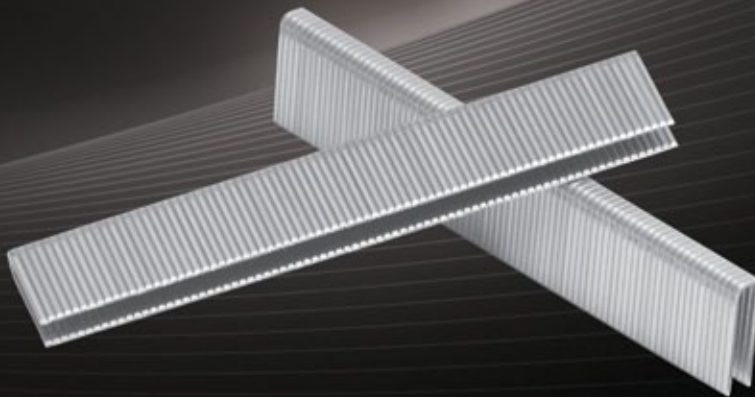


Programming Windows® 8 Apps with HTML, CSS, and JavaScript



Kraig Brockschmidt

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

Copyright © 2012 Microsoft Corporation

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

ISBN: 978-0-7356-7261-1

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

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

Complying with all applicable copyright laws is the responsibility of the user. Without limiting the rights under copyright, no part of this document may be reproduced, stored in or introduced into a retrieval system, or transmitted in any form or by any means (electronic, mechanical, photocopying, recording, or otherwise), or for any purpose, without the express written permission of Microsoft Corporation.

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

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

Acquisitions, Developmental, and Project Editor: Devon Musgrave

Cover: Twist Creative • Seattle

Table of Contents

Introduction 19

Who This Book Is For 20

What You'll Need (Can You Say "Samples"?) 21

A Formatting Note..... 22

Acknowledgements..... 23

Errata & Book Support 24

We Want to Hear from You..... 25

Stay in Touch 25

**Chapter 1: The Life Story of a Windows Store App:
Platform Characteristics of Windows 8..... 26**

Leaving Home: Onboarding to the Windows Store..... 27

Discovery, Acquisition, and Installation..... 30

Playing in Your Own Room: The App Container..... 34

Different Views of Life: View States and Resolution Scaling 37

Those Capabilities Again: Getting to Data and Devices 40

Taking a Break, Getting Some Rest: Process Lifecycle Management..... 43

Remembering Yourself: App State and Roaming 45

Coming Back Home: Updates and New Opportunities 48

And, Oh Yes, Then There's Design 50

Chapter 2: Quickstart 52

A Really Quick Quickstart: The Blank App Template..... 52

 Blank App Project Structure..... 55

QuickStart #1: Here My Am! and an Introduction to Blend for Visual Studio..... 59

 Design Wireframes..... 59

 Create the Markup..... 62

 Styling in Blend..... 64

 Adding the Code 68

 Creating a Map with the Current Location 69

 Oh Wait, the Manifest! 73

Capturing a Photo from the Camera	75
Sharing the Fun!	78
Extra Credit: Receiving Messages from the iframe	81
The Other Templates	82
Fixed Layout Template.....	82
Navigation Template.....	83
Grid Template.....	83
Split Template.....	83
What We've Just Learned	84
Chapter 3: App Anatomy and Page Navigation.....	85
Local and Web Contexts within the App Host.....	86
Referencing Content from App Data: ms-appdata.....	90
Here My Am! with ms-appdata.....	92
Sequential Async Operations: Chaining Promises	94
Error Handling Within Promises: then vs. done.....	96
Debug Output, Error Reports, and the Event Viewer	96
App Activation.....	99
Branding Your App 101: The Splash Screen and Other Visuals.....	99
Activation Event Sequence.....	101
Activation Code Paths.....	103
WinJS.Application Events	105
Extended Splash Screens.....	106
Activation Deferrals	108
App Lifecycle Transition Events and Session State	109
Suspend, Resume, and Terminate.....	109
Basic Session State in Here My Am!	114
Data from Services and WinJS.xhr	116
Handling Network Connectivity (in Brief)	119
Tips and Tricks for WinJS.xhr	120
Page Controls and Navigation	121

WinJS Tools for Pages and Page Navigation	121
The Navigation App Template, PageControl Structure, and PageControlNavigator	123
The Navigation Process and Navigation Styles.....	129
Optimizing Page Switching: Show-and-Hide	130
WinRT Events and removeEventListener	131
Completing the Promises Story.....	133
What We've Just Learned	135
Chapter 4: Controls, Control Styling, and Data Binding	136
The Control Model for HTML, CSS, and JavaScript	137
HTML Controls	138
WinJS stylesheets: ui-light.css, ui-dark.css, and win-* styles.....	141
Extensions to HTML Elements	142
WinJS Controls	142
WinJS Control Instantiation.....	144
Strict Processing and processAll Functions.....	145
Example: WinJS.UI.Rating Control.....	146
Example: WinJS.UI.Tooltip Control	147
Working with Controls in Blend.....	149
Control Styling	151
Styling Gallery: HTML Controls.....	153
Styling Gallery: WinJS Controls.....	155
Some Tips and Tricks.....	158
Custom Controls.....	159
Custom Control Examples	161
Custom Controls in Blend.....	164
Data Binding	167
Data Binding in WinJS.....	169
One-Time Binding.....	170
One-Way Binding.....	173
Implementing Two-Way Binding	175

Additional Binding Features	175
Binding Initializers.....	177
Binding Templates and Lists.....	178
What We've Just Learned	178
Chapter 5: Collections and Collection Controls.....	180
Collection Control Basics	181
Quickstart #1: The FlipView Control Sample.....	181
Quickstart #2a: The HTML ListView Essentials Sample	183
Quickstart #2b: The ListView Grouping Sample	186
ListView in the Grid App Project Template.....	191
The Semantic Zoom Control	195
FlipView Features and Styling	198
Data Sources	202
A FlipView Using the Pictures Library.....	202
Custom Data Sources	204
How Templates Really Work	205
Referring to Templates	206
Template Elements and Rendering	206
Template Functions (Part 1): The Basics.....	207
ListView Features and Styling.....	210
When Is ListView the Wrong Choice?.....	210
Options, Selections, and Item Methods.....	212
Styling.....	215
Backdrops.....	216
Layouts and Cell Spanning	216
Optimizing ListView Performance.....	223
Random Access.....	224
Incremental Loading.....	225
Template Functions (Part 2): Promises, Promises!.....	225
What We've Just Learned	232

Chapter 6: Layout.....	233
Principles of Windows Store App Layout.....	234
Quickstart: Pannable Sections and Snap Points	237
Laying Out the Hub	238
Laying Out the Sections	239
Snap Points	240
The Many Faces of Your Display.....	241
View States.....	242
Handling View States.....	245
Screen Size, Pixel Density, and Scaling.....	249
Graphics That Scale Well	252
Adaptive and Fixed Layouts for Display Size.....	253
Fixed Layouts and the ViewBox Control.....	254
Adaptive Layouts	256
Using the CSS Grid.....	258
Overflowing a Grid Cell	260
Centering Content Vertically	260
Scaling Font Size	261
Item Layout	262
CSS 2D and 3D Transforms	263
Flexbox	263
Nested and Inline Grids.....	264
Fonts and Text Overflow	266
Multicolumn Elements and Regions	267
What We've Just Learned	270
Chapter 7: Commanding UI	271
Where to Place Commands.....	272
The App Bar	276
App Bar Basics and Standard Commands.....	278
Command Events	281

App Bar Events and Methods	282
Showing, Hiding, Enabling, and Updating Commands	284
App Bar Styling.....	287
Custom Icons	288
Command Menus	290
Custom App Bars and Navigation Bars.....	291
Flyouts and Menus.....	293
WinJS.UI.Flyout Properties, Methods, and Events	294
Flyout Examples	295
Menus and Menu Commands.....	299
Context Menus.....	301
Message Dialogs.....	303
Improving Error Handling in Here My Am!.....	305
What We've Just Learned	309
Chapter 8: State, Settings, Files, and Documents	311
The Story of State.....	312
Settings and State.....	314
App Data Locations.....	315
AppData APIs (WinRT and WinJS)	317
Settings Containers.....	318
Versioning App State	320
Storage Folders and Storage Files.....	321
The FileIO, PathIO, and WinJS helper classes (plus FileReader)	325
Encryption and Compression.....	326
Using App Data APIs for State Management.....	327
Session State	327
Local and Temporary State.....	328
IndexedDB and Other Database Options	329
Roaming State.....	331
Settings Pane and UI.....	333

Design Guidelines for Settings	334
Populating Commands	336
Implementing Commands: Links and Settings Flyouts	338
Programmatically Invoking Settings Flyouts	341
User Data: Libraries, File Pickers, and File Queries	343
Using the File Picker	344
The File Picker UI	345
The File Picker API (and a Few Friends)	348
Media Libraries	352
Documents and Removable Storage	353
Rich Enumeration with File Queries	354
Here My Am! Update	360
What We've Just Learned	361
Chapter 9: Input and Sensors	363
Touch, Mouse, and Stylus Input	364
The Touch Language, Its Translations, and Mouse/Keyboard Equivalents	366
Edge Gestures	370
CSS Styles That Affect Input	371
What Input Capabilities Are Present?	372
Unified Pointer Events	374
Pointer Capture	377
Gesture Events	378
Multipoint Gestures	383
The Input Instantiable Gesture Sample	384
The Gesture Recognizer	386
Keyboard Input and the Soft Keyboard	388
Soft Keyboard Appearance and Configuration	389
Adjusting Layout for the Soft Keyboard	392
Standard Keystrokes	395
Inking	396

Geolocation.....	398
Sensors.....	401
What We've Just Learned.....	404
Chapter 10: Media	405
Creating Media Elements.....	406
Graphics Elements: Img, Svg, and Canvas (and a Little CSS).....	408
Additional Characteristics of Graphics Elements	411
Some Tips and Tricks.....	412
Img Elements.....	412
Svg Elements.....	413
Canvas Elements.....	413
Video Playback and Deferred Loading.....	416
Disabling Screen Savers and the Lock Screen During Playback.....	418
Video Element Extension APIs.....	419
Applying a Video Effect.....	420
Browsing Media Servers	421
Audio Playback and Mixing.....	421
Audio Element Extension APIs	423
Playback Manager and Background Audio.....	424
The Media Control UI.....	428
Playing Sequential Audio.....	429
Playlists.....	431
Loading and Manipulating Media.....	433
Media File Metadata.....	434
Thumbnails.....	435
Common File Properties	435
Media-Specific Properties	436
Media Properties in the Samples.....	439
Image Manipulation and Encoding.....	442
Transcoding and Custom Image Formats.....	447

Manipulating Audio and Video	448
Transcoding.....	448
Custom Decoders/Encoders and Scheme Handlers.....	451
Media Capture	453
Flexible Capture with the MediaCapture Object.....	454
Selecting a Media Capture Device.....	458
Streaming Media and PlayTo.....	460
Streaming from a Server and Digital Rights Management (DRM).....	461
Streaming from App to Network.....	462
PlayTo	463
What We Have Learned	466
Chapter 11: Purposeful Animations	468
Systemwide Enabling and Disabling of Animations.....	470
The WinJS Animations Library	471
Animations in Action	474
CSS Animations and Transitions.....	479
The Independent Animations Sample.....	483
Rolling Your Own: Tips and Tricks.....	485
What We've Just Learned	490
Chapter 12: Contracts.....	491
Share	493
Source Apps.....	495
Sharing Multiple Data Formats	499
Custom Data Formats: schema.org	499
Deferrals and Delayed Rendering	500
Target Apps	502
Long-Running Operations	508
Quicklinks.....	510
The Clipboard	512
Search.....	514

Search in the App Manifest and the Search Item Template.....	516
Basic Search and Search Activation.....	517
Providing Query Suggestions.....	520
Providing Result Suggestions.....	524
Type to Search.....	525
Launching Apps: File Type and URI Scheme Associations.....	525
File Activation.....	527
Protocol Activation.....	529
File Picker Providers.....	530
Manifest Declarations.....	531
Activation of a File Picker Provider.....	533
File Open Provider: Local File.....	535
File Open Provider: URI.....	537
File Save Provider: Save a File.....	538
File Save Provider: Failure Case.....	539
Cached File Updater.....	539
Updating a Local File: UI.....	542
Updating a Remote File: UI.....	544
Update Events.....	545
Contacts.....	548
Using the Contact Picker.....	551
Contact Picker Providers.....	553
What We've Just Learned.....	556
Chapter 13: Tiles, Notifications, the Lock Screen, and Background Tasks	557
Alive with Activity: A Visual Tour.....	558
The Four Sources of Updates and Notifications.....	568
Tiles, Secondary Tiles, and Badges.....	570
Secondary Tiles.....	571
Creating Secondary Tiles.....	572
App Activation From a Secondary Tile.....	574

Managing Secondary Tiles.....	575
Basic Tile Updates.....	576
Choosing a Tile Template.....	577
Creating the Payload, Method 1: Populating Template Content.....	580
Creating the Payload, Method 2: XML Strings.....	581
Creating the Payload, Method 3: The Notifications Extensions Library.....	581
Using Local and Web Images.....	582
Branding.....	584
Cycling, Scheduled, and Expiring Updates	585
Badge Updates	587
Periodic Updates	590
Web Services for Updates.....	592
Using the Localhost.....	595
Windows Azure	596
Toast Notifications.....	599
Creating Basic Toasts.....	600
Butter and Jam: Options for Your Toast	602
Tea Time: Scheduled Toasts.....	604
Toast Events and Activation.....	606
Push Notifications and the Windows Push Notification Service.....	606
Requesting and Caching a Channel URI (App).....	608
Managing Channel URIs (Service).....	610
Sending Updates and Notifications (Service).....	610
Raw Notifications (Service).....	612
Receiving Notifications (App).....	612
Debugging Tips.....	614
Windows Azure Toolkit and Windows Azure Mobile Services.....	614
Background Tasks and Lock Screen Apps.....	615
Background Tasks in the Manifest.....	616
Building and Registering Background Task.....	618

Conditions	619
Tasks for Maintenance Triggers	620
Tasks for System Triggers (Non-Lock Screen)	622
Lock Screen–Dependent Tasks and Triggers	624
Debugging Background Tasks	627
What We’ve Just Learned (Whew!)	628
Chapter 14: Networking.....	630
Network Information and Connectivity.....	631
Network Types in the Manifest.....	631
Network Information (the Network Object Roster)	632
The ConnectionProfile Object.....	634
Connectivity Events.....	634
Cost Awareness.....	636
Running Offline.....	639
XmlHttpRequest	642
Background Transfer.....	643
Basic Downloads	644
Basic Uploads	647
Breaking Up Large Files	648
Multipart Uploads	649
Providing Headers and Credentials.....	652
Setting Cost Policy.....	652
Grouping Transfers.....	653
Suspend, Resume, and Restart with Background Transfers	653
Authentication, Credentials, and the User Profile.....	655
The Credential Picker UI.....	656
The Credential Locker	659
The Web Authentication Broker.....	661
Single Sign On	665
Single Sign On with Live Connect	667

The User Profile (and the Lock Screen Image)	668
Encryption, Decryption, Data Protection, and Certificates.....	670
Syndication	671
Reading RSS Feeds	671
Using AtomPub	674
Sockets	675
Datagram Sockets.....	676
Stream Sockets	680
Web Sockets: MessageWebSocket and StreamWebSocket.....	683
The ControlChannelTrigger Background Task	687
Loose Ends (or Some Samples To Go).....	688
What We've Just Learned	689
Chapter 15: Devices and Printing	690
Using Devices.....	691
The XInput API and Game Controllers	692
Enumerating Devices in a Class	696
Windows Portable Devices and Bluetooth Capabilities	698
Near Field Communication and the Proximity API	700
Finding Your Peers (No Pressure!)	702
Advertising a Connection.....	703
Making a Connection	704
Tap to Connect and Tap to Activate	705
Sending One-Shot Payloads: Tap to Share.....	706
Printing Made Easy.....	707
The Printing User Experience.....	708
Print Document Sources.....	711
Providing Print Content and Configuring Options.....	712
What We've Just Learned	715

Chapter 16: WinRT Components: An Introduction.....	716
Choosing a Mixed Language Approach (and Web Workers)	718
Quickstarts: Creating and Debugging Components.....	720
Quickstart #1: Creating a Component in C#.....	721
Quickstart #2: Creating a Component in C++	726
Comparing the Results.....	729
Key Concepts for WinRT Components	731
Implementing Asynchronous Methods.....	733
JavaScript Workers.....	734
Async Basics in WinRT Components.....	737
Arrays, Vectors, and Other Alternatives.....	742
Projections into JavaScript.....	746
Scenarios for WinRT Components.....	748
Higher Performance	748
Access to Additional APIs.....	750
Obfuscating Code and Protecting Intellectual Property.....	752
Library Components	753
Concurrency	753
What We've Just Learned.....	754
Chapter 17: Apps for Everyone: Accessibility, World-Readiness, and the Windows Store	755
Your App, Your Business.....	757
Side Loading.....	758
Planning: Can the App Be a Windows Store App?.....	760
Planning for Monetization (or Not)	761
Free Apps	762
Ad-Supported Apps	763
Paid Apps and Trial Versions.....	764
In-App Purchases	766
Revenue Sharing and Custom Commerce for In-App Purchases.....	767

The Windows Store APIs.....	768
The CurrentAppSimulator Object	770
Trial Versions and App Purchase.....	774
Listing and Purchasing In-App Products.....	776
Receipts.....	780
Accessibility.....	781
Screen Readers and Aria Attributes.....	784
The ARIA Sample.....	785
Handling Contrast Variations	788
CSS Styling for High Contrast	790
High Contrast Resources	793
Scale + Contrast = Resource Qualifiers.....	794
High Contrast Tile and Toast Images	795
World Readiness and Localization.....	795
Globalization	797
User Language and Other Settings.....	798
Formatting Culture-Specific Data and Calendar Math	801
Sorting and Grouping.....	803
Fonts and Text Layout	803
Preparing for Localization	805
Part 1: Separating String Resources.....	805
Part 2: Structuring Resources for the Default Language.....	813
Creating Localized Resources: The Multilingual App Toolkit.....	816
Testing with the Pseudo Language.....	821
Localization Wrap-Up	823
Releasing Your App to the World.....	824
Promotional Screenshots, Store Graphics, and Text Copy	824
Testing and Pre-Certification Tools.....	825
Onboarding and Working through Rejection	827
App Updates.....	827

Getting Known: Marketing, Discoverability, and the Web.....	828
Connecting Your Website	829
Final Thoughts: Qualities of a Rock Star App.....	829
What We've Just Learned	831
About the Author	833
Survey Page	834

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 throughout the course of this book. Our primary focus, however, will be on the reimagined *developer* experience.

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

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

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

rather than hundreds or thousands. It would also enable direct native app development in multiple programming languages. This meant that new operating system capabilities would surface to those developers without having to wait for an update to some intermediate framework. It also meant that developers who had experience in any one of those language choices would find a natural home when writing apps for Windows 8.

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

In December 2010, I was part of the small team who set out to write the very first Windows Store apps using what parts of the new WinRT API had become available. Notepad was the text editor of choice, we built and ran apps on the command line by using abstruse Powershell scripts that required us to manually type out ungodly hash strings, we had no documentation other than oft-incomplete functional specifications, and we basically had no debugger to speak of other than the tried and true `window.alert` and `document.writeIn`. 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 apps for Windows 8, the process was becoming far more enjoyable and exceedingly 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 app in a matter of weeks. I'm hoping that this ebook, along with the extensive resources on <http://dev.windows.com>, will help you to accomplish exactly that and to reimagine your own designs.

Who This Book Is For

This book is about writing Windows Store apps using HTML5, CSS3, and JavaScript. Our primary focus will be on applying these web technologies within the Windows 8 platform, where there are unique considerations, and not on exploring the details of those web technologies themselves. For the most part, then, I'm assuming that you're already at least somewhat conversant with these standards. We will cover some of the more salient areas like the CSS grid, which is central to app layout, but otherwise I trust that you're capable of finding appropriate references for most everything else.

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 some core platform features. On the other hand, you probably also want to make the best app you can, one that performs really well and that takes advantage of the full extent of the platform. Toward this end, I've also endeavored to make this book comprehensive, helping you at least be aware of what's possible and where optimizations can be made. (Note, though, that the Store itself is discussed in Chapter 17.)

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

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

To work through this book, you should have Windows 8 installed on your development machine, along with the Windows SDK and tools. All the tools, along with a number of other resources, are listed on [Developer Downloads for programming Windows Store Apps](#). You'll specifically need Microsoft Visual Studio Express 2012 for Windows 8. We'll also acquire other tools along the way as we need them in this ebook. (Note that for all the screenshots in this book, I switched Visual Studio from its default "dark" color theme to the "light" theme, as the latter works better against a white page.)

Also be sure to download the "Sample app pack" listed on this page, or visit [Windows 8 app samples](#) and specifically download the SDK's JavaScript samples. We'll be drawing from many—if not most—of these samples in the chapters ahead, pulling in bits of their source code to illustrate how many different tasks are accomplished.

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

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

the companion content for this book, which you can download at

<http://go.microsoft.com/fwlink/?Linkid=270057>

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

Beyond all this, you’ll find that the [Windows 8 samples gallery](#) as well as the [Visual Studio sample gallery](#) also lets you search and browse additional projects that have been contributed by other developers—perhaps also you! (On the Visual Studio site, by the way, be sure to filter on Windows Store apps as the gallery covers all Microsoft platforms.) And of course, there will be many more developers who share projects on their own.

In this book I occasionally refer to posts on the [Windows 8 App Developer blog](#), which is a good resource to follow. I also recommend following the [Windows Store for Developers blog](#) for any announcements related to what is effectively your place of business. And if you’re interested in the Windows 8 backstory—that is, how Microsoft approached this whole process of reimagining the operating system—check out the [Building Windows 8 blog](#).

A Formatting Note

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

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

Acknowledgements

In many ways, this isn't *my* book—that is, it's not an account of my own experiences and opinions about writing apps for Windows 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 again the best set of samples I've ever seen for a platform. Indeed, the words in some sections come directly from conversations I've had with the people who designed and developed a particular feature. I'm grateful for their time, and I'm delighted to give them a voice through which they can share their passion for excellence with you.

A number of individuals deserve special mention for their long-standing support of this project. First to Chris Sells, with whom I co-authored the earliest versions of this book and who is now leading development efforts at Telerik. 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, who put in many long hours editing my many long chapters, many times over. My direct teammates, Kyle Marsh, Todd Landstad, Shai Hinitz, Patrick Dengler, Lora Heiny, Leon Braginski, and Joseph Ngari have also been invaluable in sharing what they've learned in working with real-world partners. A special thanks goes to Kenichiro Tanaka of Microsoft Japan, for always being the first one to return a reviewed chapter to me and for joyfully researching different areas of the platform whenever I asked. Many bows to you, my friend! Nods also to others in our international Windows Ecosystem teams who helped with localizing the Here My Am! app for Chapter 17: Gilles Peingné, Sam Chang, Celia Pipó Garcia, Juergen Schwertl, Maaten Van De Dospoort, and Li-Qun Jia (plus Shai Hinitz on Hebrew).

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

Shakil Ahmed	Scott Dickens	Kishore Kotteri	Daniel Oliver	Sam Spencer
Chris Anderson	Tyler Donahue	Victoria Kruse	Jason Olson	Ben Srouer
Erik Anderson	Brendan Elliott	Nathan Kuchta	Elliot H Omiya	Adam Stritzel
Axel Andrejs	Matt Esquivel	Elmar Langholz	Larry Osterman	Shijun Sun
Tarek Ayna	David Fields	Bonny Lau	Rohit Pagariya	Sou Suzuki
Art Baker	Erik Fortune	Travis Leithead	Ankur Patel	Simon Tao
Adam Barrus	Jim Galasyn	Chantal Leonard	Harry Pierson	Henry Tappen
Megan Bates	Gavin Gear	Cameron Lerum*	Steve Proteau	Chris Tavares
Tyler Beam	Derek Gephard	Brian LeVee	Hari Pulapaka	David Tepper

Ben Betz	Marcelo Garcia Gonzalez	Jianfeng Lin	Arun Rabinar	Sara Thomas
Johnny Bregar	Sunil Gottumukkala	Tian Luo	Matt Rakow	Ryan Thompson
John Brezak	Scott Graham	Sean Lyndersay	Ramu Ramanathan	Bill Ticehurst
John Bronskill	Ben Grover	David Machaj	Ravi Rao	Stephen Toub
Jed Brown	Paul Gusmorino	Mike Mastrangelo	Brent Rector	Tonu Vanatalu
Vincent Celie	Rylan Hawkins	Jordan Matthiesen	Ruben Rios	Jeremy Viegas
Raymond Chen	John Hazen	Ian McBurnie	Dale Rogerson	Nick Waggoner
Rian Chung	Jerome Holman	Jesse McGatha	Nick Rotondo	David Washington
Arik Cohen	Scott Hoogerwerf	Matt Merry	David Rousset	Sarah Waskom
Justin Cooperman	Stephen Hufnagel	Markus Mielke	George Roussos	Marc Wautier
Michael Crider	Sean Hume	Pavel Minaev	Jake Sabulsky	Josh Williams
Priya Dandawate	Mathias Jourdain	John Morrow	Perumaal Shanmugam	Lucian Wischik
Darren Davis	Damian Kedzierski	Feras Moussa	Edgar Ruiz Silva	Kevin Michael Woley
Jack Davis	Suhail Khalid	John Mullaly	Karanbir Singh	Charing Wong
Ryan Demopoulos	Daniel Kitchener	Jan Nelson*	Peter Smith	Michael Ziller

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

I would also like to bid adieu to the extra pounds that have accompanied my body while I've been sitting at a computer far more than I should! I'm sure you're looking forward to a resumption in our more usual fitness routines as I am.

Finally, special hugs to my wife Kristi and our young son Liam (now six), 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 ebook and its companion content. 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.

Support for developers, however, can be found on the Windows Developer Center's [support section](#), especially in the [Building Windows Store apps with HTML5/JavaScript forum](#). There is also an active community on Stack Overflow for the [windows8](#) and [winrt](#) tags.

We Want to Hear from You

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

<http://aka.ms/tellpress>

The survey is short, and we read every one of your comments and ideas. Thanks in advance for your input!

Stay in Touch

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

Chapter 1

The Life Story of a Windows Store App: Platform Characteristics of Windows 8

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

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

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

Bottom line: it's a hard choice.

Fortunately, Windows 8 provides what I personally think is a brilliant solution for apps. Early on, the Windows team set out to solve the problem of making native capabilities—the system API, in other words—*directly* available to *any* number of programming languages, including JavaScript. This is what's known as the Windows Runtime API, or just *WinRT* for short.

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

The Windows team also made it possible to write native apps that employ a variety of presentation

technologies, including DirectX, XAML, and, in the case of apps written in JavaScript, HTML5 and CSS3.

This means that Windows gives you—a developer already versed in HTML, CSS, and JavaScript standards—the ability to *use what you know* to write fully native Windows 8 apps using the WinRT API and still utilize web content! These 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!

It also means that you'll be able to leverage existing investments in JavaScript libraries and CSS template repositories: writing a native app doesn't force you to switch frameworks or engage in expensive porting work.

That said, it is also possible to use multiple languages to write an app, leveraging the dynamic nature of JavaScript for app logic while leveraging languages like C# and C++ for more computationally intensive tasks. (See "Sidebar: Mixed Language Apps" later in this chapter.)

Throughout this book we'll explore how to leverage what you know of standards-based web technologies to build great Windows 8 apps. In the next chapter we'll focus on the basics of a working app and the tools used to build it. Then we'll look at fundamentals like the fuller anatomy of an app, controls, collections, layout, commanding, state management, and input, followed by chapters on media, animations, contracts through which apps work together, networking, devices, WinRT components (through which you can use other programming languages and the APIs they can access), and the Windows Store (a topic that includes localization and accessibility). There is much to learn.

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

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

Leaving Home: Onboarding to the Windows Store

For Windows Store apps, there's really one port of entry into the world: customers always acquire, install, and update apps through the *Windows Store*. Developers and enterprise users can side-load

apps, but for the vast majority of the people you care about, they go to the Windows Store and nowhere else.

This obviously means that an app—the culmination of your development work—has to get into the Store in the first place. This happens when you take your pride and joy, package it up, and upload it to the Store by using the Store/Upload App 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!

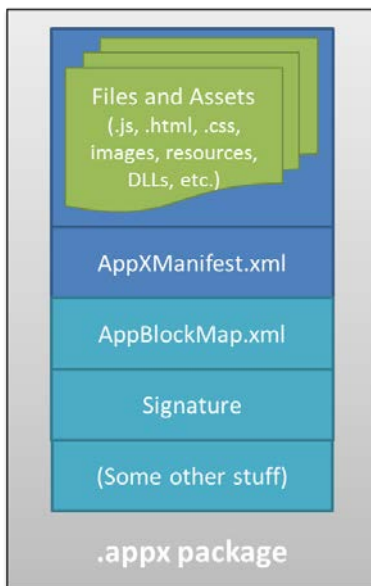


FIGURE 1-1 An appx package is simply a zip file that contains the app’s files and assets, the app manifest, a signature, and a sort of table-of-contents called the blockmap. When uploading an app, the initial signature is provided by Visual Studio; the Windows Store will re-sign the app once it’s certified. The blockmap, for its part, describes how the app’s files are broken up into 64K blocks. In addition to providing certain security functions (like detecting whether a package has been tampered with) and performance optimization, the blockmap is used to determine exactly what parts of an app have been updated between versions so the Windows Store only needs to download those specific blocks rather than the whole app anew. This greatly reduces the time and overhead that a user experiences when acquiring and installing updates.

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.

The upload process will walk you through setting your app's name (which you do ahead of time using the Store > Reserve App Name command in Visual Studio), choosing selling details (including price tier, in-app purchases, and trial periods), providing a description and graphics, and also providing notes to manual testers. After that, your app 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 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 and read the "Your App, Your Business" section of Chapter 17, "Apps for Everyone," before you start writing your app in earnest. Also, take a look at the [Preparing your app for the Store](#) topic on the Windows Developer Center.

Anyway, if your app hits any bumps along the road to certification, you'll get a report back with all the details, such as any violations of the [Windows 8 app certification requirements](#) (part of the [Windows Store agreements](#) section). Otherwise, congratulations—your app is ready for customers!

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.

Sidebar: The Store API and Product Simulator

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

Of course, this begs a question: how can an app test such features before it's even in the Store? The answer is that during development, you use these APIs through the `Windows.ApplicationModel.Store.CurrentAppSimulator` 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 `CurrentAppSimulator` to `CurrentApp` and you're good to go. (If you forget, the simulator will simply fail on a non-developer machine, like those used by the Store testers.)

Discovery, Acquisition, and Installation

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

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

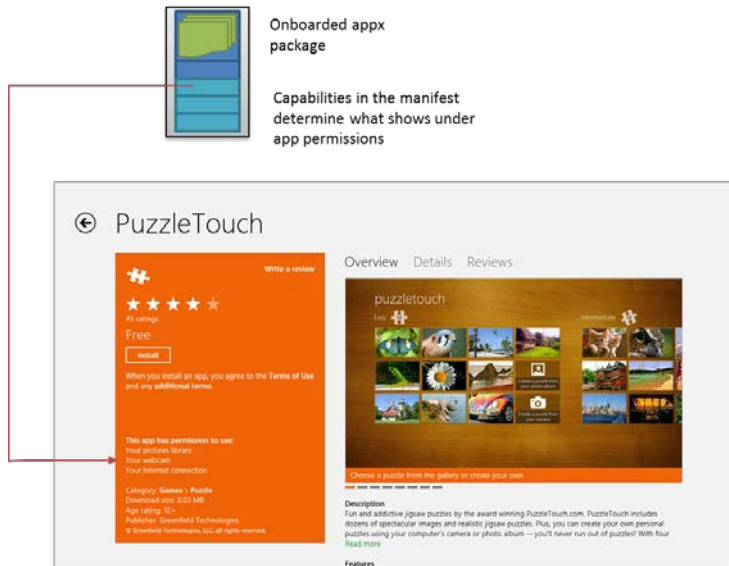


FIGURE 1-2 A typical app page in the Windows Store, where the manifest in the app package determines what appears in the app permissions. Here, for example, PuzzleTouch's manifest declares the *Pictures Library*, *Webcam*, and *Internet (Client)* capabilities.

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 "Sidebar: What is the Windows Library for JavaScript?"). As shown in Figure 1-3, the Windows deployment manager creates a folder for the app, extracts the package contents to that location, creates *appdata* folders (local, roaming, and temp, which the app can freely access, along with settings files for key-value pairs and some other system-managed folders), and does any necessary fiddling with the registry to install the app's tile on the *Start screen*, create file associations, install libraries, and do all those other things that are again described in the manifest. There are no user prompts during this process—especially not those annoying dialogs about reading the licensing agreement!

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.

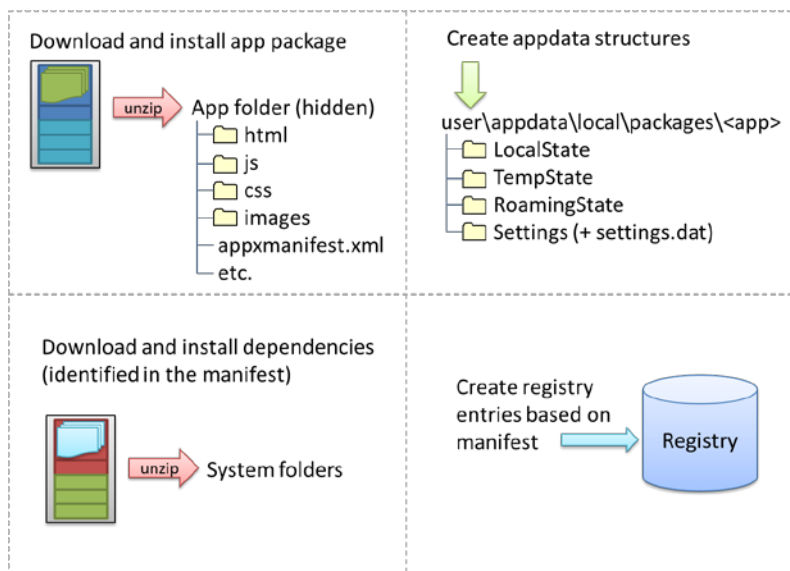


FIGURE 1-3 The installation process for Windows Store apps; the exact sequence is unimportant.

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

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

In any case, the end result of all this is that the app and its necessary structures are wholly ready to awaken on a device, as soon as the user taps a tile on the Start page or launches it through features like Search and Share. And because the system knows about everything that happened during installation, it can also completely reverse the process for a 100% clean uninstall—completely blowing away the appdata folders, for example, and cleaning up anything and everything that was put in the registry. This keeps the rest of the system entirely clean over time, even though the user may be installing and uninstalling hundreds or thousands of apps. We like to describe this like the difference between having guests in your house and guests in a hotel. In your house, guests might eat your food, rearrange the furniture, break a vase or two, feed leftovers to the pets, stash odds and ends in the backs of drawers, and otherwise leave any number of irreversible changes in their wake (and you know desktop apps that

do this, I'm sure!). In a hotel, on the other hand, guests have access only to a very small part of the whole structure, and even if they trash their room, the hotel can clean it out and reset everything as if the guest was never there.

Sidebar: What Is the Windows Library for JavaScript?

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

WinJS, for example, provides an HTML implementation of a number of controls such that they appear as part of the DOM and can be styled with CSS like other intrinsic HTML controls. This is much more natural for developers than having to create an instance of some WinRT class, bind it to an HTML element, and style it through code or some other proprietary markup scheme. Similarly, WinJS provides an animations library built on CSS that embodies the Windows 8 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 Store app written in JavaScript, and it's automatically downloaded and updated as needed when dependent apps are installed. We'll see many of its features throughout this book, though some won't cross our path. In any case, you can always explore what's available through the WinJS section of the [Windows API reference](#).

Sidebar: Third-Party Libraries

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

However, apps can freely use third-party libraries by bringing them into their own app package, provided of course that the libraries use only the APIs available to Windows Store apps.

For example, apps written in JavaScript can certainly use jQuery, Modernizer, Dojo, prototype.js, Box2D, and others, with the caveat that some functionality, especially UI and script injection, might not be supported. Apps can also use third-party binaries, known as WinRT components, that are again included in the app package. Also see "Sidebar: Mixed Language Apps" later in this chapter.

Playing in Your Own Room: The App Container

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

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

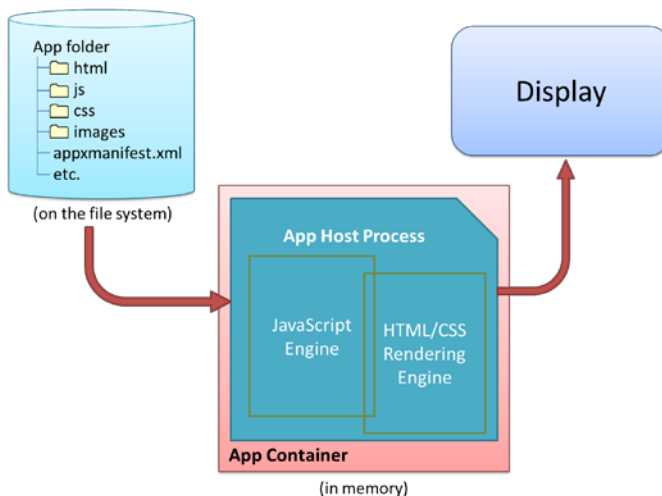


FIGURE 1-4 The app host is an executable (wwahost.exe) that loads, renders, and executes HTML, CSS, and JavaScript, in much the same way that a browser runs a web application.

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

The app host is more or less Internet Explorer 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`, `window.open`, and `window.prompt`. (Try `Windows.UI.Popups.MessageDialog` instead for some of these needs.)
- The engines support additional methods, properties, and even CSS media queries that are specific to being an app as opposed to a website. 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 Store apps.
- The default page of an app written in JavaScript runs in what's called the *local context* wherein JavaScript code has access to WinRT, can make cross-domain XMLHttpRequests, and can access remote media (videos, images, etc.). However, you cannot load remote script (from `http[s]://` sources, for example), and script is automatically filtered out of anything that might affect the DOM and open the app to injection attacks (e.g., `document.write` and `innerHTML` properties).
- Other pages in the app, as well as 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 (though you can use much of WinJS). Web context `iframes` are generally used to host web content on a locally packaged page (like a map control), as we'll see in Chapter 2, "Quickstart," or to load pages that are directly hosted on the web, while not allowing web pages to drive the app. Using such `iframe` elements, in short, allows you to build hybrid apps with both native and web content.

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

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

- All Store apps (other than some that are built into Windows) run within a dedicated environment that cannot interfere with or be interfered with other apps, nor can apps interfere with the system.
- Store apps, by default, get unrestricted read/write access only to their specific appdata folders on the hard drive (local, roaming, and temp). Access to everything else in the file system

(including removable storage) has to go through a broker. This gatekeeper provides access only if the app has declared the necessary capabilities in its manifest and/or the user has specifically allowed it. We'll see the specific list of capabilities shortly.

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

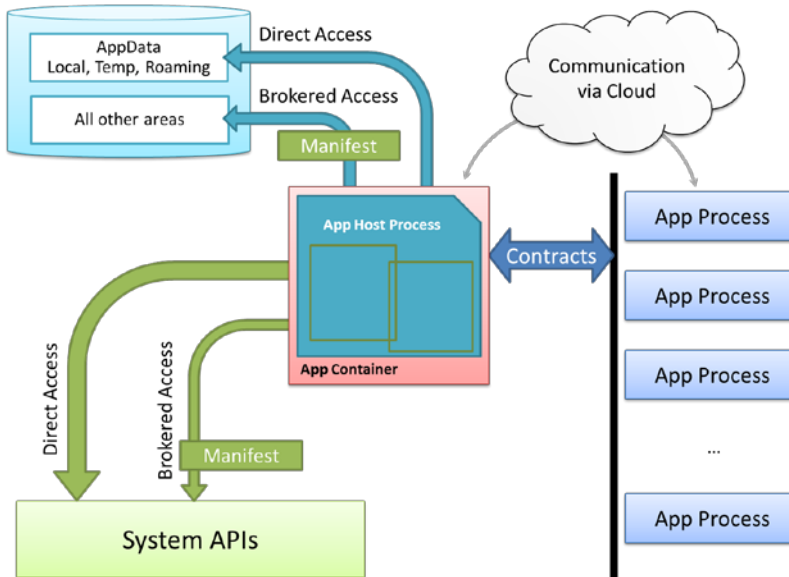


FIGURE 1-5 Process isolation for Windows Store apps.

Sidebar: Mixed Language Apps

Windows Store apps written in JavaScript can only access WinRT APIs directly; apps or libraries written in C#, Visual Basic, and C++ also have access to a subset of Win32 and .NET APIs, as documented on [Win32 and COM for Windows Store apps](#). Unfair? Not entirely, because you can write a *WinRT component* in those other languages that make functionality built with those other APIs available in the JavaScript environment (through the same projection mechanism that WinRT itself uses). Because these components are compiled into binary dynamic-link libraries (DLLs), they will also typically run faster than the equivalent code written in JavaScript and also offer some degree of intellectual property protection (e.g., hiding algorithms).

Such *mixed language apps* thus use HTML/CSS for their presentation layer and some app logic while placing the most performance critical or sensitive code in compiled DLLs. The dynamic nature of JavaScript, in fact, makes it a great language for gluing together multiple components. We'll see more in Chapter 16, "WinRT Components."

Note that mixed language apps are occasionally referred to as "hybrid" apps, but the latter term already has a meaning in the context of mobile and web development. In this book I use "mixed language apps" to avoid confusion.

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) if the user switches away. This avoids having apps that hang during startup and just sit there like a zombie, where often the user can only kill it off by using that most consumer-friendly tool, Task Manager. (Yes, I'm being sarcastic—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 the user fully stay immersed in the app experience. To be more specific, the left and right edge gestures are reserved for the system, whereas the top and bottom are for the app. Swiping up from the top or bottom edges, as you’ve probably seen, brings up the *app bar on the bottom of the screen* where an app places most of its commands, and possibly also a *navigation bar* on the top.

When running full-screen, the user’s device can be oriented in either portrait or landscape, and apps can process various events to handle those changes. An app can also specify a preferred *startup orientation* in the manifest and can also *lock* the orientation when appropriate. For example, a movie player will generally want to lock into landscape mode such that rotating the device doesn’t change the display. We’ll see these layout details in Chapter 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.) The latter two 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 *snapped region*—and a second that occupies the rest—the *filled 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 “filled” is almost the same as running in full-screen landscape, except with 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 or sidepillar region to preserve the aspect ratio of the content. Both approaches are just fine.

