other Visuals

With activation, we actually need to take a step back even *before* the app host gets loaded, back to the moment a user taps your tile on the Start screen or when your app is launched through a contract or other association. The very first thing that happens, before any app-specific code is loaded or run, is that Windows displays a splash screen composed of the image and background color you provide in your manifest.

The splash screen—which shows for at least 0.75 seconds so that it's not just a flash—gives users

something interesting to look at briefly while the app gets started (much better than an hourglass). It also occupies the whole view where the app is being launched (which might be the filled view state or the overlay area from the share or search charm), 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 is ready with its first page, the system removes the splash screen.

The splash screen, along with your app tile, is clearly one of the most important ways to uniquely brand your app, so make sure that you and your graphic artist(s) give full attention to these. There are additional graphics and settings in the manifest that 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. Thus, take a solemn vow right now that you truly, truly, cross-your-heart will *not* upload an app to the Windows Store with those defaults still in place! (For additional guidance, see Guidelines and checklist for splash screens.)

You can see that the table lists multiple sizes for various images specified in the manifest to accommodate varying pixel densities: 100%, 140%, and 180% scale factors, and even a few at 80% (don't neglect the latter: they are typically used for most desktop monitors). So while you can just provide a single 100% scale image for each of these, it's almost guaranteed that scaled-up versions of that graphic are going to look bad. So why not make your app look its best? Take the time to create each individual graphic consciously.

Manifest Tab	Section	Item	Use	Image Sizes 100%	140%	180%
Packaging	n/a	Logo	Tile/logo image used for the app on its Product Description Page in the Windows Store.	50x50	70x70	90x90
Application UI	n/a	Display Name	Appears in "all apps" view on the Start screen, search results, the Settings charm, and in the Store.	n/a	n/a	n/a
	Tile	Logo	Single-wide tile image	150x150 (+ 80% scale at 120x120)	210x210	270x270
		Wide logo (Optional)	Double-wide tile image. If provided, this is shown as the default, but user can use the single-wide tile if desired.	310x150 (+80% scale at 248x120)	434x210	558x270
		Small 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. Also used on the app tile if you elect to show a logo instead of the app name in the lower left cover of the tile.	30x30 (+80% scale at 24x24)	42x42	54x54
		Show name	Specifies whether to show the app name on your app tile (both, neither, or the single- or double-wide specifically). Set this to "no logo" if your tile images	n/a	n/a	n/a

			includes your app name.			
		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 single-wide tile	n/a	n/a	n/a
		Fore-ground 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 ration between this and the background color	n/a	n/a	n/a
		Back-ground color	Color that will be shown for transparent areas of any tile images, and buttons in app dialogs. Also provides the splash screen background color unless that is set separately.	n/a	n/a	n/a
	Notifi-cations	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
	Splash screen	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 designed.	620x300	868x420	1116x540
		Back-ground color	Color that will fill the majority of the splash screen; if not set, the Tile Background color value is used.	n/a	n/a	n/a

In the table, note that 80% scale tile graphics are used in specific cases like low DPI modes and should be provided with other scaled images. Note also that there are additional graphics besides the Packaging Logo (first item in the table) that you'll need when uploading an app to the Windows Store. See the App images topic in the docs under "Promotional images" for full details.

When saving these files, append *.scale-80*, *.scale-100*, *.scale-140*, and *.scale-180* to the filenames, before the file extension, as in *splashscreen.scale-140.png*. 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 scaled variant. Otherwise it looks for one without the suffix. No code needed! This is demonstrated in the HereMyAm3b example, where I've added all the various branded graphics (with some additional text in each graphic to show the scale). To test these different graphics, use the set resolution/scaling button in the simulator—refer back to Figure 2-5—to choose different pixel densities on a 10.6" screen (1366 x 768 =100%, 1920 x 1080 = 140%, and 2560 x 1440 = 180%). 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.

One thing you might also notice is that full-color photographic images, as I'm using in HereMyAm3b here, don't scale down very well to the smallest sizes (Store logo and small logo). This is one reason why such logos are typically simpler with Metro style design, so hopefully your designers do a better job than I have!

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 app sees in a browser. Actually, it's more rather than less! When you launch an app from its tile, here's the process as Windows sees it:

5. Windows displays a splash screen using information from the app manifest.
6. Windows launches the app host, identifying the app to launch.
7. The app host retrieves the app's Start Page setting (see the Application UI tab in the manifest editor), which identifies the HTML page to load.
8. The app host loads that page along with referenced stylesheets and script (deferring script loading if indicated in the markup). Here it's important that all files are properly encoded for best startup performance. (See the sidebar below.)
9. `document.DOMContentLoaded` fires. You can use this to do further initialization specifically related to the DOM, if desired (not common).
10. `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.
11. The splash screen is hidden once the activated event handler returns (unless the app has requested a deferral, as discussed later on).
12. `body.onload` fires. This is typically not used in Metro style apps, though it might be utilized by imported code or third party libraries.

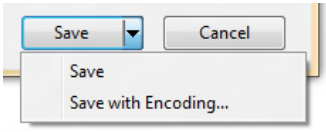
What's also very different is that an app can again be activated for many different purposes, such as contracts and associations, even while it's already running. As we'll see in later chapters, the specific page that gets loaded (step 3) can vary by contract, and if a particular page is already running,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

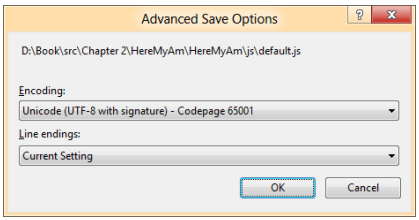
For the time being, though, let's concentrate on how we work with this core launch process, and because you'll generally do your initialization work within the `activated` event, let's examine that structure more closely.

Sidebar: File Encoding for Best Startup Performance

To optimize bytecode generation when parsing HTML, CSS, and JavaScript, 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, check this encoding: in Visual Studio's File Save As dialog (Blend doesn't have this at present), 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.





Activation Code Paths

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

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

    var app = WinJS.Application;
    var activation = Windows.ApplicationModel.Activation;
    WinJS.strictProcessing();

    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 essential code structure:

- `(function () { … })();` is again the JavaScript module pattern.
- `"use strict"` instructs the JavaScript interpreter to apply Strict Mode, a feature of

ECMAScript 5, as described at http://msdn.microsoft.com/en-us/library/br230269(v=VS.94).aspx. 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 fully qualified namespaces. This is a common practice to simplify multiple references to the same part of WinJS or WinRT.
- `WinJS.strictProcessing();` activates stricter processing rules for `WinJS.UI.processAll` and `WinJS.Binding.processAll` and `WinJS.Resources.processAll` to avoid unintentional script injection. We'll look at this more in Chapter 4, "Controls, Control Styling, and Basic Data Binding."
- `app.onactivated = function (args) {…}` assigns a handler for the `WinJS.UI.onactivated` event, which is a wrapper for `Windows.UI.WebUI.WebUIApplication.onactivated`. 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 state.
  - `WinJS.UI.processAll` instantiates WinJS controls—that is, elements that contain a `data-win-control` attribute, as we'll cover in Chapter 4.
  - `args.setPromise` instructs Windows to wait until `WinJS.UI.processAll` is complete before removing the splash screen. (See "Activation Deferrals" later in this chapter.)
    - `app.oncheckpoint` gets an empty handler in the template; we'll cover this 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 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. Entirely optional, but very helpful.

With `start`, 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 own handlers. Second, when `WinJS.Application` receives activation events, it doesn't just pass them on to the app's handlers, 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 really all there is to it.

As the template code shows, apps typically do most of their initialization work within the `activated` event, but 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 WinJS.Application.onactivated, you'll see that the exact contents of `args.details` depends on specific *kind* of activation. All activations, however, share three common properties:

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

Additional properties provide relevant data for the activation. For example, `launch` provides the `tileId` and `arguments`, which are needed with secondary tiles. (See Chapter 13, "Tiles, Notifications, the Lock Screen, and Background Tasks"). The `search` kind (the next most commonly used) provides `queryText` and `language`, `protocol` provides a `uri`, and so on. We'll see how to use many of these in the proper context. 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 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 `WinJS.Application.start`):

- `activated` Queued in the local context for `Windows.UI.WebUI.WebUIApplication.onactivated`. 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`).
- `loaded` Queued for `DOMContentLoaded` in all contexts;[20] in the web context, will be queued prior to `activated`.
- `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` This tells the app when to save the 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 8, "State, Settings, Files, and Documents.")

With most of these events (except `error` and `settings`), the `args` you receive contains a method called `setPromise`. If you need to perform an async operation within an event handler (like an XmlHttpRequest), you can obtain the promise for that work and hand it off to `setPromise` instead of calling its `then` or `done` yourself. WinJS will then *not* process the next event in the queue until that promise is fulfilled. Now to be honest, there's no actual difference between this and just calling `done` on the promise yourself within the `loaded`, `ready`, and `unload` events. It *does* make a difference with `activated` and `checkpoint` (specifically the suspending case) because Windows will otherwise assume that you've done everything you need as soon as you return from the handler; more on this in the "Activation Deferrals" section. So, in general, if you have async work within these events handlers,

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

it's a good habit to use `setPromise`. Because `WinJS.UI.processAll` is itself an async operation, the templates wrap it with `setPromise` so that the splash screen isn't removed until WinJS controls have been fully instantiated.

Anyway, 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, as we'll see later on in this chapter and in Chapter 8.

The other bits are the `queueEvent` and `stop` methods. The `queueEvent` event drops an event into the queue that will get dispatched (after any existing queue is clear) to whatever listeners you've set up on the `WinJS.Application` object. Events are simply identified with a string, so you can queue an event with any name you like, and call `WinJS.Application.addEventListener` with that same name anywhere else in the app. This can be useful for centralizing 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.

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 simulate the right 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.

Extended Splash Screens

Now, though the default splash screen helps keep the user engaged, they won't stay engaged if that same splash screen stays up for a really long time. In fact, "a really long time" for the typical consumer amounts to all of 15 seconds, at which point they'll 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 their afternoon.

Thus Windows will only wait 15 seconds for your app to get through `app.start` and the `activated` event, at which point your home page should be rendered. Otherwise, boom! Windows automatically blows your app away and returns the user to the Start screen.

The first consideration, of course, is to optimize your startup process to be as quick as possible. Still, sometimes an app really needs more than 15 seconds to get going, especially on its first run. For example, an app package might include a bunch of compressed data when downloaded from the Store, which it needs to expand onto the local file system on first run so 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 to populate a local IndexDB from data in a JSON file or download and cache a bunch of data from an online service.

Such apps thus implement an *extended splash screen,* which is just a fancy term for some clever

fakery. Simply said, if the app determines that it needs more time, it hides its real home page behind another `div` that looks exactly like the system-provided splash screen but that is under the app's control so that it can display progress indicators or other custom UI while initialization continues.

In general, Microsoft recommends that the extended splash screen initially matches the system splash screen to avoid visual jumps. (See Guidelines and checklist for splash screens.) 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. A number of apps start with a replica of the system splash screen and then animate the graphic to one side to make room for other elements. Other apps fade out the initial graphic and start a video.

Making a smooth transition 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, which is a `Windows.Foundation.Rect` containing the placement and size of the splash screen image. Because your app can be run on a variety of different display sizes, this tells you where to place the same image on your own page, where to start an animation, and/or where to place things like messages and progress indicators 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 first page comes up. Typically, this is useful to trigger the start of on-page animations, starting video playback, and so on.

We won't have a need to implement an extended splash screen in this chapter's examples, but you can refer to the Splash Screen sample in the SDK. One more detail that's worth mentioning is that because an extended splash screen is just a page in your app, it can be placed into the various view states such as snap view. So, as with every other page in your app, make sure your extended splash screen handles those states!

Activation Deferrals

As mentioned earlier, once you return from the `activated` event, Windows assumes that you've done everything you need on startup. By default, then, Windows will remove its splash screen and make your home page visible. But what if you need to complete one or more async operations before that home page is really ready, such as completing `WinJS.UI.processAll`?

This, again, is what the `args.setPromise` method inside the `activated` event is for. If you give your async operation's promise to `setPromise`, Windows will wait until that promise is fulfilled before taking down the splash screen. The templates use this to keep the system splash screen up until `WinJS.UI.processAll` is complete.

As `setPromise` just waits for a single promise to complete, how do you handle multiple async operations? You can do this a couple of ways. First, if you need to control the sequencing of those operations, you can chain them together as we already know how to do—just be sure to `return` the

last promise in the chain instead of calling its `done` method! If the sequence isn't important but you need all of them to complete, you can combine those promises by using `WinJS.Promise.join`, returning the result to `setPromise`. If you need only one of the operations to complete, you can use `WinJS.Promise.any` instead—`join` and `any` are discussed in the last section of this chapter.

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`.

Now the `setPromise` method coming from WinJS is actually implemented using a more generic deferral mechanism from WinRT. The `args` given to `Windows.UI.WebUI.WebUIApplication.onactivated` (the WinRT event) contains a little method called `getDeferral` (technically `Windows.UI.WebUI.ActivatedOperation.getDeferral`). This function returns a deferral object that then contains a `complete` method, and Windows will leave the system splash screen up until you call that method (although this doesn't change the fact that users are impatient and your app is still subject to the 15-second limit!). The code looks like this:

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

//After initialization is complete
activatedDeferral.complete();
```

Of course, `setPromise` ultimately does exactly this, and if you add a handler for the WinRT event directly, you can use the deferral yourself.

App Lifecycle Transition Events and Session State

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 snapping 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. 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.[21]

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

A similar but different condition is *visibility*. An app can be visible but not have the focus, as when

[21] You might ask whether one can configure a Metro style app to essentially lock out others, as in a "kiosk mode" or "toddler mode." I don't have the answer: the story on such a feature is still TBD at the time of writing.

it's snapped. In such cases an app would continue things like animations or updating a feed, but it would stop such activities 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. (The event works for visibility of all other 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 pick these up 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, just layout and object visibility.) At any time, an app can also retrieve the current view state through `Windows.UI.ViewManagement.ApplicationView.value`. This returns one of the `Windows.UI.ViewManagement.ApplicationViewState` values: `snapped`, `filled`, `fullScreenLandscape`, and `fullScreenPortrait`; details in Chapter 6, "Layout."

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 terminated, so the typical DOM events like `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, as discussed before, but they can use `MSApp.terminateApp` to close because of unrecoverable conditions.

Suspend, Resume, and Terminate

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

- **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!).
- **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` because it's not commonly used and WinJS has no value to add.) We'll talk more about this in the "Data from Services and WinJS.xhr" section coming up soon, because the need for this event often arises when using services.
- **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.

It's very helpful to know that you can simulate these conditions in the Visual Studio debugger by using the toolbar drop-down shown in Figure 3-8. 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!)

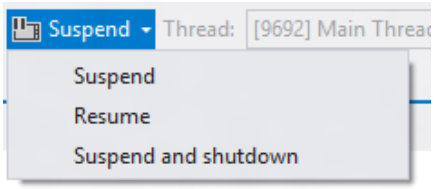


Figure 3-8 The Visual Studio toolbar drop-down to simulate suspend, resume, and terminate.

We've briefly listed those previous states before, but let's see how those 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 3-9 and the table below describes how the `previousExecutionState` values are determined.

Value of `previousExecutionState`	**Scenarios**
`notrunning`	First run after install from Store. First run after reboot or log off. App is launched within 10 seconds of being closed by user (about the time it takes to hide, suspend, and cleanly terminate the app; if the user relaunches quickly, Windows has to immediately terminate it without finishing the suspend operation). App was terminated in Task Manager while running or closes itself with `MSApp.terminateApp`.
`running`	App is *currently running* and then invoked in a way other than its app tile, such as Search, Share, secondary tiles, toast notifications, and all other contracts. When an app is running and the user taps the app tile, Windows just switches to the already-running app and without triggering activation events (though `focus` and `visibilitychange` will both be raised).
`suspended`	App is *suspended* and is invoked in a way other than the app tile (as above for `running`). In addition to focus/visibility events, the app will also receive the `resuming` event.
`terminated`	App was previously suspended and then terminated by Windows due to resource pressure. Note that this does not apply to `MSApp.terminateApp` because an app would have to be running to call that function.
`closedByUser`	App was closed by an uninterrupted close gesture (swipe down or Alt+F4). An "interrupted" close is when the user switches back to the app within 10 seconds, in which case the previous state will be `notrunning` instead.

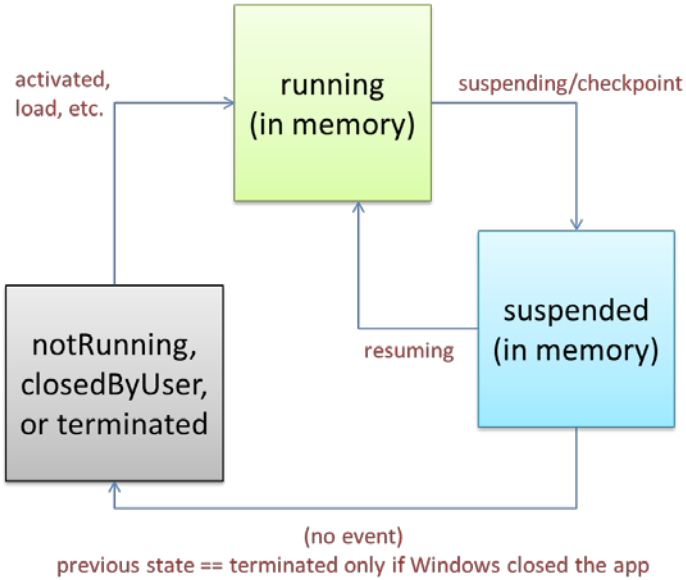


Figure 3-9 Process lifecycle events and previousExecutionState values.

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 settings (such as those in its Settings panel). 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 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.

The second requirement above is exactly why the templates provide a code structure for this case along with a `checkpoint` handler. We'll see the full details of saving and reloading state in Chapter 8. The basic idea is that an app should, when being suspended, save whatever transient session state it would need to rehydrate itself after being terminated (like form data, scroll positions, the navigation stack, and other variables). This is because although Windows might have suspended the app and dumped it from memory, *it's still running in the mind of the user*. Thus, when users activate the app

again for normal use (activation kind is `launch`, rather than through a contract), they expect that app to be right where it was before. When an app gets suspended, it must save whatever state is necessary to make this possible, and it must restore that state when activated under these conditions.

Be clear that if the user directly closes the app with Alt+F4 or the swipe-down gesture, the `suspending`/`checkpoint` events will also be raised, so the app still saves state. In these cases, however, the app will be automatically terminated after being suspended, and it won't be asked to reload that state when it's restarted because `previousExecutionState` will be `notRunning` or `closedByUser`.

The best practice is actually to save session state incrementally (as it changes) to minimize the work needed within the suspending event, because you have only five seconds to do it. Mind you, this session state does not include data that is persistent across sessions (like user files, high scores, and app settings) because an app would always reload or reapply such persistent data in each 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 8. 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 provides a single convenient place to save both session state and any other persistent data you might have.

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 this in place of other variables, so there's no need to copy variables into it separately. Within the `checkpoint` event, WinJS automatically serializes the contents of this object (using `JSON.stringify`) into a file within your local appdata folder. 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've used this object for storing variables, you only need to avoid settings those values back to their defaults when reloading your state.

Note that because the 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.

Third, if you don't want to use the `sessionState` object, 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.

A final aid ties into a deferral mechanism like the one for activation. The deferral is important because Windows will suspend your app as soon as you return from the suspending event, which could be less than five seconds. So, the event args for `WinJS.Application.oncheckpoint` provides a

`setPromise` method that ties into the underlying WinRT deferral. As before, 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.

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.

Well, hey! That sounds pretty good—is this perhaps a sneaky way to circumvent the restriction on running Metro style 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 how much time you have before Windows will forcibly suspend you regardless of any deferral. Once the first operation is complete, you can check the time and start another, and so on.

Basic Session State in Here My Am!

To demonstrate some basic state handling, I've made a few changes to Here My Am! as given in the HereMyAm3c example. 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 by prepending `app.sessionState.` to the name—in the completed handler for `getGeopositionAsync`, for example:

```
gl.getGeopositionAsync().done(function (position) {
    app.sessionState.lastPosition = {
        latitude: position.coordinate.latitude,
        longitude: position.coordinate.longitude
    };

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

Because we'll need to set the map location from here and from previously saved coordinates, I've moved that bit of code into a separate function that also makes sure 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 `app.sessionState` is initialized to an empty object by default, `{ }`, so `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 we stored `lastPosition` in `sessionState`, it will have been automatically saved in `WinJS.Application.checkpoint` when the app ran previously. When we restart from `terminated`, WinJS automatically reloads `sessionState`; if we'd saved a value there previously, it'll be there again and `updatePosition` just works.

You can test this by running the app with these changes and then using the *Suspend and shutdown* option on the Visual Studio toolbar. 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://` URL scheme to refer to it. When we capture an image, we just save that URL into `sessionState.imageURL` (the property name is arbitrary) at the end of the promise chain inside `capturePhoto`:

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

This value will also be reloaded when necessary during startup, so we can just initialize the `img src` accordingly:

```
if (app.sessionState.imageURL) {
    document.getElementById("photo").src = app.sessionState.imageURL;
}
```

This will initialize the image display from `sessionState`, but we also need to initialize `lastCapture` so that the same image is available through the Share contract. For this we need to also

save the full file path so we can re-obtain the `StorageFile` through `Windows.Storage.StorageFile.getFileFromPathAsync` (which doesn't work with `ms-appdata://` URLs). So, in `capturePhoto`:

```
app.sessionState.imagePath = newFile.path;
```

And during startup:

```
if (app.sessionState.imagePath) {
    Windows.Storage.StorageFile.getFileFromPathAsync(app.sessionState.imagePath)
        .done(function (file) {
            lastCapture = file;

            if (app.sessionState.imageURL) {
                document.getElementById("photo").src = app.sessionState.imageURL;
            }
        });
```

I've placed the code to set the `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.

Data from Services and WinJS.xhr

Though we've seen examples of using data from an app's package (via URLs or `Windows.ApplicationModel.Package.current.installedLocation`) as well as in appdata, it's very likely that your app will incorporate data from a web service and possibly send data to services as well. For this, the most common method is to employ XmlHttpRequest. You can use this in its raw (async) form, if you like, or you can save yourself a whole lot of trouble by using the `WinJS.xhr` function, which conveniently wraps the whole business inside a promise.

Making the call is quite easy, as demonstrated in the SimpleXhr example for this chapter. Here we use WinJS.xhr to retrieve the RSS feed from the Windows 8 app developer blog:

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

That is, give `WinJS.xhr` a URL and it gives back a promise that delivers its results to your completed function (in this case `processPosts`) and will even call a progress function. With the former, the results 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 extracting 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 3-10. You can look at the code to see the details.[22]

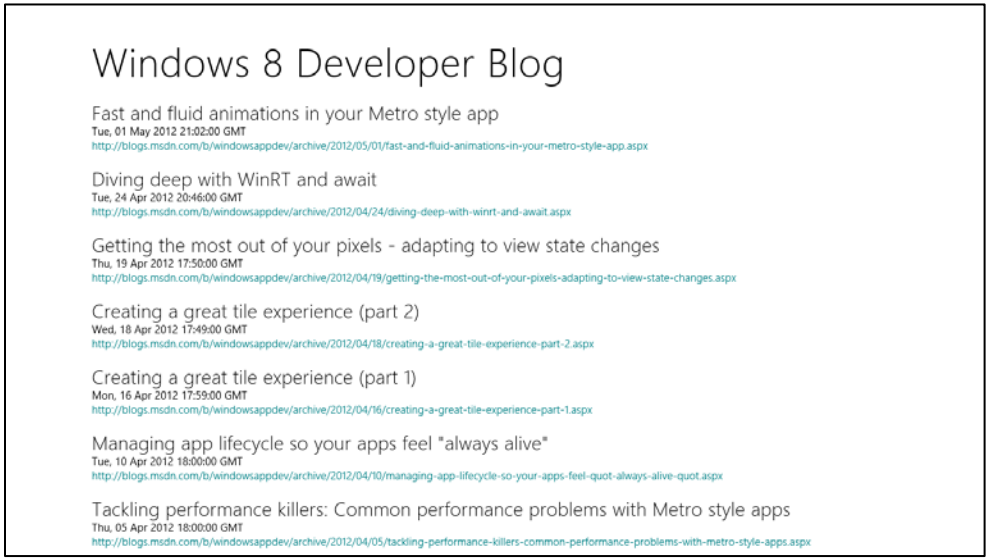


Figure 3-10 The output of the SimpleXhr app.

If you try this app, it's clear that it can use more work, and we'll use it as a basis to demonstrate additional features both in this chapter and in later ones. (You might also be interested in the tutorial called How to create a mashup in the docs.) For the moment, what concerns is not so much the mechanics of talking to services but 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 or whether

[22] It's worth mentioning that WinRT has a specific API for dealing with RSS feeds in `Windows.Web.Syndication`. 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 Metro style apps written in other languages that don't have the same built-in features as JavaScript.

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. With the Windows 8 app developer blog, we can see that new blog posts don't show up more than once a day, so refreshing only if an hour or more has passed will be entirely sufficient to keep up-to-date and to catch new posts within a reasonable timeframe.

This is implemented in SimpleXhr by first placing the `WinJS.xhr` 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 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 below 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 (the pause icon shown in Figure 3-8), 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.

It's worth mentioning here that you can use HTML5 mechanisms like `localStorage`, `indexedDB`, and the app cache for caching purposes; data for these is stored within your local appdata automatically. And speaking of databases, you may be wondering what's available other than IndexedDB for Metro style apps. One option is SQLite, as described in Using SQLite in a Metro Style App (on the blog of Tim Heuer, one of the Windows 8 engineers). You can also use the OData Library for JavaScript that's available from http://www.odata.org/libraries. It's one of the easiest ways to communicate with an online SQL Server database (or any other with an OData service), because it just uses XmlHttpRequest under the covers.

Handling Network Connectivity (in Brief)

We'll be covering network matters in Chapter 14, "Networking," but there's one important aspect that you should be aware of here. What does an app do with changes to network connectivity, such as disconnection, reconnection, and changes in bandwidth or cost (such as roaming into another provider area)?

The `Windows.Networking.Connectivity` APIs supply the details. There are three main ways to respond to such events:

- First, have a great offline story for when connectivity is lost: cache important data, queue work to be done later, and continue to provide as much functionality as you can without a connection. Clearly this is closely related to your overall state management strategy. For example, if network connectivity was lost while you were suspended, you might not be able to refresh your data at all, so be prepared for that circumstance!
- Second, listen for network changes to know when connectivity is restored, and then process your queues, recache data, and so forth.
- Third, listen for network changes to be cost-aware on metered networks. The Windows Store certification requirements, in fact, have a policy on protecting consumers from "bill shock" caused by excessive data usage on such networks. The last thing you want, to be sure, are negative reviews in the Store on issues like this.

On a simpler note, be sure to test your apps 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 map.html didn't bother to check whether the remote script for Bing Maps had actually been downloaded. Now it checks whether the `Microsoft` namespace (for the `Microsoft.Maps.Map` constructor) is valid. In SimpleXhr too, I made sure to provide an error handler to the `WinJS.xhr` promise so that I could at

least display a simple message.

Tips and Tricks for WinJS.xhr

Without opening the whole can of worms that is XmlHttpRequest, it's useful here to look at just a couple of additional points around `WinJS.xhr`.

First, notice that the single argument to this function 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.

Second 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, as we'll see in the "Completing the Promises Story" section at the end of this chapter, you can use the `WinJS.Promise.timeout` function, which allows you to set a timeout period after which the `WinJS.xhr` promise (and the async operation behind it) will be canceled.

You might have need to wrap `WinJS.xhr` in another promise, something that we'll also see at the end of this chapter. You could do this to encapsulate other intermediate processing with the XHR call 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 XHR calls together, you can use `WinJS.Promise.join`, which we'll again see later on.

We also saw how to process transferred bytes within the progress handler. You can use other data in the response and request as well. For example, the event args object contains a `readyState` property.

For release apps, using XHR with `localhost:` URL's (local loopback) is blocked by design. During development, however, when this is very useful, for instance, to debug a service without deploying it, you can enable this in Visual Studio by opening the project properties dialog (Project menu -> <project> options…), selecting Debugging on the left side, and setting Allow Local Network Loopback to true.

Finally, it's helpful to know that for security reasons cookies are automatically stripped out of XHR responses coming into the local context. One workaround to this is to make XHR calls from a web context `iframe` (in which you can use `WinJS.xhr`) and then to extract the cookie information you need and pass it to the local context via `postMessage`. Alternately, you might be able to solve the problem on the service side, such as implementing an API there that will directly provide the information you're trying to extract from the cookies in the first place.

For all other details on this function, refer to the `WinJS.xhr` function documentation and its links

to associated tutorials.

Page Controls and Navigation

Now we come to an aspect of Metro style apps that very much separates them from typical web applications. In 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 when there is, there are well-established mechanisms for doing so, such as HTML5 `sessionStorage` and `localStorage` (which work just fine in Metro style apps).

This type of navigation presents a few problems for Metro style apps, however. For one, navigating to a wholly 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 switching pages with a hyperlink. Users of web apps are accustomed to waiting a bit for a browser to acquire a new page (I've found many things to do with 15-second intervals!), but this isn't an appropriate user experience for a fast and fluid Metro style 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, Metro style apps typically implement "pages" by dynamically replacing sections of the DOM wholly within the context of a single page like default.html (which is akin to how AJAX-based apps 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 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.[23] They are, in fact, implemented like other controls in WinJS (as we'll see in Chapter 4), so you can declare them in markup, instantiate them with `WinJS.UI.process[All]`, use as many of them within a single host page as you like, and even nest them.

These APIs provide *only* the means to load and unload individual pages—they pull HTML in from other files (along with referenced CSS and JS) and attach the contents to an element in the DOM. That's it. To actually implement a page-to-page navigation structure, we need two additional pieces: something that manages a navigation stack and something that hooks navigation events to the page-loading mechanism of `WinJS.UI.Pages`.

For the first piece, you can turn to `WinJS.Navigation`, which through about 150 lines of CS101-level code supplies a basic navigation stack. This is all it does. The stack itself is just a list of URLs 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`).[24]

For the second piece, you can create your own linkage between `WinJS.Navigation` and `WinJS.UI.Pages` however you like. In fact, in the early stages of app development of Windows 8, even prior to the first public developer preview releases, people ended up writing just about the same boilerplate code over and over. In response, the team at Microsoft responsible for the templates magnanimously decided to supply a standard implementation that also adds some keyboard handling (for forward/back) and some convenience wrappers for layout matters. Hooray!

This piece is called the `PageControlNavigator`. Because it's just a piece of template-supplied code and not part of WinJS, it's entirely under your control, so you can tweak, hack, or lobotomize it however you want.[25] In any case, because it's likely that you'll often use the `PageControlNavigator` in your own apps, let's look at how it all works in the context of the Navigation App template.

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

---

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

[24] 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 returns `true`.

[25] The Quickstart: using single-page navigation topic also shows a clever way to hijack HTML hyperlinks and hook them into `WinJS.Navigation.navigate`. This can be a useful tool, especially if you're importing code from a web app.

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".
- **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 such markup, script, and style files.
- **js/navigator.js** Contains the implementation of the PageControlNavigator class.

To build upon this structure, add additional pages by using a page control template. I recommend first creating a new folder for the page under *pages*. 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` control. 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. This 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 PageControl from the item template:

```
<!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 class="win-backbutton" aria-label="Back" disabled></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 back button, which the `PageControlNavigator` automatically wires up for keyboard, mouse, and touch events. (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 help you edit a page control in Blend.)

The definition of the actual page control is in 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 relative URL (the page control identifier), 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 URL will simply add members to an existing definition (replacing those that already exist).

For a page created with the Page Control item template, you get a couple more methods in the structure (some comments omitted):

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

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

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

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

It's good to note that once you've defined a page control in this way, you can instantiate it from JavaScript with `new` by first obtaining its constructor function from `WinJS.UI.Pages.get(<page_url>)` and then calling that constructor with the parent element and an object containing its options.

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:

PageControl Method	When Called
`init`	Before elements from the page control have been copied into the DOM.
`processed`	After `WinJS.UI.processAll` is compete (that is, controls in the page have been instantiated, which is done automatically), but before page content itself has been added to the DOM.
`ready`	After the page have been added to the DOM.
`error`	If an error occurs in loading or rendering the page.
`unload`	Navigation has left the page.
`updateLayout`	In response to the `window.onresize` event, which signals changes between landscape, fill, snap, and portrait view states.

Note that `WinJS.UI.Pages` calls the first four methods; `unload` and `updateLayout`, 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 control (e.g., populate lists), wire up other page-specific event handlers, and so on. The `updateLayout` method is important when you need to adapt your page layout to new conditions, such as changing the layout of a ListView control (as we'll see in Chapter 5, "Collections and Collection Controls").

As for the `PageControlNavigator` itself, the code in navigator.js shows how it's defined and how it wires up a few 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;
                nav.onnavigated = this._navigated.bind(this);
                window.onresize = this._resized.bind(this);
```

```
                document.body.onkeyup = this._keyupHandler.bind(this);
                document.body.onkeypress = this._keypressHandler.bind(this);
                document.body.onmspointerup = this._mspointerupHandler.bind(this);
            }, {
    //...
```

First we see the definition of the `Application` namespace as a container for the `PageControlNavigator` class. Its constructor receives the `element` that contains it (the *contenthost* `div` in default.html) and the `options` declared with `data-win-options` in that element. This control creates another `div` for itself, appends that to its parent, adds a listener for the `WinJS.Navigation.onnavigated` event, and sets up its other listeners. Then it waits for someone to call `WinJS.Navigation.navigate`, which happens in the `activated` handler of default.js, to navigate to either the home page or the last page viewed if previous session state was reloaded. When that happens, the PageControlNavigator's `_navigated` 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:

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

    args.detail.setPromise(
        WinJS.Promise.timeout().then(function () {
            if (that.pageElement.winControl && that.pageElement.winControl.unload) {
                that.pageElement.winControl.unload();
            }
            return WinJS.UI.Pages.render(args.detail.location, newElement,
                args.detail.state, parented);
        }).then(function parentElement(control) {
            that.element.appendChild(newElement);
            that.element.removeChild(that.pageElement);
            that.navigated();
            parentedComplete();
        })
    );
},
```

One final important point here is that in the page control's JavaScript code, `document` will refer to that page control's contents, not to the content host in default.html.

And that, my friends, is how it works! As a concrete example of doing this in a real app, the code in the HereMyAm3d sample has been converted to use this model for its single home page. To make this conversion. I started with a new project using the Navigation App template to get the page navigation structures set up. Then I copied or imported the relevant code and resources from HereMyAm3c, primarily into the 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 this project in Blend. You'll first see that everything shows up in default.html, but you

can also open home.html and edit just that page.

You should note that WinJS calls `WinJS.UI.processAll` in the process of loading a page control, so we don't need to concern ourselves with that detail. 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 and 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 we simply write another flag into `app.sessionState` called `initFromState`. We always set this flag on startup, so any value that might be persisted between sessions is irrelevant.

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 use a module—that is, `(function() { ... })()`—to define things and then you use a namespace to export selective bits that are referenced through the namespace. In short, use a namespace anytime you're tempted to add any global objects or functions!

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 (and 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 through a namespace. Then you can use `new <namespace>.<class>` anywhere in the app.

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 page (e.g., default.html). With the `PageControlNavigator` instantiated and a page control defined via `WinJS.UI.Pages`, simply call `WinJS.Navigation.navigate` with the relative URI of that page control (its identifier). This loads that page and adds it to the DOM inside the element to which the `PageControlNavigator` is attached. This makes that page visible, thereby "navigating" to it so far as the user is concerned. You can also use the other methods of `WinJS.Navigation` to move forward and back in the nav stack, with its `canGoBack` and `canGoForward` properties allowing you to enable/disable navigation controls. 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 5) with a bunch of default items.
- Tapping a group header in the list navigates to section page (**pages/groupDetail**). This is done in groupedItems.html line 21, where the `click` event calls `WinJS.Navigation.navigate("/pages/groupDetail/groupDetail.html")` with an options argument identifying the specific group to display. That argument comes into the `ready` function of groupDetail.js.
- Tapping an item on the hub page goes to detail page (**pages/itemDetail**). The `itemInvoked` handler for the items—see groupedItems.js lines 27–37—calls `WinJS.Navigation.navigate("/pages/itemDetail/itemDetail.html")` with an options argument identifying the specific item to display. As with groups, that argument comes into the `ready` function of itemDetail.js.
- Tapping an item in the section page also goes to the details page through the same mechanism—see groupDetail.js lines 25–28.
- The back buttons on all pages are wired into `WinJS.Navigation.back` by virtue of code in the `PageControlNavigator`.

For what it's worth, the Split App template works similarly, where each list item on the items page (pages/items) is wired to navigate to pages/split when invoked.

In any case, the Grid App template also serves as an example of what we call the *Hub-Section-Detail*

navigation style. 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 5, 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 from the top edge). When using page controls and `PageControlNavigator`, navigation controls can just invoke `WinJS.Naviation.navigate` for this purpose. Note that in this style, there typically is no back button.

These styles, along with many other UI aspects of navigation, can be found on Navigation design for Metro style apps. This is an essential topic for Metro style app 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 that appear in an app. 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.

Typically, such pages appear only the first time the app is run. If the user provides a valid login, those credentials can be saved for later use via the `Windows.Security.Credentials` 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. (Note: UX guidance on this should be forthcoming but was not available at the time of writing.)

In both cases, you would typically point to such pages in default.html as the home page. In the `init` or `processed` methods of the page control, then, which are fired before the page is added to the DOM, check to see if it's not actually necessary to show the page. If that's the case, just call `WinJS.Navigation.navigate` to switch over to what will then be the first visible page.

Optimizing Page Switching: Show-and-Hide

Even with page controls, there is still lots 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 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 the list and individual items. In this case, page controls might not be the best design.

You could use a split (master-detail) view, of course, but that means splitting the screen real estate. An alternative, then, is to actually keep the list 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 (see `WinJS.UI.Pages.render`) into another `div` that occupies the whole screen and overlays the list, and then make that `div` visible. When you dismiss the details page, just hide the div and set `innerHTML` to `""`. 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.

The `PageControlNavigator` is provided by the templates as part of your app, and you can modify it however you like to provide this kind of capability in a more structured manner.

Completing the Promises Story

Whew! We've taken a long ride in this chapter through many, many fine details of how apps are built and how they run (or don't run!). One consistent theme you may have noticed is that of promises—they've come up in just about every section! Indeed, async abounds within both WinJS and WinRT, and thus so do promises.

I wanted to close this chapter, then, by flushing out the story of promises, for they provide richer functionality than we've utilized so far. (If you want the fuller *async* story, read Keeping apps fast and fluid with asynchrony in the Windows Runtime on the Windows 8 app developer blog.)

In review, let's step back for a moment to revisit what a promise really *is*. Simply said, it's an object that returns a value, however complex, sometime in the future. The way you get that value is by calling the promise's `then` or `done` method, whose first parameter is a completed function that will receive the promised value when it is ready—and that function might be called immediately if the result is already available! Furthermore, you can call `then`/`done` multiple times for the same promise, and you'll just get the same results in each place. This won't cause the system to get confused or anything.

If there's an error along the way, the second parameter to `then`/`done` is an error function that will be called instead. (Otherwise exceptions are swallowed by `then` or thrown to the event loop by `done`.)

A third parameter to `then`/`done` is a `progress` function, which is called periodically by those async

operations that support it.[26] We've already seen, for instance, how `WinJS.xhr` operations will periodically call the progress function for "ready state" changes and as the response gets downloaded.

Now there's no requirement that a promise has to wrap an async operation or async *anything*. You can, in fact, wrap *any* value in a promise by using the static method `WinJS.Promise.wrap`. Such a wrapper on an already existing value (the future is now!) will just turn right around and call the completed function with that value when you call the promise's `then` or `done` methods. This allows you to use any value, really, where a promise is expected, or return things like errors from functions that otherwise return promises for async operations. (`WinJS.Promise.wraperror` exists for this specific purpose.)

`WinJS.Promise` also provides a host of useful static methods (called directly through `WinJS.Promise`, rather than through a promise object):

- `is` determines whether an arbitrary value is a promise, It makes sure it's an object with a function named "then"; it does not test for "done".
- `as` works like `wrap` except that if you give it a promise, it just returns that promise. If you give a promise to `wrap`, it wraps it in another promise.
- `join` aggregates promises into a single one that's fulfilled when all the values given to it, including other promises, are fulfilled. This essentially groups promises with an AND operation (using `then`, so you'll want to call the join's `done` method to handle errors appropriately).
- `any` is similar to `join` but groups with an OR (again using `then`).
- `cancel` stops an async operation. If an error function is provided, it's called with a value of `Error("canceled")`.
- `theneach` applies completed, error, and progress functions to a group of promises (using `then`), returning the results as another group of values inside a promise.
- `timeout` has a dual nature. If you just give it a timeout value, it returns a promise wrapped around a call to `setTimeout`. If you also provide a promise as the second parameter, it will *cancel* that promise if it's not fulfilled within the timeout period. This latter case is essentially a wrapper for the common pattern of adding a timeout to some other async operation that doesn't have one already.
- `addEventListener`/`removeEventListener` (and `dispatchEvent`) manage handlers for the `error` event that promises will fire on exceptions (but *not* for cancellation). Listening for this event does not affect use of error functions. It's an addition, not a

[26] If you want to impress your friends while reading the documentation, know that if an async function shows it returns a value of type `IAsync[Action | Operation]WithProgress`, then it will utilize a progress function given to a promise. If it only lists `IAsync[Action | Operation]`, progress is not supported.

replacement.[27]

In addition to using functions like `as` and `wrap`, you can also create a promise from scratch by using `new WinJS.Promise(<init> [, <oncancel>)` where `<init>` is a function that accepts completed, error, and progress callbacks and `oncancel` is an optional function that's called in response to `WinJS.Promise.cancel`.

If `WinJS.Promise.as` doesn't suffice, creating a promise like this is useful to wrap other operations (not just values) within the promise structure so that it can be chained or joined with other promises. For example, if you have a library that talks to a web service through raw async XmlHttpRequest, you can wrap each API of that library with promises. You might also use a new promise to combine multiple async operations (or other promises!) from different sources into a single promise, where `join` or `any` don't give you the control you need. Another example is encapsulating specific completed, error, and progress functions within a promise, such as to implement a multiple retry mechanism on top of singular XHR operations, to hook into a generic progress updater UI, or to add under-the-covers logging or analytics with service calls so that the rest of your code never needs to know about them.

What We've Just Learned

- How the local and web contexts affect the structure of an app, for pages, page navigation, and `iframe` elements.
- How to use application content URI rules to extend resource access to web content in an `iframe`.
- Using `ms-appdata` URI scheme to reference media content from local, roaming, and temp appdata folders.
- How to execute a series of async operations with chained promises.
- How exceptions are handled within chained promises and the differences between `then` and `done`.
- Methods for getting debug output and error reports for an app, within the debugger and the Windows Event Viewer.
- 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.
- 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.
- Using data from services through WinJS.xhr and how this relates to the resuming event.

[27] Async operations from WinRT that get wrapped in promises do not fire this error event, which is why you typically use an error handler instead.

- How to achieve 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.
- All the details of promises that are common used with (but not limited to) async operations.

Chapter 4

Controls, Control Styling, and Data Binding

Controls are one of those things you just can't seem to get away from, especially within technology-addicted cultures like those that surround many of us. Even low-tech devices like bicycles and various gardening tools have controls. But this isn't a problem—it's actually a necessity. Controls are the means through which human intent is translated into the realm of mechanics and electronics, and they are entirely made to invite interaction. As I write this, in fact, I'm sitting on an airplane and noticing all the controls that are in my view. The young boy in the row ahead of me seems to be doing the same, and that big "call attendant" button above him is just begging to be pressed!

Controls are certainly essential to Metro style apps, and they will invite consumers to poke, prod, touch, click, and swipe them. (They will also invite the oft-soiled hands of many small toddlers as well; has anyone made a dishwasher-safe tablet PC yet?) Windows 8, of course, provides a rich set of controls for apps written in HTML, CSS, and JavaScript. What's most notable in this context is that from the earliest stages of design, Microsoft wanted to avoid forcing HTML/JavaScript developers to use controls that were incongruous with what those developers already know—namely, the use of HTML control elements like `<button>` that can be styled with CSS and wired up in JavaScript by using functions like `addEventListener` and `on<event>` properties.

You can, of course, use those intrinsic HTML controls in a Metro style app because those apps run on top of the same HTML/CSS rendering engine as Internet Explorer. No problem. There are even special classes, pseudo-classes, and pseudo-elements that give you fine-grained styling capabilities, as we'll see. But the real question was how to implement Windows 8-specific controls like the toggle switch and list view that would allow you to work with them in the same way—that is, declare them in markup, style them with CSS, and wire them up in JavaScript with `addEventListener` and `on<event>` properties.

The result of all this is that for you, the HTML/JavaScript developer, you'll be looking to WinJS for these controls rather than WinRT. Let me put it another way: if you've noticed the large collection of APIs in the `Windows.UI.Xaml` namespace (which constitutes about 40% of WinRT), guess what? You get to completely ignore all of it! Instead, you'll use the WinJS controls that support declarative markup, styling with CSS, and so on, which means that Windows controls (and custom controls that follow the same model) ultimately show up in the DOM along with everything else, making them accessible in all the ways you already know and understand.

The story of controls in Windows 8 is actually larger than a single chapter. Here we'll be looking primarily at those controls that represent or work with simple data (single values) and that participate

in page layout as elements in the DOM. Participating in the DOM, in fact, is exactly why you can style and manipulate all the controls (HTML and WinJS alike) through standard mechanisms, and a big part of this chapter is to just visually show the styling options you have available. In the latter part of this chapter we'll also explore the related subject of data binding: creating relationships between properties of data objects and properties of controls (including styles) so that the controls reflect what's happening in the data.

The story will then continue in Chapter 5, "Collections and Collection Controls," where we'll look at collection controls—those that work with potentially large data sets—and the additional data-binding features that go with them. We'll also give special attention to media elements (image, audio, and video) in Chapter 10, aptly titled "Media," as they have a variety of unique considerations. Similarly, those elements that are primary for defining layout (like grid and flexbox) are the subject of Chapter 6, "Layout," and we also have a number of UI elements that don't participate in layout at all, like app bars and flyouts, as we'll see in Chapter 7, Metro Style Commanding UI."

In short, having covered much of the wiring, framing, and plumbing of an app in Chapter 3, "App Anatomy and Page Navigation," we're ready to start enjoying the finish work like light switches, doorknobs, and faucets—the things that make an app really come to life and engage with human beings.

Sidebar: Essential References for Controls