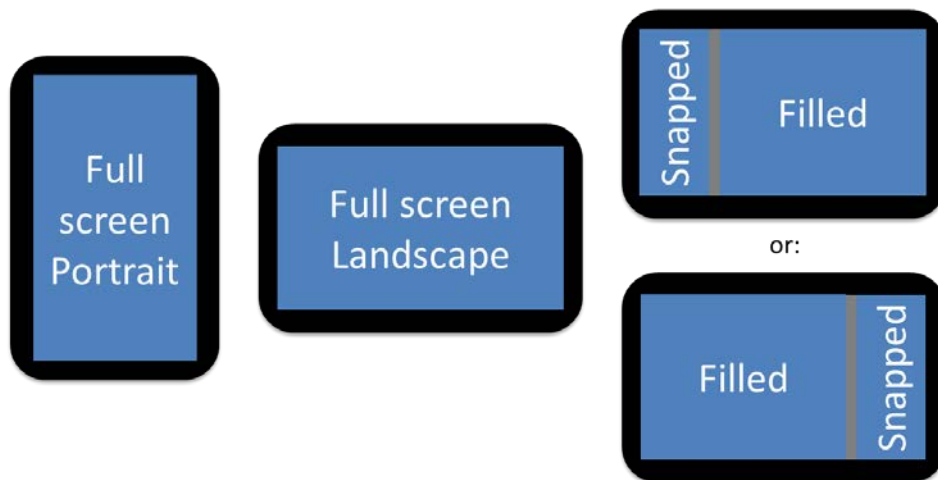


FIGURE 1-6 The four view states for Windows Store apps; all pages within the app need to be prepared to show properly in all four view states, a process that generally just involves visibility of elements and layout that can often be handled entirely within CSS media queries.

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 to 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 snapped 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 snapped 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 filled 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 [Guidelines for scaling to screens](#) and the [Designing UX for apps](#) topic.

It might also be true that you're running on a high-resolution device that also has a very small screen (high *pixel density*), 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 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 application with HTML, CSS, and JavaScript, you typically end up with a number of different HTML pages and navigate between them by using `<a href>` tags or by setting `document.location`.

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

Furthermore, switching between pages is visually abrupt: the user sees a blank screen while the new page is being loaded. This makes it difficult to provide a smooth, animated transition

between pages as generally seen within the Windows 8 personality—it’s the antithesis of “fast and fluid” and guaranteed to make designers cringe.

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

Those Capabilities Again: Getting to Data and Devices

At run time, now, even inside the app container, your app has plenty of room to play and to delight your customers. It can 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), manage state and user data (Chapter 8), and receive and process *pointer events*, which unify touch, mouse, and stylus (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). They can optimize performance and extend their capabilities through WinRT components (Chapter 16), and they can adapt themselves to different markets, provide accessibility, and work with various monetization options like advertising, trial versions, and in-app purchases (Chapter 17).

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

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

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



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

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

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

We’ll encounter many other sections of the manifest besides capabilities in this book. For example, 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 which URIs are known and trusted by your app and can thus act on the app’s behalf. The manifest is also where you declare things like your preferred orientation, *background tasks* (like playing audio or handling real-time communication), contract behaviors (such as which page in your app should be brought up in response to being invoked via a contract), custom protocols, and the appearance of tiles and notifications. You and your app will become bosom buddies with the manifest.

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

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

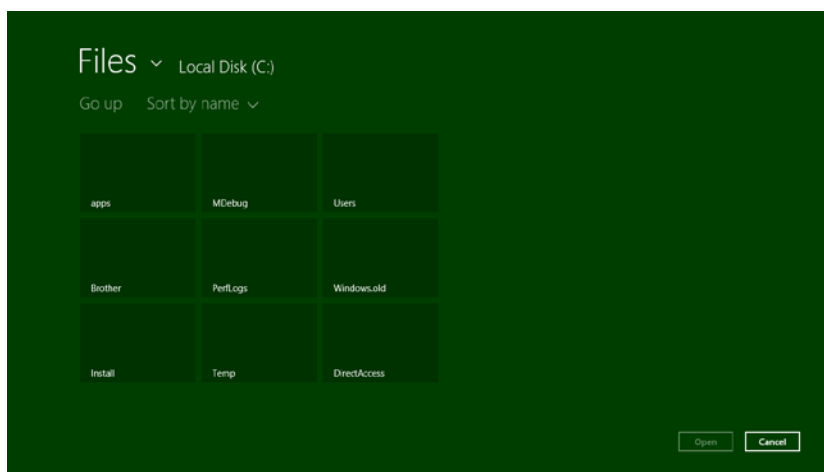


FIGURE 1-8 Using the file picker UI to access other parts of the file system from within a Store app, such as folders on a drive root (but not protected system folders). This is done by tapping the down arrow next to “Files.”

Taking a Break, Getting Some Rest: Process Lifecycle Management

Whew! We've covered a lot of ground already in this first chapter—our apps have been busy, busy, busy, and we haven't even started writing any code yet! In fact, apps can become really busy when they implement certain sides of contracts. If an app declares itself as a Search, Share, Contact, or File Picker *target* in its manifest (among other things), Windows will activate the app in response to the appropriate user actions. For example, if the user invokes the Share charm and picks your app as a Share target, Windows will activate the app with an indication of that purpose. In response, the app displays its specific share UI 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 Windows Store apps that helps conserve power and optimize battery life. One reality of traditional multitasking operating systems is that users typically leave a bunch of apps running, all of which consume power. This made sense with desktop apps because many of them can be at least partially visible at once. But for Store apps, the system is boldly taking on the job itself and using the full-screen nature of those apps to its advantage.

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

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

If the user then switches back to the app (in whatever view 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, update its layout, refresh a view of a file system library, or take a new sensor reading because any amount of time might have passed since it was suspended. Typically, though, an app will not need to reload any of its own state because it was in memory the whole time.

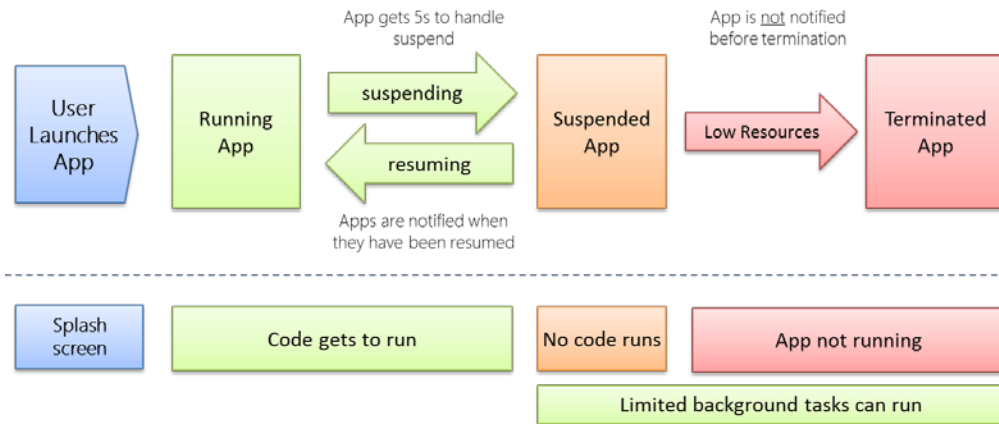


FIGURE 1-9 Process lifetime states for Windows Store apps.

There are a couple of exceptions to this. First, Windows provides a *background transfer* API—see Chapter 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 Push Notification Service, WNS) so that they need not even be running for this purpose—see Chapter 13, “Tiles, Notifications, the Lock Screen, and Background Tasks.” Second, certain kinds of apps do useful things when they’re not visible, such as audio players, communications apps, or those that need to take action when specific system events occur (like a network change, user login, etc.). With audio, as we’ll see in Chapter 10, “Media,” an app specifies background audio in its manifest (where else!) and sets certain properties on the appropriate audio elements. This allows it to continue running in the background. With system events, as we’ll also see in Chapter 13, an app declares background tasks in its manifest that are tied to specific functions in their code. In this case, Windows will wake the app from the suspended state when an appropriate trigger occurs. This is shown at the bottom of Figure 1-9.

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

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

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

Remembering Yourself: App State and Roaming

To step back for a moment, one of the key differences between traditional desktop applications and Windows Store apps is that the latter are inherently stateful. That is, once they’ve run the first time, they remember their state across invocations (unless explicitly closed by the user or unless they provide an affordance to reset the state explicitly). Some desktop applications work like this, but most suffer from a kind of identity crisis when they’re launched. Like Gilderoy Lockhart in *Harry Potter and the Chamber of Secrets*, they often start up asking themselves, “Who am I?”⁶ with no sense of where they’ve been or what they were doing before.

Clearly this isn’t a good idea with Store apps whose lifetime is being managed automatically. From the user’s point of view, apps are always running even if they’re not. It’s therefore critical that apps first manage settings that are always in effect and then also save their session state when being suspended. This way, if the app is terminated and restarted, it can reload that session state to return to the exact place it was before. (An app receives a flag on startup to indicate its previous execution state, which determines what it should do with saved session state. Details are in Chapter 3.)

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

Fortunately, Windows 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 Microsoft account, as shown in Figure 1-10.

⁶ 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?*

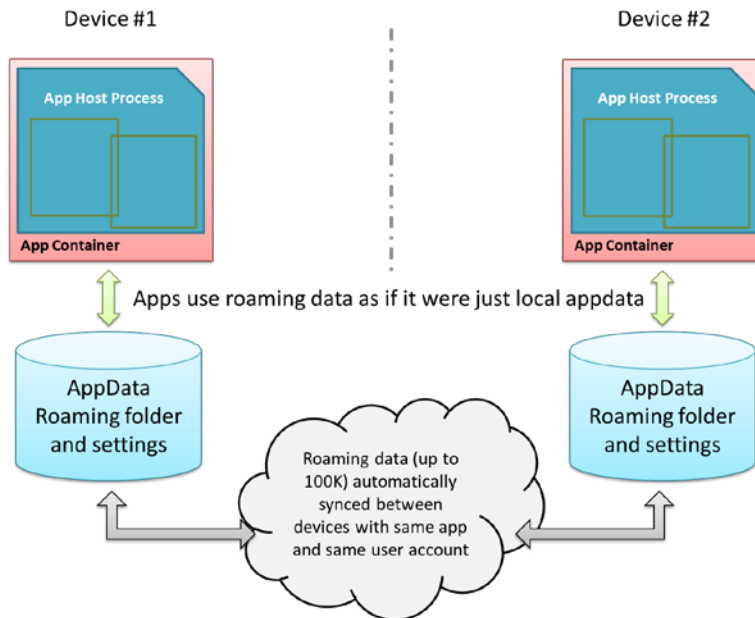


FIGURE 1-10 Automatic roaming of app roaming data (folder contents and settings) between devices.

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 LocalState, RoamingState, and TempState when the app is installed (I typically refer to them without the "State" appended.) The app can programmatically get to any of these folders at any time and can create in them all the files and subfolders to fulfill its heart's desire. There are also APIs for managing individual Local and Roaming settings (key-value pairs), along with groups of settings called *composites* that are always written to, read from, and roamed as a unit. (These are useful when implementing the app's Settings features for the Settings charm, as covered in Chapter 8, "State, Settings, Files, and Documents.")

Now, although the app can write as much as it wants to the appdata areas (up to the capacity of the file system), Windows will automatically roam the data in your Roaming sections only if you stay below an allowed quota (~100K, but there's an API for that). If you exceed the limit, the data will still be there but none of it will be roamed. Also be aware that cloud storage has different limits on the length of filenames and file paths as well as the complexity of the folder structure. So keep your roaming state small and simple. If the app needs to roam larger amounts of data, use a secondary web service like SkyDrive (see the blog post [Extending "Windows 8" apps to the cloud with SkyDrive](#)).

So the app really needs to decide what kind of state is local to a device and what should be roamed. Generally speaking, any kind of settings, data, or cached resources that are device-specific should always be local (and Temp is also local), whereas settings and data that represent the user's interaction with the app are potential roaming candidates. For example, an email app that maintains a local cache of messages would keep those local but would roam account settings (sans passwords, see Tip below) so

that the user has to configure the app on only one device. It would probably also maintain a per-device setting for how it downloads or updates emails so that the user can minimize network/radio traffic on a mobile device. A media player, similarly, would keep local caches that are dependent on the specific device's display characteristics, and it would roam playlists, playback positions, favorites, and other such settings (should the user want that behavior, of course).

Tip For passwords in particular, always store them in the Credential Locker (see Chapter 14). If the user allows password roaming (PC Settings > Sync Your Settings > Passwords), the locker's contents will be roamed automatically.

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

Along these same lines, I'm told that if a user installs an app, roams some settings, uninstalls the app, then within "a reasonable time" reinstalls the app, the user will find that those settings are still in place. This makes sense, because it would be too draconian to blow away roaming state in the cloud the moment a user just happened to uninstall an app on all their devices. There's no guarantee of this behavior, mind you, but Windows will apparently retain roaming state for an app for some time 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 is always under the app's control. Temp data, on the other hand, can be deleted if the user runs the Disk Cleanup utility. Local data is thus best used to support an app's functionality, and temp data is used to support run-time optimization at the expense of disk space.

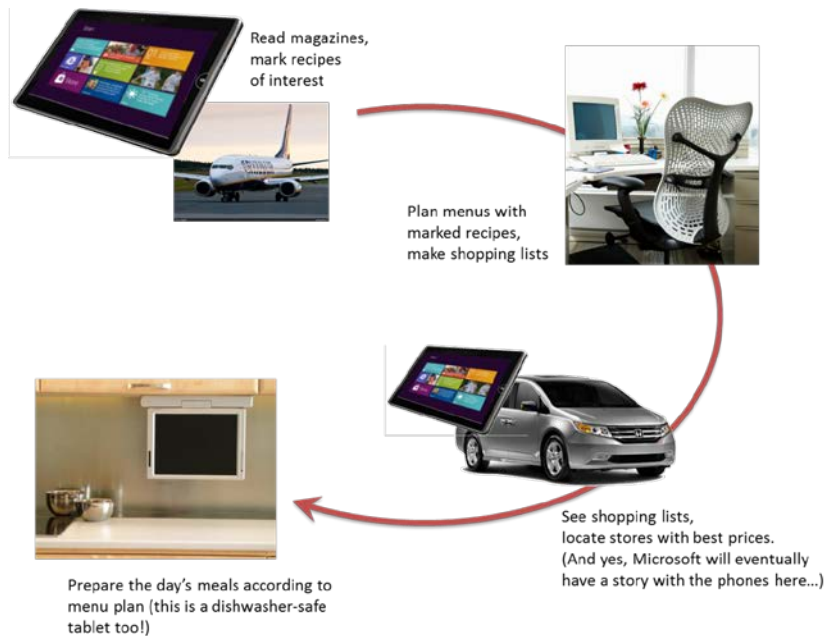
For Windows Store apps written in HTML and JavaScript, you can also use existing caching mechanisms like HTML5 local storage, IndexedDB, app cache, and so forth. All of these will be stored within the app's LocalState folder.

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

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

An example scenario is illustrated in Figure 1-11, where an app can have different personalities or flavors depending on user context and how different devices might be used in that context. It might seem rather pedestrian to think about an app for meal planning, recipe management, and shopping lists, but that's something that happens in a large number of households worldwide. Plus it's something that my wife would like to see me implement if I wrote more code than text!

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



Coming Back Home: Updates and New Opportunities

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

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

The Store will also provide you with crash *analytics* so that you can specifically identify problem areas in your app that evaded your own testing. This is incredibly valuable—maybe you’re already clapping your hands in delight!—because if you’ve ever wanted this kind of data before, you’ve had to implement the entire mechanism yourself. No longer. This is one of the valuable services you get in exchange for your annual registration with the Store. (Of course, you can still implement your own too.)

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

Updates are onboarded to the Windows Store just like the app’s first version. You create and upload an app package (with the same package name as before but a new version number), and then you update your description, graphics, pricing, and other information. After that your updated package goes through the same certification and signing process as before, and when all that’s complete your new app will be available in the Store. 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 applications.)

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. Your updated app should be prepared, then, to migrate a previous version of its state if and when it encounters such.

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

Another interesting question with updates is whether you can get a list of the customers who have acquired your app from the Store. The answer is no, because of privacy considerations. However, there is nothing wrong with including a registration feature in your app through which users can opt in to

receive additional information from you, such as more detailed update notifications. Your Settings panel is a great place to include this.

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

And, Oh Yes, Then There’s Design

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

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

I said that we’ll be treating this minimally because I simply do not consider myself a designer. I encourage you to be honest about this yourself: if you don’t have a good designer working with you, **get one**. Sure, you can probably work out an OK design on your own, but the demands of a consumer-oriented market combined with a newer design language like that employed in Windows 8—where the emphasis is on simplicity and tailored experiences—underscores the need for professional help. It’ll make the difference between a functional app and a great app, between a tool and a piece of art, between apps that consumers accept and those they *love*.

With design, I do encourage developers to peruse the material on [Designing UX for apps](#) for a better understanding of design principles. But let’s be honest: as a developer, do you really want to ponder what “fast and fluid” means (and design not just static wireframes but also the dynamic aspects of an app like animations)? 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.

Let me add that design is neither a one-time nor a static process. Developers and designers will need to work together throughout the development experience, as design needs will arise in response to how well the implementation really works. For example, the real-world performance of an app might require the use of progress indicators when loading certain pages or might be better solved with a redesign of page navigation. It may also turn out, as we found with one of our early app partners, that the kinds of

graphics called for in the design simply weren't available from the app's back-end service. The design was lovely, in other words, but couldn't actually be implemented, so a design change was necessary. So make sure that your ongoing relationship with your designers is a healthy and happy one.

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

Chapter 2

Quickstart

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

A Really Quick Quickstart: The Blank App Template

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

1. Run Visual Studio Express. If this is your first time, you’ll be prompted to obtain a developer license. Do this, because you can’t go any further without it!
2. Click New Project... in the Visual Studio window, or use the File > New Project menu command.
3. In the dialog that appears (Figure 2-1), make sure you select JavaScript under Templates on the left side, and then select Blank Application in the middle. Give it a name (**HelloWorld** will do), a folder, and click OK.

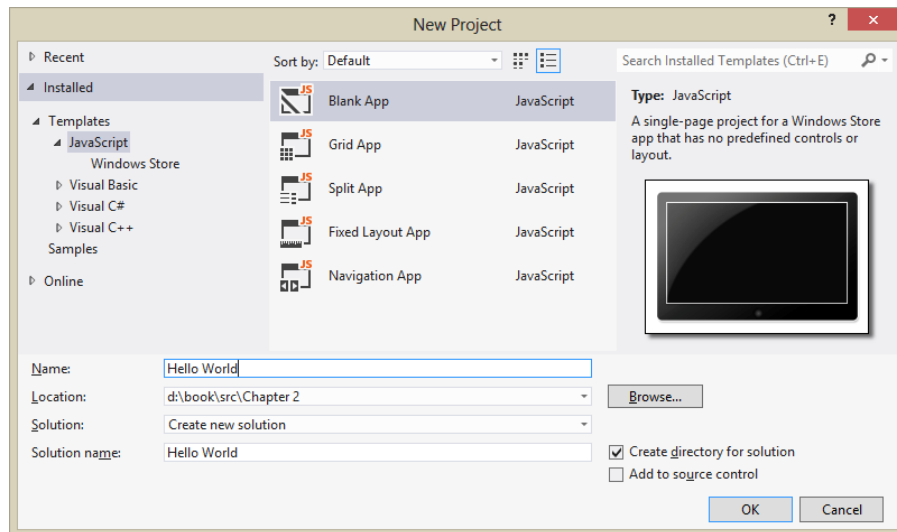


FIGURE 2-1 Visual Studio’s New Project dialog using the light UI theme. (See the Tools > Options menu command, and then change the theme in the Environment/General section). I use the light theme in this book because it looks best against a white page background.

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

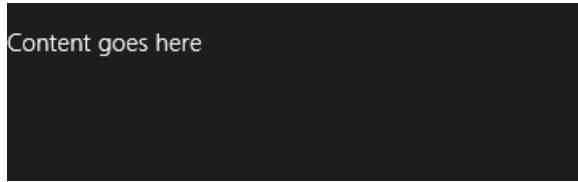


FIGURE 2-2 The only vaguely interesting portion of the Hello World app's display. The message is at least a better invitation to write more code than the standard first-app greeting!

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

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

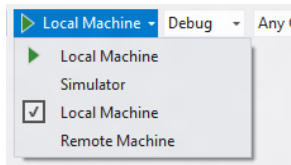


FIGURE 2-3 Visual Studio's debugging options on the toolbar.

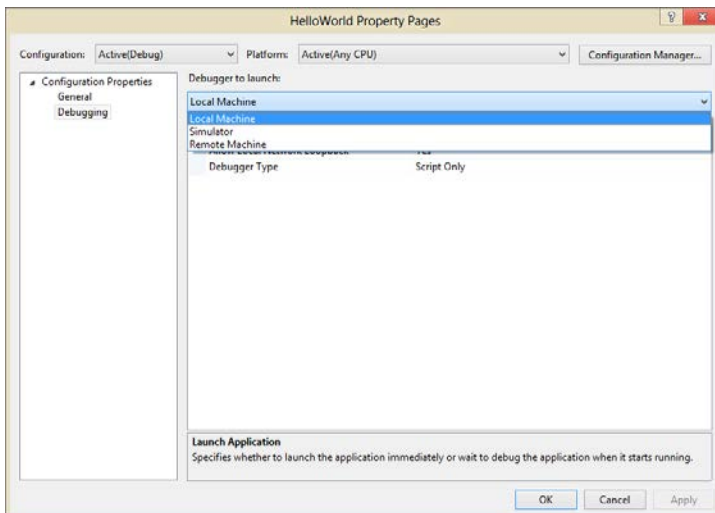


FIGURE 2-4 Visual Studio's debugging options in the app properties dialog.

The Remote Machine option allows you to run the app on a separate device, which is absolutely essential for working with devices that can't run desktop apps at all, such as ARM devices (and if you see only this option with a sample project, the build target is probably set to ARM). Setting this up is a straightforward process: see [Running apps on a remote machine](#), and I do recommend that you get familiar with it. Also, when you don't have a project loaded in Visual Studio, the Debug menu offers the Attach To Process command, which allows you to debug an already-running app. See [How to start a debugging session \(JavaScript\)](#).

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 apps in the simulator](#) topic.

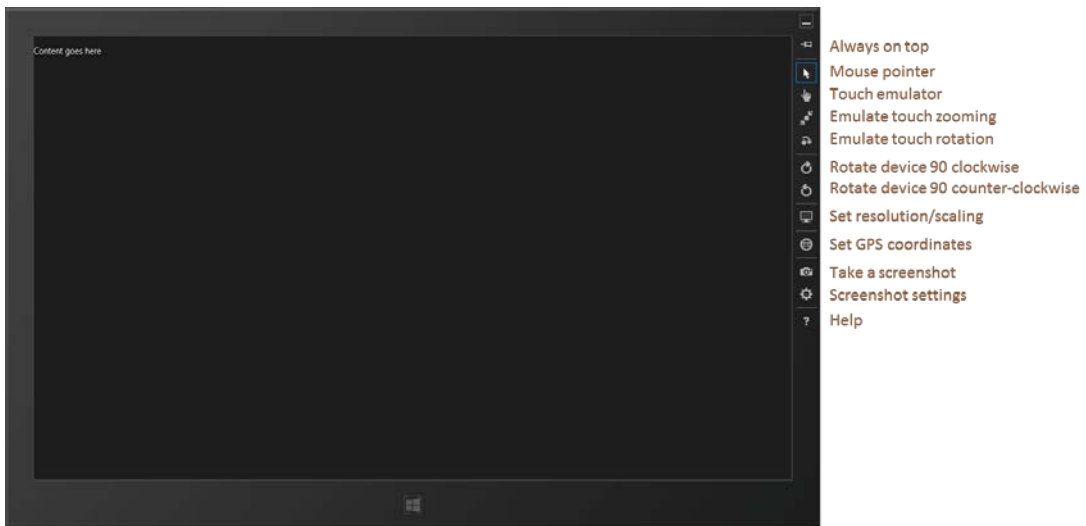


FIGURE 2-5 Hello World running in the simulator, with added labels on the right for the simulator controls. Truly, the "Blank App" template lives up to its name!

Sidebar: How Does Visual Studio Run an App?

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

There's really no magic involved: deployment can actually be done through the command line. To see the details, use the Store/Create App Package in Visual Studio, select No for a Store upload, and you'll see a dialog in which you can save your package 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 who have a developer license, allowing them to side-load your app without needing your full 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.
- **<Appname>_TemporaryKey.pfx** A temporary signature created on first run.
- **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, default.html selected as the start page, and all the places where you control different aspects of your app. We'll be seeing these throughout this book; for a complete reference, see the [App packages and deployment](#) and [Using the manifest designer](#) topics. And if you want to explore the manifest XML directly, right-click this file and select View Code.

The css folder contains a default.css file where you'll see media query structures for the four view states that all apps should honor. We'll see this in action in the next section, and I'll discuss all the details in Chapter 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 sending your app to the Store (and you'll want to provide scaled versions too, 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 and other places at run time.
- **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 solution-wide or project-wide searches.)

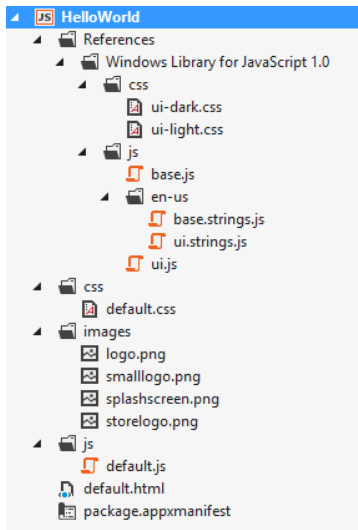


FIGURE 2-6 A Blank app project fully expanded in Solution Explorer.

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/css/ui-dark.css" rel="stylesheet">
  <script src="//Microsoft.WinJS.1.0/js/base.js"></script>
  <script src="//Microsoft.WinJS.1.0/js/ui.js"></script>

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

You will generally always have these references (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 your app

package, whereas a single / refers to the root of your package. Beyond that, everything else is standard HTML5, so feel free to play around with adding some additional HTML of your own and see the effects.

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;

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

    app.oncheckpoint = function (args) {
    };

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

We'll come back to `checkpoint` in Chapter 3. For now, remember from Chapter 1, "The Life Story of a Windows Store App," that an app can be activated in many ways. These are indicated in the `args.detail.kind` property whose 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.ApplicationModel.Activation.ApplicationExecutionState` enumeration tells the app how it was last running. Again, we'll see more on this in Chapter 3, so the comments in the default code above should satisfy your curiosity for the time being.

Now, what is that `args.setPromise(WinJS.UI.processAll())` for? As we'll see many times, `WinJS.UI.processAll` instantiates any WinJS controls that are declared in HTML—that is, any element (commonly a `div` or `span`) that contains a `data-win-control` attribute whose value is the name of a constructor function. 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.⁷ As for `args.setPromise`, that's employing something called a deferral that we'll defer to Chapter 3.

As short as it is, that little `app.start()`; at the bottom is also a very important piece. 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 throughout the module but don't appear in the global namespace.⁸

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.

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

Sidebar: Writing Code in Debug Mode

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

⁷ 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).

⁸ See Chapter 2 of Nicolas Zakas's *High Performance JavaScript* (O'Reilly, 2010) for the performance implications of scoping.

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

When my son was three years old, he never—despite the fact that he was born to two engineers parents and two engineer grandfathers—peeked around corners or appeared in a room saying “Hello world!” No, his particular phrase was “Here my am!” Using that particular variation of announcing oneself to the universe, 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 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 many of these most excellent samples in this book, I encourage you to download the whole set—go to the URL above, and locate the link for “Windows 8 app samples”. This link 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. (Again, if you open a sample that seems to support only the Remote Machine debugging option, the build target is probably set to ARM—change it to Any CPU for local debugging.)

I also *strongly* encourage you to spend a half-day, even a full day, getting familiar with Visual Studio and Blend for Visual Studio and just perusing through the samples so that you know what’s there. Such small investments will pay huge productivity dividends even in the short term!

Design Wireframes

Before we start on the code, let’s first look at design wireframes for this app. Oooh...design? Yes! Perhaps for the first time in the history of Windows, there’s a real design *philosophy* to apply to apps. In 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 year or two and invest in that training yourself. Simply said, *design matters* for Windows Store apps, and it will make the difference between apps that succeed and apps that merely exist in the Windows Store and are largely ignored. And having a design in hand will just make it easier to implement because you won't have to make those decisions when you're writing code! (If you still intend on filling designer shoes and communing with apps like Adobe Illustrator, be sure to visit [Designing UX for apps](#) for the philosophy and details of Windows Store app design, plus design resources.)

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

Note Traditional wireframes are great to show a static view of the app, but in the “fast and fluid” environment of Windows 8, the *dynamic* aspects of an app—animations and movement—are also very important. Great app design includes consideration of not just where content is placed but how and when it gets there in response to which user actions. Chapter 11, “Purposeful Animations,” discusses the different built-in animations that you can use for this purpose.

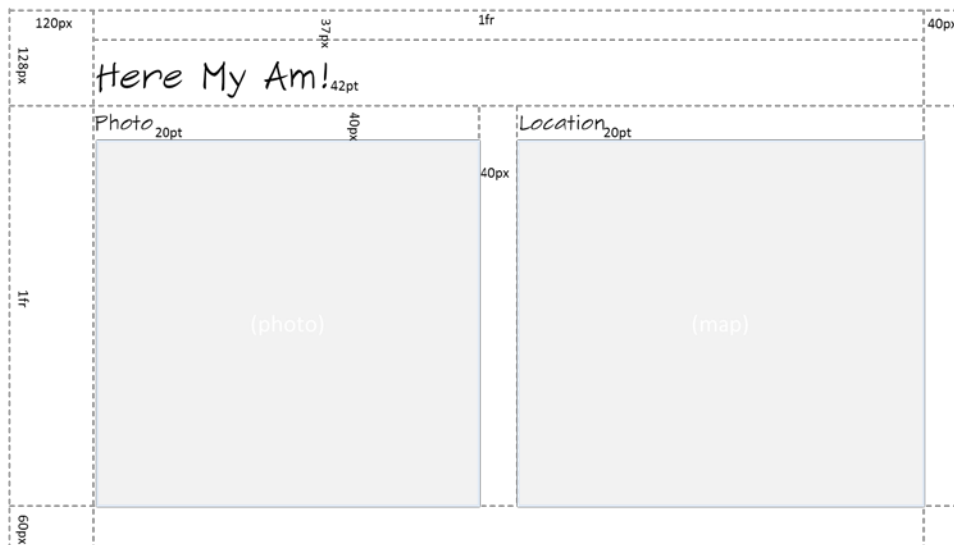


FIGURE 2-7 Full-screen landscape and filled (landscape) wireframe. These view states typically use the same wireframe (the same margins), with the proportional parts of the grid simply becoming smaller with the reduced width.

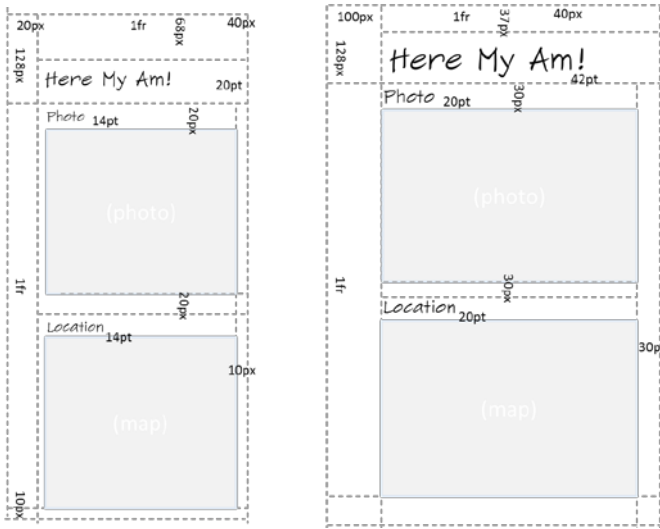


FIGURE 2-8 Snapped wireframe (left; landscape only) and full-screen portrait wireframe (right); these are not to scale.

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, and for apps written in HTML and JavaScript the work can mostly, if not entirely, be handled through CSS media queries.

One of the important aspects of Windows Store app design is understanding the layout *silhouette*: the size of the header fonts, their placement, the specific margins, grid layout, and all that (as marked in the previous figures). These recommendations 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. At the same time, the silhouette is a starting point and not a requirement—apps can and do depart from it when it makes sense. Absent a clear design, however, it's best to stay with it.

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 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 before. (Note: Video 2-2 shows all these steps together.)

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

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

Here we see the five elements in the wireframe: a main header, two subheaders, a space for a photo (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.⁹

You'll see that some elements have style classes assigned to them. Those that start with `win` come from the WinJS stylesheet.¹⁰ 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.

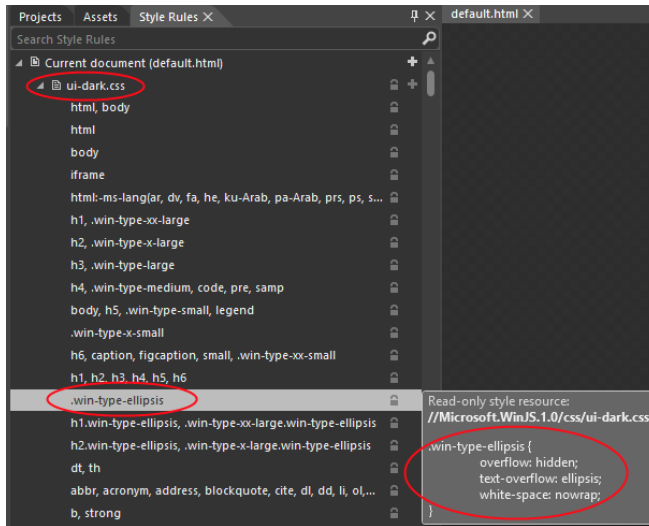


FIGURE 2-9 In Blend, the Style Rules tab lets you look into the WinJS stylesheet and see what each particular style contains. Take special notice of the search bar under the tabs. This is here so you don't waste your time visually scanning for a particular style—just start typing in the box, and let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

The page we'll load into the `iframe`, `map.html`, is part of our app package that we'll add in a moment, but note how we reference it. The `ms-appx-web:///` protocol indicates that the `iframe` and its contents will run in the web context (introduced in Chapter 1), thereby allowing us to load the remote script for the Bing maps control. The *triple slash*, for its part—or more accurately the third slash—is shorthand for “the current app package” (a value that you can obtain from `document.location.host`), so we don't need to create an absolute URI for in-package content.

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

⁹ 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.

¹⁰ 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.

your app and accessing WinRT APIs.

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.

In Chapter 17 we'll also see how to separate strings (including ARIA labels) from our markup, JavaScript, and even the manifest and place it in a resource file. This is something you might want to do from early on, so see the "Preparing for Localization" section in that chapter for the details. Note, however, that resource lookup doesn't work well in Blend, so you might want to hold off on the effort until you've done most of your styling.

Styling in Blend

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

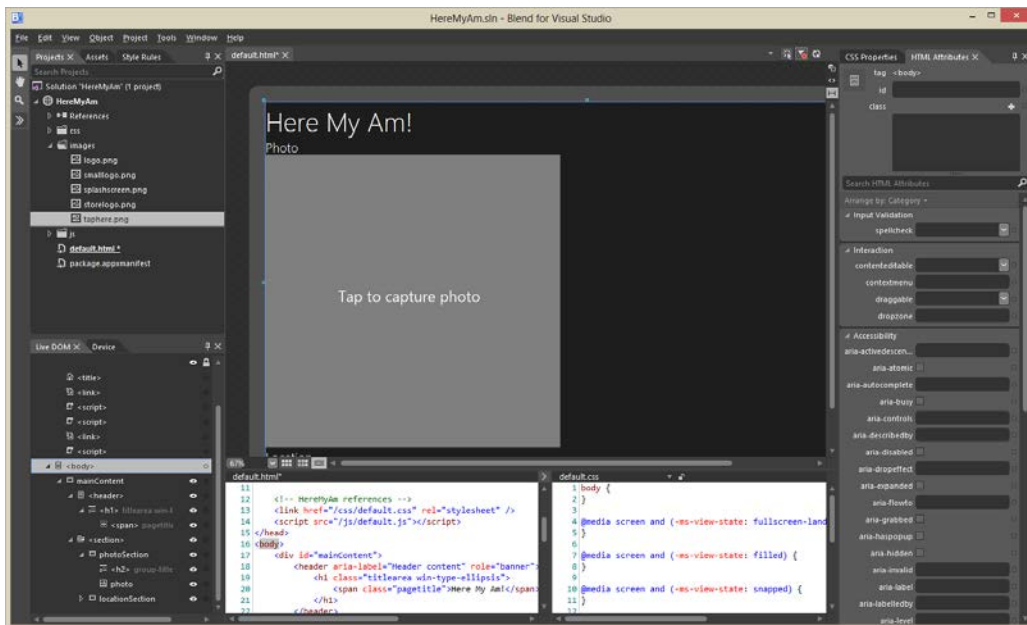


FIGURE 2-10 The app in Blend without styling, showing a view that is much like the Visual Studio simulator. If the `taphere.png` image doesn't show after adding it, use the View/Refresh menu command.

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

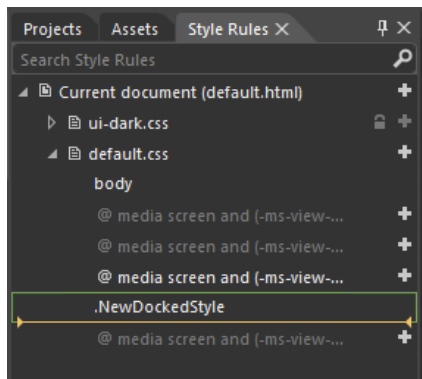
orientation, screen resolution, and view state. Clicking an element in the Live DOM here will highlight it in the designer, just like clicking an element in the designer will highlight it in the Live DOM section.

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:

- Set the insertion point of the style rule with the orange-yellow line control within Blend's Style Rules tab. This determines exactly where any new rule you create will be created:



- 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 (especially inline style), 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.

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.

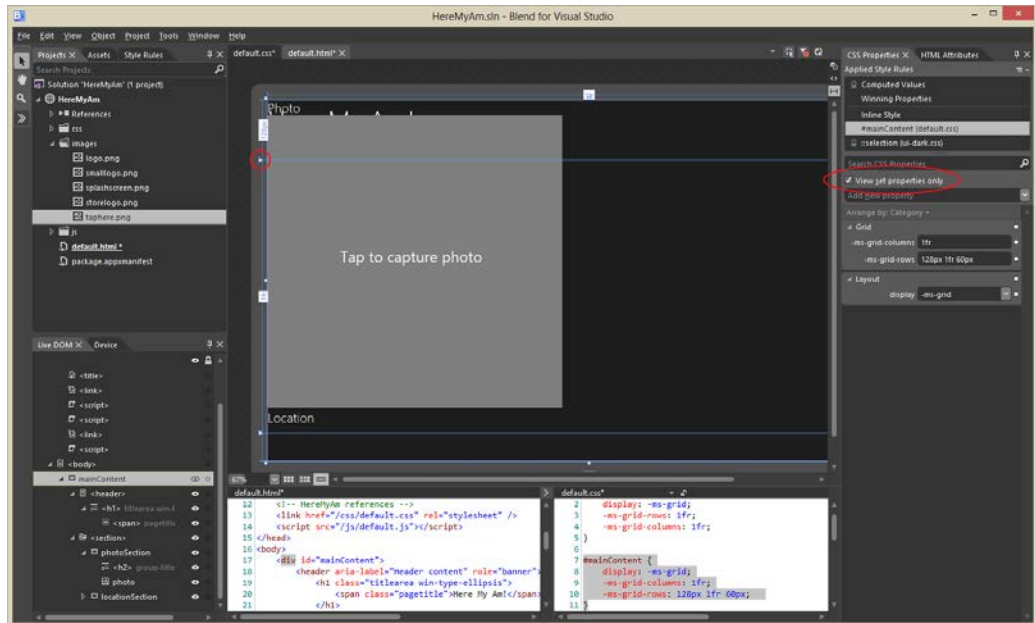


FIGURE 2-11 Setting the grid properties for the *mainContent* *div*. Notice how the View Set Properties Only checkbox (upper right) makes it easy to see what styles are set for the current rule. Also notice in the main “Artboard” how the grid rows and columns are indicated, including sliders (circled) 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 in the appropriate media queries for the view states—is best done in video. Video 2-2 (available with this book’s downloadable companion content) shows this process starting with the creation of the project, styling the different 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. It also demonstrates the amount of time it takes to style such an app once you’re familiar with the tools.

The result of all this in the simulator looks just like the wireframes—see Figures 2-12 through 2-14—and all the styling is entirely contained within the appropriate media queries of *default.css*. Most importantly, the way Blend shows us the results in real time is an enormous time-saver over fiddling with the CSS and running the app all over again, a painful process that I’m sure you’re familiar with! (And the time savings are even greater with Interactive Mode; see Video 4-1 in the companion content created for Chapter 4, “Controls, Control Styling, and Data Binding.”)

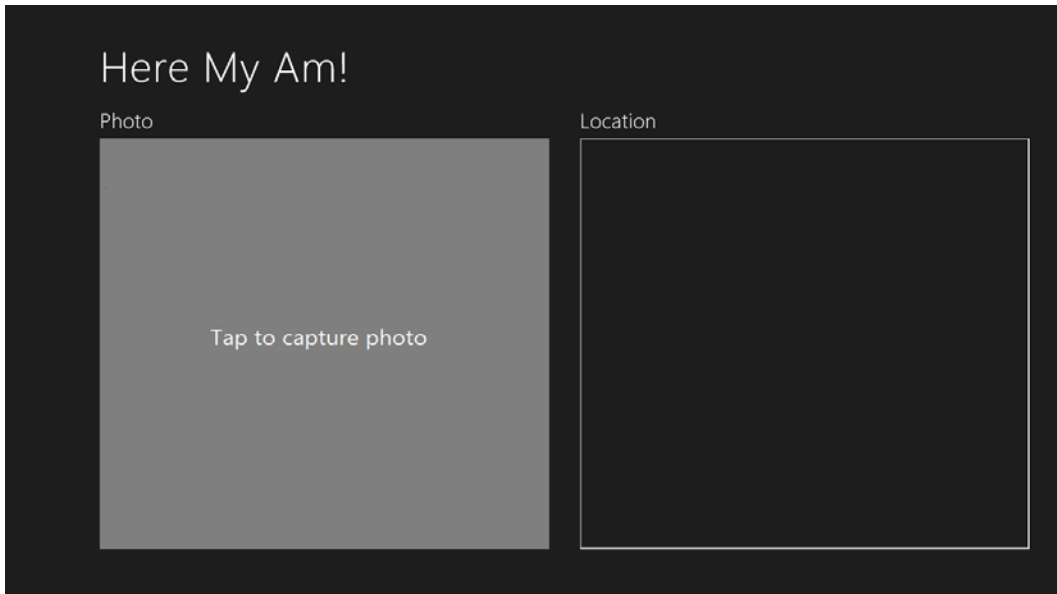


FIGURE 2-12 Full-screen landscape view.

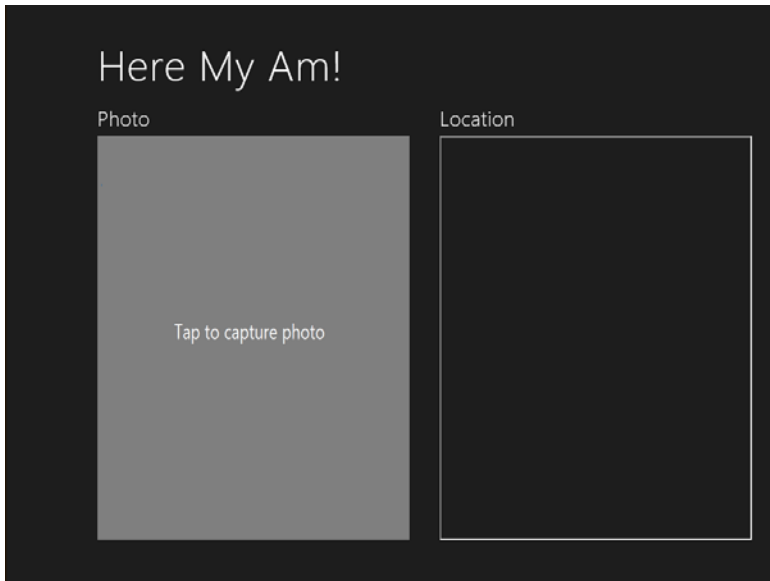


FIGURE 2-13 Filled view (landscape only).