Before we go on, you'll want to know about two essential topics on the Windows Developer Center that you'll likely refer to time and time again. First is the comprehensive Controls list (Metro style JavaScript), which identifies all the controls that are available to you, as we'll summarize later in this chapter. The second are comprehensive UX Guidelines for Metro style apps, which describes the best use cases for most controls and scenarios in which not to use them. This is a very helpful resource for both you and your designers.

The Control Model for HTML, CSS, and JavaScript

Again, when Microsoft designed the developer experience for Windows 8, we strove for a high degree of consistency between intrinsic HTML control elements, WinJS controls, and custom controls. I like to refer to all of these as "controls" because they all result in a similar user experience: some kind of widget with which the user interacts with an app. In this sense, every such control has three parts:

- Declarative markup (producing elements in the DOM)
- Applicable CSS (styles as well as special pseudo-class and pseudo-element selectors)
- Methods, properties, and events accessible through JavaScript

Standard HTML controls, of course, already have dedicated markup to declare them, like `<button>`,

<input>, and <progress>. WinJS and custom controls, lacking the benefit of existing standards, are declared using some root element, typically a or , with two custom data-* attributes: data-win-control and data-win-options. The value of data-win-control specifies the fully qualified name of a public constructor function that creates the actual control as child elements of the root. The second, data-win-options, is a JSON string containing key-value pairs separated by commas: { <key1>: <value1>, <key1>: <value2>, ... }.

The constructor function itself takes two parameters: the root (parent) element and an options object. Conveniently, WinJS.Class.define produce functions that look exactly like this, making it very handy for defining controls (as WinJS does itself). Of course, because data-* attributes are, according to the HTML5 specifications, completely ignored by the HTML/CSS rendering engine, some additional processing is necessary to turn an element with these attributes into an actual control in the DOM. And this, as I've hinted at before, is exactly the life purpose of the WinJS.UI.process and WinJS.UI.processAll methods. As we'll see shortly, these methods parse the options attribute and pass the resulting object and the root element to the constructor function identified in data-win-control.

The result of this simple declarative markup plus WinJS.UI.process/processAll is that WinJS and custom controls are just elements in the DOM like any others. They can be referenced by DOM-traversal APIs and targeted for styling using the full extent of CSS selectors (as we'll see in the styling gallery later on). They can listen for external events like other elements and can surface events of their own by implementing [add/remove]EventListener and on<event> properties. (WinJS again provides standard implementations of addEventListener, removeEventListener, and dispatchEvent for this purpose.)

Let's now look at the controls we have available for Metro style apps, starting with the HTML controls and then the WinJS controls. In both cases we'll look at their basic appearance, how they're instantiated, and the options you can apply to them.

HTML Controls

HTML controls, I hope, don't need much explaining. They are described in HTML5 references, such as http://www.w3schools.com/html5/html5_reference.asp, and shown with default "light" styling in Figure 4-1 and Figure 4-2. (See the next section for more on WinJS stylesheets.) It's worth mentioning that most embedded objects are not supported, except for a specific ActiveX controls; see Migrating a web app.

Creating or instantiating an HTML also works as you would expect. You either declare them in markup (using attributes to specify options, the rundown of which is given in Table 4-1), or you create them procedurally from JavaScript by calling new with the appropriate constructor, configuring properties and listeners as desired, and adding the element to the DOM wherever its needed. Nothing new here at all where Metro style apps are concerned.

For examples of creating and using these controls, refer to the Common HTML Controls sample in the Windows SDK, from which the images in Figure 4-1 and Figure 4-2 were obtained.

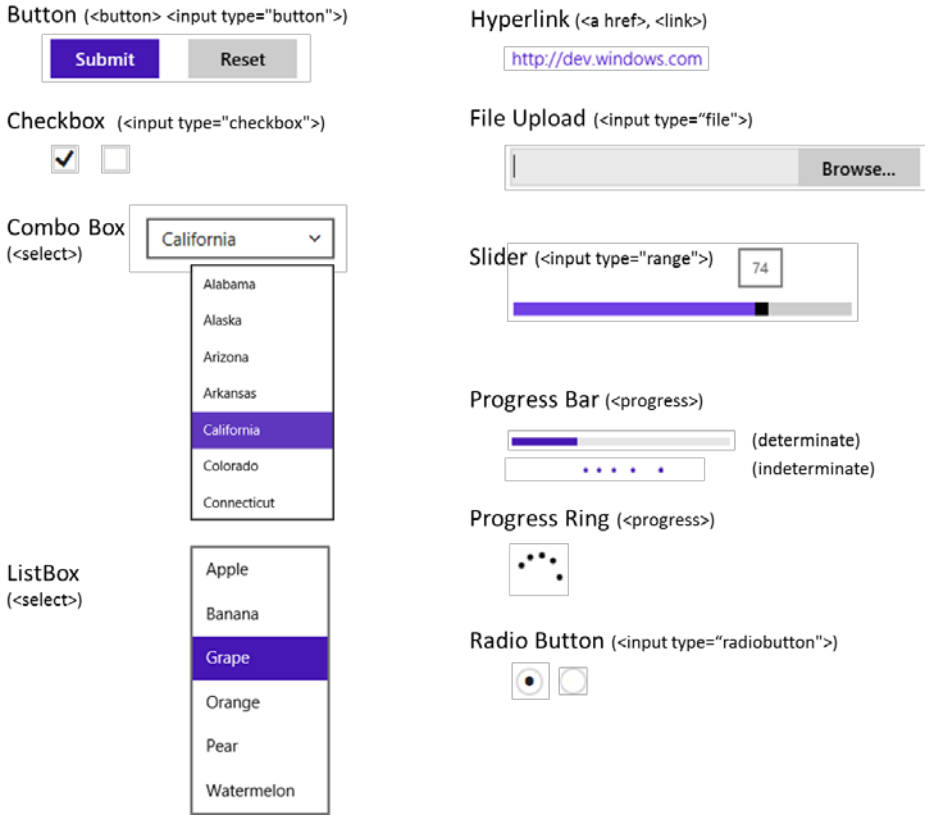


Figure 4-1 Standard HTML5 controls with default "light" styles (the ui-light.css stylesheet of WinJS).

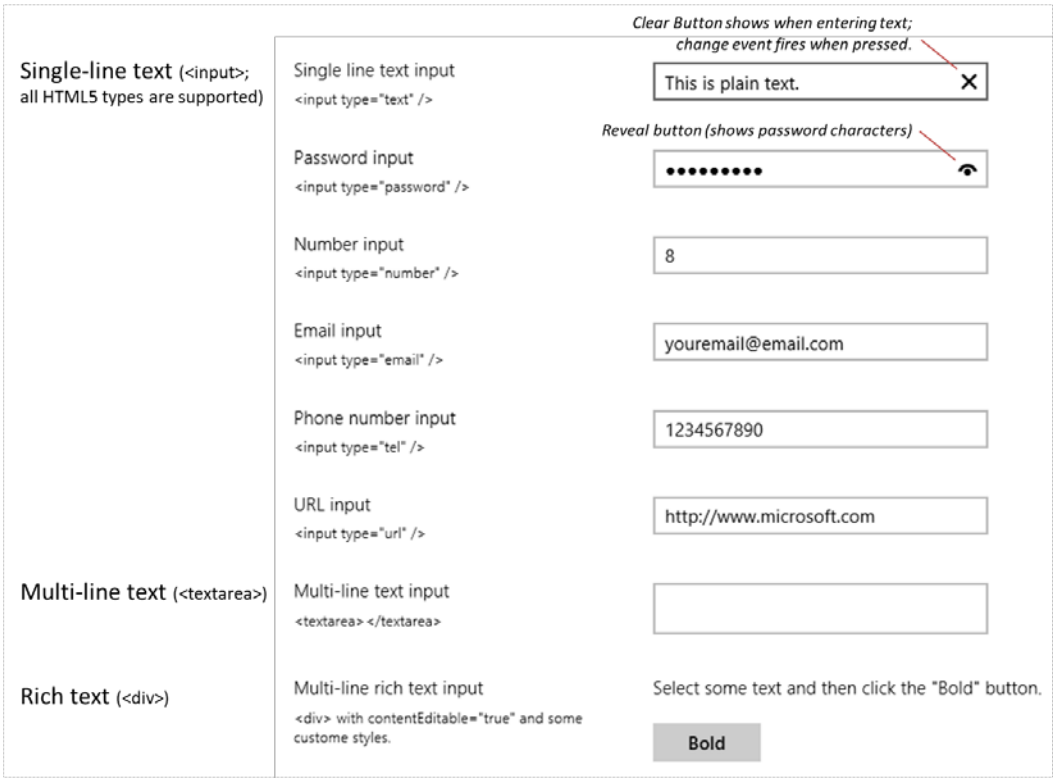


Figure 4-2 Standard HTML5 text input controls with default "light" styles (the ui-light.css stylesheet of WinJS).

Control	Markup	Common Option Attributes	Element Content (inner text/HTML)
Button	<button type="button">	Image; (note that without type, the default is "submit")	Button text
Button	<input type="button"> <input type="submit"> <input type="reset">	value (button text); image	n/a
Checkbox	<input type="checkbox">	value, checked	Checkbox label
Combo Box	<select>	size="1" (default), multiple, selectedIndex	Multiple <option> elements
Email	<input type="email">	value (initial text)	n/a
File Upload	<input type="file">	accept (mime types), mulitple	n/a
Hyperlink	<a href>	target	Link text

ListBox	<select> with size > 1	size (a number greater than 1), multiple, selectedIndex	Multiple <option> elements
Multi-line Text	<textarea>	cols, rows, readonly	Initial text content
Number	<input type="number">	n/a	n/a
Password	<input type="password">	value (initial text)	n/a
Phone Number	<input type="tel">	value (initial text)	n/a
Progress	<progress>	value (initial position), max (highest position; min is 0); no min/max makes it inderterminate	n/a
Radiobutton	<input type="radiobutton">	value, checked, defaultChecked	Radiobutton label
Rich Text		contentEditable="true"	HTML content
Slider	<input type="range">	min, max, value (initial position), step (increment)	n/a
URL	<input type="url">	value (initial text)	n/a

Table 4-1 Options (attributes) for standard HTML5 controls and how element content is applied.

Two areas that add something to HTML controls are the WinJS stylesheets and the additional methods, properties, and events that Microsoft's rendering engine adds to most HTML elements. These are the subjects of the next two sections.

WinJS stylesheets: ui-light.css, ui-dark.css, and win-* styles

WinJS comes with two parallel stylesheets that provide many default styles and style classes for Metro style apps: ui-light.css and ui-dark.css. You'll always use one or the other, as they are mutually exclusive. The first is intended for apps that are oriented around text, because dark text on a light background is generally easier to read (so this theme is often used for news readers, books, magazines, etc., including figures in published books like this!). The dark theme, on the other hand, is intended for media-centric apps like picture and video viewers where you want the richness of the media to stand out.

Both stylesheets define a number of `win-*` style classes, which I like to think of as style packages that effectively add styles and CSS-based behaviors (like the `:hover` pseudo-class) that turn standard HTML controls into a Windows 8-specific variant. These are `win-backbutton` for buttons, `win-ring`, `win-medium`, and `win-large` for circular `progress` controls, `win-vertical` for a vertical slider (range) control, and `win-textarea` for a content editable `div`. If you want to see the details, search on their names in the Style Rules tab in Blend.

Extensions to HTML Elements

As you probably know already, there are many developing standards for HTML and CSS. Until these are brought to completion, implementations of those standards in various browsers are typically made available ahead of time with vendor-prefixed names. In addition, browser vendors sometimes add their own extensions to the DOM API for various elements.

With Metro style apps, of course, you don't need to worry about the variances between browsers, but since these apps essentially run on top of the Internet Explorer engines, it helps to know about those extensions that still apply. These are summarized in Table 4-2, and you can find the full Elements Reference in the documentation for all the details your heart desires (and too much to spell out here).

If you've been working with HTML5 and CSS3 in Internet Explorer already, you might be wondering why the table doesn't show the various animation (`msAnimation*`), transition (`msTransition*`), and transform properties (`msPerspective*` and `msTransformStyle`), along with `msBackface-Visibility`. This is because these standards are now far enough along that they no longer need vendor prefixes with Internet Explorer 10 or Metro style apps (though the `ms*` variants still work).

Methods	Description
`msMatchesSelector`	Determines if the control matches a selector.
`ms[Set | Get | Release]PointerCapture`	Captures, retrieves, and releases pointer capture for an element.
Style properties (on element.style)	Description
`msGrid*`, `msRow*`	Gets or sets placement of element within a CSS grid.
Events (add "on" for event properties)	Description
`mscontentzoom`	Fires when a user zooms an element (Ctrl+ +/-, Ctrl + mousewheel), pinch gestures.
`msgesture[change | end | hold | tap | pointercapture]`	Gesture input events (see Chapter 9, "Input and Sensors").
`msinertiastart`	Gesture input events (see Chapter 9).
`mslostpointercapture`	Element lost capture (set previously with msSetPointerCapture.
`mspointer[cancel | down | hover | move | out | over | up]`	Pointer input events (see Chapter 9).
`msmanipulationstatechanged`	State of a manipulated element has changed.

Table 4-2 Microsoft extensions and other vendor-prefixed methods, properties, and events that apply to HTML controls in Metro style apps.

WinJS Controls

Windows 8 defines a number of controls that help apps fulfill Metro style design guidelines. As noted

before, these are implemented in WinJS for Metro style apps written in HTML, CSS, and JavaScript, rather than WinRT; this allows those controls to integrate naturally with other DOM elements. Each control is defined as part of the `WinJS.UI` namespace using `WinJS.Class.define`, where the constructor name matches the control name. So the full constructor name for a control like the Rating is WinJS.UI`.Rating`.

The simpler controls that we'll cover here in this chapter are `DatePicker`, `Rating`, `ToggleSwitch`, and `Tooltip`, the default styling for which are shown in Figure 4-3. The collection controls that we'll cover in Chapter 5 are `FlipView`, `ListView`, and `SemanticZoom`. App bars, flyouts, and others that don't participate in layout are again covered in later chapters. Apart from these, there is only one other, `HtmlControl`, which is simply an older (and essentially deprecated) alias for `WinJS.UI.Pages`. That is, the `HtmlControl` is the same thing as rendering a page control: it's an arbitrary block of HTML, CSS, and JavaScript that you can declaratively incorporate anywhere in a page. We've already discussed all those details in Chapter 3, so there's nothing more to add here.

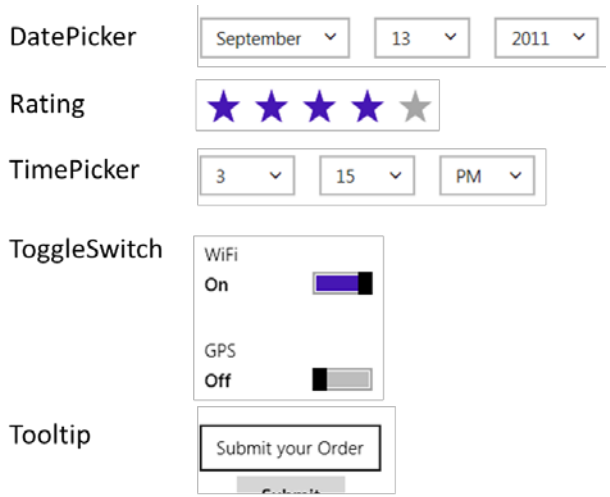


Figure 4-3 Default (light) styles on the simple WinJS controls.

The `WinJS.UI.Tooltip` control, you should know, can utilize any HTML including other controls, so it goes well beyond the plain text tooltip that HTML provides automatically for `alt` and `title` attributes. We'll see more examples later.

So again, a WinJS control is declared in markup by attaching `data-win-control` and `data-win-options` attributes to some root element. That element is typically a `div` (block element) or `span` (inline element), because these don't bring much other baggage, but any element can be used. These elements can, of course, have `id` and `class` attributes as needed. The available options for these controls are summarized in Table 4-3, which includes those events that can be wired up through the `data-win-options` string, if desired. For full documentation on all these options, start with the Controls list in the documentation and go to the control-specific topics linked from there.

Fully-qualified constructor name in data-win-control	Options in data-win-options
WinJS.UI.DatePicker	Properties: calendar, current, datePattern, disabled, maxYear, minYear, monthPattern, yearPattern Events: onchange
WinJS.UI.Rating	Properties: averageRating, disabled, enableClear, maxRating, tooltipStrings (an array of strings the size of maxRating), userRating Events: oncancel, onchange, onpreviewcancel
WinJS.UI.TimePicker	Properties: clock, current, disabled, hourPattern, minuteIncrement, periodPattern Events: onchange
WinJS.UI.ToggleSwitch	Properties: checked, disabled, labelOff, labelOn, title Events: onchange
WinJS.UI.Tooltip	Properties: contentElement, innerHtml, infotip, placement Events: onbeforeclose, onbeforeopen, onclosed, onopened Methods: open, close

Table 4-3 Available options for simple WinJS controls.

Again, the `data-win-options` string containing key-value pairs, one for each property or event, separated by commas, in the form `{ <key1>: <value1>, <key1>: <value2>, ... }`. For events, whose names in the options string always start with `on`, the value is the name of the event handler you want to assign.

In JavaScript code, you can also assign event handlers by using `<element>.addEventListener("<event>", …)` where `<element>` is the element for which the control was declared and `<event>` drops the "on" as usual. To access the properties and events directly, use `<element>.winControl.-<property>`. The `winControl` object is created when the WinJS control is instantiated and attached to the element, so that's where these options are available.

WinJS Control Instantiation

As we've seen a number of times already, WinJS controls declared in markup with `data-*` attributes are not instantiated until you call `WinJS.UI.process(<element>)` for a single control or `WinJS.UI.processAll` for all such elements in the DOM. To understand this process, here's what `WinJS.UI.process` does for a single element:

13. Parse the `data-win-options` string into an options object.
14. Extract the constructor specified in `data-win-control`, and then call `new` on that function passing the root element and the options object.
15. The constructor creates whatever child elements it needs within the root element.
16. The object returned from the constructor—the control object—is stored in the root element's `winControl` property.

Clearly, then, the bulk of the work really happens in the constructor. Once this has happened, other JavaScript code (as in your activated method) can call methods, manipulate properties, and add listeners for events on both the root element and the `winControl` object. The latter, clearly, must be used for WinJS control-specific methods, properties, and events.

`WinJS.UI.processAll`, for its part, simply traverses the DOM looking for data-win-control attributes and does `WinJS.UI.process` for each. How you use both of these is really your choice: `processAll` goes through a whole page (or just a page control—whatever the document object refers to), whereas `process` lets you control the exact sequence or instantiate controls for which you dynamically insert markup. Note that in both cases the return value is a promise, so if you need to take additional steps after processing is complete, call the promise's `done` method with a suitable completed function.

It's also good to understand that `process` and `processAll` are really just helper functions. If you need to, you can just directly call `new` on a control constructor with an element and options object. This will create the control and attach it to the given element automatically. You can also pass `null` for the element, in which case the WinJS control constructors create a new `div` element to contain the control that is otherwise unattached to the DOM. This would allow you, for instance, to build up a control offscreen and attach it to the DOM only when needed.

To see all this in action, we'll look at some examples with both the Rating and Tooltip controls in a moment. First, however, we need to discuss what that `WinJS.strictProcessing()` call we've seen in the templates is all about.

Strict Processing and processAll Functions

WinJS has three DOM-traversing functions: `WinJS.UI.processAll`, `WinJS.Binding.processAll` (which we'll see later in this chapter), and `WinJS.Resources.processAll` (which we'll see in Chapter 17, "Apps for Everyone"). Each of these looks for specific `data-win-*` attributes and then takes additional actions using those contents. Those actions, however, can involve calling a number of different types of functions:

- Functions appearing in a "dot path" for control processing and binding sources
- Functions appearing in the left-hand side for binding targets, resource targets, or control processing
- Control constructors and event handlers
- Binding initializers or functions used in a binding expression
- Any custom layout used for a ListView control

Such actions introduce a risk of injection attack if a `processAll` function is called on untrusted HTML, such as arbitrary markup obtained from the web. To mitigate this risk, WinJS has a notion of strict processing that is enabled at present by calling `WinJS.strictProcessing()` and that will be

enabled by default for the final release of Windows 8. The effect of strict processing (which cannot be subsequently disabled) is that any functions indicated in markup that `processAll` methods might encounter must be "marked for processing" or else processing will fail. The mark itself is simply a `supportedForProcessing` property on the function object that is set to `true`.

Functions returned from `WinJS.Class.define`, `WinJS.Class.derive`, `WinJS.UI.Pages.define`, and `WinJS.Binding.converter` are automatically marked in this manner. For other functions, you can either set a `supportedForProcessing` property to true directly or use marking functions like so:

```
WinJS.Utilities.markSupportedForProcessing(myfunction);
WinJS.UI.eventHandler(myHandler);
WinJS.Binding.initializer(myInitializer);

//Also OK
<namespace>.myfunction = WinJS.UI.eventHandler(function () {
});
```

Note also that appropriate functions coming directly from WinJS, such as all `WinJS.UI.*` control constructors, as well as `WinJS.Binding.*` functions, are marked by default.

So, if you reference custom functions from your markup, be sure to mark them accordingly. But this is *only* for references from *markup*: you don' t need to mark functions that you assign to `on<event>` properties in code or pass to `addEventListener`.

Example: WinJS.UI.Rating Control

OK, now that we got the strict processing stuff covered, let's see some concrete example of working with a WinJS control.

For starters, here's some markup for a `WinJS.UI.Rating` control, where the options specify two initial property values and an event handler:

```
<div id="rating1" data-win-control="WinJS.UI.Rating"
    data-win-options="{averageRating: 3.4, userRating: 4, onchange: changeRating}">
</div>
```

To instantiate this control, we need to call either of the following functions:

```
WinJS.UI.process(document.getElementById("rating1"));
WinJS.UI.processAll();
```

Again, both of these functions return a promise, but it's unnecessary to call `done` unless we need to do additional post-instantiation processing or surface exceptions that might have occurred (and that are otherwise swallowed). Also, note that the `changeRating` function specified in the markup must be globally visible and marked for processing, or else the control will fail to instantiate.

Next, we can instantiate the control and set the options procedurally. So, in markup:

```
<div id="rating1" data-win-control="WinJS.UI.Rating"></div>
```

And in code:

```
var element = document.getElementById("rating1");
WinJS.UI.process(element);
element.winControl.averageRating = 3.4;
element.winControl.userRating = 4;
element.winControl.onchange = changeRating;
```

The last three lines above could also be written as follows using the `WinJS.UI.setOptions` method, but this isn't recommended because it's harder to debug:

```
var options = { averageRating: 3.4, userRating: 4, onchange: changeRating };
WinJS.UI.setOptions(element.winControl, options);
```

We can also just instantiate the control directly. In this case the markup is nonspecific:

```
<div id="rating1"></div>
```

And we call `new` on the constructor ourselves:

```
var newControl = new WinJS.UI.Rating(document.getElementById("rating1"));
newControl.averageRating = 3.4;
newControl.userRating = 4;
newControl.onchange = changeRating;
```

Or, as mentioned before, we can skip the markup entirely, have the constructor create an element for us (a `div`), and attach it to the DOM at our leisure:

```
var newControl = new WinJS.UI.Rating(null,
    { averageRating: 3.4, userRating: 4, onchange: changeRating });
newControl.element.id = "rating1";
document.body.appendChild(newControl.element);
```

Hint If you see strange errors on instantiation with these latter two cases, check whether you forgot the `new` and are thus trying to invoke the constructor function directly.

Note also in these last two cases that the *rating1* element will have a `winControl` property that is the same as `newControl` as returned from the constructor.

To see this control in action, please refer to the HTML Rating control sample in the SDK.

Example: WinJS.UI.Tooltip Control

With most of the other simple controls—namely the DatePicker, TimePicker, and ToggleSwitch—you can work with them in the same ways as we just saw with Ratings. All that changes are the specifics of their properties and events; again, start with the Controls list page and navigate to any given control for all the specific details. Also, for working samples refer to the HTML DatePicker and TimePicker controls and the HTML ToggleSwitch control samples in the SDK.

The `WinJS.UI.Tooltip` control is a little different, however, so I'll illustrate its specific usage. First,

to attach a tooltip to a specific element, you can either add a `data-win-control` attribute to that element or place the element itself inside the control:

```
<!-- Directly attach the Tooltip to its target element -->
<targetElement data-win-control="WinJS.UI.Tooltip">
</targetElement>

<!-- Place the element inside the Tooltip -->
<span data-win-control="WinJS.UI.Tooltip">
    <!-- The element that gets the tooltip goes here -->
</span>

<div data-win-control="WinJS.UI.Tooltip">
    <!-- The element that gets the tooltip goes here -->
</div>
```

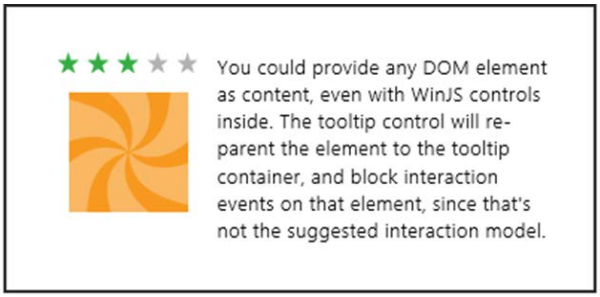
Second, the `contentElement` property of the tooltip control can name another element altogether, which will be displayed when the tooltip is invoked. For example, if we have this piece of hidden HTML in our markup (and notice that it contains other controls):

```
<div style="display: none;">
    <!--Here is the content element. It's put inside a hidden container
    so that it's invisible to the user until the tooltip takes it out.-->
    <div id="myContentElement">
        <div id="myContentElement_rating">
            <div data-win-control="WinJS.UI.Rating" class="win-small movieRating"
                data-win-options="{userRating: 3}">
            </div>
        </div>
        <div id="myContentElement_description">
            <p>You could provide any DOM element as content, even with WinJS controls inside. The
tooltip control will re-parent the element to the tooltip container, and block interaction events
on that element, since that's not the suggested interaction model.</p>
        </div>
        <div id="myContentElement_picture">
        </div>
    </div>
</div>
```

we can reference it like so:

```
<div data-win-control="WinJS.UI.Tooltip"
   data-win-options="{infotip: true, contentElement: myContentElement}">
   <span>My piece of data</span>
</div>
```

When you hover over the text (with a mouse or hover-enabled touch hardware), this tooltip will appear:



This example is taken directly from the HTML Tooltip control sample in the SDK, so you can go there to see how all this works directly.

Working with Controls in Blend

Before we move onto the subject of control styling, it's a good time to highlight a few additional features of Blend for Visual Studio where controls are concerned. As I mentioned in Video 2-1, the Assets tab in Blend gives you quick access to all the HTML elements and WinJS controls that you can just drag and drop into whatever page is showing in the artboard. (See Figure 4-4.) This will create basic markup, such as a `div` with a `data-win-control` attribute for WinJS controls; then you can go to the HTML Attributes pane (on the right) to set options in the markup. (See Figure 4-5.)

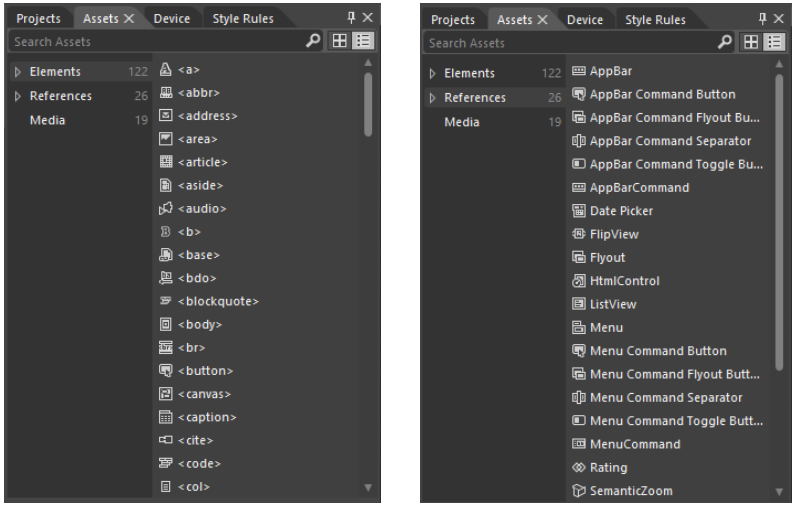


Figure 4-4 HTML elements (left) and WinJS control (right) as shown in Blend's Assets tab.

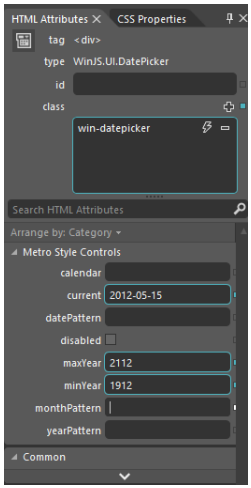


Figure 4-5 Blend's HTML Attributes tab shows WinJS control options, and editing them will affect the `data-win-options` attribute in markup.

Next, take a moment to load up the Common HTML Controls sample from the SDK into Blend. This is a great opportunity to try out Blend's Interactive Mode to navigate to a particular page and explore the interaction between the artboard and the Live DOM. (See Figure 4-6.) Once you open the project, go into interactive mode by selecting View -> Interactive Mode on the menu, pressing Ctrl+Shift+I, or clicking the small leftmost button on the upper right corner of the artboard. Then select scenario 5 (Progress introduction) in the listbox, which will take you to the page shown in Figure 4-6. Then exit interactive mode (same commands), and you'll be able to click around on that page.

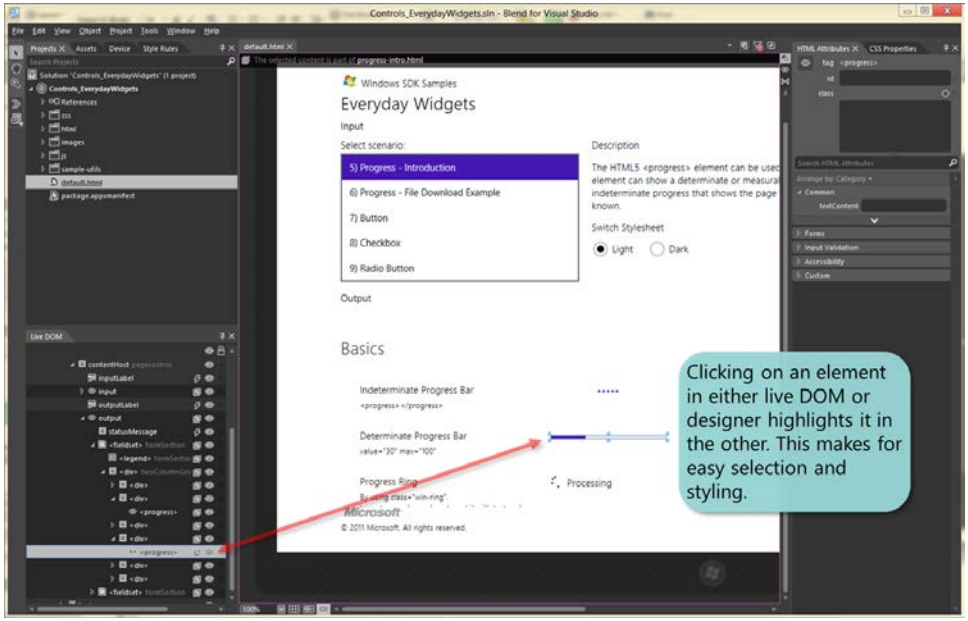


Figure 4-6 Blend's interaction between the artboard and the Live DOM.

With the Common HTML Controls sample, you'll see that there's just a single element in the Live DOM for intrinsic controls, as there should be, since all the internal details are part and parcel of the HTML/CSS rendering engine. On the other hand, load up the HTML Rating control sample instead and expand the div that contains one such control. There you'll see all the additional child elements that make up this control (shown in Figure 4-7), and you can refer to the right-hand pane for HTML attributes and CSS properties. You can see something similar (with even more detailed information), in the DOM Explorer of Visual Studio when the app is running. (See Figure 4-8.)

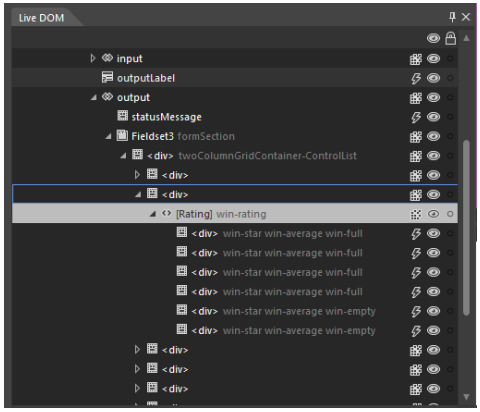


Figure 4-7 Expanding a WinJS control in Blend's Live DOM reveals the elements that are used to build it.

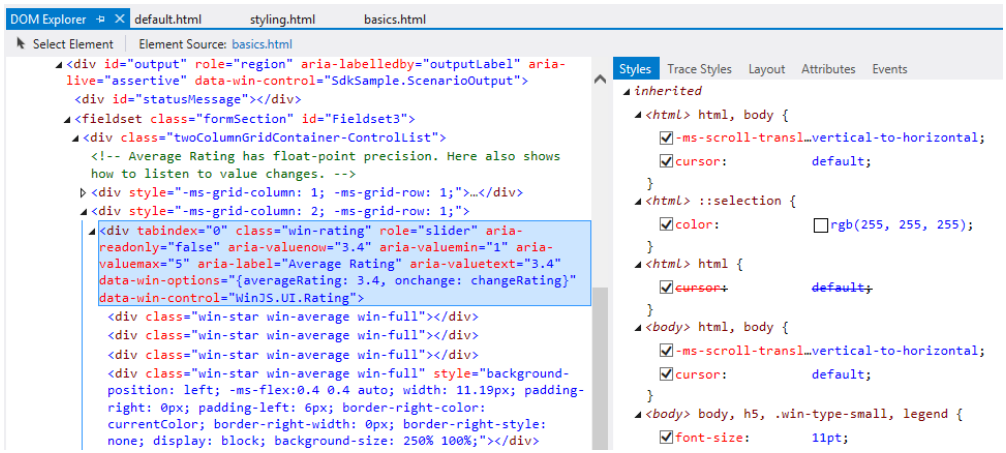


Figure 4-8 Expanding a WinJS control in Visual Studio's DOM Explorer also shows complete details for a control.

Control Styling

Now we come to a topic where we'll mostly get to look at lots of pretty pictures: the various ways in

which HTML and WinJS controls can be styled. As we've discussed, this happens through CSS all the way, either in a stylesheet or by assigning `style.*` properties, meaning that apps have full control over the appearance of controls. In fact, absolutely *everything* that's different between HTML controls in a Metro style app and the same controls on a web page, is due to styling and styling alone.

For both HTML and WinJS controls, CSS standards apply including pseudo-selectors like `:hover`, `:active`, `:checked`, and so forth, along with `-ms-*` prefixed styles for emerging standards.

For HTML controls, there are also additional `-ms-*` styles—that aren't part of CSS3—to isolate specific parts of those controls. That is, because the constituent parts of such controls don't exist separately in the DOM, pseudo-selectors—like `::-ms-check` to isolate a checkbox mark and `::-ms-fill-lower` to isolate the left or bottom part of a slider—allow you to communicate styling to the depths of the rendering engine. In contrast, all such parts of WinJS controls are addressable in the DOM, so they are just styled with specific `win-*` classes defined in the WinJS stylesheets. That is, the controls are simply rendered with those style classes. Default styles are defined in the WinJS stylesheets, but apps can override any aspect of those to style the controls however you want.

In a few cases, as already pointed out, certain `win-*` classes define style packages for use with HTML controls, such as `win-backbutton`, `win-vertical` (for a slider) and `win-ring` (for a progress control). These are intended to style standard controls to look like special system controls.

There are also a few general purpose `-ms-*` styles (not selectors) that can be applied to many controls (and elements in general), along with some general WinJS `win-*` style classes. These are summarized in Table 4-4.

Style or Class	Description
`-ms-user-select: none | element | text`	Enables or disables selection for an element.
`-ms-zoom: <percentage>`	Optical zoom (magnification).
`-ms-touch-action: auto | none` (and more)	Allows specific tailoring of a control's touch experience, enabling more advanced interaction models.
`win-interactive`	Prevents default behaviors for controls contained inside FlipView and ListView controls (see Chapter 5).
`win-swipeable`	Sets `-ms-touch-action` styles so a control within a ListView can be swiped (to select) in one direction without causing panning in the other.
`win-small`, `win-medium`, `win-large`	Size variations to some controls.
`win-textarea`	Sets typical text editing styles.

Table 4-4 General -ms-* styles and win-* classes for Metro style apps.

For all of these and more, spend some time with these three reference topics: WinJS styles for typography, WinJS CSS classes for HTML controls, and WinJS classes for WinJS controls. I also wanted to provide you with a summary (Table 4-5) of all the other vendor-prefixed styles (or selectors) that are supported within the CSS engine for Metro style apps. I made this list because the documentation here can be hard to penetrate: you have to click through the individual pages under the Cascading Style Sheets topic in the docs to see what little bits have been added to the CSS you already know.

Area	Styles
Backgrounds and borders	`-ms-background-position-[x | y]`
Box model	`-ms-overflow-[x | y]`
Basic UI	`-ms-text-overflow` (for ellipses rendering) `-ms-user-select` (sets or retrieves where users are able to select text within an element.) `-ms-zoom` (optical zoom)
Flexbox	`-ms-[inline-]flexbox` (values for `display`); `-ms-flex` and `-ms-flex-[align | direction | order | pack | wrap]`
Gradients	`-ms-[repeating-]linear-gradient`, `-ms-[repeating-]radial-gradient`
Grid	`-ms-grid` and `-ms-grid-[column | column-align | columns | column-span | grid-layer | row | row-align | rows | row-span]`
High contrast	`-ms-high-contrast-adjust`
Regions	`-ms-flow-[from | into]` along with the `MSRangeCollection` method
Text	`-ms-block-progression`, `-ms-hyphens` and `-ms-hypenate-limit-[chars | lines | zone]`, `-ms-text-align-last`, `-ms-word-break`, `-ms-word-wrap`, `-ms-ime-mode`, `-ms-layout-grid` and `-ms-layout-grid-[char | line | mode | type]`, and `-ms-text-[autospace | kashida-space | overflow | underline-position]`
Other	`-ms-writing-mode`

Table 4-5 Summary of -ms-* styles beyond CSS standards. Vendor-prefixed styles for animations, transforms, and transitions are still supported, though no longer necessary, because these standards have recently been finalized.

Styling Gallery: HTML Controls

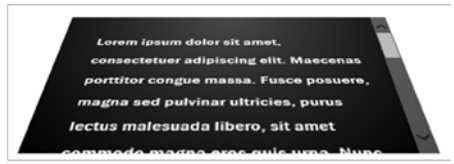
Now we get to enjoy a visual tour of styling capabilities for Metro style apps. Much can be done with standard styles, and then there are all the things you can do with special styles and classes as shown in the graphics in this section. The specifics of all these examples can be seen in the Common HTML Controls sample in the SDK.

Also check out the very cool Applying app theme color (theme roller) sample. This beauty lets you configure the primary and secondary colors for an app, shows how those colors affect different controls, and produces about 200 lines of precise CSS that you can copy into your own stylesheet. This very much helps you create a color theme for your app, which we very much encourage to establish an app's own personality within the overall Metro style design guidelines.

Button (background-color)

Progress (color)

Text Area (transform)



Select (background-color, color, border, font)

Checkbox/Radiobutton (background-image and :checked)

Button

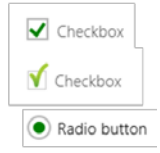
```
<button class="win-backbutton">
```

Checkbox/Radiobutton

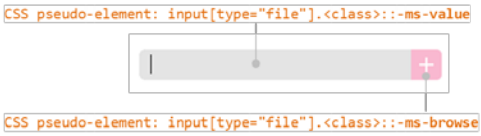
```
CSS pseudo-element: input[type="checkbox"].<class>::-ms-check (color)
```