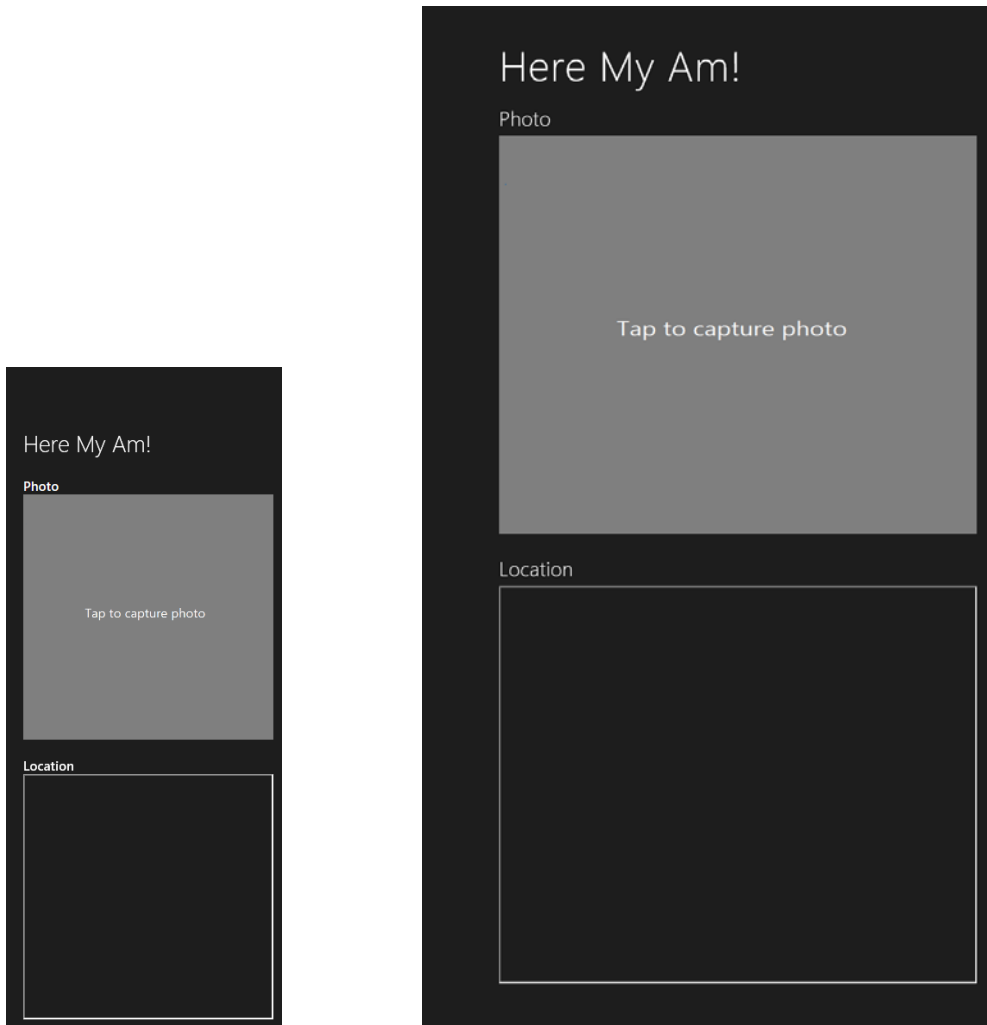


FIGURE 2-14 Snapped view (landscape only) and full-screen portrait view; these are to relative scale.

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.¹¹ 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.

¹¹ 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.

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

- A Bing maps control in the Location section showing the user's current location. We'll 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.

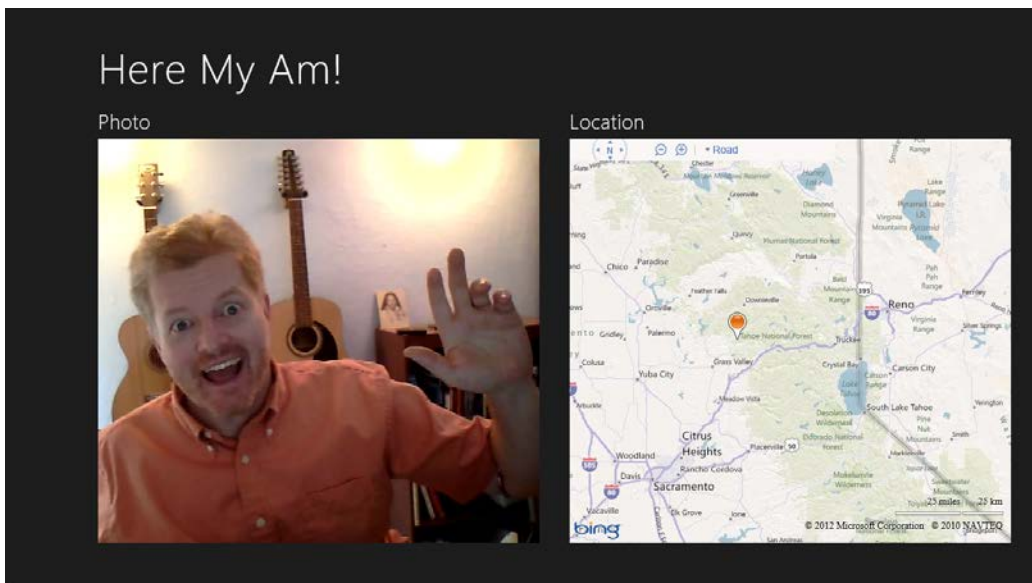


FIGURE 2-15 The completed Here My Am! app (though I zoomed out the map so you can't quite tell exactly where I live!).

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 page loads the Bing Maps control script from a remote source and thus runs in the web context. Note that we could also employ the [Bing Maps SDK](#), which provides script we can load into the local context. For the time being, I want to use the remote script approach because it gives us an opportunity to work with web content and the web context in general, something that I'm sure you'll want to understand for your own apps. We'll switch to the local control in Chapter 8, "State, Settings, Files, and Documents."

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:¹²

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

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

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

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

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

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

        var target = this[call.functionName];

        if (typeof target !== 'function') {
          throw "The function name does not resolve to an actual function";
        }

        return target.apply(this, call.args);
      }

      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);
      }
    </script>
  </head>
  <body>
    <div id="map">
      <div id="mshelp">
        Map
      </div>
    </div>
  </body>
</html>
```

¹² Note that you should replace the `credentials` inside the `init` function with your own key obtained from <https://www.bingmapsportal.com/>.

```

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

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

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

        var location = new Microsoft.Maps.Location(lat, long);
        var pushpin = new Microsoft.Maps.Pushpin(location, { 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`. This is an important point: the local and web contexts are kept separate so that arbitrary web content cannot drive an app or access WinRT APIs. The two contexts enforce a boundary between an app and the web that can only be crossed with `postMessage`.

In the code above, you can see that we pick up such messages and pass them to the `processMessage` function, a little generic routine 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: var activation = Windows.ApplicationModel.Activation;
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 points about this code. 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 Chapter 9, "Input and Sensors," in "Sidebar: HTML5 Geolocation." The API exists in WinRT because other supported languages (C# and C++) don't have access to the HTML5 geolocation APIs. We're focused on WinRT APIs in this book, so we'll just use functions in the `Windows.Devices.Geolocation` namespace.

Next, in the second parameter to `postMessage` you see a combination of `ms-appx[-web]://` with `document.location.host`. This essentially means “the current app from the local [or web] context,” which is the appropriate origin of the message. Notice that we use the same value to check the origin when receiving a message: the code in `map.html` verifies it’s coming from the app’s local context, whereas the code in `default.js` verifies that it’s coming from the app’s web context. Always make sure to check the origin appropriately; see [Validate the origin of postMessage data](#) in [Developing secure apps](#).

Finally, the call to `getGeopositionAsync` has an interesting construct, wherein we make the call and chain this function called `done` onto it, whose argument is another function. This is a very common pattern 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, the *completed handler*. It can also take two optional functions to wire up *error* and *progress 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 (Chapter 3), and how promises fit into async operations more generally (Chapter 16, “WinRT Components”).

The argument passed to the *completed handler* contains the results of the async call, which in our example above is a `Windows.Geolocation.Geoposition` object containing the last reading. (When reading the docs for an async function, you’ll see that the return type is listed like `IAsyncOperation-<Geoposition>`. The name within the `<>` indicates the actual data type of the results, so you’ll 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.¹³

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 says we neglected to set the *Location* capability in the manifest. Without that capability set, calls like this that depend on that capability will throw an exception.

¹³ 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.

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

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

The `console.log` function writes a string to the *JavaScript Console window* in Visual Studio, which is obviously a good idea. 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. (This is all we'll do with the error for now; in Chapter 7, "Commanding UI," we'll add a better message and a retry command.)

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

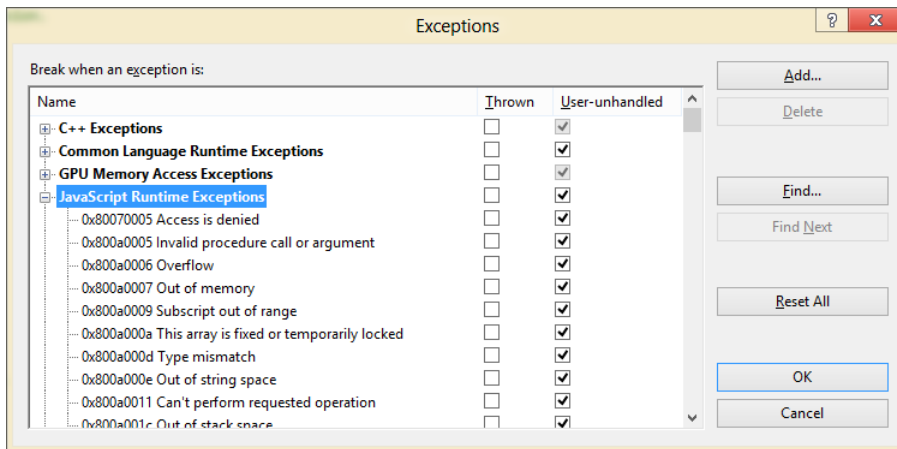


FIGURE 2-16 JavaScript run-time exceptions in the Debug/Exceptions dialog of Visual Studio.

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.

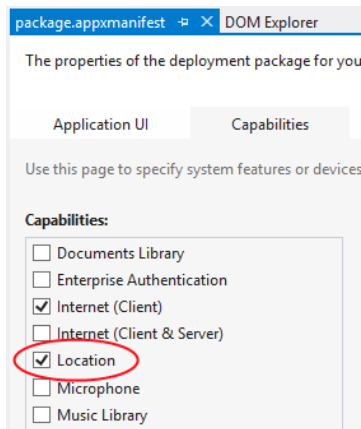


FIGURE 2-17 Setting the *Location* capability in Visual Studio's manifest editor. (Note that Blend supports editing the manifest only as XML.)

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 select 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 in Chapter 8.

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. It's a good example of how a little dynamic code like JavaScript combined with well-designed WinRT components—both those in the system and those you can write yourself—make a very powerful combination!

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

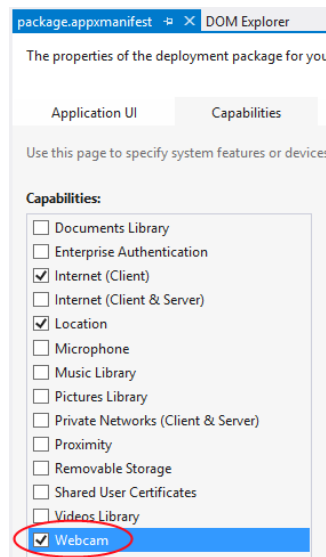


FIGURE 2-19 The camera capability in Visual Studio's manifest editor.

On first use of the camera at run time, you'll see another consent dialog 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:

```
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 many 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. That's easy as

well! You can hand a `StorageFile` object to the `URL.createObjectURL` method and get back an URI that can be directly assigned to the `img.src` attribute. The captured photo appears!¹⁴

Note that `captureFileAsync` will call the completed handler if the UI was successfully invoked but the user hit the back button and didn't actually capture anything. This is why 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.

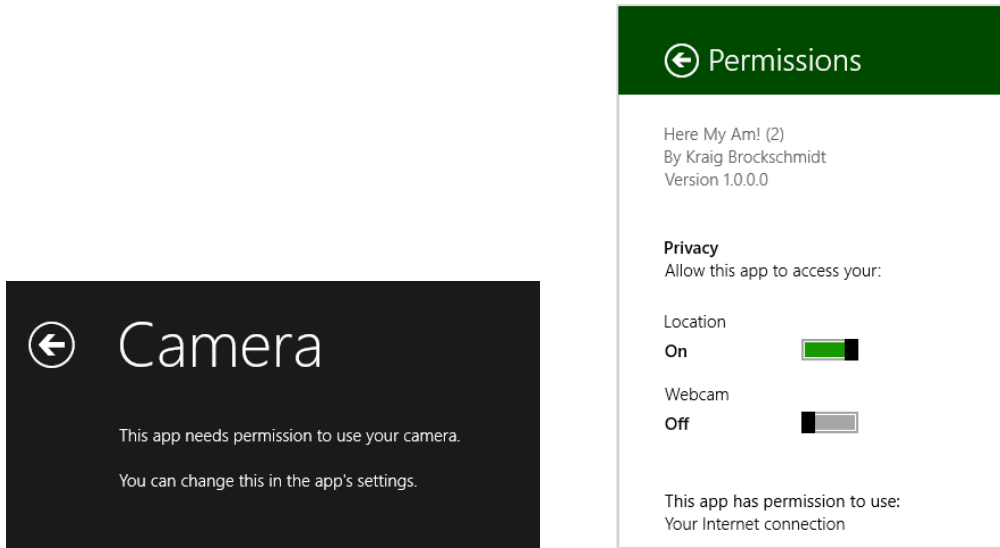


FIGURE 2-21 The camera capture UI's message when consent is denied (left); you can change permissions through the Settings Charm > Permissions pane (right).

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 asks the app for its *source* data. It then examines that data, generates a list of *target* apps that understand the data formats involved (according to their manifests), and displays that list in the Share pane. When the user selects a target, that app is activated and given the source data. In short,

¹⁴ The `{oneTimeOnly: true}` parameter indicates that the URI is not reusable and should be revoked via `URL.revokeObjectURL` when it's no longer used, as when we replace `img.src` with a new picture. Without this, we would 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 contract is an abstraction that sits between the two, so the source and target apps never need to know anything about each other.

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

So instead of adding code to our app to share the photo and location to a particular target, like Facebook, 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.¹⁵ First we just need to set up an appropriate listener—place this code in the `activated` event in `default.js` after setting up the `click` listener on the `img` element:

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

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 and then fill in text and image properties (if you want to obtain a map from Bing for sharing purposes, see [Get a static map](#)):

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

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

    data.properties.title = "Here My Am!";
```

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

```

data.properties.description = "At ("
    + lastPosition.latitude + ", " + lastPosition.longitude + ")";

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

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

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

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

```

The latter part of this code is pretty standard stuff for sharing a file-based image (which we have in `lastCapture`). I got most of this code, in fact, directly from the [Share content source app sample](#), which we'll look at more closely in Chapter 12, "Contracts."

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

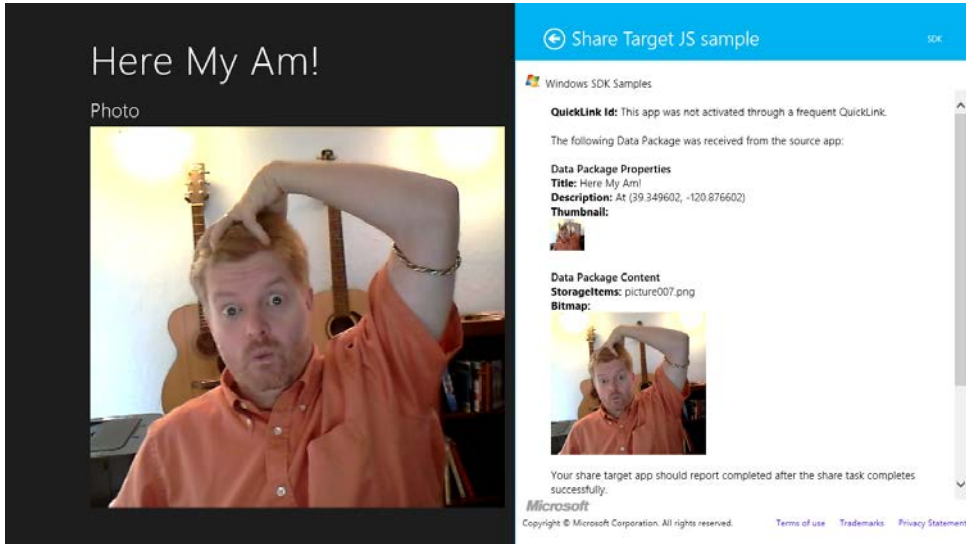


FIGURE 2-22 Sharing (monkey-see, monkey-do!) to the Share target sample in the Windows SDK. Share targets appear as a partial overlay on top of the current app, so the user never leaves the app context.

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 an object containing parameters, stringifies the whole thing, and then 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) {  
    //Verify data and origin (in this case the web context page)  
    if (!message.data || message.origin !== "ms-appx-web://" + document.location.host) {  
        return;  
    }  
  
    if (!message.data) {  
        return;  
    }  
  
    var eventObj = JSON.parse(message.data);  
  
    switch (eventObj.event) {  
        case "locationChanged":  
            lastPosition = { latitude: eventObj.latitude, longitude: eventObj.longitude };  
            break;  
  
        default:
```

```
        break;
    }
};
```

Clearly, this is more code than we'd need to handle a single message or event from an `iframe`, but I wanted to give you something that could be applied more generically in your own apps.

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 Windows Store app without any other distractions. In Chapter 3, we'll look more deeply at the anatomy of apps through a few of the other templates, yet we won't cover them all. We'll close this chapter, then, with a short introduction to these very handy tools.

Fixed Layout Template

"A project for a Windows Store app that scales using 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!*, 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.

As we'll see in Chapter 6, 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 or sidepillars 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 Store app that has predefined controls for navigation." (Blend/Visual Studio description)

The Navigation template builds on the Blank template by adding support for page navigation. As discussed in Chapter 1, Windows Store apps written in JavaScript are best implemented by having a single HTML page container into which other pages are dynamically loaded. This allows for smooth transitions (as well as animations) between those pages and preserves the script context.

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

Building on the Navigation template, the Grid template provides the basis for apps that will navigate collections of data across multiple pages. The home page shows grouped items within the collection, from which you can then navigate into the details of an item or into the details of a group and its items (from which you can then go into 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 Store app that navigates among grouped items. The first page allows group selection while the second displays an item list alongside details for the selected item." (Blend/Visual Studio description)

This last template also builds on the Navigation template and works over a collection of data. Its home page displays a list of groups, rather than grouped items as with the Grid template. Tapping a group then navigates to a group detail page split into two (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 Windows Store app from the Blank app template.
- How to run an app inside the local debugger and within the simulator, as well as the role of remote machine debugging.
- The features of the simulator that include the ability to simulate touch, set view states, and change resolutions and pixel densities.
- The basic project structure for Windows Store apps, including WinJS references.
- The core activation structure for an app through the WinJS.Application.onactivated event.
- The role and utility of design wireframes in app development, including the importance of designing for all view states, where the work is really a matter of element visibility and layout.
- The power of Blend for Visual Studio to quickly and efficiently add styling to an app's markup. Blend also makes a great CSS debugging tool.
- How to safely use web content (such as Bing maps) within a web context iframe and communicate between that page and the local context app by using the postMessage method.
- How to use the WinRT APIs, especially async methods involving promises but also geolocation and camera capture. Async operations return a promise to which you provide a completed handler (and optional error and progress handlers) to the promise's then or done method.
- Manifest capabilities determine whether an app can use certain WinRT APIs. Exceptions will result if an app attempts to use an API without declaring the associated capability.
- How to share data through the Share contract by responding to the daterequested event.
- 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 became completely invisible to the eye once the wallboard went up and the finish work was 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 inside the walls. Imagine if cabinets and trim started falling off the walls after a week or two of moving into 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. As a result, your customers will probably start looking for somewhere else to live, meaning someone else's app!

This chapter, then, is about those fundamentals: the core foundational structure of an app upon which you can build something that 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 Windows Store App," apps written with HTML, CSS, and JavaScript are not directly executable like their compiled counterparts written in C#, Visual Basic, or C++. In our app packages, there are no EXEs, just .html, .css, and .js files (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 Store apps.

Let's review what we've already learned in Chapter 1 and Chapter 2, "Quickstart," about the characteristics of the hosted environment:

- The app host (and the apps in it) use 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](#). A related topic is [Windows Store apps using JavaScript versus traditional web apps](#).
- HTML content in the app package can be loaded into the *local* or *web context*, depending on the `ms-appx:///` and `ms-appx-web:///` scheme used to reference that content (the third / 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.
- ActiveX control 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. (To explore the characteristics themselves, refer to the [Integrating](#)

[content and controls from web services sample](#).) 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 tab¹⁶, *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.

Next, 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 `` tag with `target` pointing to an `iframe`, the scheme in `href` determines the context.

A web context page, for its part, can contain only a web context `iframe` ; 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 Windows Store 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
<code><iframe src="ms-appx:///"></code>	<code>iframe</code> in local context	Not allowed
<code><iframe src="ms-appx-web:///"></code>	<code>iframe</code> in web context	<code>iframe</code> in web context
<code><iframe src="http[s]://"></code> or other scheme	<code>iframe</code> in web context	<code>iframe</code> in web context
<code></code> <code><iframe name="myFrame"></code>	<code>iframe</code> in local or web context depending on [uri]	<code>iframe</code> in web context; [uri] cannot begin with <code>ms-appx</code> .
<code></code>	Links to page in local context	Not allowed unless explicitly specified (see below)
<code></code>	Not allowed	Links to page in web context
<code></code> with any other protocol including <code>http[s]</code>	Opens default browser with [uri]	Opens default browser with [uri]

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

When an `iframe` is in the web context, note that its page can contain `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 Windows Store app cannot navigate from its top-level page (in the local context) directly to a web context page of any kind (local or remote) and remain within the app: the browser will be launched instead. 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 (as does Scenario 6 of the [Integrating content and controls from web services sample](#)). Do be careful when the URI contains query parameters, however. For example, you don't want to allow a website to navigate to something like `ms-appx:///delete.html?file=superimportant.doc!`

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, the app cache, and IndexedDB—things that web pages typically assume they can use. By default, the web context in a Store 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.

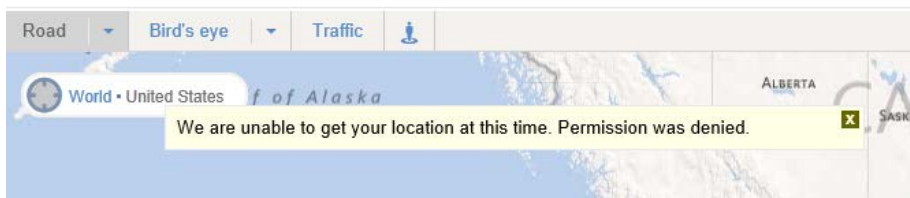


FIGURE 3-1 Use of brokered capabilities like geolocation from within a web context will generate an error.

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.

Such capabilities are blocked because web content loaded into an `iframe` can easily provide the means to navigate to other arbitrary pages. From the Bing maps page 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, such as `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, a message dialog will appear (Figure 3-3), just as if the app had made the request. (Note: When run inside the debugger, the ContentUri example might show a Permission Denied exception on startup. If so, press Continue within Visual Studio because this doesn't affect the app running outside the debugger.)

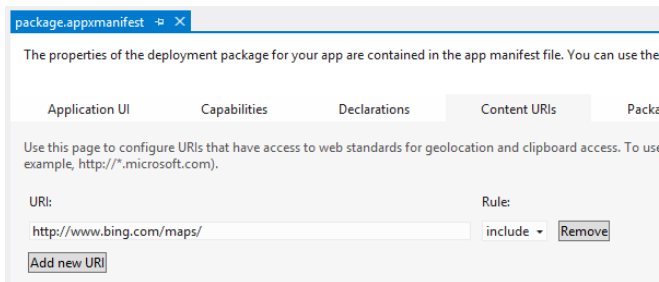


FIGURE 3-2 Adding a content URI to the app manifest; the contents of the text box is saved when the manifest is saved. Add New URI creates another set of controls in which to enter additional rules.

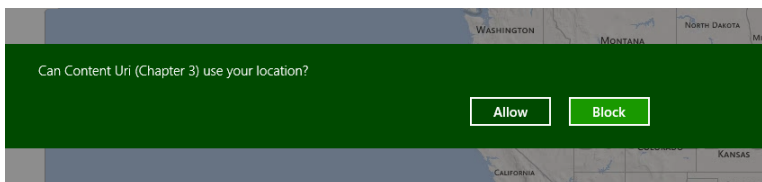


FIGURE 3-3 With a content URI rule in place, web content in an `iframe` acts like part of the app, showing why content URI rules are necessary to protect the user from pages unknown to the app that could otherwise trick the user into granting access to sensitive resources.

Sidebar: A Few iframe Tips and Cautions

As we're talking about `iframe` elements here, there are a couple extra tips you might find helpful when using them. First, to prevent selection, style the `iframe` with `-ms-user-select: none` or set its `style.msUserSelect` property to "none" in JavaScript. Second, some web pages contain frame-breaking code that prevents the page from being loaded into an `iframe`, in which case the page will be opened in the default browser and not the app. If that page is essential to your app, you'll need to work with the owner to create an alternate page that will work for you. Third, just as plug-ins aren't supported in Windows Store apps, they'll also fail to load for web pages loaded into an `iframe`. In short, pulling web content that you don't own into an app is a risky business!

Furthermore, `iframe` support is not intended to let you just build an app out of remote web pages. Section 2.4 of the [Windows 8 app certification requirements](#), in fact, specifically disallow apps that are just websites—the primary app experience must take place within the app, meaning that it doesn't happen within websites hosted in `iframe` elements. A few key reasons for this are that websites typically aren't set up well for touch interaction (which violates requirement 3.5) and often won't work well in snapped view (violating requirement 3.6). In short, overuse of web content will likely mean that the app won't be accepted by the Store.

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, the `file://` protocol is wholly blocked by design for various security reasons, even for your appdata folders to which you otherwise have full access. (Custom protocols are also unsupported in `iframe src` URIs.) 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.). This is because it wasn't feasible to create a sub-sandbox environment for such pages, without which it would be possible for a page loaded with `ms-appdata:///` to access everything in your app.

Can you do any kind of dynamic page generation, then? Well, yes: you need to load file contents and process them manually, inserting them into the DOM through `innerHTML` properties and such. 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 requires 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. Be mindful, though, that the Windows Store certification requirements specifically disallow doing this with script obtained from a remote source in the local context (see section requirement 3.9). The other inevitable caveat here is that automatic filtering is applied to that code that prevents injection of script (and other risky markup) 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 chainsaws and flaming swords 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 that I'm not including here. One such topic—the numerous variations of the `sandbox` attribute for `iframes`—is also demonstrated in the [JavaScript iframe sandbox attribute sample](#).

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

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, Windows Store apps can employ libraries like jQuery, Prototype, Dojo, and so forth, as noted in Chapter 1. However, there are some limitations and caveats.

First, because local context pages in an app cannot load script from remote sources, apps typically need to include such libraries in their packages unless only being used from the web context. WinJS, mind you, doesn't need bundling because it's provided by the Windows Store, but such "framework packages" are not enabled for third parties 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. Crucially, anything in the library that assumes a higher level of trust than the app container provides (such as open file system access) will have issues.

The most common issue comes up when libraries inject elements or script into the DOM (as through `innerHTML`), a widespread practice for web applications 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: we just take it on faith that the picture is stored somewhere so long as we get a URI. 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, to allow the user, for example, to select from previously captured pictures (as we'll do in Chapter 8, "State, Settings, Files, and Documents"). 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 `StorageFolder` 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 hyphens 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, fulfilling each 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 next promise out of each completed handler rather than fulfilling it with **done**. This allows you to indent all the async calls only once, and it 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 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 must be 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 `WinJS.Application.onerror` or `window.onerror` handlers. (The latter will get the error if the former doesn't handle it.) If you don't, the app will be 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` (as you would do if you're writing a library from which you return promises), always use `done` at the end of a chain even for a single `async` operation.¹⁸

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 Windows Store apps as quickie debugging aids. Fortunately, you have two other good options for that purpose. One is `Windows.UI.Popups.MessageDialog`, which is actually what you use for real user prompts in general. The other is `console.log`, as 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.¹⁹

Another DOM API function to which you might be accustomed is `window.close`. You can still use this as a development tool, but in released apps Windows interprets this call as a crash and generates an error report in response. This report will appear in the Store dashboard for your app, with a message

¹⁸ A number of samples in the Windows SDK use `then` instead of `done`, especially for single `async` operations. This came from the fact that `done` didn't yet exist at one point, and those samples weren't always updated.

¹⁹ 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.

telling you to not use it! After all, Store apps should not provide their own close affordances, as described in requirement 3.6 of the Store certification policy.

There might be situations, however, when a released app needs to close itself in response to unrecoverable conditions. Although you can use `window.close` for this, it's better to use `MSApp.terminateApp` because it allows you to also include information as to the exact nature of the error. These details show up in the Store dashboard, making it easier to diagnose the problem.

In addition to the Store dashboard, you should make fast friends with the Windows Event Viewer.²⁰ 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 first navigate to Application and Services Log and expand Microsoft/Windows/AppHost, left-click to select Admin (this is important), 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.

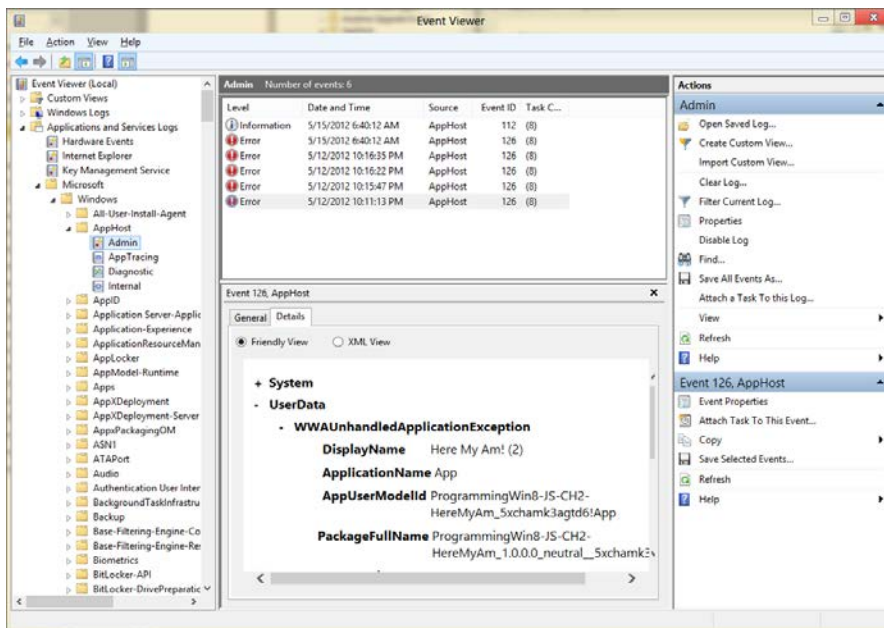


FIGURE 3-4 App host events, such as unhandled exceptions and load errors, can be found in Event Viewer.

²⁰ 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.

We already introduced Visual Studio's Exceptions dialog in Chapter 2; refer back to Figure 2-16. For each type of JavaScript exception, this dialog supplies two checkboxes labeled Thrown and User-unhandled. Checking Thrown will display a dialog box in the debugger (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.

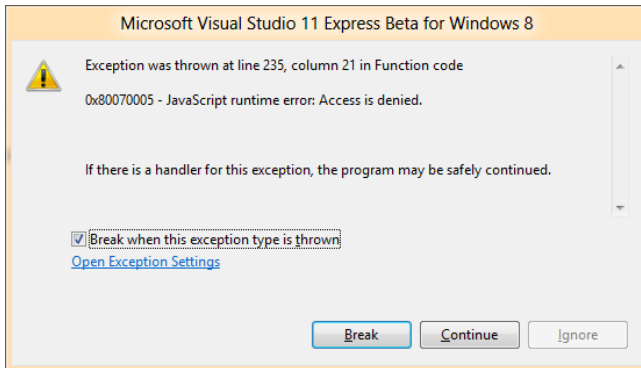


FIGURE 3-5 Visual Studio's exception dialog. As the dialog indicates, it's safe to press Continue if you have an error handler in the app; otherwise the app will terminate. Note that the checkbox in this dialog is a shortcut to toggle the Thrown checkbox for this exception type in the Exceptions dialog.

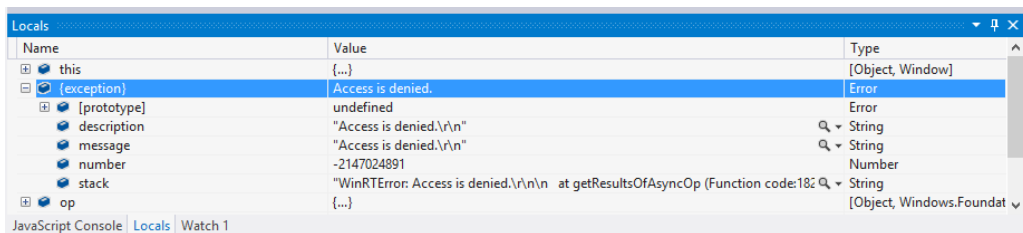


FIGURE 3-6 Information in Visual Studio's Locals pane when you Break on an exception.

The User-unhandled option (enabled for all exceptions by default) will display a similar dialog whenever an exception is thrown to the event loop, indicating that it wasn't handled by an app-provided error function ("user" code from the system's perspective).

You typically turn on Thrown only for those exceptions you care about; turning them all on can make it very difficult to step through your app! 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 here because this will include those exceptions not otherwise listed. This way you can catch (and fix) any exceptions that might abruptly terminate the app, which is something your customers should never experience.

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. This can happen a variety of ways, such as via the Start screen tile, contracts, and file type and URI scheme 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 some other way. 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 while the app gets started (much better than an hourglass). It also occupies the whole view where the app is being launched, so it's a much more directly engaging experience for your users. This view can be the filled view state, the overlay area from the share or search charm, or the snapped view if the app is immediately snapped. During this time, an instance of the app host gets launched to load, parse, and render your HTML/CSS, and load, parse, and execute your JavaScript, firing events along the way as we'll see in the next section. When the app's first page is ready, the system removes the splash screen.

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 the “all apps” view on the Start screen, search results, the Settings charm, and in the Store.	n/a	n/a	n/a
	Tile	Logo	Square tile image.	150x150 (+ 80% scale at 120x120)	210x210	270x270
		Wide logo	Optional wide tile image. If provided, this is shown as the default, but user can use the square 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 as targets. Also used on the app tile if you elect to show a logo instead of the app name in the lower left corner of the tile.	30x30 (+80% scale at 24x24)	42x42	54x54
		Show name	Specifies whether to show the app name on your app tile (both, neither, or the square or wide specifically). Set this to “no logo” if your tile images includes your app name.	n/a	n/a	n/a
		Short name	Optional: if provided, is used for the name on the tile in place of the Display Name, as Display Name may be too long for a square tile.	n/a	n/a	n/a
		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 ratio between this and the background color.	n/a	n/a	n/a
		Back-ground color	Color used for transparent areas of any tile images, the default background for secondary tiles, notification backgrounds, buttons in app dialogs, borders when the app is a provider for file picker and contact picker contracts, headers in settings panes, and the app’s page in the Store. Also provides the splash screen background color unless that is set separately.	n/a	n/a	n/a
	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 desired.	620x300	868x420	1116x540
		Back-ground color	Color that will fill the majority of the splash screen; if not set, the App UI Background color is used.	n/a	n/a	n/a

In the table, note that 80% scale tile graphics are used in specific cases like low DPI modes (generally when the DPI is less than 130 and the resolution is less than 2560 x 1440) and should be provided with other scaled images. 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 variant for the current scale. 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 to Figure 2-5 in Chapter 2—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, but note that you might need to exit and restart the simulator to see the new scaling take effect.

One thing you might also notice is that full-color photographic images don't scale down very well to the smallest sizes (store logo and small logo). This is one reason why such logos are typically simpler with Windows Store app design, which also keeps them smaller when compressed. This is an excellent consideration to keep your package size smaller when you make more versions for different contrasts and languages. We'll see more on this in Chapter 17, "Apps for Everyone."

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

Activation Event Sequence

As the app host is built on the same parsing and rendering engines as Internet Explorer, the general sequence of activation events is more or less what a web application sees in a browser. Actually, it's more rather than less! When you launch an app from its tile, here's the process as Windows sees it:

1. Windows displays a splash screen using information from the app manifest.
2. Windows launches the app host, identifying the app to launch.
3. 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.
4. 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.)

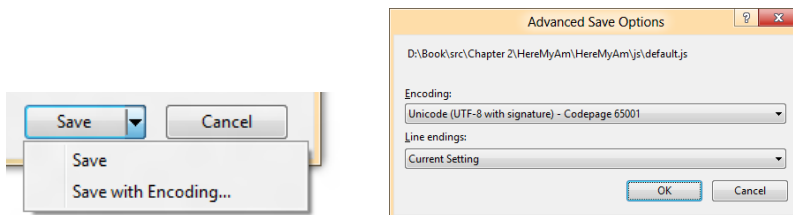
5. `document.DOMContentLoaded` fires. You can use this to do further initialization specifically related to the DOM, if desired (not common).
6. `Windows.UI.WebUI.WebUIApplication.onactivated` fires. This is typically where you'll do all your startup work, instantiate WinJS and custom controls, initialize state, and so on.
7. The splash screen is hidden once the `activated` event handler returns (unless the app has requested a deferral as discussed later in the "Activation Deferrals" section).
8. `body.onload` fires. This is typically not used in Windows Store 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.



Along these same lines, minification of JavaScript isn't particularly important for Windows Store apps. Because an app package is downloaded from the Windows Store as a unit and often contains other assets that are much larger than your code files, minification won't make much difference there. Once the package is installed, bytecode generation means that the package's JavaScript has already been processed and optimized, so minification won't have any additional performance impact.

Activation Code Paths

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

```
(function () {  
    "use strict";  
  
    var app = WinJS.Application;  
    var activation = Windows.ApplicationModel.Activation;  
  
    app.onactivated = function (args) {  
        if (args.detail.kind === activation.ActivationKind.launch) {  
            if (args.detail.previousExecutionState !==  
                activation.ApplicationExecutionState.terminated) {  
                // TODO: This application has been newly launched. Initialize  
                // your application here.  
            } else {  
                // TODO: This application has been reactivated from suspension.  
                // Restore application state here.  
            }  
            args.setPromise(WinJS.UI.processAll());  
        }  
    };  
  
    app.oncheckpoint = function (args) {  
    };  
  
    app.start();  
})();
```

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

- `(function () { ... })();` surrounding everything is again the JavaScript module pattern.
- `"use strict"` instructs the JavaScript interpreter to apply [Strict Mode](#), a feature of ECMAScript 5. This checks for sloppy programming practices, like using implicitly declared variables, so it's a good idea to leave it in place.
- `var app = WinJS.Application;` and `var activation = Windows.ApplicationModel.Activation;` both create substantially shortened aliases for commonly used namespaces. This is a common practice to simplify multiple references to the same part of WinJS or WinRT.
- `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 session state.

- `WinJS.UI.processAll` instantiates WinJS controls—that is, elements that contain a `data-win-control` attribute, as we’ll cover in Chapter 4, “Controls, Control Styling, and Data Binding.”
- `args.setPromise` instructs Windows to wait until `WinJS.UI.processAll` is complete before removing the splash screen. (See “Activation Deferrals” 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, where there are a number of potential code paths depending on the values in `args.details` (an `IActivatedEventArgs` object). If you look at the documentation for `WinJS.Application.onactivated`, you’ll see that the exact contents of `args.details` depends on specific activation kind. All activations, however, share three common properties:

args.details Property	Type (in Windows.Application-Model.Activation)	Description
<code>Kind</code>	<code>ActivationKind</code>	The reason for the activation. The possibilities are <code>launch</code> (most common); <code>search</code> , <code>shareTarget</code> , <code>file</code> , <code>protocol</code> , <code>fileOpenPicker</code> , <code>fileSavePicker</code> , <code>contactPicker</code> , and <code>cachedFileUpdater</code> (for servicing contracts); and <code>device</code> , <code>printTask</code> , <code>Settings</code> , and <code>cameraSettings</code> (generally used with device apps). For each supported activation kind, the app will have an appropriate initialization path.
<code>previousExecutionState</code>	<code>ApplicationExecutionState</code>	The state of the app prior to this activation. Values are <code>notRunning</code> , <code>running</code> , <code>suspended</code> , <code>terminated</code> , and <code>closedByUser</code> . Handling the <code>terminated</code> case is most common because that’s the one where you want to restore previously saved session state (see “App Lifecycle Transition Events”).

<code>splashScreen</code>	SplashScreen	Contains an <code>ondismissed</code> event for when the system splash screen is dismissed. This also contains an <code>imageLocation</code> property (<code>Windows.Foundation.Rect</code>) with coordinates where the splash screen image was displayed, as noted in "Extended Splash Screens."
---------------------------	------------------------------	--

Additional properties provide relevant data for the activation. For example, `launch` provides the `tileId` and `arguments` from secondary tiles. (See Chapter 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 their proper contexts, and sometimes they apply to altogether different pages than `default.html`. What's contained in the templates (and what we've already used for an app like Here My Am!) is primarily to handle normal startup from the app tile (or 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;²¹ 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 session state it needs to restart from a previous state of `terminated`. It's fired in response to both the document's `beforeunload` event, as well as `Windows.UI.WebUI.WebUIApplication.onsuspending`.
- `unload` Also fired for `beforeunload` after the `checkpoint` event is fired.

²¹ There is also `WinJS.Utilities.ready` through which you can specifically set a callback for `DOMContentLoaded`. This is used within WinJS, in fact, to guarantee that any call to `WinJS.UI.processAll` is processed after `DOMContentLoaded`.

- `settings` Fired in response to `Windows.UI.ApplicationSettings.SettingsPane.-oncommandsrequested`. (See Chapter 8.)

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 (like an `XmlHttpRequest`) within an event handler, 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, 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.

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` method drops an event into the queue that will get dispatched, after any existing queue is clear, to whatever listeners you've set up on the `WinJS.Application` object. Events are simply identified with a string, so you can queue an event with any name you like, and call `WinJS.Application.addEventListener` with that same name anywhere else in the app. This 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. (For an example of using `queueEvent`, see Scenario 2 of the [App model sample](#).)

As for `stop`, this is provided to help with unit testing so that you can simulate different activation sequences without having to relaunch the app and somehow 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.

In truth, so long as the user keeps your app in the foreground and doesn't switch away, Windows will give you all the time you need. But if the user switches to the Start screen or another app, you're subject to a 15-second timeout. If you're not in the foreground, Windows will wait only 15 seconds for an app to get through `app.start` and the `activated` event, at which point your home page should be rendered. Otherwise, boom! Windows automatically terminates your app.

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 the first time it's run after being installed, so it should let the user know that something is happening. 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 populate a local IndexedDB from data in a JSON file or download and cache a bunch of data from an online service.

It's also possible that the user is trying to launch your app shortly after rebooting the system, in which case there might be lots of disk activity going on. If you load data from disk in your activation path, your process could take much longer than usual.

In all these cases, whenever an app is at risk of exceeding 15 seconds, you want to implement an *extended splash screen*. This means hiding your 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 (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.

For an example of an extended splash screen, refer to the [Splash screen sample](#). 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 snapped 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 again 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 that the end of the chain is a promise that becomes the argument to `setPromise`—don't call its `done` method (use `then` if needed)! If the sequence isn't important but you need all of them to complete, you can combine those promises by using `WinJS.Promise.join`, passing the result to `setPromise`. If you need only one of the operations to complete, you can use `WinJS.Promise.any` instead—`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 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();

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


Of course, `setPromise` ultimately does exactly this, and if you add a handler for the WinRT `activated` 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.

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 action 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 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, as well as the `document.hidden` property. (The event works for visibility of individual elements as well.) A change in visibility is also a good time to save user data like documents or game progress.

For *view state changes*, an app can detect these in several ways. As shown in the Here My Am! example, an app typically uses media queries (in declarative CSS or in code through media query listeners) to reconfigure layout and visibility of elements, which is really all that view states should affect. (Again, view state changes never change the mode of the app, 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), suspended, and then closed, 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 due to unrecoverable conditions.

Suspend, Resume, and Terminate

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

- **Suspending** When an app is not visible in any view state, it will be suspended after five seconds (according to the wall clock) to conserve battery power. This means it remains wholly in memory but won't be scheduled for CPU time and thus won't have network or disk activity (except when using specifically allowed background tasks). When this happens, the app receives the `Windows.UI.WebUI.WebUIApplication.onsuspending` event, which is also exposed through `WinJS.Application.oncheckpoint`. Apps must return from this event within the five-second period, or Windows will assume the app is hung and terminate it (period!). During this time, apps save transient session state and should also release any exclusive resources acquired as well, like file streams or device access. (See [How to suspend an app](#).)
- **Resuming** If the user switches back to a suspended app, it receives the `Windows.UI.WebUI.WebUIApplication.onresuming` event. (This is not surfaced through `WinJS.Application` 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. In addition, if you're tracking sensor input of any kind (like compass, geolocation, or orientation), resuming is a good time to get a fresh reading. You'll also want to check license status for your app and in-app purchases if you're using trials and/or expirations (see Chapter 17). There are also times when you might want to refresh your layout (as we'll see in Chapter 6), because it's possible for your app to resume directly into a different view state than when it was suspended or to a different screen resolution as when the device has been connected to an external monitor. The same goes for enabling/disabling clipboard commands.
- **Terminating** When suspended, an app might be terminated if there's a need for more memory. There is *no event* for this, because by definition the app is already suspended and no code can run. Nevertheless, this is important for the app lifecycle because it affects `previousExecutionState` when the app restarts.

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 3-7. 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!)

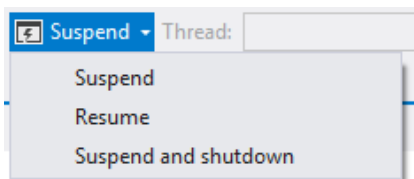


FIGURE 3-7 The Visual Studio toolbar drop-down to simulate suspend, resume, and terminate.

We've briefly listed those previous states before, but let's see how they relate to the events that get fired and the `previousExecutionState` value that shows up when the app is next launched. This can get a little tricky, so the transitions are illustrated in Figure 3-8 and the table below describes how the

`previousExecutionState` values are determined.

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

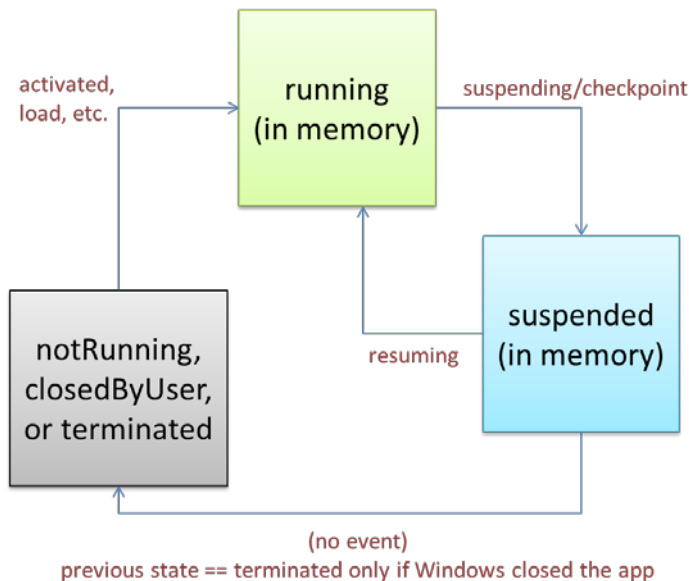


FIGURE 3-8 Process lifecycle events and `previousExecutionState` values.

The big question for the app, of course, is not so much what determines the value of `previousExecutionState` as what it should actually *do* with this value during activation. Fortunately, that story is a

bit simpler and one that we've already seen in the template code:

- If the activation kind is `launch` and the previous state is `notrunning` or `closedByUser`, the app should start up with its default UI and apply any persistent state or settings. With `closedByUser`, there might be scenarios where the app should perform additional actions (such as updating cached data) after the user explicitly closed the app and left it closed for a while.
- If the activation kind is `launch` and the previous state is `terminated`, the app should start up in the *same 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 if not sooner, save whatever transient session state it would need to rehydrate itself after being terminated. This includes unsubmitted 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 user's mind*. 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. By the time an app gets suspended, then, it needs to have saved whatever state is necessary to make this possible. It then restores that state when `previousExecutionState` is `terminated`.

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

It works out best, actually, to save session state as it changes during the app's lifetime, thereby minimizing the work needed within the `suspending` event (where you have only five seconds). Mind you, this session state does not include 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, if you haven't already done so.

Second is the `WinJS.Application.sessionState` object. On normal startup, this is just an empty

object to which you can add whatever properties you like, including other objects. A typical strategy is to just use `sessionState` directly as a container for variables. Within the `checkpoint` event, WinJS automatically serializes the contents of this object (using `JSON.stringify`) into a file within your local appdata folder (meaning that variables in `sessionState` must have a string representation). 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.

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.

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

A final aid ties into a deferral mechanism like the one for activation. The deferral is important because Windows will suspend your app as soon as you return from the `suspending` event. If you need a deferral for async operations, 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 Windows Store apps in the background? Will my app keep running indefinitely if I request a deferral by never calling `complete`?

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

A basic demonstration of using the suspending deferral, by the way, can be found in the [App activated, resume, and suspend sample](#). This also provides an example of activation through a custom

URI scheme, a subject that we'll be covering later in Chapter 12, "Contracts." An example of handling state, in addition to the updates we'll make to Here My Am! in the next section, can be found in Scenario 3 of the [App model sample](#).

Basic Session State in Here My Am!

To demonstrate some basic handling of session state, I've made a few changes to Here My Am! as given in the 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 (prepending `app.sessionState`.) as in the completed handler for `getGeopositionAsync`:

```
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://` URI scheme to refer to it. When we capture an image, we just save that URI into `sessionState.imageUrl` (the property name is arbitrary) at the end of the promise chain inside `capturePhoto`:

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

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://` URIs). 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.

Note Using `ms-appdata:///` and `getFileFromPathAsync` works because the file exists in a location that we can access programmatically by default. It also works for libraries for which we declare a capability in the manifest. If, however, we obtained a `StorageFile` from the file picker, we'd need to save that in the `Windows.Storage.AccessCache` to preserve access permissions across sessions.

Data from Services and WinJS.xhr

Though we've seen examples of using data from an app's package (via URIs 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 developer blog:

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

That is, give `WinJS.xhr` a URI and it gives back a promise that delivers its results to your completed handler (in this case `processPosts`) and will even call a progress handler if provided. With the former, the result contains a `responseXML` property, which is a `DomParser` object. With the latter, the event object contains the current XML in its `response` property, which we can easily use to display a download count:

```
function showProgress(e) {
    var bytes = Math.floor(e.response.length / 1024);
    document.getElementById("status").innerText = "Downloaded " + bytes + " KB";
}
```

The rest of the app just chews on the response text looking for `item` elements and displaying the `title`, `pubDate`, and `link` fields. With a little styling (see `default.css`), and utilizing the WinJS typography style classes of `win-type-x-large` (for `title`), `win-type-medium` (for `pubDate`), and `win-type-small` (for `link`), we get a quick app that looks like Figure 3-9. You can look at the code to see the details.²²

²² WinRT has a specific API for dealing with RSS feeds in `Windows.Web.Syndication`, which we'll see in Chapter 14. You can use this if you want a more structured means of dealing with such data sources. As it is, JavaScript has intrinsic APIs to work with XML, so it's really your choice. In a case like this, the syndication API along with `Windows.Web.AtomPub` and `Windows.Data.Xml` are very much needed by Windows 8 apps written in other languages that don't have the same built-in features as JavaScript.

Windows 8 Developer Blog

Fast and fluid animations in your Metro style app

Tue, 01 May 2012 21:02:00 GMT

<http://blogs.msdn.com/b/windowsappdev/archive/2012/05/01/fast-and-fluid-animations-in-your-metro-style-app.aspx>

Diving deep with WinRT and await

Tue, 24 Apr 2012 20:46:00 GMT

<http://blogs.msdn.com/b/windowsappdev/archive/2012/04/24/diving-deep-with-winrt-and-await.aspx>

Getting the most out of your pixels - adapting to view state changes

Thu, 19 Apr 2012 17:50:00 GMT

<http://blogs.msdn.com/b/windowsappdev/archive/2012/04/19/getting-the-most-out-of-your-pixels-adapting-to-view-state-changes.aspx>

Creating a great tile experience (part 2)

Wed, 18 Apr 2012 17:49:00 GMT

<http://blogs.msdn.com/b/windowsappdev/archive/2012/04/18/creating-a-great-tile-experience-part-2.aspx>

Creating a great tile experience (part 1)

Mon, 16 Apr 2012 17:59:00 GMT

<http://blogs.msdn.com/b/windowsappdev/archive/2012/04/16/creating-a-great-tile-experience-part-1.aspx>

Managing app lifecycle so your apps feel "always alive"

Tue, 10 Apr 2012 18:00:00 GMT

<http://blogs.msdn.com/b/windowsappdev/archive/2012/04/10/managing-app-lifecycle-so-your-apps-feel-quot-always-alive-quot.aspx>

Tackling performance killers: Common performance problems with Metro style apps

Thu, 05 Apr 2012 18:00:00 GMT

<http://blogs.msdn.com/b/windowsappdev/archive/2012/04/05/tackling-performance-killers-common-performance-problems-with-metro-style-apps.aspx>

FIGURE 3-9 The output of the SimpleXhr app.

For a fuller demonstration of XHR and related matters, refer to the [XHR, handling navigation errors, and URL schemes sample](#) along with the tutorial called [How to create a mashup](#) in the docs. I don't go into much detail with XHR in this book because it's primarily a matter of retrieving and processing data that has little to do with the Windows 8 platform. Instead, what concerns us here are the implications of suspend and resume.

In particular, an app cannot predict how long it will stay suspended before being resumed or before being terminated and restarted.

In the first case, an app that gets resumed will have all its previous data still in memory. It very much needs to decide, then, whether that data has become stale since the app was suspended and whether sessions with other servers have exceeded their timeout periods. You can also think of it this way: after what period of time will users not remember nor care what was happening the last time they saw your app? If it's a week or longer, it might be reasonable to resume or restart in a default state. Then again, if you pick up right back where they were, users gain increasing confidence that they *can* leave apps running for a long time and not lose anything. Or you can compromise and give the user options to choose from. You'll have to think through your scenario, of course, but if there's any doubt, resume where the app left off.

To check elapsed time, save a timestamp on suspend (from `new Date().getTime()`), get another timestamp in the `resuming` event, take the difference, and compare that against your desired refresh period. A Stock app, for example, might have a very short period. With the Windows 8 developer blog, on the other hand, new posts don't show up more than once a day, so a much longer period on the order of hours is sufficient to keep up-to-date and to catch new posts within a reasonable timeframe.

This is implemented in 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 (refer back to Figure 3-7), 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 folder. And speaking of databases, you may be wondering what's available for Windows Store apps other than IndexedDB. One option is SQLite, as described in [Using SQLite in a Windows Store 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 early in your development efforts. 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! On the flip side, if you were offline when suspended, check for connectivity when resuming.
- 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. Section 4.5 of the [Windows 8 app certification requirements](#), in fact, deals with 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 `html/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. There's much more you can do here, of course, but try to at least cover the basics to avoid exceptions that will terminate the app.

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 connected to it) will be canceled. Canceling is accomplished by simply calling a promise's `cancel` method.

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 Windows Store apps, using XHR with `localhost`: URI's (local loopback) is blocked by design. During development, however, this is very useful to debug a service without deploying it. You can enable local loopback in Visual Studio by opening the project properties dialog (Project menu > <project> Properties...), selecting Debugging on the left side, and setting Allow Local Network Loopback to yes. We'll see example of this in Chapter 13 where it's very useful to debug services that issue tile updates and other notifications.

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](#) documentation and its links to associated tutorials.

Page Controls and Navigation

Now we come to an aspect of Windows Store 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 Store apps).

This type of navigation presents a few problems for Store 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 applications are accustomed to waiting a bit for a browser to acquire a new page (I've found many things to do with an extra 15 seconds!), but this isn't an appropriate user experience for a fast and fluid Windows Store app. Furthermore, such a transition doesn't allow animation of various elements on and off the screen, which can help provide a sense of continuity between pages if that fits with your design.

So, although you can use direct links, Store apps typically implement "pages" by dynamically replacing sections of the DOM within the context of a single page like `default.html`, akin to how 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 is employed by the templates. Think of this as a generic wrapper around the fragment loader that lets you easily define a “page control”—simply an arbitrary unit of HTML, CSS, and JS—that you can easily pull into the context of another page as you do other controls.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. See Scenario 1 of the [HTML Page controls sample](#) for examples.

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 URIs on top of which [WinJS.Navigation](#) exposes [state](#), [location](#), [history](#), [canGoBack](#), and [canGoForward](#) properties. The stack is manipulated through the [forward](#), [back](#), and [navigate](#) methods, and the [WinJS.Navigation](#) object raises a few events—[beforenavigate](#), [navigating](#), and [navigated](#)—to anyone who wants to listen (through [addEventListener](#)).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.

Note Additional samples that demonstrate basic page controls and navigation, along with handling session state, can be found in the following SDK samples: [App activate and suspend using WinJS](#) (using the session state object in a page control), [App activated, resume and suspend](#) (described earlier; shows using the suspending deferral and restarting after termination), and [Navigation and navigation history](#).

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.

The Navigation App Template, PageControl Structure, and PageControlNavigator

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

- **default.html** Contains a single container `div` with a `PageControlNavigator` control pointing to `pages/home/home.html` as the app's home page.
- **js/default.js** Contains basic activation and state checkpoint code for the app.
- **css/default.css** Contains global styles.
- **pages/home** Contains a page control for the "home page" contents, composed of **home.html**, **home.js**, and **home.css**. Every page control typically has its own markup, script, and style files.
- **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*, like *home* in the default project structure. Then right-click that folder, select Add > New Item, and select Page Control. This will create suitably named `.html`, `.js`, and `.css` files in that folder.

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

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

All we have here is a single container `div` named *contenthost* (it can be whatever you want), in which we declare the `Application.PageControlNavigator` 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 `pages/home/home.js`; by default, the templates just provide the bare minimum:

```

(function () {
  "use strict";

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

```

The most important part is `WinJS.UI.Pages.define`, which associates a relative URI (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 URI will simply add members to an existing definition, replacing those that already exist. Be mindful that if you have a typo in the URI, including a mismatch between the URI here and the actual path to the page, the page won't load. This can be a subtle error to track down.

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_uri>)` 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
<code>init</code>	Called before elements from the page control have been created.
<code>processed</code>	Called after <code>WinJS.UI.processAll</code> is complete (that is, controls in the page have been instantiated, which is done automatically), but before page content itself has been added to the DOM.
<code>ready</code>	Called after the page have been added to the DOM.
<code>error</code>	Called if an error occurs in loading or rendering the page.
<code>unload</code>	Called when navigation has left the page.
<code>updateLayout</code>	Called in response to the <code>window.onresize</code> event, which signals changes between landscape, filled, snapped, and portrait view states.

Note that `WinJS.UI.Pages` calls the first four methods; the `unload` and `updateLayout` methods, on the other hand, are used only by the `PageControlNavigator`. Of all of these, the `ready` method is the most common one to implement. It's where you'll do further initialization of control (e.g., populate lists), wire up other page-specific event handlers, and so on. The `unload` method is also where you'll want to remove event listeners for WinRT objects, as described in "WinRT Events and removeEvent- Listener" later 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 `js/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`), or it creates a new one if none is given. The constructor also receives the `options` declared in the `data-win-options` attribute of that element. The page control then appends its contents to this root element, adds a listener for the `WinJS.Navigation.onnavigated` event, and sets up listeners for keyboard, mouse, and resizing events. It then waits for someone to call `WinJS.Navigation.navigate`, which happens in the `activated` handler of `js/default.js`, to navigate to either the home page or the last page viewed if previous session state was reloaded:

```

if (app.sessionState.history) {
  nav.history = app.sessionState.history;
}
args.setPromise(WinJS.UI.processAll().then(function () {
  if (nav.location) {
    nav.history.current.initialPlaceholder = true;
    return nav.navigate(nav.location, nav.state);
  } else {
    return nav.navigate(Application.navigator.home);
  }
}));

```

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 to the navigator control:

```

_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();
    })
    );
},

```

Here you can see how the `PageControlNavigator` calls the previous page's `unload` event. After this, the new page's content is added to the DOM, and then the old page's contents are removed. The call to `that.navigated` will then reset `this.element`.

Tip In a page control's JavaScript code you can use `this.element.querySelector` rather than `document.querySelector` if you only want to look in the page control's contents and have no need to traverse the entire DOM. Because `this.element` is just a node, however, it does not have other traversal methods like `getElementById`.

And that, my friends, is how it works! In addition to the [HTML Page controls sample](#), and to show a concrete example of doing this in a real app, the code in the `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 `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` by itself 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 are loaded and before the `PageControlNavigator` is even instantiated, we need to remember that condition so that the home page's `ready` method can later initialize itself accordingly from `app.sessionState` values. For this we simply write another flag into `app.sessionState` called `initFromState` (`true` if `previous-ExecutionState` is `terminated`, `false` otherwise.)

Sidebar: WinJS.Namespace.define and WinJS.Class.define

[WinJS.Namespace.define](#) provides a shortcut for the JavaScript namespace pattern. This helps to minimize pollution of the global namespace as each app-defined namespace is just a single object in the global namespace but can provide access to any number of other objects, functions, and so on. This is used extensively in WinJS and is recommended for apps as well, where you define everything you need in a module—that is, within a `(function() { ... })()` block—and then export selective variables or functions through a namespace. In short, use a namespace anytime you're tempted to add any global objects or functions!

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 as a namespace member. Then you can use `new <namespace>.<class>` anywhere in the app.

Sidebar: Helping Out IntelliSense

In Windows Store apps you might encounter certain markup structures within code comments, often starting with a triple slash, `///`. These are used by Visual Studio and Blend to provide rich IntelliSense within the code editors. You'll see, for example, `/// <reference path.../>` comments, which create a relationship between your current script file and other scripts to resolve externally defined functions and variables. This is explained on the [JavaScript IntelliSense](#) page in the documentation. For your own code, especially with namespaces and classes that you will use from other parts of your app, use these comment structures to describe your interfaces to IntelliSense. For details, see [Extending JavaScript IntelliSense](#), and look around the WinJS JavaScript files for many examples.

The Navigation Process and Navigation Styles

Having seen how page controls, [WinJS.UI.Pages](#), [WinJS.Navigation](#), and the [PageControlNavigator](#) all relate, it's straightforward to see how to navigate between multiple pages within the context of a single HTML 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, unloading any previous page. 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 `pages/groupedItems/groupedItems.html`, where an inline `onclick` handler event navigates to `pages/groupDetail/groupDetail.html` with an argument identifying the specific group to display. That argument comes into the `ready` function of `pages/groupDetail/groupDetail.js`.
- Tapping an item on the hub page goes to detail page (**pages/itemDetail**). The `itemInvoked` handler for the items, the `_itemInvoked` function in `pages/groupedItems/groupedItem.js`, calls `WinJS.Navigation.navigate("/pages/itemDetail/itemDetail.html")` with an argument identifying the specific item to display. As with groups, that argument comes into the `ready` function of `pages/itemDetail/itemDetail.js`.
- Tapping an item in the section page also goes to the details page through the same mechanism—see the `_itemInvoked` function in `pages/groupDetail/groupDetail.js`.

- The back buttons on all pages 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 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 along with the app bar, as we'll see in Chapter 7, "Commanding UI"). When using page controls and [PageControlNavigator](#), navigation controls can just invoke [WinJS.Navigation.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 Windows Store apps](#). This is an essential topic for designers.

Sidebar: Initial Login and In-App Licensing Agreements (EULA) Pages

Some apps might require either a login or acceptance of a license agreement to do anything, and thus it's appropriate that such pages are the first to appear in an app after the splash screen. In these cases, if the user does not accept a license or doesn't provide a login, the app should display a message describing the necessity of doing so, but it should *always* leave it to the user to close the app if desired. Do not close the app automatically.

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.PasswordVault](#) API. If the user accepts a EULA, that fact should be saved in appdata and reloaded anytime the app needs to check. These settings (login and acceptance of a license) should then always be accessible through the app's Settings charm. Legal notices, by the way, as well as license agreements, should always be accessible through Settings as well. See [Guidelines and checklist for login controls](#).

Optimizing Page Switching: Show-and-Hide

Even with page controls, there is still a lot going on when navigating from page to page: one set of elements is removed from the DOM, and another is added in. Depending on the pages involved, this can be an expensive operation. For example, if you have a page that displays a list of hundreds or

thousands of items, where tapping any item goes to a details page (as with the Grid App template), hitting the back button from a detail page will require reconstruction of the list.

Showing progress indicators can help alleviate the user's anxiety, and the recommendation is to show such indicators after two seconds and provide a means to cancel the operation after ten seconds. Even so, users are notoriously impatient and will likely want to quickly switch between a list of items and item details. In this case, page controls might not be the best design.

You could use a split (master-detail) view, of course, but that means splitting the screen real estate. An alternative 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.

Note that because the [PageControlNavigator](#) is provided by the templates as part of your app, you can modify it however you like to provide this kind of capability in a more structured manner.

WinRT Events and `removeEventListener`

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

```
var myNumber = 1;
element.addEventListener(<event>, function (e) { myNumber++; } );
```

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

To ensure that such variables are available to that anonymous function when it's later invoked as an event handler, the JavaScript engine creates a *closure*, a data structure that describes the local variables available to that function. Usually the closure is a small bit of memory, but depending on the code inside that event handler, the closure could encompass the entire global namespace—a rather large allocation!

Every such closure increases the memory footprint or *working set* of the app, so it's a good practice to keep them at a minimum. For example, declaring a separate named function—which has its own scope—will reduce any necessary closure.

More important than minimizing closures is making sure that the event listeners themselves—and their associated closures—are properly cleaned up and their memory allocations released.

Typically, this is not even something you need to think about. When object such as HTML elements are destroyed, such as when a page control is unloaded from the DOM, their associated listeners are automatically removed and closures are released. However, in a Windows Store app written in HTML and JavaScript, there are other sources of events for which the app might add event listeners, where those objects are never destroyed. These can be objects from WinJS, objects from WinRT, and [window](#) and [document](#). Those listeners must be cleaned up properly, or else the app will have memory leaks (memory that is allocated but never freed because it's never released for garbage collection).

Of special concern are events that originate from WinRT objects. Because of the nature of the projection layer that makes WinRT available in JavaScript, WinRT ends up holding references to JavaScript event handlers (known also as *delegates*) while the JavaScript closures hold references to those WinRT objects. As a result of these cross-references, those closures might never be released.

This is not a problem, mind you, if the app *always* listens to a particular event. For example, the [suspending](#) and [resuming](#) events are two that an app typically listens to for its entire lifetime, so any related allocations will be cleaned up when the app is terminated. The same is true for most listeners you might add for [window](#) and [document](#) events, which persist for the lifetime of the app.

Memory leaks occur, however, when an app listens to a WinRT object event only temporarily and neglects to explicitly call [removeEventListener](#), or when the app might call [addEventListener](#) for the same event multiple times (in which case you can end up duplicating closures). With page controls, as discussed in this chapter, it's common to call [addEventListener](#) within the page's [ready](#) method on some WinRT object. When you do this, *be sure to match that call with [removeEventListener](#) in the page's [unload](#) method to release the closures*. I've done this in [HereMyAm3d](#) with [datarequested](#) in [pages/home/home.js](#) just to be clear.

Throughout this book, the WinRT events with which you need to be concerned are highlighted with a special color, as in [datarequested](#) (except where the text is also a hyperlink). This is your cue to check whether an explicit call to [removeEventListener](#) is necessary. Again, if you'll always be listening for the event, removing the listener isn't needed, but if you add a listener when loading a page control, you almost certainly will need to make that extra call. Be especially aware that the samples don't necessary pay attention to this detail, so don't follow any examples of neglect there. Finally, note that events from WinJS objects don't need this attention because the library already handles removal of event listeners.

In the chapters that follow, I will remind you of what we've just discussed on our first meaningful encounter with a WinRT event. Keep your eyes open for the color coding in any case.

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. Demonstrations of what we'll cover here can be found in the [WinJS Promise sample](#), and if you want the fuller *async* story, read [Keeping apps fast and fluid with asynchrony in the Windows Runtime](#) on the Windows 8 developer blog.

In review, let's step back for a moment to revisit what a promise really *means*. Simply said, it's an object that returns a value, simple or complex, sometime in the future. The way you know when that value is available is by calling the promise's `then` or `done` method with a *completed handler*. That handler will be called with the promised value (the *result*) when it is ready—which will be immediately if the value is already available! Furthermore, you can call `then/done` multiple times for the same promise, and you'll just get the same result in each completed handler. 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 handler* that will be called instead of the completed handler. Otherwise exceptions are swallowed by `then` or thrown to the event loop by `done`, as we've discussed.

A third parameter to `then/done` is a *progress handler*, 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 handler with that value as soon as you call `then` or `done`. This allows you to use any value 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 specific promise instance:

- `is` determines whether an arbitrary value is a promise. It basically makes sure it's an object with a function named "then"; it does not test for "done".

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. You can learn more about this in Chapter 16.

- `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 handlers 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 handlers. It's an addition, not a 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>]`. Here `<init>` is a function that accepts completed, error, and progress *dispatchers*, and `oncancel` is an optional function that's called in response to `WinJS.Promise.cancel`. The dispatchers are what you call to trigger any completed, error, or progress handlers given to the promise's `then` or `done` methods, whereas `oncancel` is your own function that the promise will call if it's canceled. Creating a new promise in this way is typically done when you create an async function of your own. For example, we'll see how this is used to encapsulate an async web worker in Chapter 16, "WinRT Components."

Also, 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

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.

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, and deferrals when the app needs to use async operations on startup.
- 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.
- 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 Windows 8 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 Windows 8 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, "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](#) that identifies all the controls that are available to you, as we'll summarize later in this chapter. The second are comprehensive [UX Guidelines for Windows 8 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 `<div>` 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>, ... }`.

Hint If you've just made changes to `data-win-options` and your app seems to terminate without reason (and without an exception) when you next launch it, check for syntax errors in the options string. Forgetting the closing `}`, for example, will cause this behavior.

The constructor function itself takes two parameters: the root (parent) element and an options object. Conveniently, `WinJS.Class.define` 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 Windows 8 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 HTML controls works as you would expect. You can declare them in markup by using attributes to specify options, the rundown of which is given in the table following Figure 4-2. You can also create them procedurally from JavaScript by calling [new](#) with the appropriate constructor, configuring properties and listeners as desired, and adding the element to the DOM wherever its needed. Nothing new here at all where Windows 8 apps are concerned.

For examples of creating and using these controls, refer to the [HTML essential controls sample](#) in the Windows SDK, from which the images in Figure 4-1 and Figure 4-2 were obtained.

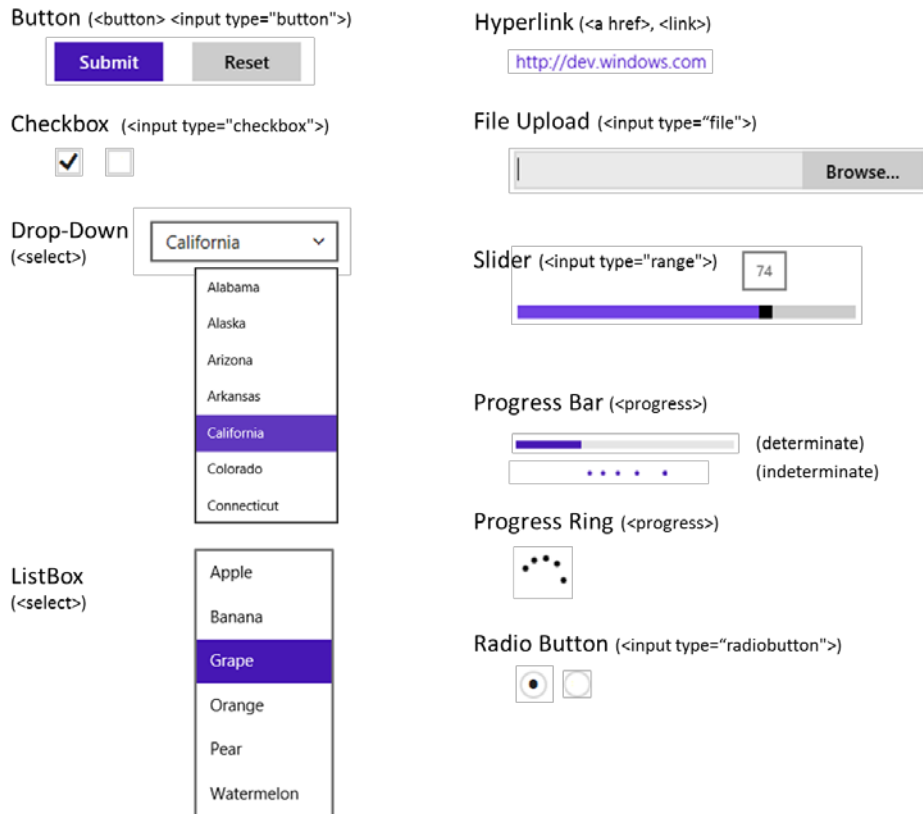


FIGURE 4-1 Standard HTML5 controls with default "light" styles (the ui-light.css stylesheet of WinJS).

Single-line text (<input>; all HTML5 types are supported)

Single line text input
<input type="text" />

Password input
<input type="password" />

Number input
<input type="number" />

Email input
<input type="email" />

Phone number input
<input type="tel" />

URL input
<input type="url" />

Clear Button shows when entering text; oninput event fires when pressed.

Reveal button (shows password characters)

Select some text and then click the "Bold" button.

This is plain text.

.....

8

youremail@email.com

1234567890

http://www.microsoft.com

Multi-line text (<textarea>)

Multi-line text input
<textarea></textarea>

Rich text (<div>)

Multi-line rich text input
<div> with contentEditable="true" and some custome styles.

Bold

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">	(note that without type, the default is "submit")	button text
Button	<input type="button"> <input type="submit"> <input type="reset">	value (button text)	n/a
Checkbox	<input type="checkbox">	value, checked	n/a (use a label element around the input control to add clickable text)
Drop Down List	<select>	size="1" (default), multiple, selectedIndex	multiple <option> elements
Email	<input type="email">	value (initial text)	n/a
File Upload	<input type="file">	accept (mime types), mulitple	n/a

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

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 Windows Store 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-small` for a rating control, `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 Windows Store 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 engine, it helps to know about those extensions that still apply. These are summarized in the table below, 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 [msBackfaceVisibility](#). This is because these standards are now far enough along that they no longer need vendor prefixes with Internet Explorer 10 or Store apps (though the [ms*](#) variants still work).

Methods	Description
msMatchesSelector	Determines if the control matches a selector.
ms[Set Get Release]PointerCapture	Captures, retrieves, and releases pointer capture for an element.
Style properties (on <code>element.style</code>)	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.

WinJS Controls

Windows 8 defines a number of controls that help apps fulfill Windows app design guidelines. As noted before, these are implemented in WinJS for apps written in HTML, CSS, and JavaScript, rather than WinRT; this allows those controls to integrate naturally with other DOM elements. Each control is

defined as part of the `WinJS.UI` namespace using `WinJS.Class.define`, where the constructor name matches the control name. So the full constructor name for a control like the Rating is `WinJS.UI.Rating`.

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.

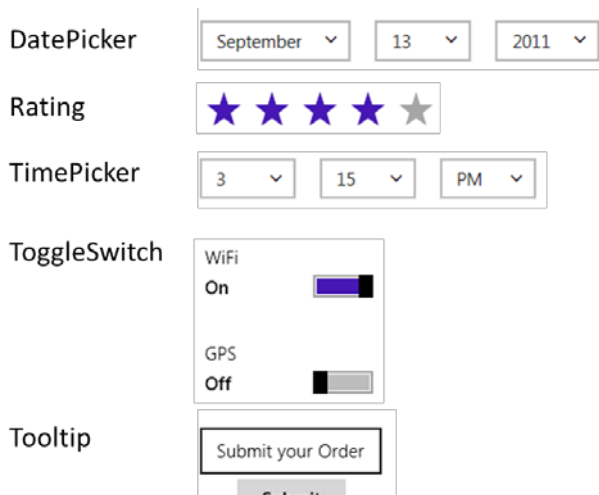


FIGURE 4-3 Default (light) styles on the simple WinJS controls.

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 the `title` attribute. 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 the table below, 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 (note that event names use the 'on' prefix in the attribute syntax)
WinJS.UI.DatePicker	Properties: <code>calendar</code> , <code>current</code> , <code>datePattern</code> , <code>disabled</code> , <code>maxYear</code> , <code>minYear</code> , <code>monthPattern</code> , <code>yearPattern</code> Events: <code>onchange</code>
WinJS.UI.Rating	Properties: <code>averageRating</code> , <code>disabled</code> , <code>enableClear</code> , <code>maxRating</code> , <code>tooltipStrings</code> (an array of strings the size of <code>maxRating</code>), <code>userRating</code> Events: <code>oncancel</code> , <code>onchange</code> , <code>onpreviewchange</code>
WinJS.UI.TimePicker	Properties: <code>clock</code> , <code>current</code> , <code>disabled</code> , <code>hourPattern</code> , <code>minuteIncrement</code> , <code>periodPattern</code> . (Note that the date portion of <code>current</code> will always be July 15, 2011 because there are no known daylight savings time transitions on this day.) Events: <code>onchange</code>
WinJS.UI.ToggleSwitch	Properties: <code>checked</code> , <code>disabled</code> , <code>labelOff</code> , <code>labelOn</code> , <code>title</code> Events: <code>onchange</code>
WinJS.UI.Tooltip	Properties: <code>contentElement</code> , <code>innerHTML</code> , <code>infoTip</code> , <code>extraClass</code> , <code>placement</code> Events: <code>onbeforeclose</code> , <code>onbeforeopen</code> , <code>onclosed</code> , <code>onopened</code> Methods: <code>open</code> , <code>close</code>

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:

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

Clearly, then, the bulk of the work really happens in the constructor. Once this takes place, other JavaScript code (as in your `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, provide a completed handler to the promise's `done` method.

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

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 a matter referred to as *strict processing*.

Strict Processing and processAll Functions

WinJS has three DOM-traversing functions: `WinJS.UI.processAll`, `WinJS.Binding.processAll` (which we'll see later in 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 enforced within all HTML/JavaScript apps. The effect of strict processing is that any functions indicated in markup that `processAll` methods might encounter must be "marked for

processing” or else processing will fail. The mark itself is simply a property named `supportedForProcessing` on the function object that is set to `true`.

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

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

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

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

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

Example: WinJS.UI.Rating Control

OK, now that we got the strict processing stuff covered, let’s see some concrete examples 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 either of the following calls:

```
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 handle exceptions that might have occurred (and that are otherwise swallowed). Also, note that the `changeRating` function specified in the markup must be globally visible and marked for processing, or else the control will fail to instantiate.

Alternately, we can instantiate the control and set the options procedurally. In markup:

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

And in code:

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

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

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

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

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

and we call `new` on the constructor ourselves:

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

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

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

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

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

To see this control in action, please refer to the [HTML Rating control sample](#).

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.

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, consider this piece of hidden HTML in our markup that contains other controls:

```

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

```

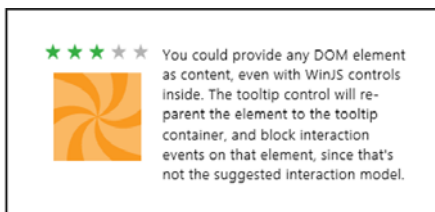
We can reference it like so:

```

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

```

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



This example is taken directly from the [HTML Tooltip control sample](#), so you can go there to see how all this works 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 2-2, the Assets tab in Blend gives you quick access to all the HTML elements and WinJS controls (among many other elements) that you can just drag and drop into whatever page is showing in the artboard. (See Figure 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.)

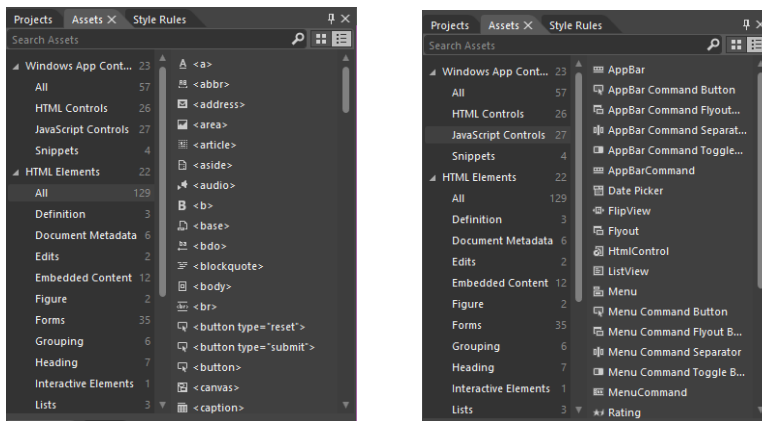


FIGURE 4-4 HTML elements (left) and WinJS control (right) as shown in Blend's Assets tab.

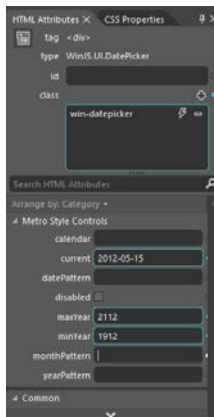


FIGURE 4-5 Blend's HTML Attributes tab shows WinJS control options, and editing them will affect the `data-win-options` attribute in markup.

Next, take a moment to load up the [HTML essential controls sample](#) into Blend. This is a great opportunity to try out Blend's Interactive Mode to navigate to a particular page and explore the interaction between the artboard and the Live DOM. (See Figure 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. A short demonstration of using interactive mode in this way is given in Video 4-1 in this chapter's companion content.

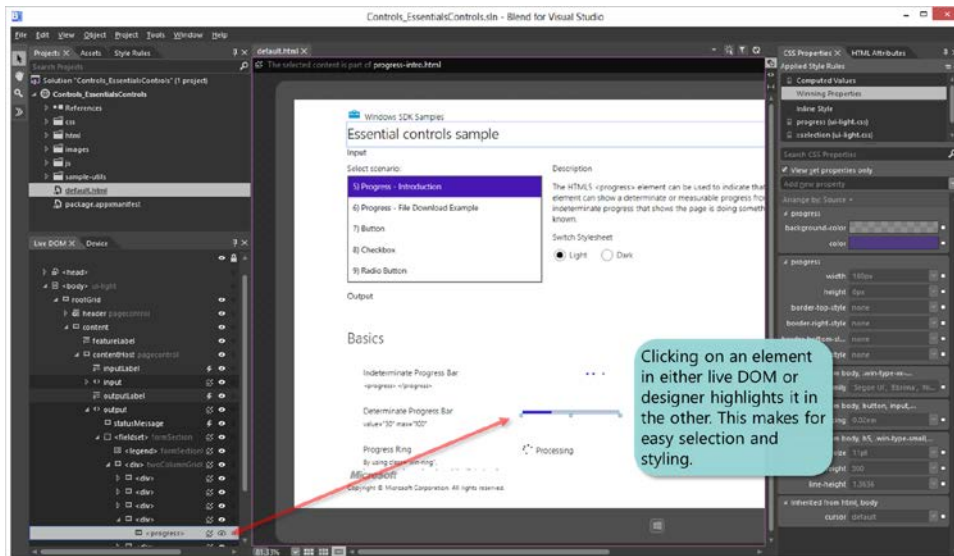


FIGURE 4-6 Blend's interaction between the artboard and the Live DOM.

With the HTML essential controls sample, you'll see that there's just a single element in the Live DOM for intrinsic controls, as there should be, 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.)

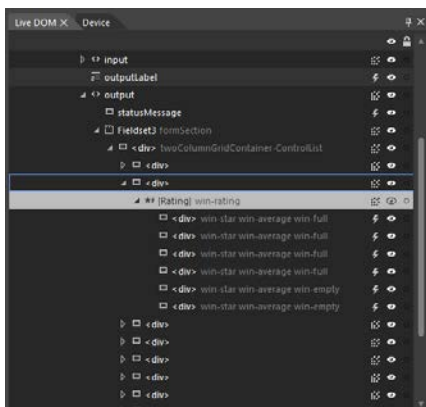


FIGURE 4-7 Expanding a WinJS control in Blend's Live DOM reveals the elements that are used to build it.

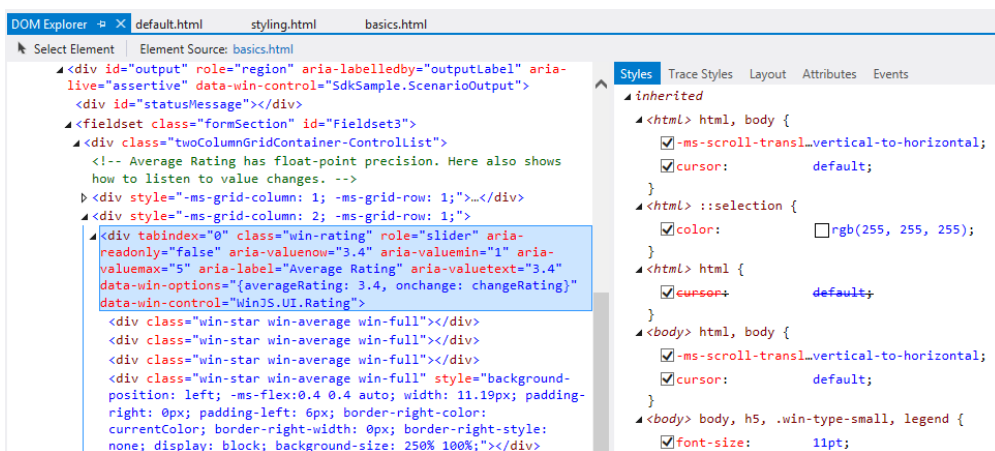


FIGURE 4-8 Expanding a WinJS control in Visual Studio's DOM Explorer also shows complete details for a control.

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 Store 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 the following table.

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

For all of these and more, spend some time with these three reference topics: [WinJS CSS classes for typography](#), [WinJS CSS classes for HTML controls](#), and [CSS classes for WinJS controls](#). I also wanted to provide you with a summary of all the other vendor-prefixed styles (or selectors) that are supported within the CSS engine for Store apps; see the next table. Vendor-prefixed styles for animations, transforms, and transitions are still supported, though no longer necessary, because these standards have recently been finalized. I made this list because the documentation here can be hard to penetrate: you have to click through the individual pages under the [Cascading Style Sheets](#) topic in the docs to see what little bits have been added to the CSS you already know.

Area	Styles
Backgrounds and borders	<code>-ms-background-position-[x y]</code>
Box model	<code>-ms-overflow-[x y]</code>
Basic UI	<code>-ms-text-overflow</code> (for ellipses rendering) <code>-ms-user-select</code> (sets or retrieves where users are able to select text within an element) <code>-ms-zoom</code> (optical zoom)


Flexbox	<code>-ms-[inline-]flexbox</code> (values for <code>display</code>); <code>-ms-flex</code> and <code>-ms-flex-[align direction order pack wrap]</code>
Gradients	<code>-ms-[repeating-]linear-gradient</code> , <code>-ms-[repeating-]radial-gradient</code>
Grid	<code>-ms-grid</code> and <code>-ms-grid-[column column-align columns column-span grid-layer row row-align rows row-span]</code>
High contrast	<code>-ms-high-contrast-adjust</code>
Regions	<code>-ms-flow-[from into]</code> along with the <code>MSRangeCollection</code> method
Text	<code>-ms-block-progression</code> , <code>-ms-hyphens</code> and <code>-ms-hyphenate-limit-[chars lines zone]</code> , <code>-ms-text-align-last</code> , <code>-ms-word-break</code> , <code>-ms-word-wrap</code> , <code>-ms-ime-mode</code> , <code>-ms-layout-grid</code> and <code>-ms-layout-grid-[char line mode type]</code> , and <code>-ms-text-[autospace kashida-space overflow underline-position]</code>
Other	<code>-ms-writing-mode</code>

Styling Gallery: HTML Controls


Now we get to enjoy a visual tour of styling capabilities for Windows Store apps. Much can be done with standard styles, and then there are all the things you can do with special styles and classes as shown in the graphics in this section. The specifics of all these examples can be seen in the [HTML essential controls sample](#).

Also check out the very cool [Applying app theme color \(theme roller\) sample](#). This beauty lets you configure the primary and secondary colors for an app, shows how those colors affect different controls, and produces about 200 lines of precise CSS that you can copy into your own stylesheet. This very much helps you create a color theme for your app, which we very much encourage to establish an app's own personality within the overall Windows 8 design guidelines and not try to look like the system itself. (Do note that controls in system-provided UI, like the confirmation flyout when creating secondary tiles, will be styled with system colors. These cannot be controlled by the app.)

Button (background-color)



Progress (color)



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

Apple

▼

Fruits

Apple

Banana

Grape

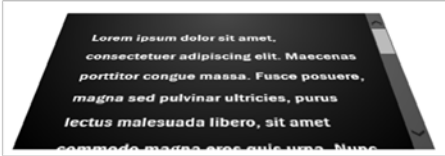
Vegetables

Broccoli


Carrot


Eggplant

Text Area (transform)



Checkbox/Radiobutton (background-image and :checked)





Button

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

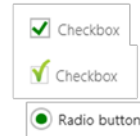


Checkbox/Radiobutton