```
CSS pseudo-element: input[type="checkbox"].<class>::-ms-check (image)
```

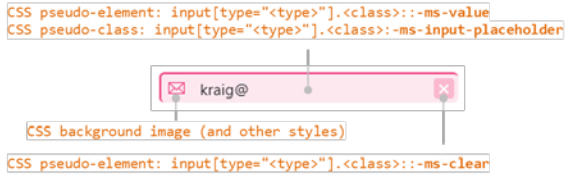
```
CSS pseudo-element: input[type="radio"].<class>::-ms-check (color)
```



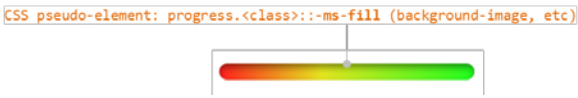
File upload



Text Input (most forms)

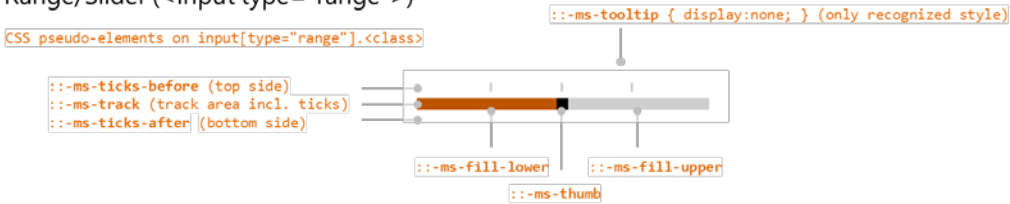


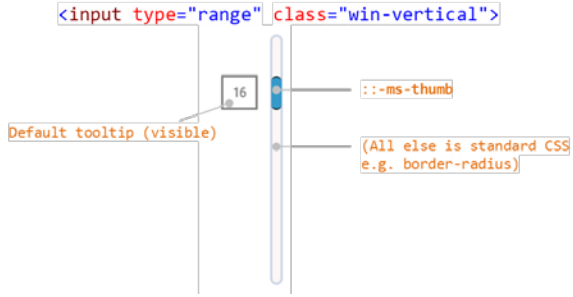
Progress

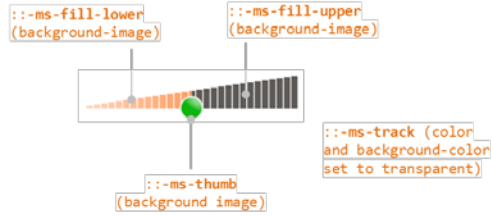


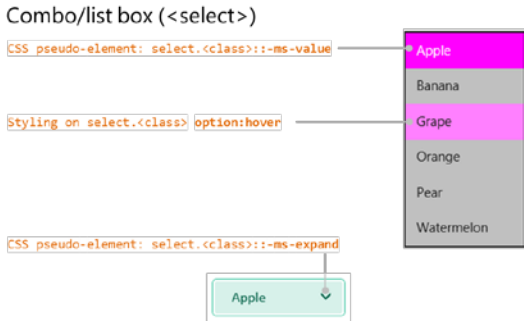
Text Input (password)

Range/Slider (<input type="range">)





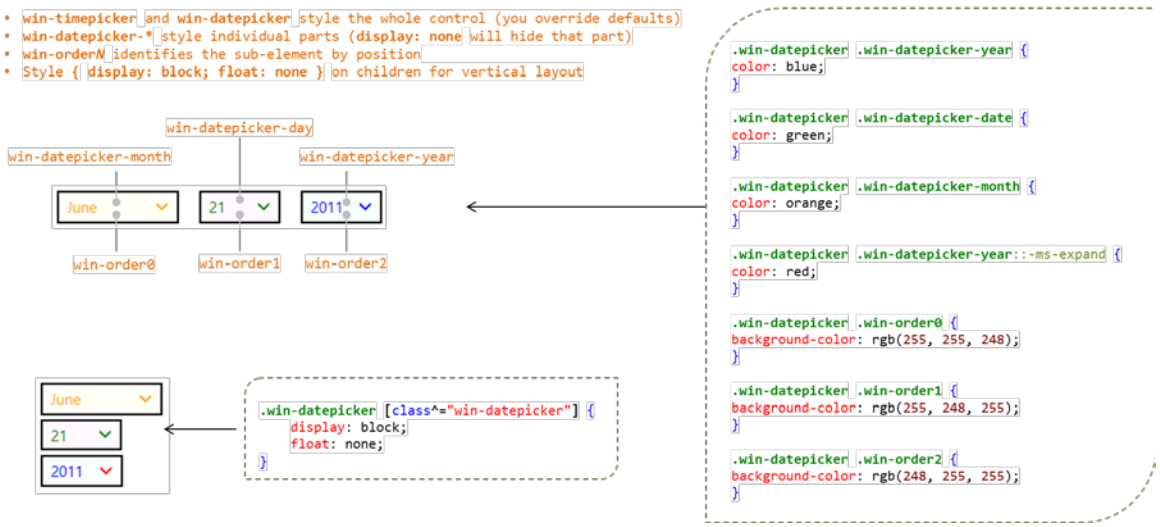


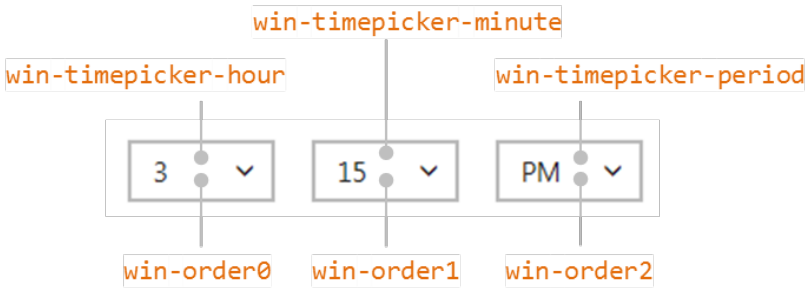


Styling Gallery: WinJS Controls

Similarly, here is a visual rundown of styling for WinJS controls, drawing again from the samples in the SDK: HTML DatePicker and TimePicker controls, HTML Rating control, HTML ToggleSwitch control, and HTML Tooltip control.

For the WinJS DatePicker and TimePicker, refer to styling for the HTML `select` element along with the `::-ms-value` and `::-ms-expand` pseudo-elements. I will note that the sample isn't totally comprehensive, so the visuals below highlight the finer points:



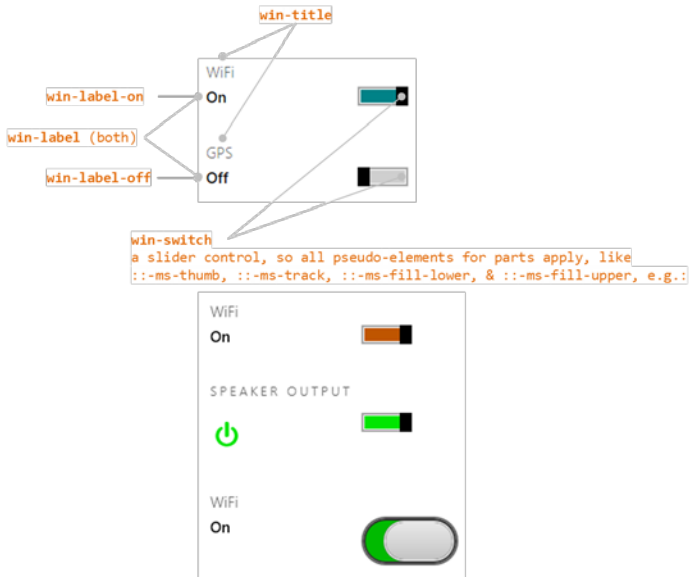


The Rating control has states that can be styled in addition to its stars and the overall control. `win-*` classes identify these individually; combinations style all the variations as in Table 4-6.

Style Class	Part
`win-rating`	Styles the entire control.
`win-star`	Styles the control's stars generally.
`win-empty`	Styles the control's empty stars.
`win-full`	Styles the control's full stars.
.win-star Classes	State
`win-average`	Control is displaying an average rating (user has not selected a rating and the `averageRating` property is non-zero).
`win-disabled`	Control is disabled.
`win-tentative`	Control is displaying a tentative rating.
`win-user`	Control is displaying user-chosen rating.
Variation	Classes (selectors)
Average empty stars	`.win-star.win-average.win-empty`
Average full stars	`.win-star.win-average.win-full`
Disabled empty stars	`.win-star.win-disabled.win-empty`
Disabled full stars	`.win-star.win-disabled.win-full`
Tentative empty stars	`.win-star.win-tentative.win-empty`
Tentative full stars	`.win-star.win-tentative.win-full`
User empty stars	`.win-star.win-user.win-empty`
User full stars	`.win-star.win-user.win-full`

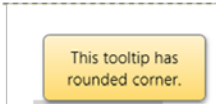
Table 4-6 Styles and variations for WinJS.UI.Rating

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



And finally, for Tooltip, `win-tooltip` is a single class for the tooltip as a whole; the control can then contain any other HTML to which CSS applies using normal selectors:

```
#roundedCornerTooltip {
    border-radius: 6px;
    max-width: 140px;
    padding: 10px;
    border: 1px solid rgb(108, 108, 108);
    background-image:
        -ms-linear-gradient(-90deg, rgb(255, 242, 207),
        rgb(254, 218, 108));
    box-shadow: 1px 2px 6px rgba(0, 0, 0, 0.4);
    text-align: center;
}
```



Rounded Corner

```
#ovalTooltip {
    display: -ms-box;
    -ms-box-orient: vertical;
    -ms-box-align: middle;
}

#oval {
    border-radius: 100px 100px 100px 100px / 50px 50px 50px 50px;
    width: 212px;
    height: 100px;
    border: 1px solid black;
    background-color: white;
    text-align: center;
    line-height: 100px;
    color: black;
}

#beak {
    background-image: url("images/beak.svg");
    background-repeat: no-repeat;
    background-size: 100% 100%;
    width: 12px;
    height: 16px;
    margin-top: -1px;
}
```

Oval

Some Tips and Tricks

1. In the current implementation, tooltips on a slider (`<input type="range">`) are always numerical values; there isn't a means to display other forms of text, such as *Low*, *Medium*, and *High*. For something like this, you could consider a `WinJS.UI.Rating` control with three values, using the `tooltipStrings` property to customize the tooltips.

2. The `::-ms-tooltip` pseudo-selector for the slider affects only visibility (with `display: none`); it cannot be used to style the tooltip generally. This is useful to hide the default tooltips if you want to implement custom UI of your own.

3. There are additional types of `input` controls (different values for the `type` attribute) that I haven't mentioned. This is because those types have no special behaviors and just render as a text box. Those that have been specifically identified might also just render as a text box, but they can affect, for example, what on-screen keyboard configuration is displayed on a touch

device.

4. If you don't find `width` and `height` properties working for a control, try using `style.width` and `style.height` instead.

5. You'll notice that there are two kinds of button controls: `<button>` and `<input type="button">`. They're visually the same, but the former is a block tag and can display HTML inside itself, whereas the latter is an inline tag that displays only text. A `button` also defaults to `<input type="submit">`, which has its own semantics, so you generally want to use `<button type="button">` to be sure.

6. If a `WinJS.UI.Tooltip` is getting clipped, you can override the `max-width` style in the `win-tooltip` class, which is set to 30em in the WinJS stylesheets. Again, peeking at the style in Blend's Style Rules tab is a quick way to see the defaults.

7. The HTML5 `meter` element is not supported for Metro style apps in Windows 8.

8. To turn off the focus rectangle for a control, use `<selector>:focus { outline: none; }`.

9. Metro style apps can use the `window.getComputedStyle` method to obtain a `currentStyle` object that contains the applied styles for an element, *or* for a pseudo-element. This is very helpful, especially for debugging, because pseudo-elements like `::-ms-thumb` for an HTML slider control never appear in the DOM, so the styling is not accessible through the element's `style` property nor does it surface in tools like Blend. Here's an example of retrieving the background color style for a slider thumb:

```
var styles = window.getComputedStyle(document.getElementById("slider1"), "::-ms-thumb");
styles.getPropertyValue("background-color");
```

Custom Controls

As extensive as the HTML and WinJS controls are, there will always be something you wish the system provided but doesn't. "Is there a calendar control?" is a question I've often heard. "What about charting controls?" These clearly aren't included directly in Windows 8, and despite any wishing to the contrary, it means you or another third-party will need to create a custom control.

Fortunately, everything we've learned so far, especially about WinJS controls, applies to custom controls. In fact, WinJS controls are entirely implemented using the same model that you can use directly, and since you can look at the WinJS source code anytime you like, you already have a bunch of reference implementations available.

To go back to our earlier definition, a control is just declarative markup (creating elements in the DOM) plus applicable CSS plus methods, properties, and events accessible from JavaScript. To create such a control in the WinJS model, generally follow this pattern:

1. Define a namespace for your control(s) by using `WinJS.Namespace.define` to both provide a

naming scope and to keep excess identifiers out of the global namespace. (Do *not* add controls to the WinJS namespace.) Remember that you can call `WinJS.Namespace.define` many times to add new members, so typically an app will just have a single namespace for all its custom controls.

2. Within that namespace, define the control constructor by using `WinJS.Class.define` (or `derive`), assigning the return value to the name you want to use in `data-win-control` attributes. That fully qualified name will be `<namespace>.<constructor>`.

3. Within the constructor (of the form <constructor>(`element`, `options`):
   a. You can recognize any set of options you want; these are arbitrary. Simply ignore any that you don't recognize.
   b. If `element` is `null` or `undefined`, create a `div` to use in its place.
   c. Assuming `element` is the root element containing the control, be sure to set `element.winControl=this` and `this.element=element` to match the WinJS pattern.

4. Within `WinJS.Class.define`, the second argument is an object containing your public methods and properties (those accessible through an instantiated control instance); the third argument is an object with static methods and properties (those accessible through the class name without needing to call `new`).

5. For your events, mix (`WinJS.Class.mix`) your class with the results from `WinJS.Utilities.createEventProperties(<events>)` where `<events>` is an array of your event names (without on prefixes). This will create `on<event>` properties in your class for each name in the list.

6. Also mix your class with `WinJS.UI.DOMEventMixin` to add standard implementations of `addEventListener`, `removeEventListener`, `dispatchEvent`, and `setOptions`.[28]

7. In your implementation (markup and code), refer to classes that you define in a default stylesheet but that can be overridden by consumers of the control. Consider using existing `win-*` classes to align with general styling.

8. A typical best practice is to organize your custom controls in per-control folders that contain all the html, js, and css files for that control. Remember also that calls to `WinJS.Namespace.define` for the same namespace are additive, so you can populate a single namespace with controls that are defined in separate files.

You might consider using `WinJS.UI.Pages` if what you need is mostly a reusable block of HTML/CSS/JavaScript for which you don't necessarily need a bunch of methods, properties, and events. `WinJS.UI.Pages` is, in fact, implemented as a custom control. Along similar lines, if what you need is

---

[28] Note that there is also a `WinJS.Utilities.eventMixin` that is similar (without `setOptions`) that is useful for noncontrol objects that won't be in the DOM but still want to fire events. The implementations here don't participate in DOM event bubbling/tunneling.

a reusable block of HTML in which you want to do run-time data binding, check out `WinJS.Binding.Template` (which we'll see toward the end of this chapter), which exists for that purpose. This isn't a control as we've been describing here—it doesn't support events, for instance—but might be exactly what you need.

It's also worth reminding you that everything in WinJS, like `WinJS.Class.define` and `WinJS.UI.DOMEventMixin` are just helpers for common patterns. You're not in any way required to use these, because in the end, custom controls are just elements in the DOM like any others and you can create and manage them however you like. The WinJS utilities just make most jobs cleaner and easier.

Custom Control Examples

To see these recommendations in action, here are a couple of examples. First is what Chris Tavares, one of the WinJS engineers who has been a tremendous help with this book, described as the "dumbest control you can imagine." Yet it certainly shows the most basic structures:

```
WinJS.Namespace.define("AppControls", {
    HelloControl: function (element, options) {
        element.winControl = this;
        this.element = element;

        if (options.message) {
            element.innerText = options.message;
        }
    }
});
```

With this, you can then use the following markup so that `WinJS.UI.process/processAll` will instantiate an instance of the control (as an inline element because we're using `span` as the root):

```
<span data-win-control="AppControls.HelloControl"
    data-win-options="{ message: 'Hello, World'}">
</span>
```

Note that the control definition code must be executed before `WinJS.UI.process/processAll` so that the constructor function named in `data-win-control` actually exists at that point.

For a more complete control, you can take a look at the HTML SemanticZoom for custom controls sample in the Windows SDK. My friend Kenichiro Tanaka of Microsoft Tokyo also created the calendar control shown in Figure 4-9 and provided in the CalendarControl sample for this chapter.

Following the guidelines given earlier, this control is defined using `WinJS.Class.define` within a Controls namespace (calendar.js lines 4–10 shown here [with a comment line omitted]):

```
WinJS.Namespace.define("Controls", {
    Calendar : WinJS.Class.define(
        function (element, options) {
            this.element = element || document.createElement("div");
            this.element.className = "control-calendar";
```

```
        this.element.winControl = this;
```

The rest of the constructor (lines 12–63) builds up the child elements that define the control, making sure that each piece has a particular class name that, when scoped with the `control-calendar` class placed on the root element above, allows specific styling of the individual parts. The defaults for this are in calendar.css; specific overrides that differentiate the two controls in Figure 4-9 are in default.css.

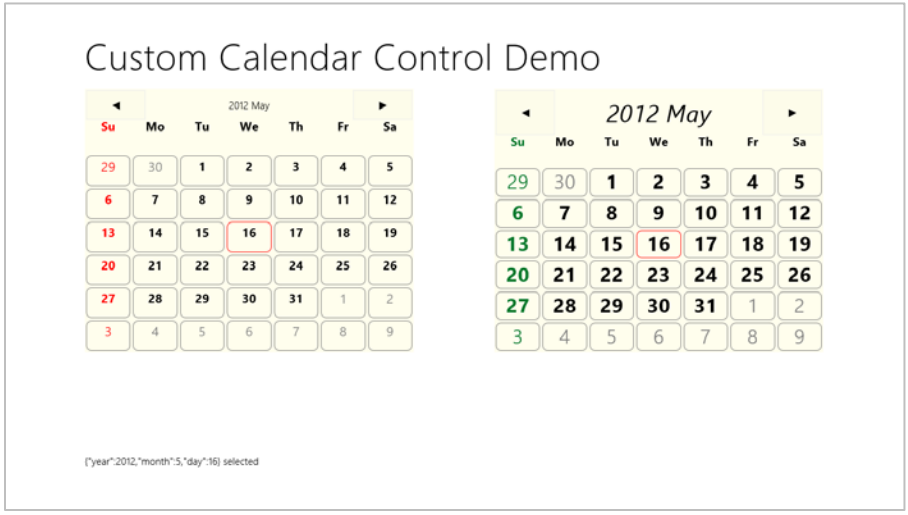


Figure 4-9 Output of the Calendar Control demo sample.

Within the constructor you can also see that the control wires up its own event handlers for its child elements, such as the previous/next buttons and each date cell. In the latter case, clicking a cell uses dispatchEvent to raise a `dateselected` event from the overall control itself.

Lines 63–127 then define the members of the control. There are two internal methods, `_setClass` and `_update`, followed by two public methods, `nextMonth` and `prevMonth`, followed by three public properties, `year`, `month`, and `date`. Those properties can be set through the `data-win-options` string in markup or directly through the control object as we'll see in a moment.

At the end of calendar.js you'll see the two calls to `WinJS.Class.mix` to add properties for the events (there's only one here), and the standard DOM event methods like `addEventListener`, `removeEventListener`, and `dispatchEvent`, along with `setOptions`:

```
WinJS.Class.mix(Controls.Calendar, WinJS.Utilities.createEventProperties("dateselected"));
WinJS.Class.mix(Controls.Calendar, WinJS.UI.DOMEventMixin);
```

Very nice that adding all these details is so simple—thank you, WinJS![29]

[29] Technically speaking, `WinJS.Class.mix` accepts a variable number of arguments, so you can actually combine the two

Between calendar.js and calendar.css we have the definition of the control. In default.html and default.js we can then see how the control is used. In Figure 4-9, the control on the left is declared in markup and instantiated through the call to `WinJS.UI.processAll` in default.js.

```
<div id="calendar1" class="control-calendar" aria-label="Calendar 1"
    data-win-control="Controls.Calendar"
    data-win-options="{ year: 2012, month: 5, ondateselected: CalendarDemo.dateselected}">
</div>
```

You can see how we use the fully qualified name of the constructor as well as the event handler we're assigning to `ondataselected`. But remember that functions referenced in markup like this have to be marked for strict processing. The constructor is automatically marked through `WinJS.Class.define`, but the event handler needs extra treatment: we place the function in a namespace (to make it globally visible) and use `WinJS.UI.eventHandler` to do the marking:

```
WinJS.Namespace.define("CalendarDemo", {
    dateselected: WinJS.UI.eventHandler(function (e) {
        document.getElementById("message").innerText = JSON.stringify(e.detail) + " selected";
    })
});
```

Again, if you forget to mark the function in this way, the control won't be instantiated at all. (Remove the `WinJS.UI.eventHandler` wrapper to see this.)

To demonstrate creating a control outside of markup, the control on the right of Figure 4-9 is created as follows, within the *calendar2* `div`:

```
//Since we're creating this calendar in code, we're independent of WinJS.UI.processAll.
var element = document.getElementById("calendar2");

//Since we're providing an element, this will be automatically added to the DOM
var calendar2 = new Controls.Calendar(element);

//Since this handler is not part of markup processing, it doesn't need to be marked
calendar2.ondateselected = function (e) {
    document.getElementById("message").innerText = JSON.stringify(e.detail) + " selected";
}
```

There you have it!

Custom Controls in Blend

Blend is an excellent design tool for working with controls directly on the artboard, so you might be wondering how custom controls integrate into that story.

First, since custom controls are just elements in the DOM, Blend works with them like all other parts of the DOM. Try loading the Calendar Control Demo into Blend to see for yourself.

---

calls above into a single one.

Next, a control can determine if it's running inside Blend's design mode if the `Windows.ApplicationModel.DesignMode.designModeEnabled` property is `true`. One place where this is very useful is when handling resource strings. We won't cover resources in full until Chapter 17, but it's important to know here that resource lookup, through `Windows.ApplicationModel.Resources.ResourceLoader` in Blend's design mode, doesn't work the same as when the app is actually running for real. To be blunt, it doesn't work at all and throws exceptions! So you can use the design-mode flag to just provide a suitable default instead of doing the lookup.

For example, one of the early partners I worked with had a method to retrieve a localized URL to their back-end services, which was failing in design mode. Using the design mode flag, then, we just had to change the code to look like this:

```
WinJS.Namespace.define("App.Localization", {
    getBaseUrl: function () {
        if (Windows.ApplicationModel.DesignMode.designModeEnabled) {
            return "www.default-base-service.com";
        } else {
            var resources = new Windows.ApplicationModel.Resources.ResourceLoader();
            var baseUrl = resources.getString("baseUrl");
            return baseUrl;
        }
    }
});
```

Finally, it is possible to have custom controls show up in the Assets tab alongside the HTML elements and the WinJS controls. As of this writing, however, I haven't gotten this to work yet, so I'll have to defer the details to a later revision of this chapter!

Data Binding

As I mentioned in the introduction to this chapter, the subject of data binding is closely related to controls because it's how you create relationships between properties of data objects and properties of controls (including styles). This way, controls reflect what's happening in the data, which is often exactly what you want to accomplish in your user experience.

I want to start this discussion with a review of data binding in general, for you may be familiar with the concept to some extent, as I was, but unclear on a number of the details. At times, in fact, especially if you're talking to someone who has been working with it for years, data binding seems to become shrouded in some kind of impenetrable mystique. I don't at all count myself among such initiates, so I'll try to express the concepts in prosaic terms.

The general idea of data binding is again to connect or "bind" properties of two different objects together, typically a data object and a UI object, which we can generically refer to as a source and a target. A key here is that data binding generally happens between *properties*, not objects.

The binding can also involve converting values from one type into another, such as converting a set

of separate source properties into a single string as suitable for the target. It's also possible to have multiple target objects bound to the same source object or one target bound to multiple source objects. This flexibility is exactly why the subject can become somewhat nebulous, because there are so many possibilities! Still, for most scenarios, we can keep the story simple.

A common data binding scenario is shown in Figure 4-10, where we have specific properties of two UI elements, a `span` and an `img`, bound to properties of a data object. There are three bindings here: (1) the `span.innerText` property is bound to the source.name property; (2) the `img.src` property is bound to the `source.photoURL` property; and (3) the `span.style.color` property is bound to the output of a converter function that changes the `source.userType` property into a color.

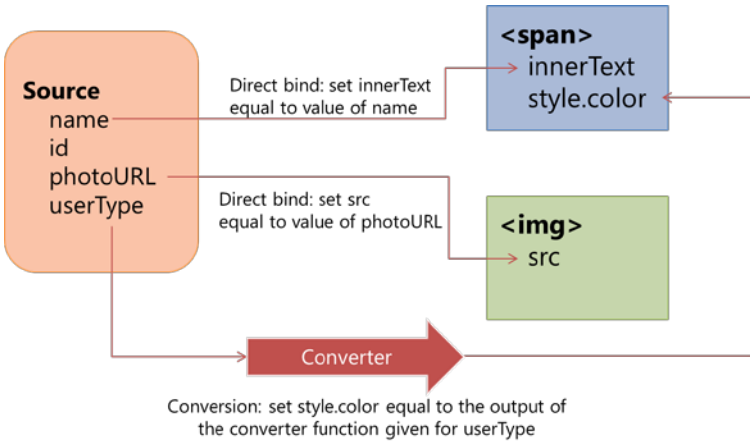
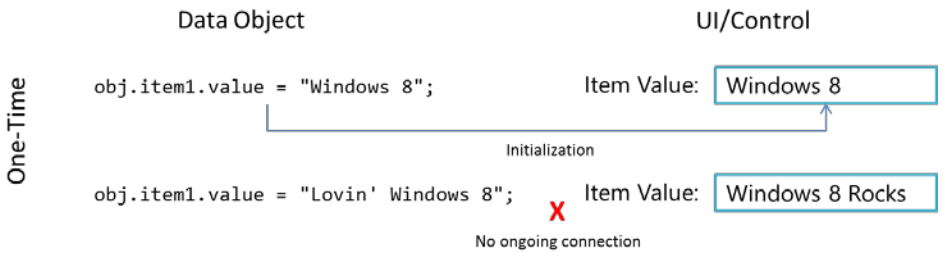


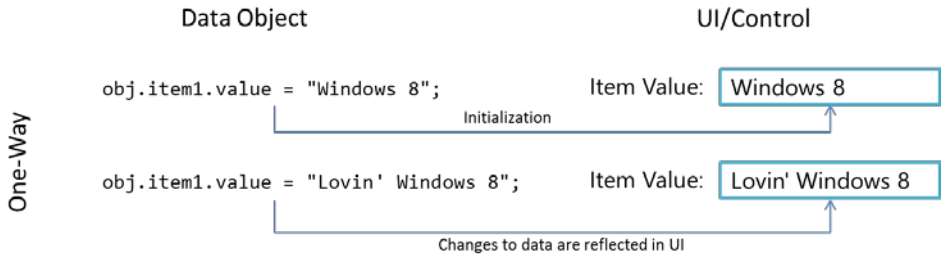
Figure 4-10 A common data-binding scenario between a source data object and two target UI elements, involving two direct bindings and one binding with a conversion function.

How these bindings actually behave at run time then depends on the particular *direction* of each binding, which can be one of the following:

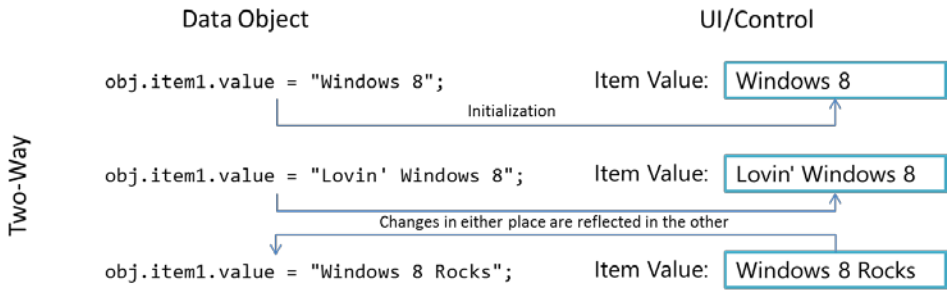
One-time: the value of the source property (possibly with conversion) is copied to the target property at some point, after which there is no further relationship. This is what you automatically do when passing variables to control constructors, for instance, or simply assigning target property values using source properties. (What's useful here is to have a declarative means to make such assignments directly in element attributes.)



One-way: the target object listens for change events on bound source properties so that it can update itself with new values. This is typically used to update a UI element in response to underlying changes in the data. Changes within the target element (like a UI control), however, are not reflected back to the data itself (but can be sent elsewhere as with form submission, which could in turn update the data through another channel).



Two-way: essentially one-way binding in both directions, as the source object also listens to change events from the target object. Changes made within a UI element like a text box are thus saved back in the bound source property, just as changes to the data source property update the UI element. Obviously, there must be some means to not get stuck in an infinite loop; typically, both objects avoid firing another change event if the new value is the same as the existing one.



Data Binding in WinJS

Now that we've seen what data binding is all about, we can see how they can be implemented within a Metro style app. If you like, you can create whatever scheme you want for data binding or use a third-party JavaScript library for the job: it's just about connecting properties of source objects with properties of target objects.

Now, if you're anything like a number of my paternal ancestors, who seemed to wholly despise relying on anyone to do anything they could do themselves (like drilling wells, mining coal, and manufacturing engine parts), you may very well be content with engineering your own data-binding solution. But if you have a more tempered nature like I do (thanks to my mother's side), I'm delighted

when someone is thoughtful enough to create a solution for me. Thus my gratitude goes out to the WinJS team who, knowing of the common need for data binding, created the `WinJS.Binding` API. This supports one-time and one-way binding, both declaratively and procedurally, along with converter functions. At present, WinJS does not provide for two-way binding, but such structures aren't difficult to set up in code, as we'll see.

Within the WinJS structures, multiple target elements can be bound to a single data source. `WinJS.Binding`, in fact, provides for what are called *templates*, basically collections of target elements that are together bound to the same data source. Though we don't recommend it, it's possible to bind a single target element to multiple sources, but this gets tricky to manage properly. A better approach in such cases is to wrap those separate sources into a single object and bind to that instead.

The best way, now, to understand `WinJS.Binding` is to first see look at how we'd write our own binding code and then see the solution that WinJS offers. For these examples, we'll use the same scenario as shown in Figure 4-10, where we have a source object bound to two separate UI elements, with one converter that changes a source property into a color.

One-Time Binding

One-time binding, as mentioned before, is essentially what you do whenever you just assign values to properties of an element. So, given this HTML:

```
<!-- Markup: the UI elements we'll bind to a data object -->
<section id="loginDisplay1">
    <p>You are logged in as <span id="loginName1"></span></p>
    <img id="photo1"></img>
</section>
```

and the following data source object:

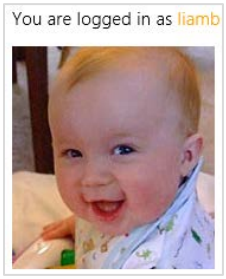
```
var login1 = { name: "liam", id: "12345678",
    photoURL: "http://www.kraigbrockschmidt.com/images/Liam07.png", userType: "kid"};
```

we can bind as follows, also using a converter function in the process:

```
//"Binding" is done one property at a time, with converter functions just called directly
var name = document.getElementById("loginName1");
name.innerText = login1.name;
name.style.color = userTypeToColor1(login1.userType);
document.getElementById("photo1").src = login1.photoURL;

 function userTypeToColor1(type) {
     return type == "kid" ? "Orange" : "Black";
 }
```

This gives the following result, in which I shamelessly publish a picture of my kid:



The code for this can be found in Test 1 of the BindingTests example for this chapter. With WinJS we can accomplish the same thing by using a declarative syntax and a processing function. In markup, we use the attribute `data-win-bind` to map target properties of the containing element to properties of the source object that is given to the processing function, `WinJS.Binding.processAll`.

The value of `data-win-bind` is a string of property pairs. Each pair's syntax is `<source property> : <target property> [<converter>]` where the converter is optional. Each property identifier can use dot notation as needed, and property pairs are separated by a semicolon as shown in the HTML:

```
<section id="loginDisplay2">
    <p>You are logged in as
        <span id="loginName2"
            data-win-bind="innerText: name; style.color: userType Tests.userTypeToColor">
        </span>
    </p>
    <img id="photo2" data-win-bind="src: photoURL"/>
</section>
```

Sidebar: Data-Binding Properties of WinJS Controls

When targeting properties on a WinJS control and not its root (containing) element, the target property names should begin with `winControl`. Otherwise you'll be binding to nonexisting properties on the root element. When using `winControl`, the bound property serves the same purpose as specifying a fixed value in `data-win-options`. For example, the markup used earlier in the "Example: WinJS.UI.Rating Control" section could use data binding for its `averageRating` and `userRating` properties as follows (assuming `myData` is an appropriate source):

```
<div id="rating1" data-win-control="WinJS.UI.Rating"
    data-win-options="{onchange: changeRating}">
    data-win-bind="{winControl.averageRating: myData.average,
      winControl.userRating: myData.rating}">
</div>
```

Anyway, assuming we have a data source as before:

```
var login2 = { name: "liamb", id: "12345678",
    photoURL: "http://www.kraigbrockschmidt.com/images/Liam07.png", userType: "kid"};
```

We convert the markup to actual bindings using `WinJS.Binding.processAll`:

```
//processAll scans the element's tree for data-win-bind, using given object as data context
WinJS.Binding.processAll(document.getElementById("loginDisplay2"), login2);
```

This code, Test2 in the example, produces the same result as Test 1. The one added bit here is that we need to define the converter function so that it's globally accessible and marked for processing. This can be accomplished with a namespace that contains a function (actually called an initializer, as we'll discuss in the "Binding Initializers" section near the end of this chapter) created by `WinJS.Binding.converter`:

```
//Use a namespace to export function from the current module so WinJS.Binding can find it
WinJS.Namespace.define("Tests", {
    userTypeToColor: WinJS.Binding.converter(function (type) {
        return type == "kid" ? "Orange" : "Black";
    })
});
```

As with control constructors defined with `WinJS.Class.define`, `WinJS.Binding.converter` automatically marks the functions it returns as safe for processing.

We could also put the data source object and applicable converters within the same namespace.[30] For example (in Test 3), if we placed our `login` data object and the `userTypeToColor` function in a `LoginData` namespace, the markup and code would look like this:

```
<span id="loginName3"
    data-win-bind="innerText: name; style.color: userType LoginData.userTypeToColor">
</span>


WinJS.Binding.processAll(document.getElementById("loginDisplay3"), LoginData.login);

WinJS.Namespace.define("LoginData", {
        login : {
            name: "liamb", id: "12345678",
            photoURL: "http://www.kraigbrockschmidt.com/images/Liam07.png",
            userType: "kid"
        },

        userTypeToColor: WinJS.Binding.converter(function (type) {
            return type == "kid" ? "Orange" : "Black";
        })
    });
```

In summary, for one-time binding `WinJS.Binding` simply gives you a declarative syntax to do exactly what you'd do in code, with a lot less code. Because it's all just some custom markup and a processing function, there's no magic here, though such useful utilities are magical in their own way! In fact, the code here is really just one-way binding without having the source fire any change events.

---

[30] More commonly, converters would be part of a namespace in which applicable UI elements are defined, because they're more specific to the UI than to a data source.

We'll see how to do that with `WinJS.Binding.as` in a moment after a couple more notes.

First, `WinJS.Binding.processAll` is actually an async function that returns a promise. Any completed handler given to its `done` method will be called when the processing is finished, if you have additional code that's depending on that state. Second, you can call `WinJS.Binding.processAll` more than once on the same target element, specifying a different source object (data context) each time. This won't replace any existing bindings, mind you—it just adds new ones, meaning that you could end up binding the same target property to more than one source, which could become a big mess. So again, a better approach is to combine those sources into a single object and bind to that, using dot notation to identify nested properties.

One-Way Binding

The goal for one-way binding is, again, to update a target property, typically in a UI control, when the bound source property changes. That is, one-way binding means to effectively repeat the one-time binding process whenever the source property changes.

In the code we saw above, if we changed `login.name` after calling `WinJS.Binding.processAll`, nothing will happen in the output controls. So how can we automatically update the output?

Generally speaking, this requires that the data source maintains a list of *bindings*, where each binding could describe a source property, a target property, and a converter function. The data source would also need to provide methods to manage that list, like *addBinding*, *removeBinding*, and so forth. Thirdly, whenever one of its bindable (or *observable*) properties changes it goes through its list of bindings and updates any affected target property accordingly.

These requirements are quite generic; you can imagine that their implementation would pretty much join the ranks of classic boilerplate code. So, of course, WinJS provides just such an implementation! In this context, sources are called *observable objects*, and the function `WinJS.Binding.as` wraps any arbitrary object with just such a structure. (It's a no-op for nonobjects.) Conversely, `WinJS.Binding.unwrap` removes that structure if there's a need. Furthermore, `WinJS.Binding.define` creates a constructor for observable objects around a set of properties (described by a kind of empty object that just has property names). Such a constructor allows you to instantiate source objects dynamically, as when processing data retrieved from an online service.

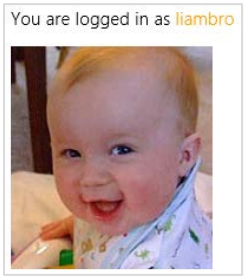
So let's see some code. Going back to the last example above (Test 3), anytime before or after `WinJS.Binding.processAll` we can take the `LoginData.login` object and make it observable as follows:

```
var loginObservable = WinJS.Binding.as(LoginData.login)
```

This is actually all we need to do—with everything else the same as before, we can now change a bound property within the `loginObservable` object:

```
loginObservable.name = "liambro";
```

This will update the target property:



Here's how we'd then create and use a reusable class for an observable object (Test 4 in the BindingTests example). Notice the object we pass to `WinJS.Binding.define` contains property names, but no values (they'll be ignored):

```
WinJS.Namespace.define("LoginData", {
    //...

    //LoginClass becomes a constructor for bindable objects with the specified properties
    LoginClass: WinJS.Binding.define({name: "", id: "", photoURL: "", userType: "" }),
});
```

With that in place, we can create an instance of that class, initializing desired properties. In this example, we're using a different picture and leading `userType` uninitialized:

```
var login4 = new LoginData.LoginClass({ name: "liamb",
    photoURL: "http://www.kraigbrockschmidt.com/images/Liam08.jpg" });
```

Binding to this `login` object, we'd see that the username initially comes out black.

```
//Do the binding (initial color of name would be black)
WinJS.Binding.processAll(document.getElementById("loginDisplay"), login4);
```

Updating the `userType` property in the source (as below) would then cause an update the color of the target property, which happens through the converter automatically:

```
login4.userType = "kid";
```

Implementing Two-Way Binding

To implement two-way binding, the process is straightforward:

1. Add listeners to the appropriate UI element events that relate to bound data source properties.
2. Within those handlers, update the data source properties.

The data source should be smart enough to know when the new value of the property is already the same as the target property, in which case it shouldn't try to update the target lest you get caught in a loop. The observable object code that WinJS provides does this type of check for you.

To see an example of this, refer to the Declarative Binding sample in the SDK, which listens for the `change` event on text boxes and updates values in its source accordingly.

Additional Binding Features

If you take a look at the WinJS.Binding reference in the documentation, you'll see a number of other goodies in the namespace. Let me briefly outline the purpose of these.

If you already have a defined class (from `WinJS.Class.define`) and want to make it observable, use `WinJS.Class.mix` as follows:

```
var MyObservableClass = WinJS.Class.mix(MyClass, WinJS.Binding.mixin,
    WinJS.Binding.expandProperties(MyClass));
```

`WinJS.Binding.mixin` here contains a standard implementation of the binding functions that WinJS expects. `WinJS.Binding.expandProperties` creates an object whose properties match those in the given object (the same names), with each one wrapped in the proper structure for binding. Clearly, this type of operation is useful only when doing a mix, and it's exactly what `WinJS.Binding.define` does with the oddball, no-values object we give to it.

If you remember from a previous section, one of the requirements for an observable object is that is contains methods to manage a list of bindings. An implementation of such methods is contained in the `WinJS.Binding.observableMixin` object. Its methods are:

- `bind` Saves a binding (property name and a function to invoke on change).
- `unbind` Removes a binding created by bind.
- `Notify` Goes through the bindings for a property and invokes the functions associated with it. This is where WinJS checks that the old and new values are actually different and where it also handles cases where an update for the same target is already in progress.

Building on this is yet another mixin, `WinJS.Binding.dynamicObservableMixin` (which is what `WinJS.Binding.mixin` is), which adds methods for managing source properties as well:

- `setProperty` Updates a property value and notifies listeners if the value changed.
- `updateProperty` Like `setProperty`, but returns a promise that completes when all listeners have been notified (the result in the promise is the new property value).

- `getProperty` Retrieves a property value as an observable object itself, which makes it possible to bind within nested object structures (`obj1.obj2.prop3`, etc.).
- `addProperty` Adds a new property to the object that is automatically enabled for binding.
- `removeProperty` Removes a property altogether from the object.

Why would you want all of these? Well, there are some creative uses. You can call `WinJS.Binding.bind`, for example, directly on any observable source when you want to hook up another function to a source property. This is like adding event listeners for source property changes, and you can have as many listeners as you like. This is helpful for wiring up two-way binding, and it doesn't in any way have to be related to manipulating UI. The function just gets called on the property change. This could be used to autosync a back-end service with the source object.

The Declarative Binding sample in the SDK (again, found here) also shows calling `bind` with an object as the second parameter, a form that allows for binding to nested members of the source. The syntax looks like this: `bind(rootObject, { property: { sub-property: function(value) { ... } } })`—whatever matches the source object. With such an object in the second parameter, bind will make sure to invoke all the functions assigned to the nested properties. In such a case, the return value of `bind` is an object with a `cancel` method that will clear out this complex binding.

The `notify` method, for its part, is something you can call directly to trigger notifications. This is useful with additional bindings that don't necessarily depend on the values themselves, just the fact that they changed. The major use case here is to implement computed properties—ones that change in response to another property value changing.

The system here also has some intelligent handling of multiple changes to the same source property. After the initial binding, further change notifications are asynchronous and multiple pending changes to the same property are coalesced. So, if in our example we made several changes to the name property in quick succession:

```
login.name = "Kenichiro";
login.userType = "Josh";
login.userType = "Chris";
```

only one notification for the last value would be sent and that would be the value that shows up in bound targets.

Finally, here are a few more functions hanging off `WinJS.Binding`:

- `oneTime` A function that just loops through the given target (destination) properties and sets them to the value of the associated source properties. This function can be used for true one-time bindings, as is necessary when binding to WinRT objects. It can also be used directly as an initializer within `data-win-bind` if the source is a WinRT object.

- `defaultBind` A function that does the same as `oneTime` but establishes one-way binding between all the given properties. This also serves as the default initializer for all relationships in `data-win-bind` when specific initializer isn't specified.
- `declarativeBind` The actual implementation of `processAll`. (The two are identical.) In addition to the common parameters (the root target element and the data context), it also accepts a `skipRoot` parameter (if true, processing does not bind properties on the root element, only its children, which is useful for template objects) and `bindingCache` (an optimization for holding the results of parsing the data-win-bind expression when processing template objects).

Binding Initializers

In our earlier examples we saw some uses of converter functions that turn some bit of source data into whatever a target property expects. But the function you specify in `data-win-bind` is more properly called an *initializer* because in truth it's only ever called once.

Say what? Aren't converters used whenever a bound source property gets copied to the target? Well, yes, but we're actually talking about two different functions here. Look carefully at the code structure for the `userTypeToColor` function we used earlier:

```
userTypeToColor: WinJS.Binding.converter(function (type) {
    return type == "kid" ? "Orange" : "Black";
})
```

The `userTypeToColor` function itself is an *initializer*. When it's called—once and only once—its *return value* from `WinJS.Binding.converter` is the *converter* that will then be used for each property update. That is, the real converter function is not `userTypeToColor`—it's actually a structure that wraps the anonymous function given to `WinJS.Binding.converter`.

Under the covers, `WinJS.Binding.converter` is actually using `bind` to set up relationships between source and target properties, and it inserts your anonymous conversion function into those relationships. Fortunately, you generally don't have to deal with this complexity and can just provide that conversion function, as shown above.

Still, if you want a raw example, check out the Declarative Binding sample again, as it shows how to create a converter for complex objects directly in code without using `WinJS.Binding.converter`. In this case, that function needs to be marked as safe for processing if it's referenced in markup. Another function, `WinJS.Binding.initializer` exists for that exact purpose; the return value of `WinJS.Binding.converter` passes through that same method before it comes back to your app.

Binding Templates and Lists

Did you think we'd exhausted `WinJS.Binding` yet? Well, my friend, not quite! There are two more pieces to this rich API that lead us directly into the next chapter. (Now you know the real reason I put this entire section where I did!). The first is `WinJS.Binding.List`, a bindable *collection* data source

that—not surprisingly—is very useful when working with collection controls.

`WinJS.Binding.Template` is also a unique kind of custom control. In usage, as you can again see in the Declarative Binding sample, you declare an element (typically a `div`) with `data-win-control = "WinJS.Binding.Template"`. In that same markup, you specify the template's contents as child elements, any of which can have `data-win-bind` attributes. What's unique is that when `WinJS.UI.process` or `processAll` hits this markup, it instantiates the template and actually pulls everything but the root element *out* of the DOM entirely. So what good is it then?

Well, once that template exists, anyone can call its `render` method to create a copy of that template within some other element, using some data context to process any `data-win-bind` attributes therein (typically skipping the root element itself, hence that `skipRoot` parameter in the `WinJS.Binding.declarativeBind` method). Furthermore, rendering a template multiple times into the same element creates multiple siblings, each of which can have a different data source.

Ah ha! Now you can start to see how this all makes perfect sense for collection controls and collection data sources. Given a collection data source and a template, you can iterate over that source and render a copy of the template for each source item into some other element. Add a little navigation or layout within that containing element and voila! You have the beginnings of what we know as the `WinJS.UI.FlipView` and `WinJS.UI.ListView` controls, as we'll explore next.

What We've Just Learned

- The overall control model for HTML and WinJS controls, where every control consists of declarative markup, applicable CSS, and methods, properties, and events accessible through JavaScript.
- Standard HTML controls have dedicated markup; WinJS controls use `data-win-control` attributes, which are processed using `WinJS.UI.process` or `WinJS.UI.processAll`.
- Both types of controls can also be instantiated programmatically using `new` and the appropriate constructor, such as `Button` or `WinJS.UI.Rating`.
- All controls have various options that can be used to initialize them. These are given as specific attributes in HTML controls and within the `data-win-options` attribute for WinJS controls.
- All controls have standard styling as defined in the WinJS stylesheets: ui-light.css and ui-dark.css. Those styles can be overridden as desired, and some style classes, like `win-backbutton`, are used to style a standard HTML control to look like a Windows-specific control.
- Metro style apps have rich styling capabilities for both HTML and WinJS controls alike. For HTML controls, `-ms-*`-prefixed pseudo-selectors allow you to target specific pieces of those controls. For WinJS controls, specific parts are styled using `win-*` classes that you can override.
- Custom controls are implemented in the same way WinJS controls are, and WinJS provides standard implementations of methods like `addEventListener`.

- WinJS provides declarative data-binding capabilities for one-time and one-way binding, which can employ conversion functions. It even provides the capability to create an observable (one-way bindable) data source from any other object.
- WinJS also provides support for bindable collections and templates that can be repeatedly rendered for different source objects into the same containing element, which is the basis for collection controls.

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 key partners on building apps for Windows 8 and bringing knowledge gained in that experience to the wider developer community. His other books include *Inside OLE* (two editions), *Mystic Microsoft*, *The Harmonium Handbook,* and *Finding Focus*. His website is www.kraigbrockschmidt.com.

What do you think of this book?

We want to hear from you!

To participate in a brief online survey, please visit:

microsoft.com/learning/booksurvey

Tell us how well this book meets your needs—what works effectively, and what we can do better. Your feedback will help us continually improve our books and learning resources for you.

Thank you in advance for your input!

Microsoft® Press