```
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

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

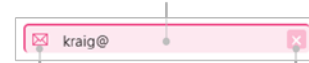


```
CSS pseudo-element: input[type="file"].<class>::-ms-browse
```

Text Input (most forms)

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

```
CSS pseudo-class: input[type="<type>"].<class>::-ms-input-placeholder
```



```
CSS background image (and other styles)
```

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

Progress

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



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



Text Input (password)

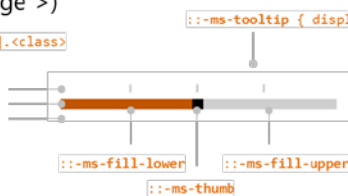
```
CSS pseudo-element: input[type="password"].<class>::-ms-reveal
```



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

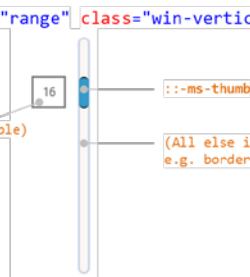
```
CSS pseudo-elements on input[type="range"].<class>
```

```
::-ms-ticks-before (top side)
::-ms-track (track area incl. ticks)
::-ms-ticks-after (bottom side)
```



```
<input type="range" class="win-vertical">
```

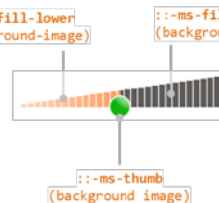
```
Default tooltip (visible)
```



```
(All else is standard CSS
e.g. border-radius)
```

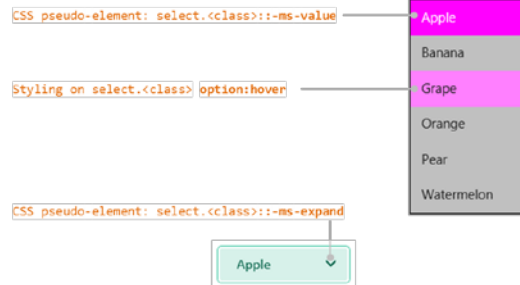
```
::-ms-tooltip { display:none; } (only recognized style)
```

```
::-ms-fill-lower (background-image)
::-ms-fill-upper (background-image)
```



```
::-ms-track (color
and background-color
set to transparent)
```

Combo/list box (<select>)



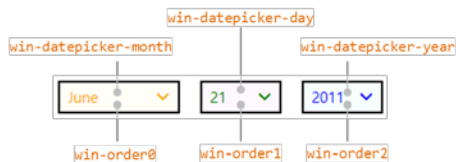
Note Though not shown here, you can also use the `-ms-scrollbar-*` styles for scrollbars that appear on pannable content in your app.

Styling Gallery: WinJS Controls

Similarly, here is a visual rundown of styling for WinJS controls, drawing again from the samples in the SDK: [HTML DatePicker and TimePicker controls](#), [HTML Rating control](#), [HTML ToggleSwitch control](#), 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:

- `win-timepicker` and `win-datepicker` style the whole control (you override defaults)
- `win-datepicker-*` style individual parts (display: none will hide that part)
- `win-orderN` identifies the sub-element by position
- Style `{ display: block; float: none }` on children for vertical layout



```
.win-datepicker [class^="win-datepicker"] {
  display: block;
  float: none;
}
```

```
.win-datepicker .win-datepicker-year {
  color: blue;
}

.win-datepicker .win-datepicker-date {
  color: green;
}

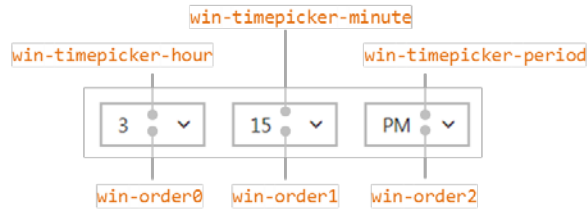
.win-datepicker .win-datepicker-month {
  color: orange;
}

.win-datepicker .win-datepicker-year::-ms-expand {
  color: red;
}

.win-datepicker .win-order0 {
  background-color: rgb(255, 255, 248);
}

.win-datepicker .win-order1 {
  background-color: rgb(255, 248, 255);
}

.win-datepicker .win-order2 {
  background-color: rgb(248, 255, 255);
}
```



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 this table:

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

`.win-rating .win-star.win-user.win-full (colors)`



`.win-rating .win-star (font size)`



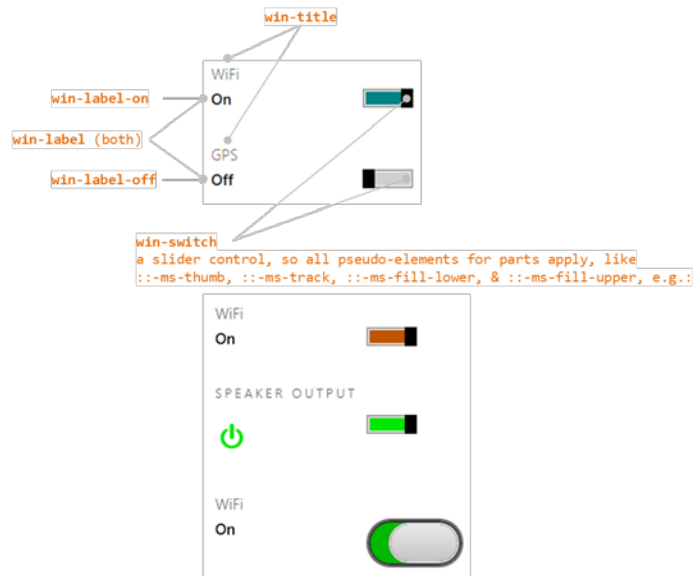
`class="win-small"`



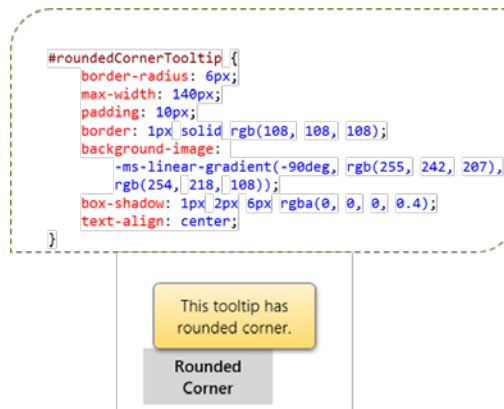
`.win-rating .win-star (background image)`



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



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:





Some Tips and Tricks

- 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.
- The `::-ms-tooltip` pseudo-selector for the slider affects only visibility (with `display: none`); it cannot be used to style the tooltip generally. This is useful to hide the default tooltips if you want to implement custom UI of your own.
- There are additional types of `input` controls (different values for the `type` attribute) that I haven't mentioned. This is because those types have no special behaviors and just render as a text box. Those that have been specifically identified might also just render as a text box, but they can affect, for example, what on-screen keyboard configuration is displayed on a touch device (see Chapter 9) and also provide specific input validation (e.g., the number type only accepts digits).
- The WinJS attribute, `data-win-selectable`, when set to `true`, specifies that an element is selectable in the same way that all `input` and `contenteditable` elements are.
- If you don't find `width` and `height` properties working for a control, try using `style.width` and `style.height` instead.
- You'll notice that there are two kinds of button controls: `<button>` and `<input type="button">`. They're visually the same, but the former is a block tag and can display HTML inside itself, whereas the latter is an inline tag that displays only text. A `button` also defaults to `<input type="submit">`, which has its own semantics, so you generally want to use `<button type="button">` to be sure.

- If a `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.
- The HTML5 `meter` element is not supported for Store apps.
- There's a default dotted outline for a control when it has the focus (tabbing to it with the keyboard or calling the `focus` method in JavaScript). To turn off this default rectangle for a control, use `<selector>:focus { outline: none; }` in CSS.
- Store apps can use the `window.getComputedStyle` method to obtain a `currentStyle` object that contains the applied styles for an element, or for a pseudo-element. This is very helpful, especially for debugging, because pseudo-elements like `::-ms-thumb` for an HTML slider control never appear in the DOM, so the styling is not accessible through the element's `style` property nor does it surface in tools like Blend. Here's an example of retrieving the background color style for a slider thumb:

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

Custom Controls

As extensive as the HTML and WinJS controls are, there will always be something you wish the system provided but doesn't. "Is there a calendar control?" is a question I've often heard. "What about charting controls?" These clearly aren't included directly in 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, follow this general 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 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. 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.

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.

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: WinJS.Class.define(function (element, options) {
    element.winControl = this;
    this.element = element;

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

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

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

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

For a more complete control, you can take a look at the [HTML SemanticZoom for custom controls sample](#). My friend Kenichiro Tanaka of Microsoft Tokyo also created the calendar control shown in Figure 4-9 and provided in the CalendarControl example for this chapter. (Note that this is example is only partly sensitive to localized calendar settings; it is not meant to be full-featured.)

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`.

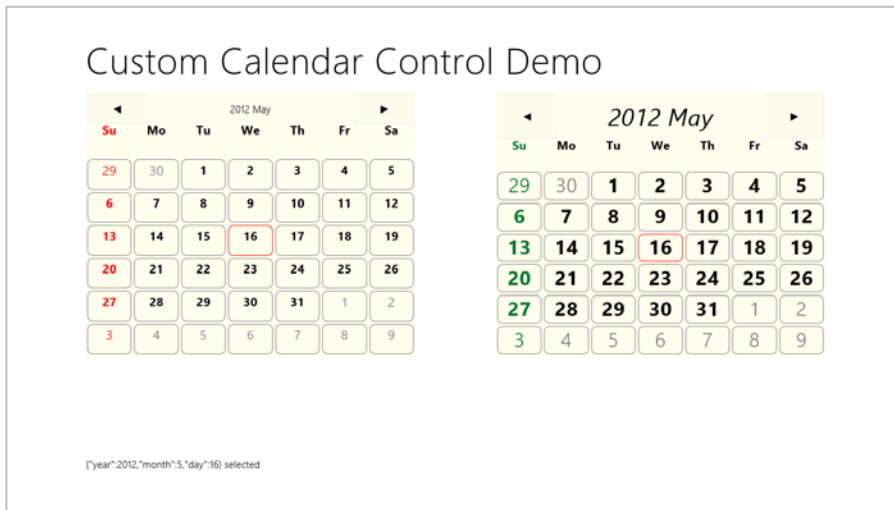


FIGURE 4-9 Output of the Calendar Control example.

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

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 calls above into a single one.

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, ondataselected: CalendarDemo.dataselected}">
</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", {
    dataselected: 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.ondataselected = function (e) {
    document.getElementById("message").innerText = JSON.stringify(e.detail) + " selected";
}
```

There you have it!

Note For a control you really intend to share with others, you'll want to include the necessary comments that provide metadata for IntelliSense. See the "Sidebar: Helping Out IntelliSense" in Chapter 3 for more details. You'll also want to make sure that the control fully supports considerations for accessibility and localization, as discussed in Chapter 17, "Apps for Everyone."

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.

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`, doesn't work in Blend's design mode as it does when the app is actually running for real. To be blunt, it 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 URI to their back-end services, which was failing in design mode. Using the design mode flag, then, we just had to change the code to look like this:

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

Finally, it is possible to have custom controls show up in the Assets tab alongside the HTML elements and the WinJS controls. For this you'll first need an [OpenAjax Metadata XML \(OAM\) file](#) that provides all the necessary information for the control, and you already have plenty of references to draw from. To find them, search for `*_oam.xml` files within *Program Files (x86)*. You should find some under the *Microsoft Visual Studio 11.0* folder and *deep* down within *Microsoft SDKs* where WinJS metadata lives. In both places you'll also find plenty of examples of the 12x12 and 16x16 icons you'll want for your control.

If you look in the controls/calendar folder of the CalendarControl example with this chapter, you'll find `calendar_oam.xml` and two icons alongside the `.js` and `.css` files. The OAM file (that must have a filename ending in `_oam.xml`) tells Blend how to display the control in its Assets panel and what code it should insert when you drag and drop a control into an HTML file. Here are the contents of that file:


```

<?xml version="1.0" encoding="utf-8"?>
<!-- Use underscores or periods in the id and name, not spaces. -->
<widget version="1.0"
  spec="1.0"
  id="http://www.kraigbrockschmidt.com/scehmas/ProgrammingWin8_JS/Controls/Calendar"
  name="ProgWin8_JS.Controls.Calendar"
  xmlns="http://openajax.org/metadata">

  <author name="Kenichiro Tanaka" />

  <!-- title provides the name that appears in Blend's Assets panel
    (otherwise it uses the widget.name). -->
  <title type="text/plain"><![CDATA[Calendar Control]]></title>

  <!-- description provides the tooltip fir Assets panel. -->
  <description type="text/plain"><![CDATA[A single month calendar]]></description>

  <!-- icons (12x12 and 16x16 provide the small icon next to the control
    in the Assets panel. -->
  <icons>
    <icon src="calendar.16x16.png" width="16" height="16" />
    <icon src="calendar.12x12.png" width="12" height="12" />
  </icons>

  <!-- This element describes what gets inserted into the .html file;
    comment out anything that's not needed -->
  <requires>
    <!-- The control's code -->
    <require type="javascript" src="calendar.js" />

    <!-- The control's stylesheet -->
    <require type="css" src="calendar.css" />

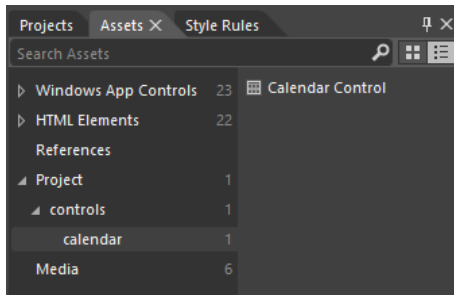
    <!-- Any inline script for the document head -->
    <require type="javascript"><![CDATA[WinJS.UI.processAll();]]></require>

    <!-- Inline CSS for the style block in the document head -->
    <!--<require type="css"><![CDATA[.control-calendar{}]]></require>-->
  </requires>

  <!-- What to insert in the body for the control; be sure this is valid HTML
    or Blend won't allow insertion -->
  <content>
    <![CDATA[
      <div class="control-calendar" data-win-control="Controls.Calendar"
        data-win-options="{ year: 2012, month: 6 }"></div>
    ]]>
  </content>
</widget>

```

When you add all five files into a project in Blend, you'll see the control's icon and title in the Assets tab (and hovering over the control shows the tooltip):



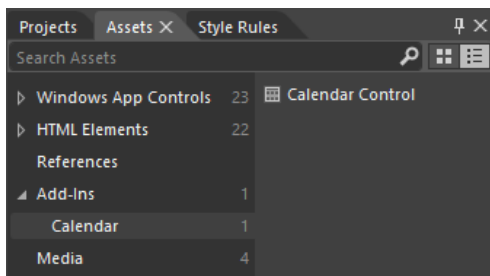
If you drag and drop that control onto an HTML page, you'll then see the different bits added in:

```
<!DOCTYPE html>
<html>
<head>
  <!-- ... -->
  <script src="calendar.js" type="text/javascript"></script>
  <link href="calendar.css" rel="stylesheet" type="text/css">
</head>
<body>
  <div class="control-calendar" data-win-control="Controls.Calendar"
    data-win-options="{month:6, year:2012}"></div>
</body>
</html>
```

But wait! What happened to the `WinJS.UI.processAll()` call that the XML indicated a `script` tag in the header? It just so happens that Blend singles out this piece of code to check if it's already being called somewhere in the loaded script. If it is (as is typical with the project templates), Blend doesn't repeat it. If it does include it, or if you specify other code here, Blend will insert it in a `<script>` tag in the header.

Also, errors in your OAM file will convince Blend that it shouldn't insert the control at all, so you'll need to fix those errors. When making changes, Blend won't reload the metadata unless you reload the project or rename the OAM file (preserving the `_oam.xml` part). I found the latter is much easier, as Blend doesn't care what the rest of the filename looks like. In this renaming process too, if you find that the control disappeared from the Assets panel, it means you have an error in the OAM XML structure itself, such as attribute values containing invalid characters. For this you'll need to do some trial and error, and of course you can refer to all the OAM files already on your machine for details.

You can also make your control available to all projects in Blend. To do this, go to *Program Files (x86)\Microsoft Visual Studio 11.0\Blend*, create a folder called *Addins* if one doesn't exist, create a subfolder therein for your control (using a reasonably unique name), and copy all your control assets there. When you restart Blend, you'll see the control listed under *Addins* in the Assets tab:



This would be appropriate if you create custom controls for other developers to use; your desktop installation program would simply place your assets in the Addins folder. As for using such a control, when you drag and drop the control to an HTML file, its required assets (but not the icons nor the OAM file) are copied to the project into the root folder. You can then move them around however you like, patching up the file references, of course.

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 (or *context*) 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.

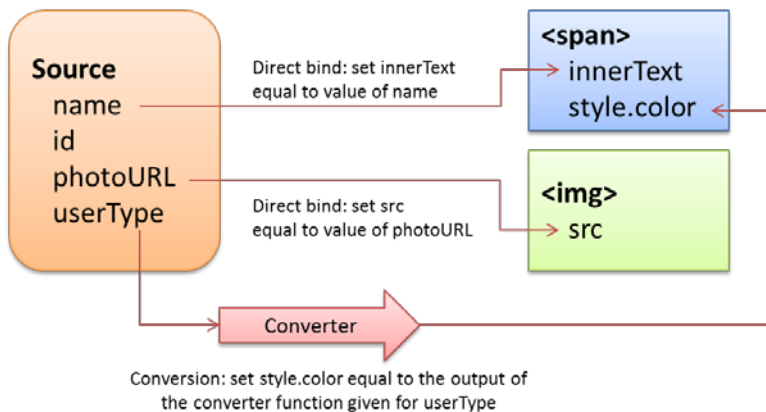
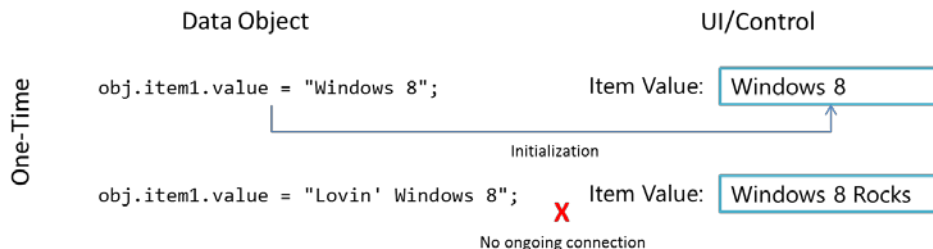


FIGURE 4-10 A common data-binding scenario between a source data object and two target UI elements, involving two direct bindings and one binding with a conversion function.

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, as we'll see.



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

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 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:

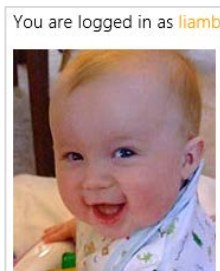
```
var login1 = { name: "liam", id: "12345678",
  photoURL: "http://www.kraigbrockschmidt.com/images/Liam07.png", userType: "kid"};
```

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 as a baby:



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 `<target property> : <source 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>
```

Note that array lookup on the source property using `[]` is not supported, though a converter could do that. On the target, if that object has a JavaScript property that you want to refer to using a hyphenated identifier, you can use the following syntax:

```
<span data-win-bind="this['funky-property']: source"></span>
```

A similar syntax is necessary for data-binding target *attributes*, such as the `aria-*` attributes for accessibility. Because these are not JavaScript properties, a special converter (or initializer as it is more properly called) named `WinJS.Binding.setAttribute` is needed:

```
<label data-win-bind="this['aria-label']: title WinJS.Binding.setAttribute"></label>
```

Also see `WinJS.Binding.setAttributeOneTime` for one-time binding for attributes.

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 (again, it's 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), we could place our `login` data object and the `userTypeToColor` function in a `LoginData` namespace, and 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. 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.

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.

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 nonexistent 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>
```

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), any time 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:

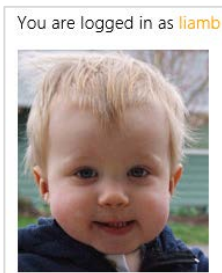
```
var login4 = new LoginData.LoginClass({ name: "liamb",  
    photoURL: "http://www.kraigbrockschmidt.com/images/Liam08.jpg" });
```

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. (Also refer to the [Programmatic binding sample](#) for a few demonstrations.)

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 earlier, one of the requirements for an observable object is that it 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](#) 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.

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

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

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

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've 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. (And 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 individual item in that source into its own 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 in the next chapter.

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.

- Windows 8 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`. Custom controls can also be shown in Blend's Assets panel either for a single project or for all projects.
- 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.

Chapter 5

Collections and Collection Controls

It's a safe bet to say that wherever you are, right now, you're probably surrounded by quite a number of collections. This book you're reading is a collection of chapters, and chapters are a collection of pages. Those pages are collections of paragraphs, which are collections of words, which are collections of letters, which are (assuming you're reading this electronically) collections of pixels. On and on....

Your body, too, has collections on many levels, which is very much what one studies in college-level anatomy courses. Looking around my office and my home, I see even more collections: a book shelf with books; scrapbooks with pages and pages with pictures; cabinets with cans, boxes, and bins of food; my son's innumerable toys; the DVD case...even the forest outside is a collection of trees and bushes, which then have branches, which then have leaves. On and on....

We look at these things *as* collections because we've learned how to generalize specific instances of unique things—like leaves or pages or my son's innumerable toys—into categories or groups. This gives us powerful means to organize and manage those things (except for the clothes in my closet, as my wife will attest). And just as the physical world around us is very much made of collections, the digital world that we use to represent the physical is naturally full of collections as well. Thus programming languages like JavaScript have constructs like arrays to organize and manage collection data, and environments like Windows 8 provide collection controls through which we can visualize and manipulate that data.

In this chapter we'll turn our attention to the two collection controls provided by WinJS: the FlipView, which shows one item from a collection at a time, and the ListView, which shows many items in different arrangements. As you might expect, the ListView is the richer of the two. As it's really the centerpiece of many app designs, we'll be spending the bulk of this chapter exploring its depths, along with the concept and implementation of *semantic zoom* (another control, in fact).

As both collection controls can handle items of arbitrary complexity (both in terms of data and presentation, unlike the simple HTML listbox and combobox controls), as well as an arbitrary number of items, they naturally build on the foundations of data binding and template controls we just saw at the end of Chapter 4, "Controls, Control Styling, and Data Binding." They also have a close relationship to collection data sources, which we'll specifically examine as well, and their own styling and behavioral considerations.

But let's not exhaust our minds here at the outset of this chapter with theory or architectural intricacies! Instead, let's just jump into some code to explore the core aspects of both controls.

Collection Control Basics

To seek the basics of the collection controls, we'll first look at the FlipView which will introduce us to item templates and data sources. We'll then see how these also apply to the ListView control, then look at grouping items within a ListView.

Quickstart #1: The FlipView Control Sample

As shown in Figure 5-1, the [FlipView control sample](#) is both a great piece of reference code for this control and a great visual tool through which to explore the control itself. (I'm also extremely grateful that I've not had to write such samples for this book!) For the purposes of this Quickstart, let's just look at the first scenario of populating the control from a simple data source and using a template for rendering the items, as these mechanisms are shared with the ListView. We'll come back to the other FlipView scenarios later in the chapter.

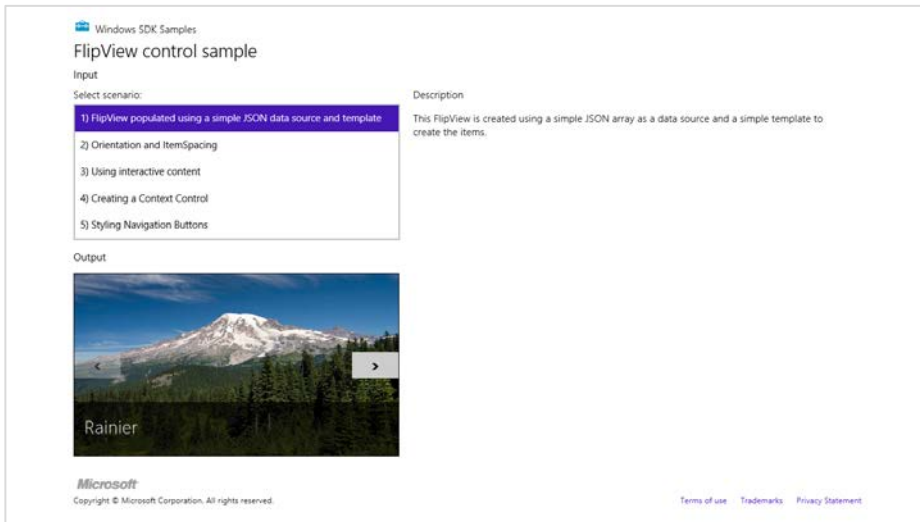


FIGURE 5-1 The FlipView control sample; the FlipView is the control displaying the picture.

As FlipView is a WinJS control, whose constructor is `WinJS.UI.FlipView`, we declare it in markup with `data-win-control` and `data-win-options` attributes (see `html/simpleFlipview.html`):

```
<div id="simple_FlipView" class="flipView" data-win-control="WinJS.UI.FlipView"
    data-win-options="{ itemDataSource: DefaultData.bindingList.dataSource,
        itemTemplate: simple_ItemTemplate }">
</div>
```

And of course, `WinJS.UI.processAll` is called in the page-loading process to instantiate the control. In the FlipView's options we can immediately see the two critical pieces to make the control work: a data source that provides the goods for each item and a template to render them.

If you were paying attention at the end of Chapter 4, you’ve probably guessed that the template is an instance of `WinJS.Binding.Template`. And you’re right! That piece of markup, in fact, comes just before the control declaration in `html/simpleFlipview.html`.

```
<div id="simple_ItemTemplate" data-win-control="WinJS.Binding.Template" style="display: none">
  <div class="overlaidItemTemplate">
    <img class="image" data-win-bind="src: picture; alt: title" />
    <div class="overlay">
      <h2 class="ItemTitle" data-win-bind="innerText: title"></h2>
    </div>
  </div>
</div>
```

Note that a template must *always* be declared in markup before any controls that reference them: `WinJS.UI.processAll` must instantiate the template first because the collection control will be asking the template to render its contents for each item in the data source. Also remember from Chapter 4 that instantiating a template removes its contents from the DOM so that it cannot be altered at run time. You can see this when running the sample: expand the nodes in Visual Studio’s DOM Explorer or Blend’s Live DOM pane, and you’ll see the root `div` of the template but none of its children.

In the sample, the prosaically named `ItemTemplate` is made of an `img` element and another `div` containing an `h2`. The `overlay` class on that latter `div`, if you look at Figure 5-1 carefully, is clearly styled with a partially transparent background color (see `css/default.css` for the `.overlaidItemTemplate .overlay` selector). This shows that you can use any elements you want in a template, including other WinJS controls. In the latter case, these are picked up when `WinJS.UI.process/ processAll` is invoked on the template.³¹

You can also see that the template uses WinJS data-binding attributes, where the `img.src`, `img.alt`, and `h2.innerText` properties are bound to data properties called `picture` and `title`. This shows how properties of two target elements can be bound to the same source property. (Remember that if you’re binding to properties of the WinJS control itself, rather than its child elements, those properties must begin with `winControl`.)

For the data source, the FlipView’s `itemDataSource` option is assigned the value of `DefaultData.bindingList.dataSource` that you can find in `js/DefaultData.js`:

```
var array = [
  { type: "item", title: "Cliff", picture: "images/Cliff.jpg" },
  { type: "item", title: "Grapes", picture: "images/Grapes.jpg" },
  { type: "item", title: "Rainier", picture: "images/Rainier.jpg" },
  { type: "item", title: "Sunset", picture: "images/Sunset.jpg" },
  { type: "item", title: "Valley", picture: "images/Valley.jpg" }
];
var bindingList = new WinJS.Binding.List(array);

WinJS.Namespace.define("DefaultData", {
```

³¹ Note that for such controls to be fully interactive, assign the `win-interactive` class to them, otherwise the surrounding control (and this applies to ListView as well) will swallow input events before they reach those controls.

```
bindingList: bindingList,  
array: array  
});
```

We briefly met `WinJS.Binding.List` at the end of Chapter 4; its purpose is to turn an in-memory array into an observable data source for one-way binding. The `WinJS.Binding.List` wrapper is also necessary because the `FlipView` and `ListView` controls cannot work directly against a simple array, even for one-time binding. They expect their data sources to provide the methods of the `WinJS.UI.IListDataSource` interface. The `dataSource` property of a `WinJS.Binding.List`, as in `bindingList.dataSource`, provides exactly this, and you'll always use this property in conjunction with `FlipView` and `ListView`. (It exists for no other purpose, in fact.) If you forget and attempt to just bind to the `WinJS.Binding.List` directly, you'll see an exception that says, "Object doesn't support property or method 'createListBinding'."

Suffice it to say that `WinJS.Binding.List` will become your intimate friend for in-memory data sources. Of course, you won't typically be using hard-coded data like the sample. You'll instead load array data from a file or obtain it from a web service, at which point `WinJS.Binding.List` makes it accessible to collection controls.

Do note that `WinJS.Binding.List` fully supports dynamic data. If you look at its [reference page](#) in the documentation, you'll see that it looks a whole lot like a JavaScript array, with a `length` property and the whole set of array methods from `concat` and `indexOf` to `push`, `pop`, and `unshift`. This is entirely intentional: no need to make you relearn the basics!

It's also important to note with `FlipView`, as well as `ListView`, that setting the control's `itemDataSource` property automatically sets up one-way binding, so any changes to the list object or even the array on which it is built will trigger an automatic update in the bound control.

Quickstart #2a: The HTML ListView Essentials Sample

As I said before, the basic mechanisms for data sources and templates apply to the `ListView` control exactly as it does to `FlipView`, which we can now see in the [HTML ListView essentials sample](#) (shown in Figure 5-2), specifically its first two scenarios of creating the control and responding to item events.

Because `ListView` can display multiple items at the same time, it needs one more piece in addition to the data source and the template: something to describe how those items visually relate to one another. This is the `ListView`'s `layout` property, which we see in the markup for Scenario 1 of this sample along with a few other behavioral options (`html/scenario1.html`):

```
<div id="listView" data-win-control="WinJS.UI.ListView"
    data-win-options="{ itemDataSource: myData.dataSource,
        itemTemplate: smallListIconTextTemplate, selectionMode: 'none',
        tapBehavior: 'none', swipeBehavior: 'none', layout: { type: WinJS.UI.GridLayout } }">
</div>
```

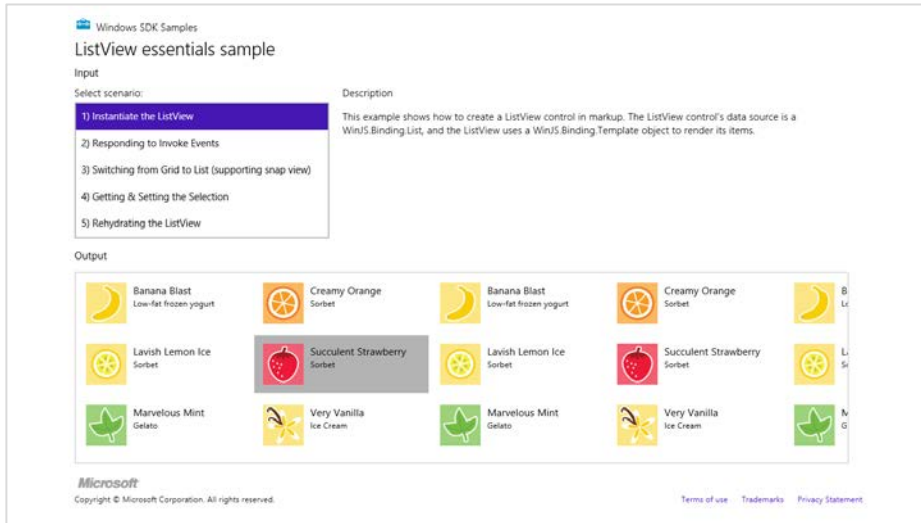


FIGURE 5-2 The HTML ListView essentials sample.

The ListView's constructor, `WinJS.UI.ListView`, is, of course, called by the ubiquitous `WinJS.UI.processAll` when the page control is loaded. The data source for this list is set to `myData.dataSource` where `myData` is again a `WinJS.Binding.List` (defined at the top of `js/data.js` over a simple array) and its `dataSource` property provides the needed interface.

The control's item template is defined earlier in `default.html` with the id of `smallListIconTextTemplate` and is essentially the same sort of thing we saw with the FlipView (an `img` and some text elements), so I won't list it here.

In the control options we see three behavioral properties: `selectionMode`, `tapBehavior`, and `swipeBehavior`. These are all set to `'none'` in this sample to disable selection and click behaviors entirely, making the ListView a passive display. It can still be panned, but the items don't respond to input. (Also see the "Item Hover Styling" sidebar.)

As for the `layout` property, this is an object of its own, whose `type` property indicates which layout to use. `WinJS.UI.GridLayout`, as we're using here, is a two-dimensional top-to-bottom then left-to-right algorithm, suitable for horizontal panning. WinJS provides another layout type called `WinJS.UI.ListLayout`, a one-dimensional top-to-bottom organization that's suitable for vertical panning, especially in snapped view. (We'll see this with the Grid App project template shortly; the ListView essentials sample lacks a good snapped view.)

Now while the ListView control in Scenario 1 only displays items, we often want those items to respond to a click or tap. Scenario 2 shows this, where the `tapBehavior` property is set to `'invoke'` (see [html/scenario2.html](#)). This is the same as using `tapBehavior: WinJS.UI.TapBehavior.toggleSelect`, as that's just defined in the [enumeration](#) as "invoke". This behavior will select or deselect an item, depending on its state, and then invoke it. Other variations are `directSelect`, where an item is always selected and then invoked, and `invokeOnly` where the item is invoked without changing the selection state. You can also set the behavior to `none` so that clicks and taps are ignored.

When an item is invoked, the ListView control fires an `itemInvoked` event. You can wire up a handler by using either `addEventListener` or the ListView's `oniteminvoked` property. Here's how Scenario 2 does it (slightly rearranged from `js/scenario2.js`):

```
var listView = element.querySelector('#listView').winControl;
listView.addEventListener("iteminvoked", itemInvokedHandler, false);

function itemInvokedHandler(eventObject) {
    eventObject.detail.itemPromise.done(function (invokedItem) {
        // Act on the item
    });
}
```

Note that we're listening for the event on the WinJS *control*, but it also works to listen for the event on the containing element thanks to bubbling. This can be helpful if you need to add listeners to a control before it's instantiated, since the containing element will already be there in the DOM.

In the code above, you could also assign a handler by using the `listView.oniteminvoked` property directly, or you can specify the handler in the `iteminvoked` property `data-win-options` (in which case it must be marked safe for processing). The event object you then receive in the handler contains a *promise* for the invoked item, not the item itself, so you need to call its `done` or `then` method to obtain the actual item data. It's also good to know that you should never change the ListView's data source properties directly within an `iteminvoked` handler, because you'll probably cause an exception. If you have need to do that, wrap the change code inside a call to `setImmediate` so that you can yield back to the UI thread first.

Sidebar: Item Hover Styling

While disabling selection and tap behaviors on a ListView creates a passive control, hovering over items with the mouse (or suitable touch hardware) still highlights each item; refer back to Figure 5-2. You can control this using the `.win-container:hover` pseudo-selector for the desired control. For example, the following style rule removes the hover effect entirely:

```
#myListView .win-container:hover {
    background-color: transparent;
    outline: 0px;
}
```

Quickstart #2b: The ListView Grouping Sample

Displaying a list of items is great, but more often than not, a collection really needs another level of organization—what we call grouping. This is readily apparent when I open the file drawer next to my desk, which contains a collection of various important and not so important papers. Right away, on the file folder tabs, I see my groups: Taxes, Financials, Community, Insurance, Cars, Writing Projects, and Miscellany (among others). Clearly, then, we need a grouping facility within a collection control and ListView is happy to oblige.

A core demonstration of grouping can be found in the [HTML ListView grouping and Semantic Zoom sample](#) (shown in Figure 5-3). As with the Essentials sample, the code in `js/groupedData.js` contains a lengthy in-memory array around which we create a `WinJS.Binding.List`. Here's a condensation to show the item structure (I'd show the whole array, but this is making me hungry for some dessert!):

```
var myList = new WinJS.Binding.List([
  { title: "Banana Blast", text: "Low-fat frozen yogurt", picture: "images/60Banana.png" },
  { title: "Lavish Lemon Ice", text: "Sorbet", picture: "images/60Lemon.png" },
  { title: "Creamy Orange", text: "Sorbet", picture: "images/60Orange.png" },
  ...
]);
```

Here we have a bunch of items with `title`, `text`, and `picture` properties. We can group them any way we like and even change the groupings on the fly. As Figure 5-3 shows, the sample groups these by the first letter of the title.

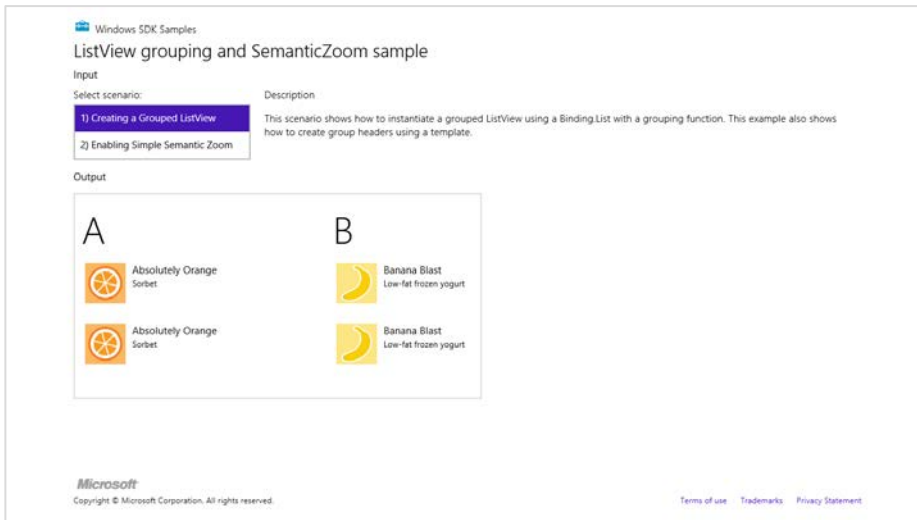


FIGURE 5-3 The HTML ListView grouping and Semantic Zoom sample.

If you take a peek at the [ListView reference](#), you'll see that the control works with two templates and two collections: that is, alongside its `itemTemplate` and `itemDataSource` properties are ones called `groupHeaderTemplate` and `groupDataSource`. These are used with the ListView's `GridLayout` (the default) to organize the groups and create the headers above the items.

The header template in `html/scenario1.html` is very simple (and the item template is like what we've already seen):

```
<div id="headerTemplate" data-win-control="WinJS.Binding.Template">
  <div class="simpleHeaderItem">
    <h1 data-win-bind="innerText: title"></h1>
  </div>
</div>
```

This is referenced in the control declaration (other options omitted):

```
<div id="listView" data-win-control="WinJS.UI.ListView"
  data-win-options="{ groupDataSource: myGroupedList.dataSource,
    groupHeaderTemplate: headerTemplate }">
</div>
```

For the data sources, you can see that we're now using a variable called `myGroupedList` with a property inside it called `groups`. What's all this about?

Well, let's take a short conceptual detour. Although computers have no problem chewing on a bunch of raw data like the `myList` array, human beings like to view data with a little more organization. The three primary ways of doing this are *grouping*, *sorting*, and *filtering*. Grouping organizes items into groups, as shown in Figure 5-3; sorting orders items according to various rules; and filtering provides a subset of items that match certain criteria. In all three cases, however, you don't want such operations to actually change the underlying data: a user might want to group, sort, or filter the same data in different ways from moment to moment.

Grouping, sorting, and filtering, then, are thus referred to as *projections* of the data: they're all connected to the same underlying data such that a change to an item in the projection will be propagated back to the source, just as changes in the source are reflected in the projection.

The `WinJS.Binding.List` object provides methods to create these projections: `createGrouped`, `createSorted`, and `createFiltered`. Each method produces a special form of a `WinJS.Binding.List`: `GroupedSortedListProjection`, `SortedListProjection`, and `FilteredListProjection`, respectively. That is, each projection is a bindable list in itself, with a few extra methods and properties that are specific to the projection. You can even create a projection from a projection. For instance, `createGrouped(...).createFiltered(...)` will create a filtered projection on top of a grouped projection. (Note, however, that the list's `sort` method does not create a projection. It applies the sorting in-place, just like the JavaScript array's `sort`.)

Now that we know about projections, we can see how `myGroupedList` is created:

```
var myGroupedList = myList.createGrouped(getGroupKey, getGroupData, compareGroups);
```

This method takes three functions. The first, the *group key* function, associates an item with a group: it receives an item and returns the appropriate group string, known as the key. The key—which must be a string—can be something that’s directly included in an item or it can be derived from item properties. In the sample, the `getGroupKey` function returns the first character of the item’s `title` property (in upper case). Note, however, that the original sample just uses `charAt` to obtain the grouping character, but this won’t work for a large number of languages. Instead, use the [Windows.Globalization.Collation.CharacterGroupings](#) class and its `lookup` method as shown below, which will normalize casing automatically so that calling `toLocaleUpperCase` isn’t necessary:

```
var cg = Windows.Globalization.Collation.CharacterGroupings();

function getGroupKey(dataItem) {
    return cg.lookup(dataItem.title);
}
```

This code, and other changes made below, can be found in the modified version of this sample included with this chapter’s companion content.

Be clear that this first function, referred to as the *group key* function, determines *only* the association between the item and a group, nothing more. It also gets called for every item in the collection when `createGrouped` is called, so it should be a quick operation. For this reason the creation of `CharacterGroupings` is done one outside of the function.

Tip If deriving the group key from an item at run time required an involved process, you’ll improve overall performance by storing a prederived key in the item instead and just returning that from the group key function.

The data for the groups themselves, which is the collection to which the header template is bound to, isn’t actually created until the group projection’s `groups` method is invoked, as happens when our `ListView`’s `groupedDataSource` option gets processed. At that point, the second function passed to `createGrouped`—the *group data* function—gets called only once per group with a *representative* item for that group. In response, your function returns an object for that group containing whatever properties you need for data binding.

In the sample, the `getGroupData` function (passed to `createGrouped`) simply returns an object with a single `groupTitle` property that’s the same as the group key, but of course you can make that value anything you want. This code is also modified from the original sample to be attentive to globalization concerns, which we do by reusing `getGroupKey`:

```
function getGroupData(dataItem) {
    return {
        groupTitle: getGroupKey(dataItem)
    };
}
```


In the modified sample I changed name the `title` property of this group data object to a more distinct `groupTitle` to make it very clear that it has nothing whatsoever to do with the `title` property of the *items*. This meant changing the header templates in `html/scenario1.html` and `html/scenario2.html` to refer to `groupTitle` as well. This helps us be clear that the data contexts in the header and item templates are completely different. For the header template, it's the collection generated by the return values of your group data function; for the item template, it's the grouped projection from `WinJS.Binding.List.createGrouped`. Two different collections—remember that!

So why do we have the group data function separated out at all? Why not just create that collection automatically from the group keys? It's because you often want to include additional properties within the group data for use in the header template or in a zoomed-out view (with semantic zoom). Think of your group data function as providing summary information for each group. (The header text is really only the most basic such summary.) Since this function is only called once per group, rather than once per item, it's the proper time to calculate or otherwise retrieve summary-level data. For example, to show an item count in the group headers, we just need to include that property in the objects returned by the group data function, then data-bind an element in the header template to that property.

In the modified sample, I use `WinJS.Binding.List.createFiltered` to obtain a projection of the list filtered by the current key.³² The `length` property of this projection is then the number of items in the group:

```
function getGroupData(dataItem) {
    var key = getGroupKey(dataItem);

    //Obtain a filtered projection of our list, checking for matching keys
    var filteredList = myList.createFiltered(function (item) {
        return key == getGroupKey(item);
    });

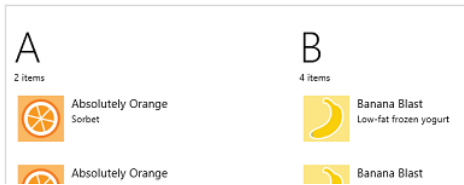
    return {
        title: key,
        count: filteredList.length
    };
}
```

With this `count` property in the collection, we can use it in the header template:

```
<div id="headerTemplate" data-win-control="WinJS.Binding.Template" style="display: none">
    <div class="simpleHeaderItem">
        <h1 data-win-bind="innerText: groupTitle"></h1>
        <h6><span data-win-bind="innerText: count"></span> items</h6>
    </div>
</div>
```

³² Creating a filtered projection is also useful to intentionally limit the number of items you want to display in a control, where you make sure that true is only returned for a fixed number of items.

After a small tweak in `css/scenario1.css`—changing the `simpleHeaderItem` class height to 65px to make a little more room—the list will now appear as follows:



Finally, back to `WinJS.Binding.List.createGrouped`, the third (and optional) function here is a *group sorter* function, which is called to sort the group data collection and therefore the order in which those groups appear in the `ListView`.³³ This function receives two group keys and returns zero if they're equal, a negative number if the first key sorts before the second, and a positive if the second sorts before the first. The `compareGroups` function in the sample does an alphabetical sort, which I've updated in the modified version to again use world-ready sort ordering:

```
function compareGroups(left, right) {
    return groupCompareGlobalized(left, right);
}

function groupCompareGlobalized(left, right) {
    var charLeft = cg.lookup(left);
    var charRight = cg.lookup(right);

    // If both are under the same grouping character, treat as equal
    if (charLeft.localeCompare(charRight) == 0) {
        return 0;
    }

    // In different groups, we must rely on locale-sensitive sort order of items since the names
    // of the groups don't sort the same as the groups themselves for some locales.
    return left.localeCompare(right);
}
```

For a two-level sort, first by the descending item count and then by the first character, we could write the following (this is in the modified sample; refer to this in the call to `myList.createGrouped` to see it in action):

```
function compareGroups2(left, right) {
    var leftLen = filteredLengthFromKey(left);
    var rightLen = filteredLengthFromKey(right);

    if (leftLen != rightLen) {
        return rightLen - leftLen;
    }
}
```

³³ This is entirely separate from creating a sorted projection of the *items*, for which you'd use `WinJS.Binding.List.createSorted`.

```

    return groupCompareGlobalized(left, right);
}

function filteredLengthFromKey(key) {
    var filteredList = myList.createFiltered(function (item) {
        return key == getGroupKey(item);
    });

    return filteredList.length;
}

```

Debugging Your Grouping Functions

If your various grouping functions don't seem to be working right, you can set breakpoints and step through the code a few times, but this becomes tedious as the functions are called many, many times for even modest collections. Instead, try using `console.log` to emit the parameters sent to those functions and/or your return values, allowing you to review the overall results much more quickly. To see what's coming into the group sorting function, for example, try this code:

```
console.log("Comparing left = " + left + " to right = " + right);
```

ListView in the Grid App Project Template

Now that we've covered the details of the ListView control and in-memory data sources, we can finally understand the rest of the Grid App project template in Visual Studio and Blend. As we covered in the "The Navigation Process and Navigation Styles" section of Chapter 3, "App Anatomy and Page Navigation," this project template provides an app structure built around page navigation: the home page (pages/groupedItems) displays a collection of sample data (see `js/data.js`) in a ListView control, where each item's presentation is described by a `WinJS.Binding.Template` as are the group headings. Figure 5-4 shows the layout of the home page and identifies the relevant ListView elements. As we also discussed before, tapping an item navigates to the pages/itemDetail page and tapping a heading navigates to the pages/groupDetail page, and now we can see how that all works with the ListView control.

The ListView in Figure 5-4 occupies the lower portion of the app's contents. Because it can pan horizontally, it actually extends all the way across; various CSS margins are used to align the first items with the layout silhouette while allowing them to bleed to the left when the ListView is panned.

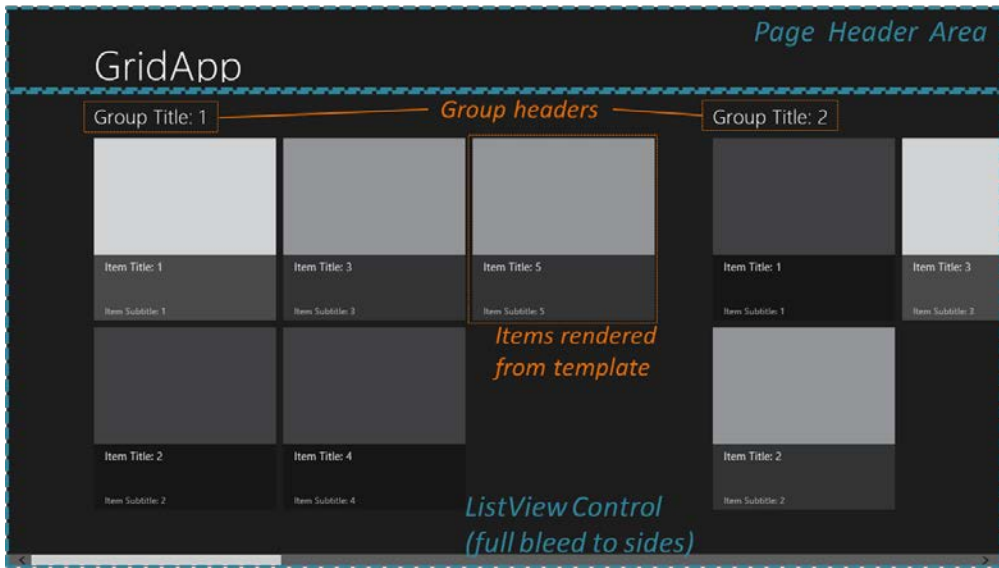


FIGURE 5-4 ListView elements as shown in the Grid App template home page. (All colored items are added labels and lines.)

There's quite a bit going on with the ListView in this project, so we'll take one part at a time. For starters, the control's markup in `pages/groupedItems/groupedItems.html` is very basic, where the only option is to indicate that the items have no selection behavior:

```
<div class="groupeditemslist win-selectionstylefilled" aria-label="List of groups"
  data-win-control="WinJS.UI.ListView" data-win-options="{ selectionMode: 'none' }">
</div>
```

Switching over to `pages/groupedItems/groupedItems.js`, the page's `ready` method handles initialization:

```
ready: function (element, options) {
    var listView = element.querySelector(".groupeditemslist").winControl;
    listView.groupHeaderTemplate = element.querySelector(".headerTemplate");
    listView.itemTemplate = element.querySelector(".itemtemplate");
    listView.oniteminvoked = this._itemInvoked.bind(this);

    // (Keyboard handler initialization omitted)...

    this.initializeLayout(listView, appView.value);
    listView.element.focus();
},
```

Here you can see that the control's templates can be set in code just as easily as from markup, and in this case we're using a class to locate the template element instead of an id. Why does this work? It's because we've actually been referring to elements the whole time: the app host automatically creates a variable for an element that's named the same as its id. It's the same thing. In code you can also provide

a function instead of a declarative template, which allows you to dynamically render each item individually. More on this later.

You can also see how this page assigns a handler to the `itemInvoked` events (above `ready`), calling `WinJS.Navigation.navigate` to go to the `groupDetail` or `itemDetail` pages as we saw in Chapter 3:

```
_itemInvoked: function (args) {
    if (appView.value === appViewState.snapped) {
        // If the page is snapped, the user invoked a group.
        var group = Data.groups.getAt(args.detail.itemIndex);
        this.navigateToGroup(group.key);
    } else {
        // If the page is not snapped, the user invoked an item.
        var item = Data.items.getAt(args.detail.itemIndex);
        nav.navigate("/pages/itemDetail/itemDetail.html", {
            item: Data.getItemReference(item) });
    }
}

navigateToGroup: function (key) {
    nav.navigate("/pages/groupDetail/groupDetail.html", { groupKey: key });
},
```

In this case we retrieve item data from the underlying collection (the `getAt` methods) rather than using the item data itself. This is because the group information needed for the first case isn't part of an item directly. We also see here that the page interprets item invocations differently depending on the view state. This is because it actually switches both its layout and its data source when the view state changes. This is handled in the page's internal `_initializeLayout` method, called both on startup and from the page's `updateLayout` function:

```
initializeLayout: function (listView, viewState) {
    if (viewState === appViewState.snapped) {
        listView.itemDataSource = Data.groups.dataSource;
        listView.groupDataSource = null;
        listView.layout = new ui.ListLayout();
    } else {
        listView.itemDataSource = Data.items.dataSource;
        listView.groupDataSource = Data.groups.dataSource;
        listView.layout = new ui.GridLayout({ groupHeaderPosition: "top" });
    }
},
```

A `ListView`'s layout, in short, can be changed at any time by setting its `layout` property. When the view state is snapped, this is set to `WinJS.UI.ListLayout`, otherwise `WinJS.UI.GridLayout` (whose `groupHeaderPosition` property can be `"top"` or `"left"`). You can also see that you can change a `ListView`'s data source on the fly: in snapped state it's a list of groups, otherwise it's the list of items.

I hope you can now see why I introduced page navigation well before we got to `ListView`, because this project gets quite complicated down in its depths! In any case, let's now look at the templates for this page (`pages/groupedItems/groupedItems.html`):

```

<div class="headertemplate" data-win-control="WinJS.Binding.Template">
  <button class="group-header win-type-x-large win-type-interactive"
    data-win-bind="groupKey: key" role="link" tabindex="-1" type="button"
    onclick="Application.navigator.pageControl.navigateToGroup(event.srcElement.groupKey)" >
    <span class="group-title win-type-ellipsis" data-win-bind="textContent: title"></span>
    <span class="group-chevron"></span>
  </button>
</div>

<div class="itemtemplate" data-win-control="WinJS.Binding.Template">
  <div class="item">
    
    <div class="item-overlay">
      <h4 class="item-title" data-win-bind="textContent: title"></h4>
      <h6 class="item-subtitle win-type-ellipsis"
        data-win-bind="textContent: subtitle"></h6>
    </div>
  </div>
</div>

```

Again, we have the same use of `WinJS.Binding.Template` and various bits of data-binding syntax sprinkled around the markup, not to mention the `click` handler assigned to the header text itself, which, like an item in snapped view, navigates to the group detail page.

As for the data itself (that you'll likely replace), this is again defined in `js/data.js` as an in-memory array that feeds into `WinJS.Binding.List`. In the `sampleItems` array each item is populated with inline data or other variable values. Each item also has a `group` property that comes from the `sampleGroups` array. Unfortunately, this latter array has almost identical properties as the items array, which can get confusing. To help clarify that a bit, here's the complete property structure of an item:

```

{
  group : {
    key,
    title,
    subtitle,
    backgroundImage,
    description
  },
  title,
  subtitle,
  description,
  content,
  backgroundImage
}

```

As we saw with the `ListView` grouping sample earlier, the `Grid App` project template uses `createGrouped` to set up the data source. What's interesting to see here is that it sets up an initially empty list, creates the grouped projection (omitting the optional sorter function), and then adds the items by using the list's `push` method:

```

var list = new WinJS.Binding.List();
var groupedItems = list.createGrouped(
    function groupKeySelector(item) { return item.group.key; },
    function groupDataSelector(item) { return item.group; }
);

generateSampleData().forEach(function (item) {
    list.push(item);
});

```

This clearly shows the dynamic nature of lists and ListView: you can add and remove items from the data source, and one-way binding will make sure the ListView is updated accordingly. In such cases you do *not* need to refresh the ListView's layout—that happens automatically. I say this because there's been some confusion with the ListView's `forceLayout` method, which you only need to call, as the documentation states, "when making the ListView visible again after its `style.display` property had been set to 'none'." You'll find, in fact, that the Grid App code doesn't use this method at all.

In `js/data.js` there are also a number of other utility functions, such as `getItemsFromGroup`, which uses `WinJS.Binding.List.createFiltered` as we did earlier. Other functions provide for cross-referencing between groups and items, as is needed to navigate between the items list, group details (where that page shows only items in that group), and item details. All of these functions are wrapped up in a namespace called `Data` at the bottom of `js/data.js`, so references to anything from this file are prefixed elsewhere with `Data..`

And with that, I think you'll be able to understand everything that's going on in the Grid App project template to adapt it to your own needs. Just remember that all the sample data, like the default logo and splash screen images, is intended to be wholly replaced with real data that you obtain from other sources, like a file or `WinJS.xhr`, and that you can wrap with `WinJS.Binding.List`. Some further guidance on this can be found in the [Create a blog reader tutorial](#) on the Windows Dev Center, and although the tutorial uses the Split App project template, there's enough in common with the Grid App project template that the discussion is really applicable to both.

The Semantic Zoom Control

Since we've already loaded up the [HTML ListView grouping and Semantic Zoom sample](#), and have completed our first look at the collection controls, now is a good time to check out another very interesting WinJS control: [Semantic Zoom](#).

Semantic zoom lets users easily switch between two views of the same data: a zoomed-in view that provides details and a zoomed-out view that provides more summary-level information. The primary use case for semantic zoom is a long list of items (especially ungrouped items), where a user will likely get really bored of panning all the way from one end to the other, no matter how fun it is to swipe the

screen with a finger. With semantic zoom, you can zoom out to see headers, categories, or some other condensation of the data, and then tap on one of those items to zoom back into its section or group. The design guidance recommends having the zoomed-out view fit on one to three screenfuls at most, making it very easy to see and comprehend the whole data set.

Go ahead and try semantic zoom through Scenario 2 of the ListView grouping and Semantic Zoom sample. To switch between the views, use pinch-zoom touch gestures, Ctrl+/Ctrl- keystrokes, Ctrl+mouse wheel, and/or a small zoom button that automatically appears in the lower-right corner of the control, as shown in Figure 5-5. When you zoom out, you'll see a display of the group headers, as also shown in the figure.

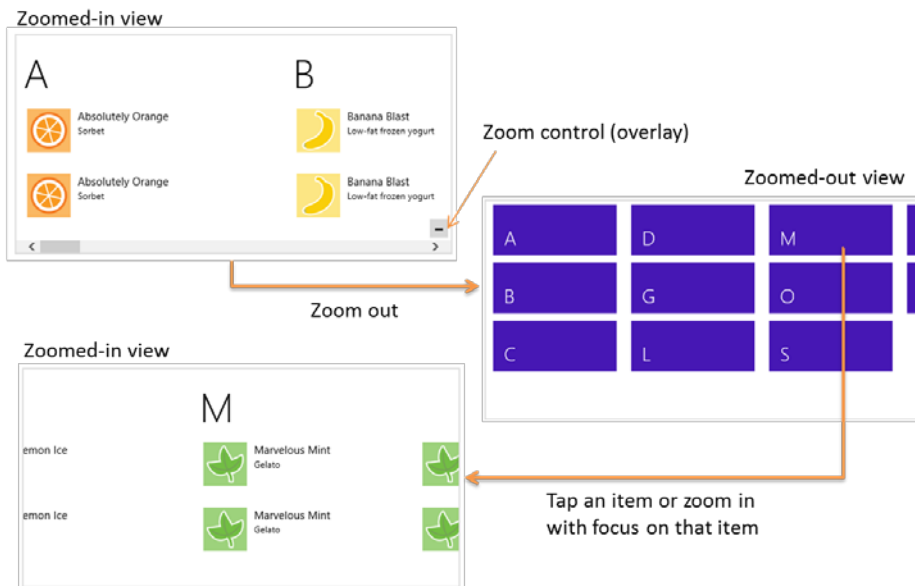


FIGURE 5-5 Semantic zoom between the two views in the ListView grouping and Semantic Zoom sample.

The control itself is quite straightforward to use. In markup, declare a WinJS control using the `WinJS.UI.SemanticZoom` constructor. Within that element you then declare two (and only two) child elements: the first defining the zoomed-in view, and the second defining the zoomed-out view—always in that order. Here's how the sample does it with two ListView controls (plus the template used for the zoomed-out view; I'm showing the code in the modified sample included with this chapter's companion content):

```
<div id="semanticZoomTemplate" data-win-control="WinJS.Binding.Template" >
  <div class="semanticZoomItem">
    <h2 class="semanticZoomItem-Text" data-win-bind="innerText: groupTitle"></h2>
  </div>
</div>

<div id="semanticZoomDiv" data-win-control="WinJS.UI.SemanticZoom">
  <div id="zoomedInListView" data-win-control="WinJS.UI.ListView"
```



```

        data-win-options="{ itemDataSource: myGroupedList.dataSource,
            itemTemplate: mediumListIconTextTemplate,
            groupDataSource: myGroupedList.groups.dataSource,
            groupHeaderTemplate: headerTemplate,
            selectionMode: 'none', tapBehavior: 'none', swipeBehavior: 'none' }">
    </div>

    <div id="zoomedOutListView" data-win-control="WinJS.UI.ListView"
        data-win-options="{ itemDataSource: myGroupedList.groups.dataSource,
            itemTemplate: semanticZoomTemplate,
            selectionMode: 'none', tapBehavior: 'invoke', swipeBehavior: 'none' }" >
    </div>
</div>

```

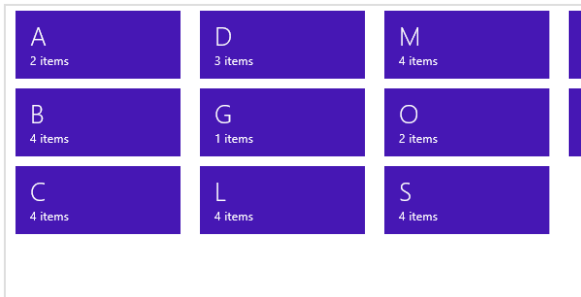
The first child, *zoomedInListView*, is just like the *ListView* for Scenario 1 with group headers and items; the second, *zoomedOutListView*, uses the groups as items and renders them with a different template. The semantic zoom control simply switches between the two views on the appropriate input gestures. When the zoom changes, the semantic zoom control fires a `zoomchanged` event where the `args.detail` value in the handler is `true` when zoomed out, `false` when zoomed in. You might use this event to make certain app bar commands available for the different views, such as commands in the zoomed-out view to change sorting or filtering, which would then affect how the zoomed-in view is displayed. We'll see the app bar in Chapter 7, "Commanding UI."

The control has a few other properties, such as `enableButton` (a Boolean to control the visibility of the overlay button; default is `true`), `locked` (a Boolean that disables zooming in either direction and can be set dynamically to lock the current zoom state; default is `false`), and `zoomedOut` (a Boolean indicating if the control is zoomed out, so you can initialize it this way; default is `false`). There is also a `forceLayout` method that's used in the same case as the *ListView*'s `forceLayout`: namely, when you remove a `display: none` style.

The `zoomFactor` property is an interesting one that determines how the control animates between the two views. The animation is a combination of scaling and cross-fading that makes the zoomed-out view appear to drop down from or rise above the plane of the control, depending on the direction of the switch, while the zoomed-in view appears to sink below or come up to that plane. To be specific, the zoomed-in view scales between 1 and `zoomFactor` while transparency goes between 1 and 0, and the zoomed-out view scales between $1/\text{zoomFactor}$ and 1 while transparency goes between 0 and 1. The default value for `zoomFactor` is 0.65, which creates a moderate effect. Lower values (minimum is 0.2) emphasize the effect, and higher values (maximum is 0.8) minimize it.

Where styling is concerned, you do most of what you need directly to the Semantic Zoom's children. However, to style the Semantic Zoom control itself you can override styles in `win-semanticzoom` (for the whole control) and `win-semanticzoomactive` (for the active view). The `win-semanticzoombutton` style also lets you style the zoom control button if needed.

It's important to understand that semantic zoom is intended to switch between two views of the same data and not to switch between completely different data sets (see [Guidelines and checklist for the Semantic Zoom control](#)). Also, the control does not support nesting (that is, zooming out multiple times to different levels). Yet this doesn't mean you have to use the same kind of control for both views: the zoomed-in view might be a list, and the zoomed-out view could be a chart, a calendar, or any other visualization that makes sense. The zoomed-out view, in other words, is a great place to show summary data that would be otherwise difficult to derive from the zoomed-in view. For example, using the same changes we made to include the item count with the group data for Scenario 1 (see "Quickstart #2b" above), we can just add a little more to the zoomed-out item template (as done in the modified sample in this chapter's companion content):



The other thing you need to know is that the semantic zoom control does not work with arbitrary child elements. An exception about a missing `zoomableView` property will tell you this! Each child control must provide an implementation of the [WinJS.UI.IZoomableView](#) interface through a property called `zoomableView`. Of all built-in HTML and WinJS controls, only `ListView` does this, which is why you typically see semantic zoom in that context. However, you can certainly provide this interface on a custom control, where the object returned by the constructor should contain a `zoomableView` property, which is an object containing the methods of the interface. Among these methods are `beginZoom` and `endZoom` for obvious purposes, and `getCurrentItem` and `setCurrentItem` that enable the semantic zoom control to zoom in to the right group when it's tapped in the zoomed-out view.

For more details, check out the [HTML SemanticZoom for custom controls sample](#), which also serves as another example of a custom control.

FlipView Features and Styling

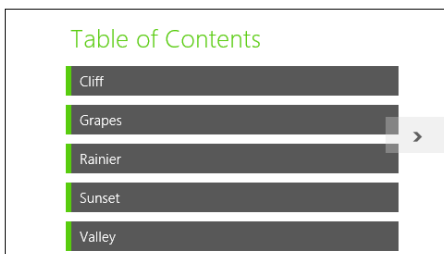
For all the glory that `ListView` merits as the richest and most sophisticated control in all of WinJS, we don't want to forget the humble `FlipView`! Thus before we delve wholly into `ListView`, let's spend a few pages covering `FlipView` and its features through the other scenarios in the [FlipView control sample](#). It's worth mentioning too that although this sample demonstrates the control's capabilities in a relatively small area, a `FlipView` can be any size, even occupying most of the screen. A common use for the control, in fact, is to let users flip through full-sized images in a photo gallery. Of course, the control can

be used anywhere it's appropriate, large or small. See [Guidelines for FlipView controls](#) for more on how best to use the control.

Anyway, Scenario 2 in the sample ("Orientation and Item Spacing") demonstrates the control's [orientation](#) property. This determines the placement of the arrow controls: left and right ([horizontal](#)) or top and bottom ([vertical](#)) as shown below. It also determines the enter and exit animations of the items and whether the control uses the left/right or up/down arrow keys for keyboard navigation. This scenario also let you set the [itemSpacing](#) property, which determines the amount of space between items when you swipe items using touch (below right). Its effect is not visible when using the keyboard or mouse to flip; to see it, you may need to use touch emulation in the Visual Studio simulator to partly drag between items.

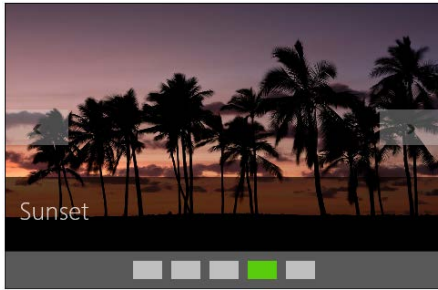


Scenario 3 ("Using interactive content") shows the use of a *template function* instead of a declarative template. We'll talk more of such functions in "How Templates Really Work" later in this chapter, but put simply, a template function or *renderer* creates elements and sets their properties procedurally, which is essentially what [WinJS.Binding.Template](#) does from the markup you give it. This allows you to render an item differently (that is, create different elements or customize style classes) depending on its actual data. In Scenario 3, the data source contains a "table of contents" item at the beginning, for which the renderer (a function called [mytemplate](#) in `js/interactiveContent.js`) creates a completely different item:



The scenario also sets up a listener for `click` events on the TOC entries, the handler for which flips to the appropriate item by setting the FlipView's `currentPage` property. The picture items then have a back link to the TOC. See the `clickHandler` function in the code for both of these actions.

Scenario 4 ("Creating a context control") demonstrates adding a navigation control overlay to each item:



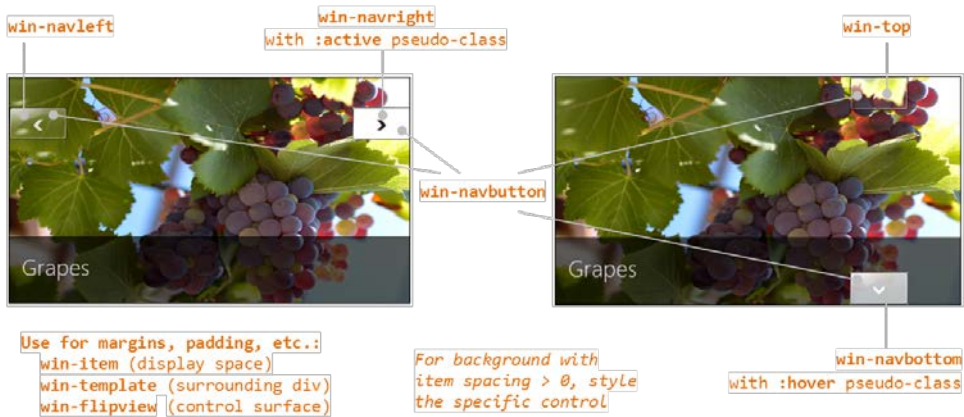
The items themselves are again rendered using a declarative template, which in this case just contains a placeholder `div` called `ContextContainer` for the navigation control (html/context-Control.html):

```
<div>
  <div id="contextControl_FlipView" class="flipView" data-win-control="WinJS.UI.FlipView"
    data-win-options="{ itemDataSource: DefaultData.bindingList.dataSource,
      itemTemplate: contextControl_ItemTemplate }">
  </div>
  <div id="ContextContainer"></div>
</div>
```

When the control is initialized in the `processed` method of `js/contextControl.js`, the sample calls the FlipView's async `count` method. The completed handler, `countRetrieved`, then creates the navigation control using a row of styled radiobuttons. The `onpropertychange` handler for each radiobutton then sets the FlipView's `currentPage` property.

Scenario 4 also sets up listeners for the FlipView's `pageselectd` and `pagevisibilitychanged` events. The first is used to update the navigation radiobuttons when the user flips between pages. The other is used to prevent clicks on the navigation control while a flip is happening. (The event occurs when an item changes visibility and is fired twice for each flip, once for the previous item, and again for the new one.)

Scenario 5 ("Styling Navigation Buttons") demonstrates the styling features of the FlipView, which involves various `win-*` styles and pseudo-classes as shown here:



If you were to provide your own navigation buttons in the template (wired to the [next](#) and [previous](#) methods), hide the default by adding `display: none` to the `<control selector> .win-navbutton` style rule.

Finally, there are a few other methods and events for the FlipView that aren't used in the sample, so here's a quick rundown of those:

- [pageCompleted](#) is an event that is raised when flipping to a new item is fully completed (that is, the new item has been rendered). In contrast, the aforementioned [pageselectd](#) event will fire when a *placeholder* item (not fully rendered) has been animated in. See "Template Functions (Part 2)" at the end of this chapter.
- [datasourcecountchanged](#) is an event raised for obvious purpose, which something like Scenario 4 would use to refresh the navigation control if items could be added or removed from the data source.
- [next](#) and [previous](#) are methods to flip between items (like [currentPage](#)), which would be useful if you provided your own navigation buttons.
- [forceLayout](#) is a method to call specifically when you make a FlipView visible by removing a `display: none` style. (The FlipView sample actually calls this whenever you change scenarios, but it's not necessary because it never changes the style.)
- [setCustomAnimations](#) allows you to control the animations used when flipping forward, flipping backward, and jumping to a random item.

For details on all of these, refer to the [WinJS.UI.FlipView](#) documentation.

Data Sources

In all the examples we've seen thus far, we've been using an in-memory data source built on [WinJS.Binding.List](#). Clearly, there are other types of data sources and it certainly doesn't make sense to load everything into memory first. How, then, do we work with such sources?

WinJS provides some help in this area. First is the [WinJS.UI.StorageDataSource](#) object that works with files in the file system, as the next section demonstrates with a FlipView and the Pictures Library. The other is [WinJS.UI.VirtualizedDataSource](#), which is meant for you to use as a base class for a custom data source of your own, an advanced scenario that we'll touch on only briefly.

A FlipView Using the Pictures Library

For everything we've seen in the FlipView sample already, it really begs for the ability to do something completely obvious: flip through pictures files in a folder. Using what we've learned so far, how would we implement something like that? We already have an item template containing an `img` tag, so perhaps we just need some URLs for those files. Perhaps we could make an array of these using an API like [Windows.Storage.KnownFolders.picturesLibrary.GetFilesAsync](#) (declaring the *Pictures Library* capability in the manifest, of course!). This would give us a bunch of [StorageFile](#) objects for which we could call [URL.createObjectURL](#). We could store those URLs in an array and then wrap it up with [WinJS.Binding.List](#):

```
var myFlipView = document.getElementById("pictures_FlipView").winControl;

Windows.Storage.KnownFolders.picturesLibrary.GetFilesAsync()
    .done(function (files) {
        var pixURLs = [];

        files.forEach(function (item) {
            var url = URL.createObjectURL(item, {oneTimeOnly: true });

            pixURLs.push({type: "item", title: item.name, picture: url });
        });

        var pixList = new WinJS.Binding.List(pixURLs);
        myFlipView.itemDataSource = pixList.dataSource;
    });
```

Although this approach works, it can consume quite a bit of memory with a larger number of high-resolution pictures because each picture has to be fully loaded to be displayed in the FlipView. This might be just fine for your scenario but in other cases would consume more resources than necessary. It also has the drawback that the images are just stretched or compressed to fit into the FlipView without any concern for aspect ratio, and this doesn't produce the best results.

A better approach is to use the [WinJS.UI.StorageDataSource](#) that again works directly with the file system instead of an in-memory array. I've implemented this as a Scenario 8 in the modified FlipView

sample code in this chapter's companion content. (Another example can be found in the [StorageDataSource and GetVirtualizedFilesVector sample](#).) Here we can use a shortcut to get a data source for the Pictures library:

```
myFlipView.itemDataSource = new WinJS.UI.StorageDataSource("Pictures");
```

"Pictures" is a shortcut because the first argument to [StorageDataSource](#) is actually something called a file query that comes from the [Windows.Storage.Search](#) API, a subject we'll see in more detail in Chapter 8, "State, Settings, Files, and Documents." These queries, which feed into the powerful [Windows.Storage.StorageFolder.createFileQueryWithOptions](#) function, are ways to enumerate files in a folder along with metadata like album covers, track details, and thumbnails that are cropped to maintain the aspect ratio. Shortcuts like "Pictures" (also "Music", "Documents", and "Videos" that all require the associated capability in the manifest) just create typical queries for those document libraries.

The caveat with [StorageDataSource](#) is that it doesn't directly support one-way binding, so you'll get an exception if you try to refer to item properties directly in a template. To work around this, you have to explicitly use [WinJS.Binding.oneTime](#) as the initializer function for each property:

```
<div id="pictures_ItemTemplate" data-win-control="WinJS.Binding.Template">
  <div class="overlaidItemTemplate">
    <img class="image" data-win-bind="src: thumbnail InitFunctions.thumbURL;
      alt: name WinJS.Binding.oneTime" />
    <div class="overlay">
      <h2 class="ItemTitle" data-win-bind="innerText: name WinJS.Binding.oneTime"></h2>
    </div>
  </div>
</div>
```

In the case of the `img.src` property, the file query gives us items of type [Windows.Storage.-BulkAccess.FileInformation](#) (the `source` variable in the code below), which contains a thumbnail image, not a URL. To convert that image data into a URL, we need to use our own binding initializer:

```
WinJS.Namespace.define("InitFunctions", {
  thumbURL: WinJS.Binding.initializer(function (source, sourceProp, dest, destProp) {
    if (source.thumbnail) {
      dest.src = URL.createObjectURL(source.thumbnail, { oneTimeOnly: true });
    }
  })
});
```

In this initializer, the `src : thumbnail` part of `data-win-bind` is actually ignored because we're just setting the image's `src` property directly to `source.thumbnail`. This is just a form of one-way binding.

Note that thumbnails aren't always immediately available in the [FileInformation](#) object, which is why we have to verify that we actually have one before creating a URL for it. This means that quickly flipping through the images might show some blanks. To solve this particular issue, we can listen for the [FileInformation.onthumbnailupdated](#) event and update the item at that time. The best way to accomplish this is to use the [StorageDataSource.loadThumbnail](#) helper, which makes sure to call [removeEventListener](#) for this WinRT event. (See "WinRT Events and removeEventListener" in Chapter 3.)

You can use this method within a binding initializer, as demonstrated in Scenario 1 of the aforementioned [StorageDataSource and GetVirtualizedFilesVector sample](#), or within a rendering function that takes the place of the declarative template. We'll do this for our FlipView sample later on, in "How Templates Really Work," which also lets us avoid the one-time binding tricks.

As a final note, Scenario 6 of the FlipView sample contains another example of a different data source, specifically one working with Bing Search. For that, let's look at custom data sources.

Custom Data Sources

Now that we've seen a collection control like FlipView working against two different data sources, you're probably starting to correctly guess that all data sources share some common characteristics and a common programmatic interface. This is demonstrated again in Scenario 6 of the FlipView sample as well as in the [HTML ListView working with data sources sample](#) shown in Figure 5-6, as we'll see in this section.

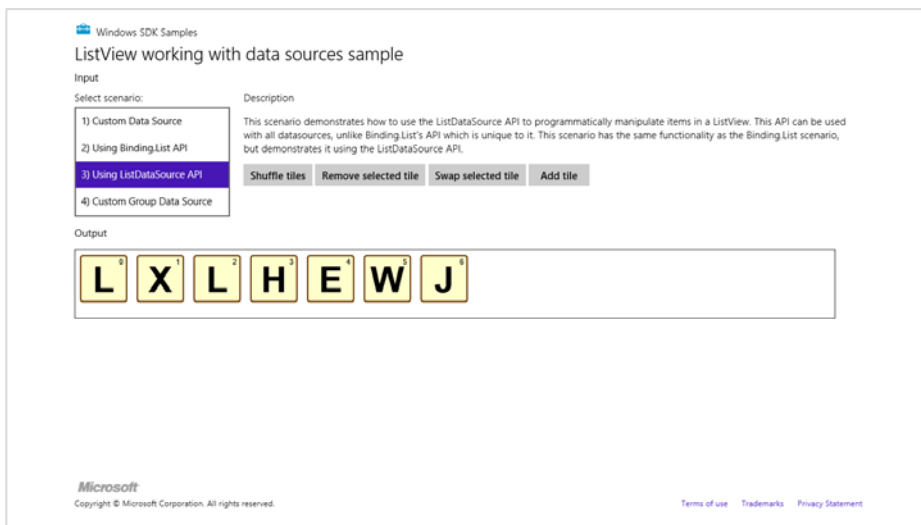


FIGURE 5-6 The HTML ListView working with data sources sample.

Scenarios 2 and 3 of this sample both work against a `WinJS.Binding.List` data source, as we've already seen, and provide buttons to manipulate that data source. Those changes are reflected in the output. The difference between the two scenarios is that Scenario 2 manipulates the data through `WinJS.Binding.List` methods like `move`, whereas Scenario 3 manipulates the underlying data source directly through a more generic [ListDataSource](#) API.

Because of data binding, changes to the data are reflected in the ListView control either way, but there are three important differences. First, the [ListDataSource](#) interface is common to all data sources, so any code you write against it will work for any kind of data source. Second, its methods are generally asynchronous because a data source might be connected to an online service or other such resource. Third, [ListDataSource](#) provides for batching changes together by calling [beginEdits](#), which will defer any change notifications to any external bound objects until [endEdits](#) is called. This allows you to do bulk data editing in ways that can improve ListView performance.

Scenarios 1 and 4 of the sample demonstrate how to create custom data sources. Scenario 1 creates a data source for Bing searches; Scenario 4 creates one for an in-memory array that you could adapt to work against a data feed that's only brought down from a service a little at a time. What's important for all these is that they implement something called a data adapter, which is an object with the methods of the [WinJS.UI.IListDataAdapter](#) interface. This provides for capabilities like caching, virtualization, change detection, and so forth. Fortunately, you get most of these methods by deriving your class from [WinJS.UI.VirtualizedDataSource](#) and then implementing those methods you need to customize. In the sample, for instance, the [bingImageSearchDataSource](#) is defined as follows (see [js/BingImageSearchDataSource.js](#)):

```
bingImageSearchDataSource = WinJS.Class.derive(WinJS.UI.VirtualizedDataSource,
    function (devkey, query) {
        this._baseDataSourceConstructor(new bingImageSearchDataAdapter(devkey, query));
    });
```

where the [bingImageSearchDataAdapter](#) class implements only the [getCount](#) and [itemsFromIndex](#) methods directly.

For a further deep-dive on this subject beyond the sample, I refer you to a session from the 2011 //Build conference entitled [APP210-T: Build data-driven collection and list apps in HTML5](#). Some of the details have since changed (like the [ArrayDataSource](#) is now [WinJS.Binding.List](#)), but on the whole it very much explains all the mechanisms. It's also helpful to remember that you can use other languages like C# and C++ to write custom data sources as well. Such languages could offer much higher performance within the data source and have access to higher-performance APIs than JavaScript.

How Templates Really Work

Earlier, when we looked at the Grid App project template, I mentioned that you can use a function instead of a declarative template for properties like [itemTemplate](#) (FlipView and ListView) and [groupHeaderTemplate](#) (ListView). This is an important capability because it allows you to dynamically render items in a collection individually, using its particular contents to customize its view. It also allows you to initialize item elements in ways that can't be done in the declarative form, such as cell spanning, delay-loading images, adding event handlers for specific parts of an item, and optimizing performance.

We'll return to some of these topics later on. For the time being, it's helpful to understand exactly what's going on with declarative templates and how that relates to custom template functions.

Referring to Templates

As I noted before, when you refer to a declarative template in the FlipView or ListView controls, what you're actually referring to is an *element*, not an element id. The id works because the app host creates variables with those names for the elements they identify. However, we don't actually recommend this approach, especially within page controls (which you'll probably use often). The first concern is that only one element can have a particular id, which means you'll get really strange behavior if you happen to render the page control twice in the same DOM.

The second concern is a timing issue. The element id variable that the app host provides isn't created until the chunk of HTML containing the element is added to the DOM. With page controls, `WinJS.UI.processAll` is called before this time, which means that element id variables for templates in that page won't yet be available. As a result, any controls that use an id for a template will either throw an exception or just show up blank. Both conditions are guaranteed to be terribly, terribly confusing.

To avoid this issue with a declarative template, place the template's name in its `class` attribute:

```
<div data-win-control="WinJS.Binding.Template" class="myItemTemplate" ...></div>
```

Then in your control declaration, use the syntax `select("<selector>")` in the options record, where `<selector>` is anything supported by `element.querySelector`:

```
<div data-win-control="WinJS.UI.ListView"
    data-win-options="{ itemTemplate: select('.myItemTemplate') }"></div>
```

There's more to this, actually, than just a `querySelector` call. The `select` function within the options searches from the root of its containing page control. If no match is found, it looks for another page control higher in the DOM, then looks in there, continuing the process until a match is found. This lets you safely use two page controls at once that both contain the same class name for different templates, and each page will use the template that's most local.

You can also retrieve the template element using `querySelector` directly in code and assign the result to the `itemTemplate` property. This would typically be done in a page's `ready` function, as demonstrated in the Grid App project, and doing so avoids both concerns identified here because `querySelector` will be scoped to the page contents and will happen after `WinJS.UI.processAll`.

Template Elements and Rendering

The next interesting question about templates is, what, exactly, do we get when instantiating a `WinJS.Binding.Template`? This is more or less another WinJS control that turns into an element when you call `WinJS.UI.processAll`. But it's different in that it removes all its child elements from the DOM, so it never shows up by itself. It doesn't even set a `winControl` property on its containing element.

What *does* have, however, is this exceptionally useful function called `render`. Given a data context (an object with properties) and an element, `render` creates a full copy of the template inside the element, resolving any data-binding relationships in the template (in both `data-win-bind` and `data-win-options` attributes) using the data context. In short, think of a declarative template as a set of instructions that the `render` method uses to do all the necessary `createElement` calls along with setting properties and doing data binding.

As shown on the [How to use templates to bind data](#) topic, you can just instantiate and render a template anywhere you want:

```
var templateElement = document.getElementById("templateDiv");
var renderHere = document.getElementById("targetElement");
renderHere.innerHTML = "";

WinJS.UI.process(templateElement).then(function (templateControl) {
    templateControl.render(myDataItem, renderHere);
});
```

It should be wholly obvious that this is exactly what FlipView and ListView controls do for each item in a given data source. In the case of FlipView, it calls its item template's `render` method each time you switch to a different item in the data source. ListView iterates over its `itemDataSource` and calls the item template's `render` for each item, and does something similar for its `groupDataSource` and the `groupHeaderTemplate`.

Template Functions (Part 1): The Basics

Knowing now that a `WinJS.Binding.Template` control is basically just a set of declarative instructions for its `render` method, you can just create a custom function to do the same job directly. That is, in addition to an element, the FlipView/ListView `itemTemplate` properties and the ListView `groupHeaderTemplate` property can also accept a renderer function. The controls use `typeof` at run time to determine what you've assigned to these properties, so if you provide a template element, the controls will call its `render` method; if you provide a function, the controls will just call that function for each item that needs to be rendered. This provides a great deal of flexibility to customize the template based on individual item data.

Indeed, a renderer allows you to individually control not only *how* the elements for each item are constructed but also *when*. As such, renderers are the primary means through which you can implement five progressive levels of optimization, especially for ListView. Warning! There be promises ahead! Well, I'll save most of that discussion for the end of the chapter, because we need to look at other ListView features first. But here let's at least look at the core structure of a renderer that applies to both FlipView and ListView, which you can see in the [HTML ListView item templates](#) and the [HTML ListView optimizing performance](#) samples. We'll be drawing code from the latter.

For starters, you can specify a renderer by name in `data-win-options` for both the FlipView and ListView controls. That function must be marked for processing as discussed in Chapter 4 since it definitely participates in `WinJS.UI.processAll`, so this is a great place to use `WinJS.Utilities.-`

`markSupportForProcessing`. Note that if you assign a function to an `itemTemplate` or `groupHeaderTemplate` property in JavaScript, it doesn't need the mark.

In its basic form, a template function receives an item promise as its first argument and returns a promise whose completed handler creates the elements for that item. Huh? Yeah, that confuses me too! So let's look at the basic structure in terms of two functions:

```
function basicRenderer(itemPromise) {
    return itemPromise.then(buildElement);
};

function buildElement (item) {
    var result = document.createElement("div");

    //Build up the item, typically using innerHTML
    return result;
}
```

The renderer is the first function above. It simply says, "When `itemPromise` is fulfilled, meaning the item is available, call the `buildElement` function with that item." By returning the promise from `itemPromise.then` (not `done`, mind you!), we allow the collection control that's using this renderer to chain the item promise and the element-building promise together. This is especially helpful when the item data is coming from a service or other potentially slow feed, and it's very helpful with incremental page loading because it allows the control to cancel the promise chain if the page is scrolled away before those operations complete. In short, it's a good idea!

Just to show it, here's how we'd make a renderer directly usable from markup, as in `data-win-options = "{itemTemplate: Renderers.basic }"`:

```
WinJS.Namespace.define("Renderers", {
    basic: WinJS.Utilities.markSupportedForProcessing(function (itemPromise) {
        return itemPromise.then(buildElement);
    })
});
```

It's also common to just place the contents of a function like `buildElement` directly within the renderer itself, resulting in a more concise expression of the exact same structure:

```
function basicRenderer(itemPromise) {
    return itemPromise.then(function (item) {
        var result = document.createElement("div");

        //Build up the item, typically using innerHTML

        return result;
    });
};
```

What you then do inside the element creation function (whether named or anonymous) defines the item's layout and appearance. Returning to Scenario 8 that we've added to the FlipView sample, we can take the following declarative template, where we had to play some tricks to get data binding to work:

```
<div id="pictures_ItemTemplate" data-win-control="WinJS.Binding.Template">
  <div class="overlaidItemTemplate">
    <img class="image" data-win-bind="src: thumbnail InitFunctions.thumbURL;
      alt: name WinJS.Binding.oneTime" />
    <div class="overlay">
      <h2 class="ItemTitle" data-win-bind="innerText: name WinJS.Binding.oneTime"></h2>
    </div>
  </div>
</div>
```

and turn it into the following renderer, keeping the two functions here separate for the sake of clarity:

```
//Earlier: assign the template in code
myFlipView.itemTemplate = thumbFlipRenderer;

//The renderer (see Template Functions (Part 2) later in the chapter for optimizations)
function thumbFlipRenderer(itemPromise) {
  return itemPromise.then(buildElement);
};

//A function that builds the element tree
function buildElement (item) {
  var result = document.createElement("div");
  result.className = "overlaidItemTemplate";

  var innerHTML = "<img class='thumbImage'>";
  var innerHTML += "<div class='overlay'>";
  innerHTML += "<h2 class='ItemTitle'>" + item.data.name + "</h2>";
  innerHTML += "</div>";

  result.innerHTML = innerHTML;

  //Set up a listener for thumbnailUpdated that will render to our img element
  var img = result.querySelector("img");
  WinJS.UI.StorageDataSource.loadThumbnail(item, img).then();

  return result;
}
```

Because we have the individual `item` in hand already, we don't need to quibble over the details of declarative data binding and converters: we can just directly use the properties we need from `item.data`. As before, remember that the `thumbnail` property of the `FileInformation` item might not be set yet. This is where we can use the `StorageDataSource.loadThumbnail` method to listen for the `FileInformation.onthumbnailupdated` event. This helper function will render the thumbnail into our `img` element when the thumbnail becomes available (with a little animation to boot!).

Tip You might also notice that I'm building most of the element by using the root `div.innerHTML` property instead of calling `createElement` and `appendChild` and setting individual properties explicitly. Except for very simple structures, setting `innerHTML` on the root element is more efficient because we minimize the number of DOM API calls. This doesn't matter so much for a `FlipView` control whose items are rendered one at a time, but it becomes very important for a `ListView` with potentially thousands of items. Indeed, when we start looking at performance optimizations, we'll also want to render the item in various stages, such as delay-loading images. We'll see all the details in the "Template Functions (Part 2): Promises, Promises!" section at the end of this chapter.

ListView Features and Styling

Having already covered data sources and templates along with a number of `ListView` examples, we can now explore the additional features of the `ListView` control, such as layouts, styling, and cell spanning for multisize items. Optimizing performance then follows in the last section of this chapter. First, however, let me answer a very important question.

When Is ListView the Wrong Choice?

`ListView` is the hands-down richest control in all of Windows. It's very powerful, very flexible, and, as we're already learning, very deep and intricate. But for all that, sometimes it's also just the wrong choice! Depending on the design, it might be easier to just use basic HTML/CSS layout.

Conceptually, a `ListView` is defined by the relationship between three parts: a data source, templates, and layout. That is, items in a data source, which can be grouped, sorted, and filtered, are rendered using templates and organized with a layout (typically with groups and group headers). In such a definition, the `ListView` is intended to help visualize a collection of similar and/or related items, where their groupings also have a relationship of some kind.

With this in mind, the following factors strongly suggest that a `ListView` is a *good* choice to display a particular collection:

- The collection can contain a variable number of items to display, possibly a very large number, showing more when the app runs on a larger display.
- It makes sense to organize and reorganize the items in various groups.
- Group headers help to clarify the common properties of the items in those groups, and they can be used to navigate to a group-specific page.
- It makes sense to sort and/or filter the items according to different criteria.
- Different groupings of items and information about those groups suggest ways in which semantic zoom would be a valuable user experience.

- The groups themselves are all similar in some way, meaning that they each refer to a similar kind of thing. Different place names, for example, are similar; a news feed, a list of friends, and a calendar of holidays are not similar.
- Items might be selectable individually or in groups, such that app bar commands could act on them.

On the flip side, opposite factors suggest that a ListView is *not* the right choice:

- The collection contains a limited or fixed number of items, or it isn't really a collection of related items at all.
- It doesn't make sense to reorganize the groupings or to filter or sort the items.
- You don't want group headers at all.
- You don't see how semantic zoom would apply.
- The groups are highly dissimilar—that is, it wouldn't make sense for the groups to sit side-by-side if the headers weren't there.

Let me be clear that I'm not talking about *design* choices here—your designers can hand you any sort of layout they want and as a developer it's *your* job to implement it! What I'm speaking to is how you choose to approach that implementation, whether with controls like ListView or just with HTML/CSS layout.

I say this because in working with the developers who created the very first apps for the Windows Store, we frequently saw them trying to use ListView in situations where it just wasn't needed. An app's hub page, for example, might combine a news feed, a list of friends, and a calendar. An item details page might display a picture, a textual description, and a media gallery. In both cases, the page contains a limited number of sections and the sections contain very different content, which is to say that there isn't a similarity of items across the groups. Because of this, using a ListView is more complicated than just using a single pannable `div` with a CSS grid in which you can lay out whatever sections you need.

Within those sections, of course, you might use ListView controls to display an item collection, but for the overall page, a simple `div` is all you need. I've illustrated these choices in Figure 5-7 using an image from the [Navigation design for Windows Store apps](#) topic, since you'll probably receive similar images from your designers. Ignoring the navigation arrows, the hub and details pages typically use a `div` at the root, whereas a section page is often a ListView. Within the hub and details pages there might be some ListView controls, but where there is essentially fixed content (like a single item), the best choice is a `div`.

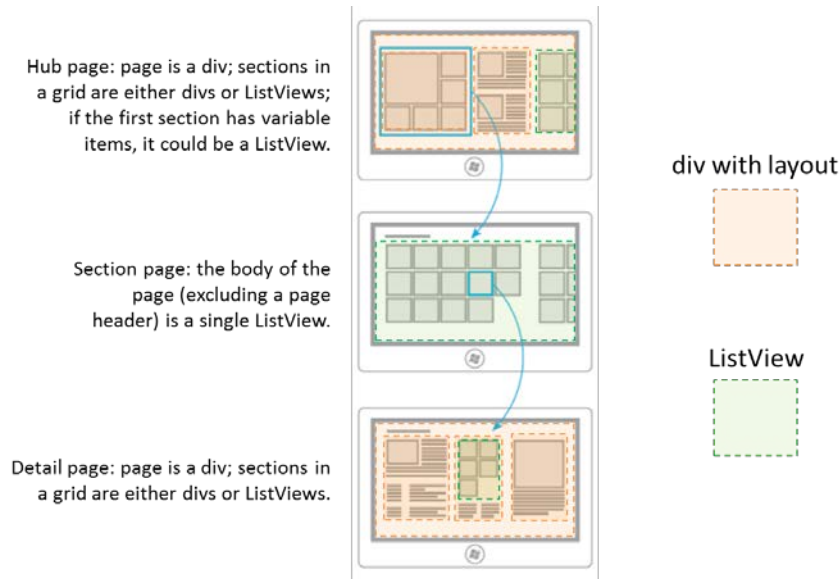


FIGURE 5-7 Breaking down typical hub-section-detail page designs into `div` elements and `ListView` controls.

A clue that you're going down the wrong path, by the way, is if you find yourself trying to combine multiple collections of unrelated data into a single source, binding that source to a `ListView`, and implementing a renderer to tease all the data apart again so that everything renders properly! All that extra work could be avoided simply by using HTML/CSS layout.

For more on `ListView` design, see [Guidelines and checklist for ListView controls](#).

Options, Selections, and Item Methods

In previous sections we've already seen some of the options you can use when creating a `ListView`, options that correspond to the control's properties that are accessible also from JavaScript. Let's look now at the complete set of properties, methods, and events, which I've organized into a few groups—after all, those properties and methods form quite a collection in themselves! Since the details for the individual properties are found on the [WinJS.UI.ListView](#) reference page, what's most useful here is to understand how the members of these groups relate:

- Addressing items** The `currentItem` property gets or sets the item with the focus, and the `elementFromIndex` and `indexOfElement` methods let you cross-reference between an item index and the DOM element for that item. The latter could be useful if you have other controls in your item template and need to determine the surrounding item in an event handler.
- Item visibility** The `indexOfFirstVisible` and `indexOfLastVisible` properties let you know what indices are visible, or they can be used to scroll the `ListView` appropriate for a given item. The `ensureVisible` method brings the specified item into view, if it's been loaded. There is also the `scrollPosition` property that contains the distance in pixels between the first item in the list

and the current viewable area. Though you can set the scroll position of the ListView with this property, it's reliable only if the control's `loadingState` (see "Loading state" group below) is `ready`, otherwise the ListView may not yet know its actual dimensions. It's thus recommended that you instead use `ensureVisible` or `indexOfFirstVisible` to control scroll position.

- **Item invocation** The `itemInvoked` event, as we've seen, fires when an item is tapped, unless the `tapBehavior` property is not set to `none`, in which case no invocation happens. Other `tapBehavior` values from the [WinJS.UI.TapBehavior](#) enumeration will always fire this event but determine how the item selection is affected by the tap. Do note that you can override the selection behavior on a per-item basis using the `selectionchanging` event and suppress the animation if needed. See the "Tap/Click Behaviors" sidebar after this list.
- **Item selection** The `selectionMode` property contains a value from the [WinJS.UI.-SelectionMode](#) enumeration, indicating single-, multi-, or no selection. At all times the `selection` property contains a [ListViewItems](#) object whose methods let you enumerate and manipulate the selected items (such as setting selected items through its `set` method). Changes to the selection fire the `selectionchanging` and `selectionchanged` events; with `selectionchanging`, its `args.detail.newSelection` property contains the newly selected items. For more on this, refer to the [HTML ListView customizing interactivity sample](#).
- **Swiping** Related to item selection is the `swipeBehavior` property that contains a value from the [WinJS.UI.SwipeBehavior](#) enumeration. "Swiping" or "cross-slide" is the touch gesture on an item to select it where the gesture moves perpendicular to the panning direction of the list. If this is set to `none`, swiping has no effect on the item and the gesture is bubbled up to the parent elements, allowing a vertically oriented ListView or its surround page to pan. If this is set to `select`, the gesture is processed by the item to select it.
- **Data sources and templates** We've already seen the `groupDataSource`, `groupHeaderTemplate`, `itemDataSource`, and `itemTemplate` properties already. There are two related properties, `resetGroupHeader` and `resetItem`, that contain functions that the ListView will call when recycling elements. This is explained in "Template Functions (Part 2): Promises, Promises!" section.
- **Layout** As we've also seen, the `layout` property (an object) describes how items are arranged in the ListView, which we'll talk about more in "Layouts and Cell Spanning" below. We've also seen the `forceLayout` function that's specifically used when a `display: none` style is removed from a ListView and it needs to re-render itself.
- **Loading behavior** As explained in the "Optimizing ListView Performance" section later on, this group determines how the ListView loads pages of items (which is why `ensureVisible` doesn't always work if a page hasn't been loaded). When the `loadingBehavior` property is set to `"randomaccess"` (the default), the ListView's scrollbar reflects the total number of items but only five total pages of items (to a maximum of 1000) are kept in memory at any given time as the user pans around. (The five pages are the current page, plus two buffer pages both ahead and behind.) The other value, `"incremental"`, is meant for loading some number of pages initially

and then loading additional pages when the user scrolls toward the end of the list (keeping all items in memory thereafter). Incremental loading works with the `automaticallyLoadPages`, `pagesToLoad`, and `pagesToLoadThreshold` properties, along with the `loadMorePages` method, as we'll see.

- **Loading state** The read-only `loadingState` property contains either `"itemsLoading"` (the list is requesting items and headers from the data source), `"viewportLoaded"` (all items and headers that are visible have been loaded), `"itemsLoaded"` (all remaining nonvisible buffered items have been loaded), or `"complete"` (all items are loaded, content in the templates is rendered, and animations have finished). Whenever this property changes, which is basically whenever the `ListView` needs to update its layout due to panning, the `loadingStateChanged` event fires.
- **Miscellany** `addEventListener`, `removeEventListener`, and `dispatchEvent` are the standard DOM methods for handling and raising events. These can be used with any other event that the `ListView` supports, including `contentanimating` that fires when the control is about to run an item entrance or transition animation, allowing you to either prevent or delay those animations. The `zoomableView` property contains the `IZoomableView` implementation as required by semantic zoom (apps will never manipulate this property).

Sidebar: Tap/Click Behavior

When you tap or click an item in a `ListView` with the `tapBehavior` property set to something other than `none`, there's a little ~97% scaling animation to acknowledge the tap. If you have some items in a list that can't be invoked (like those in a particular group or ones that you show as disabled because backing data isn't yet available), they'll still show the animation because the `tapBehavior` setting applies to the whole control. To remove the animation for any specific item, you can add the `win-interactive` class to its element within a renderer function, which is a way of saying that the item internally handles tap/click events, even if it does nothing but eat them. If at some later time the item becomes invocable, you can, of course, remove that class.

If you need to suppress selection for an item, add a handler for the `ListView`'s `selection-changing` event and call its `args.detail.preventTapBehavior` method. This works for all selection methods, including swipe, mouse click, and the Enter key.

Styling

Following the precedent of Chapter 4 and the earlier section on ListView, styling is best understood visually as in Figure 5-8, where I've applied some garish CSS to some of the `win-*` styles so that they stand out. I highly recommend you look at the [Styling the ListView and its items](#) topic in the documentation, which details some additional styles that are not shown here.

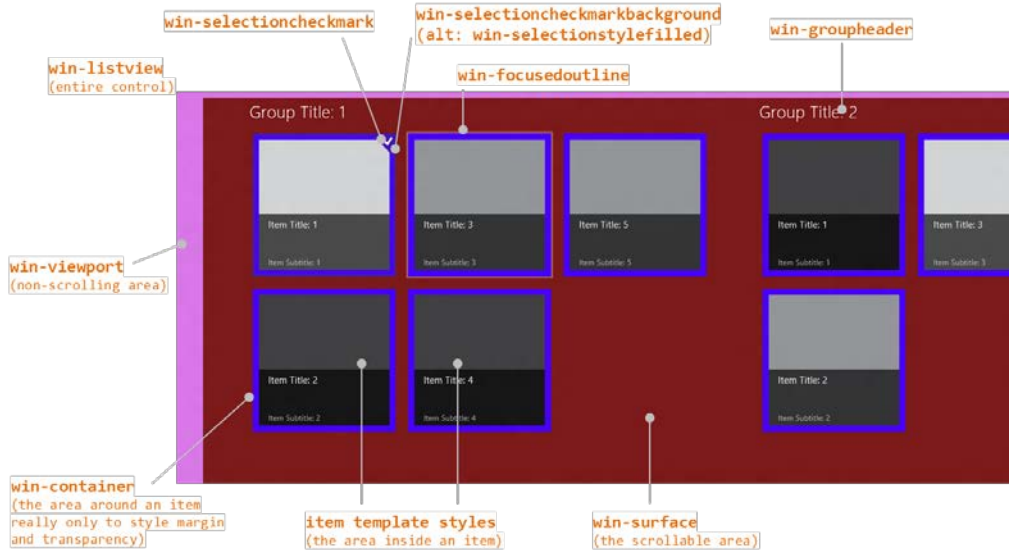
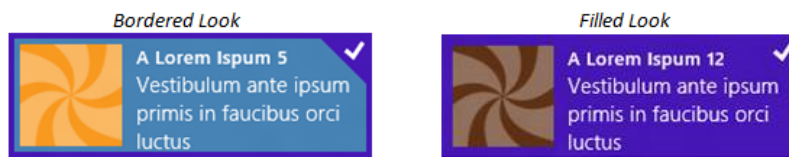


FIGURE 5-8 Style classes as utilized by the ListView control.

A few notes about styling:

- Remember that Blend is your best friend here!
- As with styling the FlipView, a class like `win-listview` is most useful with styles like margins and padding, since a property like its background color won't actually show through anywhere (unlike `win-viewport` and `win-surface`).
- `win-viewport` styles the nonscrolling background of the ListView and is rarely used, perhaps for a nonscrolling background image. `win-surface` styles the scrolling background area.
- `win-container` primarily exists for two things. One is to create space between items using `margin` styles, and the other is to override the default background color, often making its background transparent so that the `win-surface` or `win-viewport` background shows through. Note that if you set a `padding` style here instead of `margin`, you'll create areas *around* what the user will perceive as the item that are still invoked as the item. Not good. So always use `margin` to create space between items.

- Though `win-item` is listed as a style, it's deprecated and may be removed in the future: just style the item template directly.
- The documentation points out that styles like `win-container` and `win-surface` are used by multiple WinJS controls. (FlipView uses a few of them.) If you want to override styles for a ListView, be sure to scope your selectors them with other classes like `.win-listview` or a particular control's id or class.
- The default ListView height is 400px, and the control does *not* automatically adjust itself to its content. You'll almost always want to override that style in CSS or set it from JavaScript when you know the space that the ListView should occupy, as we'll cover in Chapter 6, "Layout."
- Styles not shown in the figure but described on [Styling the ListView and its items](#) include `win-focusedoutline`, `win-selection`, `win-selected`, `win-selectionborder`, `win-selectionbackground`, and `win-selectionhint`. There is also the `win-selectionstylefilled` class that you add to an item to use a *filled* selection style rather than the default *bordered* style, as shown here:



Backdrops

There is another ListView visual that is a bit like styling but not affected by styling. This is called the backdrop, an effect that's turned on by default when you use the GridLayout. On low-end hardware, especially low-power mobile devices, panning around quickly in a ListView can very easily outpace the control's ability to load and render items. To give the user a visual indication of what they're doing, the GridLayout displays a simple backdrop of item outlines based on the default item size and pans that backdrop until such time as real items are rendered. As we'll see in the next section, you can disable this feature with the GridLayout's `disableBackdrop` property and override its default gray color with the `backdropColor` property.

Layouts and Cell Spanning

The ListView's `layout` property, which you can set at any time, contains an object that's used to organize the list's items. WinJS provides two prebuilt layouts: `WinJS.UI.GridLayout` and `WinJS.UI.ListLayout`. The first, already described earlier in this chapter, provides a horizontally panning two-dimensional layout that places items in columns (top to bottom) and then rows (left to right). The second is a one-dimensional top-to-bottom layout, suitable for vertical lists (as in snapped view). These both follow the recommended design guidelines for presenting collections.

Technically speaking, the `layout` property is an object in itself, containing some number of other properties along with a `type` property. Typically, you see the syntax `layout: { type: <layout> }` in a ListView's `data-win-options` string, where `<layout>` is `WinJS.UI.GridLayout` or `WinJS.UI.ListLayout` (technically, the name of a constructor function). In the declarative usage, `layout` can also contain options that are specific to the type. For example, the following configures a GridLayout with headers on the left and four rows:

```
layout: { type: WinJS.UI.GridLayout, groupHeaderPosition: 'left', maxRows: 4 }
```

If you create the layout object in JavaScript by using `new` to call the constructor directly (and assigning it to the `layout` property), you can specify additional options with the constructor. This is done in the Grid App project template's `initializeLayout` method in `pages/groupedItems/groupedItems.js`:

```
listView.layout = new ui.GridLayout({ groupHeaderPosition: "top" });
```

You can also set properties on the ListView's `layout` object in JavaScript once it's been created, if you want to take that approach. Changing properties will generally update the layout.

In any case, each layout has its own unique options. For GridLayout, we have these:

- `groupHeaderPosition` controls the placement of headers in relation to their groups; can be set to `"left"` or `"top"`.
- `maxRows` controls the number of items the layout will place vertically before starting another column.
- `backdropColor` provides for customizing the default backdrop color (see "Backdrops" in the previous section), and `disableBackdrops` turns off the effect entirely.
- `groupInfo` identifies a function that returns an object whose properties indicate whether cell spanning should be used and the size of the cell (see below). This is called only once within a layout process.
- `itemInfo` identifies a function for use with cell spanning that returns an object of properties describing the exact size for each item and whether the item should be placed in a new column (see below).

The GridLayout also has a read-only property called `horizontal` that's always `true`. As for the ListLayout, its `horizontal` property is always `false` and has no other configurable properties.

Now, because the ListView's `layout` property is just an object (or the name of a constructor for such an object), can you create a custom layout function of your own? Yes, you can: create a class that provides the same public methods as the built-in layouts, as described by the (currently under-documented) `WinJS.UI.Layout` class. From there the layout object can provide whatever other options (properties and methods) are applicable to it.

Now before you start thinking that you might need a custom layout, the GridLayout provides for something called *cell spanning* that allows you to create variable-sized items (not an option for ListView). This is what its `groupInfo` and `itemInfo` properties are for, as demonstrated in Scenarios 4 and 5 of the [HTML ListView item templates sample](#) and shown in Figure 5-9.

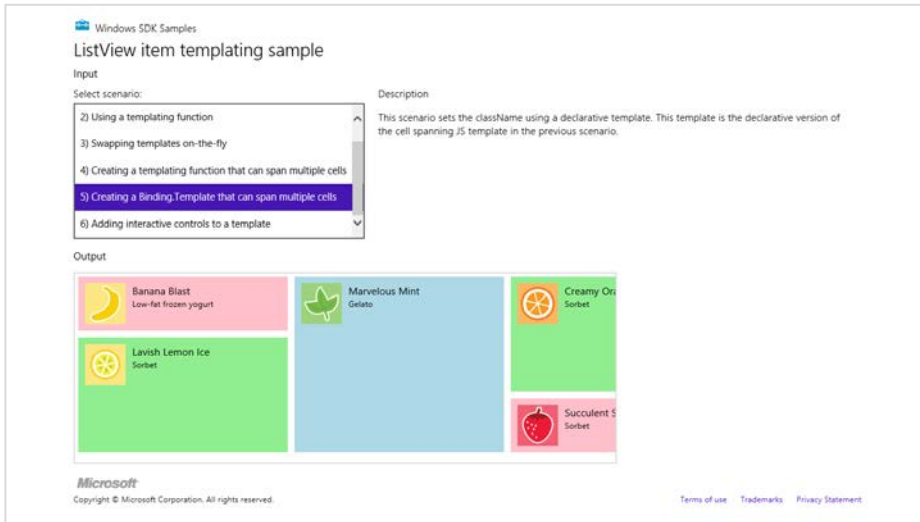


FIGURE 5-9 The ListView item templates sample showing multisize items through cell spanning.

The basic idea of cell spanning is to define a grid for the GridLayout based on the size of the smallest item (including padding and margin styles). For best performance, make the grid as coarse as possible, where every other element in the ListView is a multiple of that size.

You turn on cell spanning through the GridLayout’s `groupInfo` property. This is a function that returns an object with three properties: `enableCellSpanning`, which should be set to `true`, and `cellWidth` and `cellHeight`, which contain the pixel dimensions of your minimum cell (which, by the way, is what the GridLayout’s backdrop feature will use for its effects in this case). In the sample (see `js/data.js`), this function is named *groupInfo* like the layout’s property. I’ve given it a different name here for clarity:

```
function cellSpanningInfo() {
    return {
        enableCellSpanning: true,
        cellWidth: 310,
        cellHeight: 80
    };
}
```

You then specify this function as part of the `layout` property in `data-win-options`:

```
layout: { type: WinJS.UI.GridLayout, groupInfo: cellSpanningInfo }
```

or you can set `layout.groupInfo` from JavaScript. In any case, once you've announced your use of cell spanning, your item template should set each item's `style.width` and `style.height` properties, plus applicable padding values, to multiples of your `cellWidth` and `cellHeight` according to the following formulae (which are two arrangements of the same formula):

$$\begin{aligned} \text{templateSize} &= ((\text{cellSize} + \text{margin}) \times \text{multiplier}) - \text{margin} \\ \text{cellSize} &= ((\text{templateSize} + \text{margin}) / \text{multiplier}) - \text{margin} \end{aligned}$$

In the sample, these styles are set by assigning each item one of three class names: `smallListItemIconTextItem`, `mediumListItemIconTextItem`, and `largeListItemIconTextItem`, whose relevant CSS is as follows (from `css/scenario4.css` and `css/scenario5.css`):

```
.smallListItemIconTextItem {  
    width: 300px;  
    height: 70px;  
    padding: 5px;  
}  
  
.mediumListItemIconTextItem {  
    width: 300px;  
    height: 160px;  
    padding: 5px;  
}  
  
.largeListItemIconTextItem {  
    width: 300px;  
    height: 250px;  
    padding: 5px;  
}
```

Because each of these classes has padding, their actual sizes from CSS are 310x80, 310x170, and 310x260. The margin to apply in the formula comes from the `win-container` style in the WinJS stylesheet, which is 5px. Thus:

$$\begin{aligned} ((80 + 10) * 1) - 10 &= 80; \text{ minus 5px padding top and bottom} = \text{a height of 70px in CSS} \\ ((80 + 10) * 2) - 10 &= 170; \text{ minus 5px padding} = \text{height of 160px} \\ ((80 + 10) * 3) - 10 &= 260; \text{ minus 5px padding} = \text{height of 250px} \end{aligned}$$

The only difference between Scenario 4 and Scenario 5 is that the former assigns class names to the items through a template function. The latter does it through a declarative template and data-binds the class name to an item data field containing those values.

As for the `itemInfo` function, this is a way to optimize the performance of a `ListView` when using cell spanning. Without assigning a function to this property, the `GridLayout` has to manually determine the width and height of each item as it's rendered, and this can get slow if you pan around quickly with a large number of items. Since you probably already know item sizes yourself, you can return that information through the `itemInfo` function. This function receives an item index and returns an object with the item's `width` and `height` properties. (We'll see a working example in a bit.)

```
function itemInfo(itemIndex) {
  //determine values for itemWidth and itemHeight given itemIndex
  return {
    newColumn: false,
    itemWidth: itemWidth,
    itemHeight: itemHeight
  };
}
```

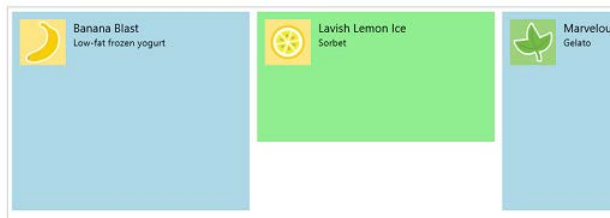
Clearly, this function will be called for every item in the list but only if cell spanning has been turned on through the `groupInfo` function. Again, unless your list is relatively small, you'll very much improve performance by providing item dimensions through this function, but be sure to return as quickly as you can.

You probably also noticed that `newColumn` property in the return value. As you might guess, setting this to `true` instructs the `GridLayout` to start a new column with this item, allowing you to control that particular behavior. You can even use `newColumn` by itself, if you like, with a smallish list.

Now you might be asking: what happens if I set different sizes in my item template but don't actually announce cell spanning? Well, you'll end up with overlapping (and rather confusing) items. This is because *the GridLayout takes the first item in a group as the basic measure for the rest of the items* (and the backdrop grid as well). It does *not* attempt to automatically size each item according to content. Try this with Scenarios 4 or 5: remove the `layout.groupInfo` property from the `ListView`'s `data-winoptions` in `html/scenario4.html` or `html/scenario5.html` and restart the app, and you'll see the medium and large items bleeding into those that follow:



Then go into `js/data.js`, set the first item's style in the `myCellSpanningData` array to be `largeListIconTextItem`, and restart; the `ListView` then does layout with that as the basic item size:



Using the first item's dimension like this underscores the fact that a `ListView` with cell spanning will take more time to render because it must measure each item as it gets laid out, with or without the `itemInfo` function. For this reason, cell spanning is probably best avoided for large lists.

Where all this gets a little more interesting, which the sample doesn't show, is how the `GridLayout` deals with items that vary in width. Its basic algorithm is still to lay out columns from top to bottom and then left to right, but it will now infill empty spaces next to smaller items when larger ones create those gaps. To demonstrate this, let's modify the sample so that the smallest item is 155x80 (half the original size), the medium item is 310x80, and the large item is 310x160. Here are the changes to make that happen:

1. Undo any changes from the previous tests: in `html/scenario4.html`, add `groupInfo` back to `data-win-options`, and in `js/data.js`, change the class in the first item of `myCellSpanningData` back to `smallListItemIconTextItem`.
2. In `js/data.js`, change the `cellWidth` in `groupInfo` to 155 (half of 310) and leave `cellHeight` at 80. For clarity, also insert an incrementing number at the start of each item's text in `myCellSpanningData` array.
3. In `css/scenario4.css`:
 - a. Change the width of `smallListItemIconTextItem` to 145px. Applying the formula, $((145+10) * 1) - 10 = 145$. Height is 70px.
 - b. Change the width of `mediumListItemIconTextItem` to 310px and the height to 70px.
 - c. Change the width of `largeListItemIconTextItem` to 310px and the height to 160px. Applying the formula to the height, $((80+10) * 2) - 10 = 170$ px.
 - d. Set the `width` style in the `#listview` rule to 800px and the `height` to 600px (to make more space in which to see the layout).

I recommend making these changes in Blend where your edits are reflected more immediately than running the app from Visual Studio. In any case, the results are shown in Figure 5-10 where the numbers show us the order in which the items are laid out (and apologies for clipping the text...experiments must make sacrifices at times!). A copy of the sample with these modifications is provided in the companion content for this chapter.



FIGURE 5-10 Modifying the ListView item templates sample to show cell spanning more completely.

In the modified sample I've also included an `itemInfo` function in `js/data.js`, as you may have already noticed. It returns the item dimensions according to the type specified for the item:

```
function itemInfo(index) {
    //getItem(index).data retrieves the array item from a WinJS.Binding.List
    var item = myCellSpanningData.getItem(index).data;
    var width, height;

    switch (item.type) {
        case "smallListIconTextItem":
            width = 145;
            height = 70;
            break;

        case "mediumListIconTextItem":
            width = 310;
            height = 70;
            break;

        case "largeListIconTextItem":
            width = 310;
            height = 160;
            break;
    }

    return {
        newColumn: false,
        itemWidth: width,
        itemHeight: height
    };
}
```

You can set a breakpoint in this function and verify that it's being called for every item; you can also see that it produces the same results. Now change the return value of `newColumn` as follows, to force a column break before item #7 and #15 in Figure 5-10, because they oddly span columns:

```
newColumn: (index == 6 || index == 14), //Break on items 7 and 15 (index is 6 and 14)
```

The result of this change is shown in Figure 5-11.



FIGURE 5-11 Using new columns in cell spanning on items 7 and 15.

One last thing I noticed while playing with this sample is that if the item size in a style rule like `smallListItemIconTextItem` ends up being smaller than the size of a child element, such as `.regularListItemIconTextItem` (which includes margin and padding), the larger size wins in the layout. As you experiment, you might want to remove the default 5px margin that's set for `win-container`. This is what creates the space between the items in Figure 5-10 but has to be added into the equations. The following rule will set that margin to 0px:

```
#listView > .win-horizontal .win-container {
    margin: 0px;
}
```

Optimizing ListView Performance

I've often told people that there's so much you can do and learn about ListView that it could be a book in itself! Indeed, it would have been easy for Microsoft to have just created a basic control that let you create templated items and have left it at that. However, knowing that the ListView would be utterly central to a large number of apps (perhaps the majority outside the gaming category), and expecting that the ListView would be called upon to host thousands or even tens of thousands of items, a highly skilled and passionate group of engineers has gone to great extremes to provide many levels of sophistication that will help your apps perform their best.

One optimization is the ability to demand-load pages of items as determined by the `ListView's loadingBehavior` property, as described in the next two sections. The other optimization is to use template functions to delay-load different parts of each item, such as images, as well as to defer actions like animations until an item actually becomes visible, which is covered in the third section below. In all cases, the whole point of these optimizations is to help the `ListView` display the most important items or parts of items as quickly as possible, deferring the loading and rendering of other items or less important item elements until they're really needed.

I did want to point out that the [Using ListView](#) topic in the documentation contains even more suggestions than I'm able to include here. (I do have other chapters to write!) I encourage you to study that topic as well, and who knows—maybe you'll be the one to write the complete `ListView` book! Furthermore, additional guidance on appwide performance can be found on [Performance best practices for Windows Store apps using JavaScript](#), which contains the `Using ListView` topic.

Random Access

If you're like myself and others in my family, you probably have an ever-increasing stockpile of digital photographs that make you glad that 1TB+ hard drives keep dropping in price. In other words, it's not uncommon for many consumers to have ready access to collections of tens of thousands of items that they will at some point want to pan through in a `ListView`. But just imagine the overhead of trying to load thumbnails for every one of those items into memory to display in a list. On low-level and low-power hardware, you'd probably be causing every suspended app to be quickly terminated, and the result will probably be anything but "fast and fluid"! The user might end up waiting a *really* long time for the control to become interactive and will certainly get tired of watching a progress ring.

With this in mind, the default `loadingBehavior` property for a `ListView` is set to `"randomaccess"`. In this mode, the `ListView's` scrollbar will reflect the total extent of the list so that the user has some idea of its size, but the `ListView` keeps a total of only five pages or screenfuls of items in memory at any given time (with an overall limit of 1000 items). For most pages, this means the visible page (in the viewport) plus two buffer pages ahead and behind. (If you're viewing the first page, the buffer extends four pages ahead; if you're on the last page, the buffer extends four pages behind—you get the idea.)

Whenever the user pans to a location in the list, any pages that fall out of the viewport or buffer zone are discarded (almost—we'll come back to this in a moment), and loading of the new viewport page and its buffer pages begins. Thus the `ListView's loadingState` property will start again at `itemsLoading`, then to `viewportLoaded` when the visible items are rendered, then `itemsLoaded` when the buffered pages are loaded, and then `complete` when everything is done. Again, at any given time, only five pages of items are loaded into memory.

Now when I said that previously loaded items get discarded when they move out of the viewport/buffer range, what actually happens is that the items get *recycled*. One of the most expensive parts of rendering an item is creating its DOM elements, so the `ListView` actually takes those elements, moves them to a new place in the list, and fills them in with new content. This will become important when we look at optimization in template functions.

Incremental Loading

Apart from potentially very large but known collections, other collections are, for all intents and purposes, essentially unbounded, like a news feed that might have millions of items stretching back to the Cenozoic Era (at least by Internet reckoning!). With such collections, you probably won't know just how many items there are at all; the best you can really do is just load another chunk when the user wants them.

This is what the `loadingBehavior` of `"incremental"` is for. In this mode, the `ListView`'s scrollbar will reflect only what's loaded in the list, but if the user passes a particular threshold—for instance, they pan to the end of the list—the `ListView` will ask the data source for more pages of items and add them to the list, updating the scrollbar. In this case, all of the loaded items remain loaded, providing for very quick panning within the loaded list but with potentially more memory consumption than random access.

The incremental loading behavior is demonstrated in Scenarios 2 and 3 of the [ListView loading behaviors sample](#). (Scenario 1 covers random access, but it's nothing different than we've already seen.) Incremental loading activates the following characteristics:

- The `ListView`'s `pagesToLoad` property indicates how many pages or screenfuls of items get loaded at a time. The default value is five.
- The `automaticallyLoadPages` property indicates whether the `ListView` should load new pages automatically as you pan through the list. If `true` (the default), as demonstrated in Scenario 2, as you pan toward the end of the list you'll see the scrollbar change as new pages are loaded. If `false`, as demonstrated in Scenario 3, new pages are not loaded until you call the `loadMorePages` method directly.
- When `automaticallyLoadPages` is `true`, the `pagesToLoadThreshold` property indicates how close the user can get to the current end of the list before new page loads are triggered. The default value is two.
- When new pages start to load (either automatically or in response to `loadMorePages`), the `ListView` will start updating the `loadingState` property firing `loadingstatechanged` events as described already.

Template Functions (Part 2): Promises, Promises!

What we just discussed with the `ListView`'s loading behavior options pertains to the incremental loading of pages. It's helpful now to combine this with incremental loading of *items*. For that, we need to look at what's sometimes referred to as the *rendering pipeline* as implemented in template functions.

When we first looked at template functions earlier (see "How Templates Really Work"), I noted that they give us control over both how and *when* items are constructed and that such functions—again, called renderers—are *the* means through which you can implement five progressive levels of optimization for `ListView` (and `FlipView`, though this is less common). Just using a renderer, as we

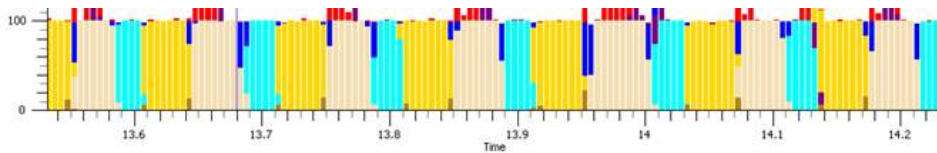
already saw, is level 1; now we're ready to see the other four levels. This is a fascinating subject, because it shows the kind of sophistication that the `ListView` has implemented for us!

Our context for this discussion is the [HTML `ListView` optimizing performance](#) sample that demonstrates all these levels and allows you to see their respective effects. Here's an overview:

- A *simple* or basic renderer allows control over the rendering on a per-item basis.
- A *placeholder* renderer separates creation of the item element into two stages. The first stage returns only those elements that define the shape of the item. This allows the `ListView` to quickly do its overall layout before all the details are filled in, especially when the data is coming from a potentially slow source. When item data is available, the second stage is then invoked to copy that data into the item elements and create additional elements that don't contribute to the shape.
- A *recycling placeholder* renderer adds the ability to reuse an existing chunk of DOM for the item, which is much faster than having to create it all from scratch. For this purpose, the `ListView`, knowing that it will be frequently paged around, holds onto some number of item elements when they go offscreen. In your renderer, you add a code path to clean up a recycled element if one is given to you, and return that as your placeholder. You then populate it with values in the second stage of rendering.
- A *multistage* renderer extends the recycling renderer both to delay-load images and other media until the item element is fully built up in the `ListView` and also to delay any visibility-related actions, such as animations, until the item is actually on-screen.
- Finally, a multistage *batching* renderer adds the ability to add images and other media as a batch, thereby rendering and possibly animating their entrance into the `ListView` as a group such that the system's GPU can be employed more efficiently.

With all of these renderers, you should strive to make them execute as fast as possible. Especially minimize the use of DOM API calls, which includes setting individual properties. Use an `innerHTML` string where you can to create elements rather than discrete calls, and minimize your use of `getElementById`, `querySelector`, and other DOM-traversal calls by caching the elements you refer to most often. This will make a big difference.

To visualize the effect of these improvements, the following graphic shows an example of how unoptimized `ListView` rendering typically happens:



The yellow bars indicate execution of the app's JavaScript—that is, time spent inside the renderer. The beige bars indicate the time spent in DOM layout, and aqua bars indicate actual rendering to the screen. As you can see, when elements are added one by one, there's quite a bit of breakup in what code is executing when, and the kicker here is that most display hardware refreshes only every 10–20 milliseconds (50–100Hz). As a result, there's lots of choppiness in the visual rendering.

After making improvements, the chart can look like the one below, where the app's work is combined in one block, thereby significantly reducing the DOM layout process (the beige):



As for all the other little bits in these graphics, they come from the performance tool called XPerf that's part of the Windows SDK (see sidebar). Without studying the details, what ultimately matters is that we understand the steps you can take to achieve these ends—namely, the different forms of renderers that you can employ as demonstrated in the sample.

Sidebar: XPerf and msWriteProfilerMark

The XPerf tool in the Windows SDK, which is documented on [Windows Performance Analysis Tools](#), can very much help you understand how your app really behaves on a particular system. Among other things, it logs calls you make to `msWriteProfilerMark`, as you'll find sprinkled throughout the WinJS source code itself. For these to show up in xperf, however, you need to start logging with this command:

```
xperf -start user -on PerfTrack+Microsoft-IE:0x1300
```

and end logging with this one, where `<trace_filename>` is any path and filename of your choosing:

```
xperf -stop user -d <trace_filename>.etl
```

Launching the .etl file you save will run the Windows Performance Analyzer and show a graph of events. Right-click the graph, and then click "Summary Table". In that table, expand Microsoft-IE and then look for and expand the Mshtml_DOM_CustomSiteEvent node. The Field3 column should have the text you passed to `msWriteProfilerMark`, and the Time(s) column will help you determine how long actions took.

As a baseline for our discussion, here is a *simple renderer*:

```
function simpleRenderer(itemPromise) {
  return itemPromise.then(function (item) {
    var element = document.createElement("div");
    element.className = "itemTempl";
    element.innerHTML = "<img src='" + item.data.thumbnail +
      "' alt='Databound image' /><div class='content'>" + item.data.title + "</div>";
    return element;
  });
}
```

Again, this structure waits for the item data to become available, and it returns a promise for the element that will be fulfilled at that time.

A *placeholder renderer* separates building the element into two stages. The return value is an object that contains a minimal placeholder in the `element` property and a `renderComplete` promise that does the rest of the work when necessary:

```
function placeholderRenderer(itemPromise) {
  // create a basic template for the item which doesn't depend on the data
  var element = document.createElement("div");
  element.className = "itemTempl";
  element.innerHTML = "<div class='content'>...</div>";

  // return the element as the placeholder, and a callback to update it when data is available
  return {
    element: element,

    // specifies a promise that will be completed when rendering is complete
    // itemPromise will complete when the data is available
    renderComplete: itemPromise.then(function (item) {
      // mutate the element to include the data
      element.querySelector(".content").innerText = item.data.title;
      element.insertAdjacentHTML("afterBegin", "<img src='" +
        item.data.thumbnail + "' alt='Databound image' />");
    })
  };
}
```

The `element` property, in short, defines the item's shape and is returned immediately from the renderer. This lets the `ListView` do its layout, after which it will fulfill the `renderComplete` promise. You can see that `renderComplete` essentially contains the same sort of thing that a simple renderer returns, minus the already created placeholder elements. (For another example, the added Scenario 8 of the `FlipView` example in this chapter's companion content has commented code that implements this approach.)

A *recycling placeholder* renderer now adds awareness of a second parameter called `recycled` that the `ListView` (but not the `FlipView`) can provide to your rendering function when the `ListView`'s `loadingBehavior` is set to `"randomaccess"`. If `recycled` is given, you can just clean out the element, return it as the placeholder, and then fill in the data values within the `renderComplete` promise as before.

If it's not provided (as when the `ListView` is first created or when `loadingBehavior` is `"incremental"`), you'll create the element anew:

```
function recyclingPlaceholderRenderer(itemPromise, recycled) {
  var element, img, label;
  if (!recycled) {
    // create a basic template for the item which doesn't depend on the data
    element = document.createElement("div");
    element.className = "itemTempl";
    element.innerHTML = "<img alt='Databound image' style='visibility:hidden;' />" +
      "<div class='content'>...</div>";
  }
  else {
    // clean up the recycled element so that we can re-use it
    element = recycled;
    label = element.querySelector(".content");
    label.innerHTML = "...";
    img = element.querySelector("img");
    img.style.visibility = "hidden";
  }
  return {
    element: element,
    renderComplete: itemPromise.then(function (item) {
      // mutate the element to include the data
      if (!label) {
        label = element.querySelector(".content");
        img = element.querySelector("img");
      }
      label.innerText = item.data.title;
      img.src = item.data.thumbnail;
      img.style.visibility = "visible";
    })
  };
}
```

In `renderComplete`, be sure to check for the existence of elements that you don't create for a new placeholder, such as `label`, and create them here if needed.

If you'd like to clean out recycled items, you can also provide a function to the `ListView`'s `resetItem` property that would contain the same code as shown above for that case. The same is true for the `resetGroupHeader` property, because you can use template functions for group headers as well as items. We haven't spoken of these as much because group headers are far fewer and don't typically have the same performance implications. Nevertheless, the capability is there.

Next we have the *multistage renderer*, which extends the recycling placeholder renderer to delay-load images and other media until the rest of the item is wholly present in the DOM, and to further delay effects like animations until the item is truly on-screen.

The hooks for this are three methods called `ready`, `loadImage`, and `isOnScreen` that are attached to the item provided by the `itemPromise`. The following code shows how these are used (where `element.querySelector` traverses only a small bit of the DOM, so it's not a concern):

```
renderComplete: itemPromise.then(function (item) {
    // mutate the element to update only the title
    if (!label) { label = element.querySelector(".content"); }
    label.innerText = item.data.title;

    // use the item.ready promise to delay the more expensive work
    return item.ready;
    // use the ability to chain promises, to enable work to be cancelled
}).then(function (item) {
    //use the image loader to queue the loading of the image
    if (!img) { img = element.querySelector("img"); }
    return item.loadImage(item.data.thumbnail, img).then(function () {
        //once loaded check if the item is visible
        return item.isOnScreen();
    });
}).then(function (onscreen) {
    if (!onscreen) {
        //if the item is not visible, then don't animate its opacity
        img.style.opacity = 1;
    } else {
        //if the item is visible then animate the opacity of the image
        WinJS.UI.Animation.fadeIn(img);
    }
})
```

I warned you that there would be promises aplenty in these performance optimizations! But all we have here is the basic structure of chained promises. The first async operation in the renderer updates simple parts of the item element, such as text. It then returns the promise in `item.ready`. When that promise is fulfilled—or, more accurately, *if* that promise is fulfilled—you can use the item's async `loadImage` method to kick off an image download, returning the `item.isOnScreen` promise from that completed handler. When and if that `isOnScreen` promise is fulfilled, you can perform those operations that are relevant only to a visible item.

I emphasize the *if* part of all this because it's very likely that the user will be panning around within the `ListView` while all this is happening. Having all these promises chained together makes it possible for the `ListView` to cancel the async operations any time these items are scrolled out of view and/or off any buffered pages. Suffice it to say that the `ListView` control has gone through a *lot* of performance testing!

Which brings us to the final multistage *batching* renderer, which combines the insertion of images in the DOM to minimize layout and repaint work. In the sample, this uses a function called `createBatch` that utilizes `WinJS.Promise.timeout` method with a 64-millisecond period to combine the image-loading promises of the multistage renderer. Honestly, you'll have to trust me on this one, because you really have to be an expert in promises to understand how it works!

```

//During initialization (outside the renderer)
thumbnailBatch = createBatch();

//Within the renderComplete chain

//...
}).then(function () {
    return item.loadImage(item.data.thumbnail);
}).then(thumbnailBatch()
).then(function (newimg) {
    img = newimg;
    element.insertBefore(img, element.firstChild);
    return item.isOnScreen();
}).then(function (onscreen) {
//...

//The implementation of createBatch

function createBatch(waitPeriod) {
    var batchTimeout = WinJS.Promise.as();
    var batchedItems = [];

    function completeBatch() {
        var callbacks = batchedItems;
        batchedItems = [];

        for (var i = 0; i < callbacks.length; i++) {
            callbacks[i]();
        }
    }

    return function () {
        batchTimeout.cancel();
        batchTimeout = WinJS.Promise.timeout(waitPeriod || 64).then(completeBatch);

        var delayedPromise = new WinJS.Promise(function (c) {
            batchedItems.push(c);
        });
        return function (v) { return delayedPromise.then(function () { return v; }); };
    };
}

```

Did I warn you about there being promises in your future? Well, fortunately, we've now exhausted the subject of template functions, but it's time well spent because optimizing ListView performance, as I said earlier, will greatly improve consumer perception of apps that use this control.

What We've Just Learned

- In-memory collection data is managed through `WinJS.Binding.List`, which integrates nicely with collection controls like `FlipView` and `ListView`. In-memory collections can come from sources like `WinJS.xhr` and data loaded from files.
- The `WinJS.UI.FlipView` control displays one item at a time; `WinJS.UI.ListView` displays multiple items according to a specific layout.
- Central to both controls is the idea that there is a data source and an item template used to render each item in that source. Templates can be either declarative or procedural.
- `ListView` works with the added notion of layout. WinJS provides two built-in layouts. `GridLayout` is a two-dimensional, horizontally panning list; `ListLayout` is for a one-dimensional vertically panning list. It is also possible to implement custom layouts.
- `ListView` provides the capability to display items in groups; `WinJS.BindingList` provides methods to create grouped, sorted, and filtered projections of items from a data source.
- The Semantic Zoom control (`WinJS.UI.SemanticZoom`) provides an interface through which you can switch between two different views of a data source, a zoomed-in (details) view and a zoomed-out (summary) view. The two views can be very different in presentation but should display related data. The `IZoomableView` interface is required on each of the views so that the Semantic Zoom control can switch between them and scroll to the correct item.
- WinJS provides a `StorageDataSource` to create a collection over `StorageFile` items.
- It is possible to implement custom data sources, as shown by samples in the Windows SDK.
- Procedural templates are implemented as template function, or renderers. These functions can implement progressive levels of optimization for delay-loading images and adding items to the DOM in batches.
- Both `FlipView` and `ListView` controls provide a number of options and styling capabilities. `ListView` also provides for item selection and different selection behaviors.
- The `ListView` control provides built-in support for optimizing random access of large data sources, as well as incremental access of effectively unbounded data sources.
- The `ListView` control supports the notion of cell spanning in its `GridLayout` to support items of variable size, which should all be multiples of a basic cell size.

Chapter 6

Layout

Compared to other members of my family, I seem to need the least amount of sleep and am often up late at night or up before dawn. To avoid waking the others, I generally avoid turning on lights and just move about in the darkness (and here in the rural Sierra Nevada foothills, it can get *really* dark!). Yet because I know the layout of the house and the furniture, I don't need to see much. I only need a few reference points like a door frame, a corner on the walls, or the edge of the bed to know exactly where I am. What's more, my body has developed a muscle memory for where doorknobs are located, how many stairs there are, how many steps it takes to get around the bed, and so on. It's really helped me understand how visually impaired people "see" their own world.

If you observe your own movements in your home and your workplace—probably when the spaces are lit!—you'll probably find that you move in fairly regular patterns. This is actually one of the most important considerations in home design: a skilled architect looks carefully at how people in the home might move between primarily spaces like the kitchen, dining room, and living room, and even within a single workspace like the kitchen. Then they design the home's layout so that the most common patterns of movement are easy and free from obstructions. If you've ever lived in a home where it *wasn't* designed this way, you can very much appreciate what I'm talking about!

There are two key points here: first, good layout makes a huge difference in the usability of any space, and second, human beings quickly form habits around how they move about within a space, habits that hopefully make their movement more efficient and productive.

Good app design clearly follows the same principles, which is exactly why Microsoft recommends following consistent patterns with your apps, as described on [Designing UX for apps](#) and [Design guidance for Windows Store apps](#). Those recommendations are not in any way whimsical or haphazard: they are the result of many years of research and investigation into what would really work best for apps and for Windows 8 as a whole. The placement of the charms, for instance, as well as commands on an app bar (as we'll see in Chapter 7, Commanding UI), arise from the reality of human anatomy, namely how far we can move our thumbs around the edges of the screen when holding a tablet device.

With page layout, in particular, the recommendations on [Laying out an app page](#)—about where headers and body content are ideally placed, the spacing between items, and so forth—can seem rather limiting, if not draconian. The silhouette, however, is meant to be a good starting point, not a hard-and-fast rule. What's most important is that the shape of an app's layout helps users develop a visual and physical muscle memory that can be applied across many apps. Research has showed that users will actually develop such habits very quickly, even within a matter of minutes, but of course those habits are not exact to specific pixels! In other words, the silhouette represents a general shape that helps users immediately understand how an app works and where to go for certain functions, just like you can easily recognize the letter "S" in many different fonts. This is very efficient and productive. On

the other hand, when presented with an app that used a completely different layout (or worse, a layout that was similar to the silhouette but behaves differently), users must expend much more energy just figuring out where to look and where to tap, just as I would have to be much more careful late at night if you moved all my furniture around!

The bottom line is that there are very good reasons behind *all* the Windows Store app design recommendations, layout included. As I've said before, if you're fulfilling the designer role for your app, study the guidelines referred to above. If someone else is fulfilling that role, make sure *they* study the guidelines! Either way, we'll be reviewing the key principles in the first section of this chapter.

After that, our focus will be on how we implement layout designs, not creating the designs themselves. (Although I apparently got the mix of my parent's genes that bestowed an aptitude for technical communication, my brother got the most of the genes for artistry!) For example, how does an app respond to view state changes to show the correct page design (for full-screen landscape, filled, snapped, and portrait)? How does the app handle varying display sizes and varying pixel densities?

We'll also spend a little time with the CSS grid and a few other CSS layout features like flexbox and multicolumn text. Generally speaking, these are all CSS standards, so I expect that you already know quite a bit about them or can find full documentation elsewhere.³⁴ We'll thus cover the basics only briefly, spending more time understanding how these features are best applied within an app and those aspects that are unique to the Windows 8 environment (such as what are called *snap points* on a pannable/scrollable `div`).

I'll remind you again that there are other UI elements like the app bar and flyouts that don't participate in layout; I'll cover these in other chapters. There are also auxiliary app pages that service contracts (such as Search and Settings) and exist outside your main navigation flow. These will employ the same layout principles covered in this chapter, but how and when they appear will be covered later.

Principles of Windows Store App Layout

Layout is truly one of the most important considerations in Windows app design. The principle of "content before chrome" means that most of what you display on any given app page is content, with little in the way of commanding surfaces, persistent navigation tabs, and passive graphical elements like separators, blurs, and gradients that don't in themselves mean anything. Another way of putting this is that content itself should be directly interactive rather than composed of passive elements that are acted upon when the user invokes some other command. Semantic zoom is a good example of such interactive content—instead of needing buttons or menus elsewhere in the app to switch between views, the capability is inherent in the control itself, with the small zoom button appearing only when needed for mouse users. Other app commands, for the most part, are similarly placed on UI surfaces

³⁴ The specifications can be found on <http://www.w3c.org>; specifically start with <http://www.w3.org/standards/webdesign/htmlcss> for both. I also highly recommend the well-designed and curated resources from [Smashing Magazine](#) for learning the nuances of CSS, which I must admit still seems mysterious to me at times.

that appear when needed through app bars and other flyouts, as we'll see in Chapter 7.

In short, “content before chrome” means immersing the user in the experience of the content rather than distracting them with nonessentials. In Windows app design, then, emphasis is given to the *space* around and between content, which serves to organize and group content without the need for lines and boxes. These essentially transparent “space frames” help consumer’s eyes focus on the content that really matters. Windows app design also uses typography (font size, weight, color, etc.) to convey a sense of structure, hierarchy, and relative importance of different content. That is, because the text on a page is already content, why not use its characteristics—the typography—to communicate what is often done with extraneous chrome? (As with the layout silhouette, the general use of the Segoe UI font within app design is not a hard-and-fast requirement, but a starting point. Having a consistent *type ramp* for different headings is more important than the font.)

As an example, Figure 6-1 shows a typical desktop or web application design for an RSS reader. Notice the persistent chrome along the top and bottom: search commands, navigation tabs, navigation controls, and so forth. This takes up perhaps 20% of the screen space. In what remains, nearly two-thirds is taken up by organizational elements, leaving 20–25% of the screen space for the content we actually care about: the article.

Figure 6-2 shows a Windows Store app design for the same content. Notice how all the ancillary commands have been moved offscreen. Search would be accomplished through the Search charm; Settings through the Settings charm; adding feeds, refresh, and navigation through commands on to the app bar; and switching views through semantic zoom. Typography is used to convey the hierarchy instead of a folder control, which then leaves the bulk of the display—nearly 75%—for the content. As a result, we can see much more of that content than before, which creates a much more immersive and engaging experience, don’t you think?

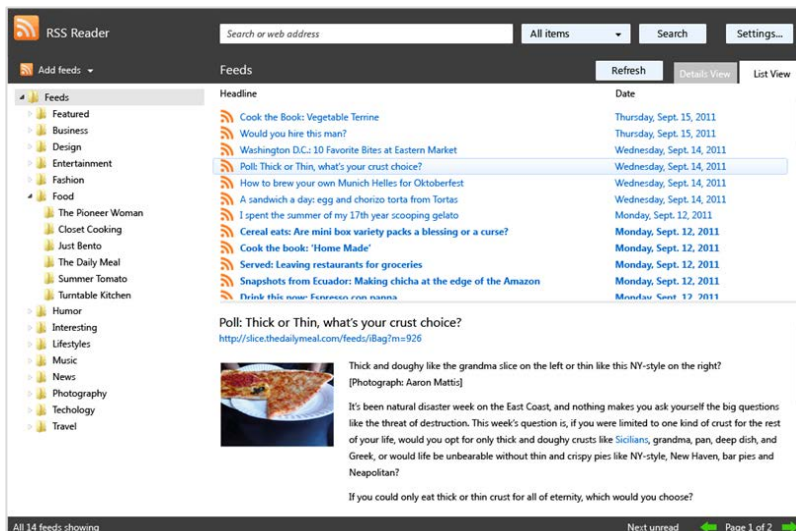


FIGURE 6-1 A typical desktop or web application design that emphasizes chrome at the expense of content.

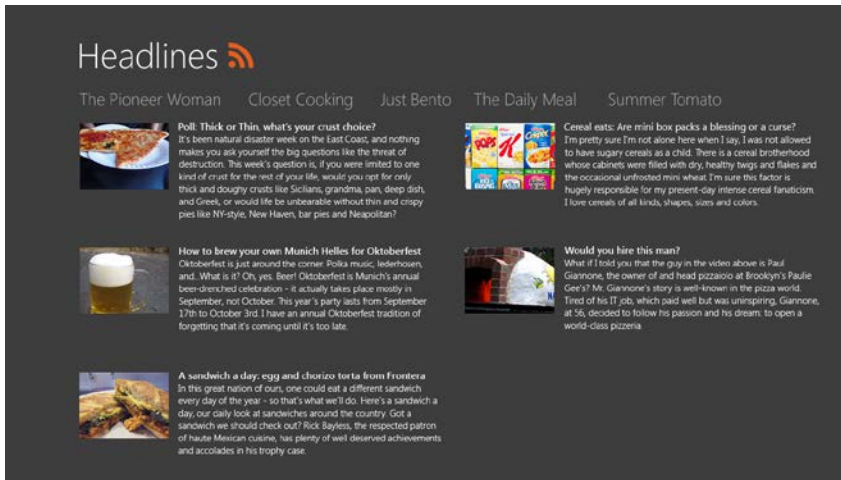


FIGURE 6-2 The same app as Figure 6-1 reimaged with one possible application of Windows app design, where most of the chrome has disappeared, leaving much more space for content. An alternate design could emphasize images much more than text.

Even where typography is concerned, Windows app design encourages the use of distinct font sizes, again called the typographic ramp, to establish a sense of hierarchy. The default WinJS stylesheets—`ui-light.css` and `ui-dark.css`—provide four fixed sizes where each level is proportionally larger than the previous (42pt = 80px, 20pt = 40px, etc.), as shown in Figure 6-3. These proportions allow users to easily establish an understanding of content structure with just a glance. Again, it's a matter of encouraging habit and muscle memory, and Microsoft's research has shown that beyond this size granularity, users are generally unable to differentiate where a piece of content fits in a hierarchy.

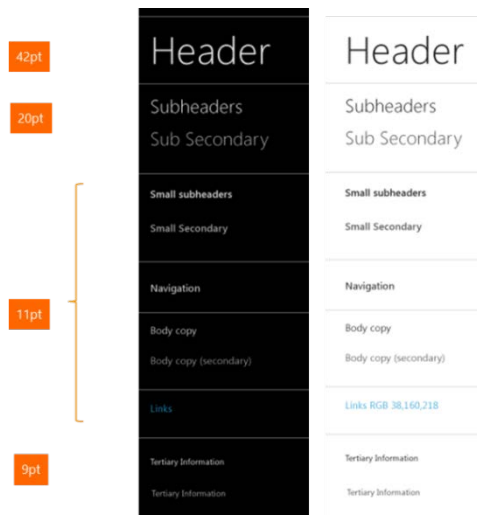


Figure 6-3 The typographic ramp of Windows Store app design, shown in both the `ui-dark.css` (left) and `ui-light.css` (right) stylesheets.

Within the body of content, then, Windows app design encourages these layout principles:

- Let content flow from edge to edge.
- Keep ergonomics in mind: pan along the long edge of the view (primarily horizontal in landscape views, vertical in snapped view and possibly portrait).
- Pan on a single axis only to create a sense of stability and to support swiping to select (as with the ListView controls), or employ rails to limit panning directions to a single axis.
- Create visual alignment, structure, and clarity with the Windows 8 silhouette, aligning elements on a grid for consistency. Refer again to [Laying out an app page](#). This shape is what allows a consumer's eyes to recognize something as a Store app without having to think about it, which provides a feeling of familiarity and confidence.

As I've mentioned before, the project templates in Visual Studio and Blend have these principles baked right in and thus provide a convenient starting point for apps. Even if you start with the Blank App template, the others like the Grid App will serve as a reference point. This is exactly what we did with the Here My Am! app in Chapter 2, "Quickstart."

The other important guiding principle that's relevant to layout is "snap and scale beautifully." This means making sure you design every page in your app to handle all four view states and to be appropriately adaptive across different display resolutions and pixel densities. We'll look at this subject in the "View States and the Many Faces of Your Display" section below. First, however, let's look at a little piece of core layout code.

Quickstart: Pannable Sections and Snap Points

In Chapter 5, "Collections and Collection Controls," we spent a little time looking at when a ListView control was the right choice and when it wasn't. One of the primary cases where developers have inappropriately attempted to use a ListView is to implement a home or hub page that contains a variety of distinct content groups arranged in columns, as shown in Figure 6-4 and explained on [Navigation design for Windows Store apps](#). At first glance this might look like a ListView, but because the data it's representing really isn't a collection, just a layout of fixed content, it makes sense to use tried-and-true HTML and CSS for the job!

I point this out because with all the great controls that WinJS provides, it's easy to forget that everything you know about HTML and CSS still applies in Store apps. After all, those controls are in themselves just blocks of HTML and CSS with some additional methods, properties, and events.

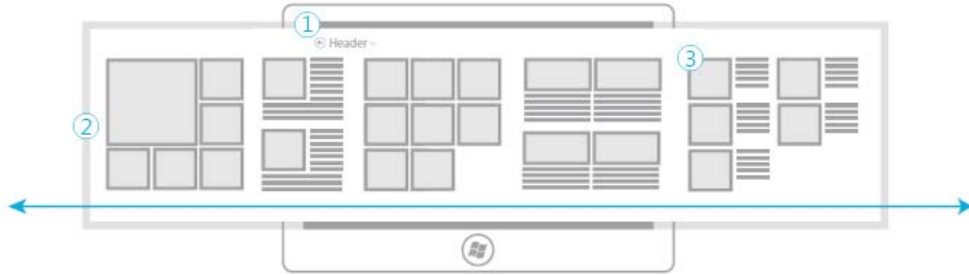


FIGURE 6-4 The layout of a typical home or hub page of a Store app with a fixed header (1), a horizontally pannable section (2), and content sections or categories (3).

Laying Out the Hub

Let's see how we'd use straight HTML and CSS to implement the pannable section of the hub page in Figure 6-4. Referring first to [Laying out an app page](#), we know that the padding between groups should be four units of 20px each, or 80px. Most of the groups themselves should be square, except for the second one which is only half the width. On a baseline 1366x768 display, the height of each section would be 768px minus 128px (for the header) minus the minimum 50px on the bottom, which leaves 590px (if we added group headings for each section, we'd subtract another 40px). So a square group on the baseline display would be 590px wide (we'd set the actual height to 100% of its containing grid cell). The total width of the section will then be (590 * 4 full-size sections) + (295 * 1 half-width section) + (80 * 4 for the separator gaps). This equals 2975px. To this we'll add border columns of 120px on the left (according to the silhouette) and 80px on the right, for a total of 3175px.

To create a section with exactly this layout, we can use a CSS grid within a block element. To demonstrate this, run Blend and create a new project with the Navigation App template (so we just get a basic page with the silhouette and not all the secondary pages). Within the `section` element of `pages/home/home.html`, create another `div` element and give it a class of `hubSections`:

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

In `pages/home/home.css`, add a few style rules. Give `overflow-x: auto` to the `section` element, and lay out the grid in the `hubSections` `div`, using added columns on the left and right for spacing (removing the `margin-left: 120px` from the `section` and adding it as the first column in the `div`):

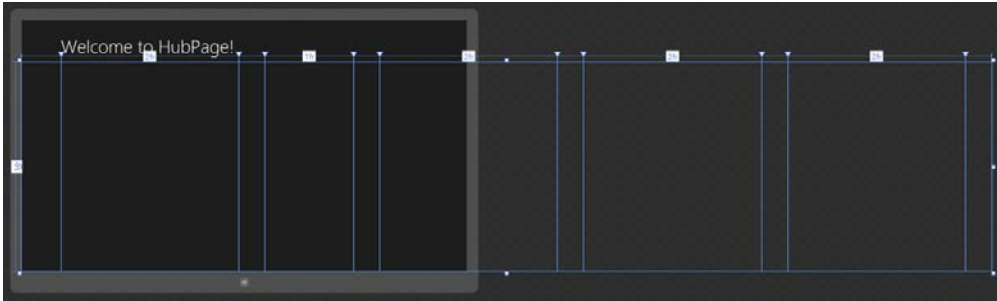
```
.homepage section[role=main] {
  overflow-x: auto;
}
.homepage .hubSections {
  width: 2975px;
  height: 100%;
  display: -ms-grid;
  -ms-grid-rows: 1fr 50px;
```

```

-ms-grid-columns: 120px 2fr 80px 1fr 80px 2fr 80px 2fr 80px 2fr 80px;
}

```

With just these styles we can already see the hub page taking shape in Blend by zooming out in the artboard:



Now let's create the individual sections, each one starting as a `div` that we add in `pages/home/home.html`:

```

<section aria-label="Main content" role="main">
  <div class="hubSections">
    <div class="hubSection1"></div>
    <div class="hubSection2"></div>
    <div class="hubSection3"></div>
    <div class="hubSection4"></div>
    <div class="hubSection5"></div>
  </div>
</section>

```

and style them into their appropriate grid cells with 100% `width` and `height`. I'm showing `hubSection1` here as the others are the same with just a different column number (4, 6, 8, and 10, respectively):

```

.homepage .hubSection1 {
  -ms-grid-row: 1;
  -ms-grid-column: 2; /* 4 for hubSection2, 6 for hubSection3, etc. */
  width: 100%;
  height: 100%;
}

```

All of this is implemented in the HubPage example included with this chapter.

Laying Out the Sections

Now we can look at the contents of each section. Depending on what you want to display and how you want those sections to interact, you can again just use layout (CSS grids or perhaps flexbox) or use a control like `ListView`. `hubSection3` and `hubSection5` have gaps at the end, so they might be `ListView` controls with variable items. Note that if we created lists with more than 9 or 6 items, respectively, we'd want to adjust the column size in the overall grid and make the `section` element width larger, but let's assume the design calls for a maximum of 9 and 6 items in those sections.

Let's also say that we want each section to be interactive, where tapping an item would navigate to a details page. (Not shown in this example are group headers to navigate to a group page.) We'll just then use a `ListView` in each, where each `ListView` has a separate data source. For *hubSection1* we'll need to use cell spanning, but the rest of the groups can just use declarative templates. The key consideration with all of these is to style the items so that they fit nicely into the basic dimensions we're using. And referring again back to the silhouette, the spacing between image items should be 10px and the spacing between columns of mixed content (*hubSection4* and *hubSection5*) should be 40px (which can be set with appropriate CSS margins).

Hint If you need to make certain areas of your content unselectable, use the `-ms-user-select` attribute in CSS for a div element. Refer to the [Unselectable content areas with -ms-user-select CSS attribute sample](#). How's that for a name?

Snap Points

If you run the `HubPage` example and pan around a bit using inertial touch gestures (that is, those that continue panning after you've released your finger, explained more in Chapter 9, "Input and Sensors"), you'll notice that panning can stop in any position along the way. You or your designers might like this, but it also makes sense in many scenarios to automatically stop on a section or group boundary. This can be accomplished for touch interactions using CSS styles for *snap points* as described in the following table. These are styles that you add to a pannable element alongside overflow styles, otherwise they have no effect. Documentation for these (and some others) can be found on the CSS reference for [Touch: Zooming and Panning](#).

Style	Description	Value Syntax
<code>-ms-scroll-snap-points-x</code>	Defines snap points along the x-axis	<code>snapInterval(start<length>, step<length>)</code> <code>snapList(list<lengths>)</code>
<code>-ms-scroll-snap-points-y</code>	Defines snap points along the y-axis	<code>snapInterval(start<length>, step<length>)</code> <code>snapList(list<lengths>)</code>
<code>-ms-scroll-snap-type</code>	Defines what type of snap points should be used for the element: <code>none</code> turns off snap points, <code>mandatory</code> always adjusts panning to land on a snap-point (which includes ending inertial panning), and <code>proximity</code> changes the panning only if a panning motion naturally ends "close enough" to a snap point. Using <code>mandatory</code> , then, will enforce a one-section/item-at-a-time panning behavior, whereas <code>proximity</code> would pan past interim snap points if enough inertia is applied. Note also that dragging with a finger (not using an inertia gesture) will allow the user to pan directly past snap points.	<code>none</code> <code>proximity</code> <code>mandatory</code>

<code>-ms-scroll-snap-x</code>	Shorthand to combine <code>-ms-scroll-snap-type</code> and <code>-ms-scroll-snap-points-x</code>	<code><-ms-scroll-snap-type></code> <code><-ms-scroll-snap-points-x></code>
<code>-ms-scroll-snap-y</code>	Shorthand to combine <code>-ms-scroll-snap-type</code> and <code>-ms-scroll-snap-points-y</code>	<code><-ms-scroll-snap-type></code> <code><-ms-scroll-snap-points-y></code>

In the table, `<length>` is a floating-point number, followed by an absolute units designator (`cm`, `mm`, `in`, `pt`, or `pc`) or a relative units designator (`em`, `ex`, or `px`).

To add snap points for each of our hub sections, then, we only need to add two snap points styles after `overflow-x`:

```
.homepage section[role=main] {
    overflow-x: auto;
    -ms-scroll-snap-type: mandatory;
    -ms-scroll-snap-points-x: snapList(0px, 670px, 1045px, 1715px, 1795px);
}
```

Note that the snap points indicated here include the 120px left border so that each one aligns the section underneath the header text. The 0px point thus snaps to the first section, 670px to the second (80px separator plus 590px width of the first section), and so on. The last snap point of 1795px, however, doesn't follow this rule because the `div` can't pan any further past that point. This means we'll snap partway into the next-to-last section, but bring the last section and its 80px right border into view.

With these changes you'll now find that panning around stops nicely (with animations) on the section boundaries. Do note that for a hub page like this, proximity snapping is usually more appropriate. Mandatory snap points are intended more for items that can't be interacted with or consumed without seeing their entirety, such as flipping between pictures, articles, and so on. (The `FlipView` control uses these.)

For more on this topic, including some of the other `-ms-scroll-*` and `-ms-content-zoom-*` styles, such as scroll rails, refer to the [HTML scrolling, panning, and zooming sample](#). Do note also that snap points are not presently supported on the `ListView` control, as they are intended for use with your own layout.

Also be clear that snap points are a touch-only feature; if you want to provide the same kind of behavior with mouse and/or keyboard input, you'll need to do such work manually along the lines of how the `FlipView` control handles transition between items.

The Many Faces of Your Display


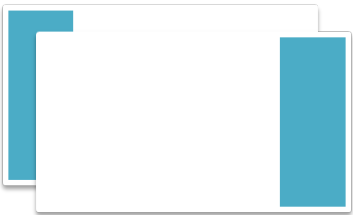
If there's one certainty about layout for a Windows Store app, it's that its display space will likely change over the lifetime of an app and change frequently. For one, auto-rotation—especially on tablet and slate devices—makes it very quick and simple to switch between landscape and portrait orientations



(unlike having to configure a display driver). Second, a device may be connected to an external display, meaning that apps need to adjust themselves to different resolutions on the fly and possibly also different pixel densities. Third, users have the ability in landscape mode to “snap” apps to the left or right side of the screen, where the snapped app is shown in a 320px wide area and another in the “filled” area that occupies the remainder of the display. This is accomplished using touch or mouse gestures, or using the Windows+. (period) and Windows+ > (shift+period) keystrokes. (Snapped view requires a display that’s at least 1366x768; otherwise it’s disabled.)

You definitely want to test your app with all of these variances: view states, display sizes, and pixel densities. View states can be tested directly on any given machine, but for the latter two, the Visual Studio simulator and the Device tab of Blend let you simulate different conditions. Our question now is how an app handles them.

View States

We already got an introduction to the four view states in Chapter 1, “The Life Story of a Windows Store app” (see Figure 1-6). Let’s now add the next level of precision as described in the following table, which includes an image of the space occupied by the app, a description of the view state, and the identifiers for that state as found in both WinRT (in the [Windows.UI.ViewManagement.-ApplicationViewState](#) enumeration) and the [-ms-view-state](#) media feature for CSS media queries:

Space Occupied by the App (Blue)	Details
	<p>App is in landscape mode occupying the entire screen.</p> <p>WinRT: <code>FullScreenLandscape</code></p> <p>-ms-view-state: <code>fullscreen-landscape</code></p>
	<p>App is occupying either left or right side of a landscape screen, in an area that is always 320 pixels wide. This means you do <i>not</i> need to design for all possible sizes between snapped, filled (see below), and full-screen states.</p> <p>WinRT: <code>snapped</code></p> <p>-ms-view-state: <code>snapped</code></p>

	<p>WinRT: <code>filled</code></p> <p>-ms-view-state: <code>filled</code></p> <p>App is occupying the area of the screen next to a snapped app. The width will be the screen size minus 320px minus 22px for the splitter.</p>
	<p>WinRT: <code>fullScreenPortrait</code></p> <p>-ms-view-state: <code>fullscreen-portrait</code></p> <p>App is in portrait mode</p>

Remember again that *every page of your app needs to be prepared for all four view states* (with some exceptions as described in the sidebar below, “Preferred Orientation and Locking Orientation”). View states are always under the user’s control, so any page can be placed into any view state at any time, even on startup. Repeat this like a mantra, because many designers and developers forget this fact!

Note It’s possible that your app might be launched directly into snapped view, as through a user gesture that pulls the app from the left edge of the screen to a snapped state. So be prepared for this possibility. Remember also that any extended splash screen in your app is a page that is also subject to view states. In fact, it’s highly likely that a user will snap an app that’s taking a while to load! At the same time, you cannot programmatically control your app’s view state on activation, so it never needs to be saved or restored as part of session state.

An app’s design should thus include all view states for each page, just like we did with the Here My Am! wireframes in Chapter 2. At the same time, handling view states for every page this does *not* mean four distinct *implementations* of the app. View states are just that: they are different views of the same page content as described on [Guidelines for snapped and fill views](#). That is, switching between view states *always* maintains the state of the app and the page—it *never* changes modes or navigates to another page. The only exception to this rule is that if an app can’t reasonably operate in snap state (like a game that needs a certain amount of screen space to be playable), it can display a message to that effect along with instructions to “Tap here to resume,” which reflects the user’s goal in such a gesture. In response to such a tap, the app can call [Windows.UI.ViewManagement.ApplicationView.tryUnsnap](#),

as demonstrated in the [Snap sample](#).³⁵ Don't use this as an excuse to cut corners, however; try as much as possible to keep the app functional in the snapped state.

Hint Think of the snapped view of a page as a kind of *heads-up view* in which the most essential information from a page is really highlighted. In other words, see snapped view as an opportunity rather than a burden.

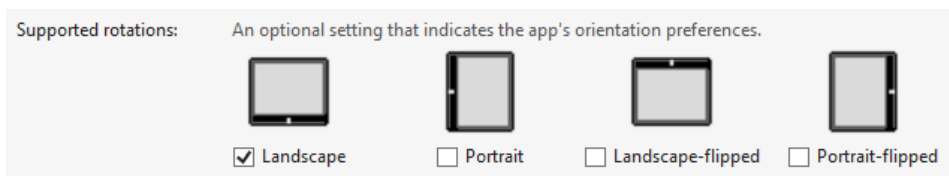
On the flip side, some apps should think about what to do with extra vertical space. A widescreen video in the snapped state will occupy only a small portion of that space, leaving room for, say, additional information about the video, recommendations, playlists, and so on, that wouldn't normally be available when running full screen. In this way, users will find added value in switching to the snapped state.

Sidebar: Preferred Orientation and Locking Orientation

View states aside, it's appropriate for some apps to start in a specific orientation and/or to lock the orientation, effectively ignoring portrait/landscape changes. A movie player, for instance, will generally want to stay in landscape mode, meaning that the fullscreen-landscape and fullscreen-portrait modes are identical—then you can watch videos while laying sideways with a tablet propped up on a chair.

To be clear, the app must still honor the three landscape view states: fullscreen-landscape, filled, and snapped. Preferred orientation is specifically about portrait vs. landscape, and this affects the orientation of your splash screen and other pages in your app. It also enables automatic orientation switching when you switch between your app and others that don't have the same preference.

To tell Windows about your preferences, check the appropriate Supported Orientation boxes in the Application UI tab of the manifest designer:



The many details about how all this works are found on the [InitialRotationPreference page](#) in the documentation. It will also tell you about the [Windows.Graphics.Display.-DisplayProperties.autoRotationPreferences](#) and [currentOrientation](#) properties to programmatically control orientation behaviors. For demonstrations, refer to the [Device auto rotation preferences sample](#).

³⁵ [tryUnsnap](#) is the only programmatic API that can affect view states. View states are otherwise always user-initiated, and there are no APIs to set a view state and no way to specify a view state on startup.

Handling View States

As I just mentioned, handling the different view states doesn't mean changing the mode of an app nor reimplementing a page. Generally, you should try to have feature parity across the states, but in cases like snapped view, especially, the reduced screen real estate will necessitate simplifying the content.

It's best to think about view states simply in terms of the visibility of elements, the size of elements, and their layout on the page. In this way, most of what you need to do can be achieved through CSS media queries using the `-ms-view-state` feature. We saw this again in the Here My Am! app of Chapter 2. The Grid App project template also demonstrates this. Here's how those media queries appear in CSS:

```
@media screen and (-ms-view-state: fullscreen-landscape) {  
    /* ... */  
}  
  
@media screen and (-ms-view-state: filled) {  
    /* ... */  
}  
  
@media screen and (-ms-view-state: snapped) {  
    /* ... */  
}  
  
@media screen and (-ms-view-state: fullscreen-portrait) {  
    /* ... */  
}  
  
/* Syntax for combining media queries (comma-separated) */  
@media screen and (-ms-view-state: fullscreen-landscape),  
screen and (-ms-view-state: fullscreen-portrait), screen and (-ms-view-state: filled) {  
    /* ... */  
}
```

It's also perfectly reasonable to add other clauses to these queries, such as `and (min-width: "1600px")`, as you might be making various other adjustments based on screen sizes.

For Store apps, use the view state features in media queries instead of the CSS `orientation` states (landscape and portrait), which are simply derived from the relative width and height of the display and don't distinguish states like `snapped`. In other words, the Windows view states are more specific to the platform and reflect states that the standard CSS does not, helping your app understand not only its available real estate but also the mode in which it's running.³⁶

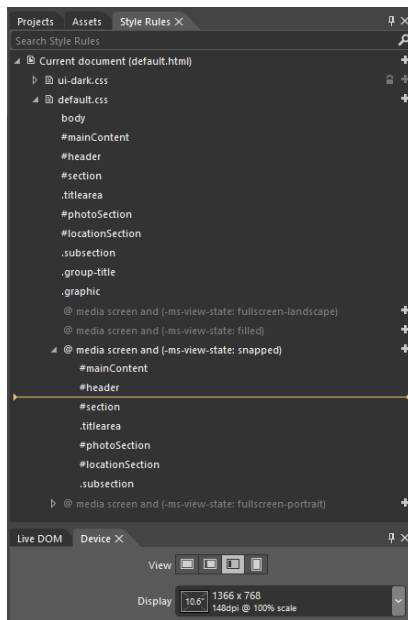
For example, according to the standard CSS algorithm, both the `fullscreen-portrait` and `snapped` states will appear as `orientation: portrait` because the aspect ratio is more vertical than horizontal. However, snapped view implies a different user intent than `fullscreen-portrait`: in snapped view you want to show the most essential parts of an app rather than trying to replicate your portrait layout in a

³⁶ That said, view states are *not* reported to pages loaded into a web context `iframe`. Such pages can use the standard CSS media queries to infer the view state, or the surrounding local context page can pass the view state to the `iframe` through `postMessage`.

320-pixels-wide space.

The general practice is to place all your full-screen landscape rules at the top of your CSS file and then make specific adjustments within the specific media queries. We did this with Here My Am! in Chapter 2, where the default styles worked for `fullscreen-landscape` and `filled` as-is, so we needed specific rules only for `snapped` and `fullscreen-viewport`.

Tip When styling your app in Blend, there’s a visual affordance in the Style Rules pane that lets you control the exact insertion point of any new CSS styles in the given stylesheet. With this—the orange line shown in the graphic below and shown in Video 2-2 of the companion content—you can indicate where to insert styles for specific media queries and within that media query:



In a few cases, handling media queries in declarative CSS alone won’t be sufficient. When the primary content display on a page is a horizontally panning `ListView` with `GridLayout`, you typically switch that control over to `ListLayout` in snapped view. You might also, as suggested on [Guidelines for snapped and fill views](#), change a list of buttons to a single drop-down `select` element to offer the same functionality through a more compact UI. Such things require a little bit of JavaScript.

For these purposes you can employ the standard Media Query Listener API in JavaScript. This interface (part of the W3C CSSOM View Module, see <http://dev.w3.org/csswg/cssom-view/>) allows you to add handlers for media query state changes. To listen for the snapped state, for instance, you can use code like this:

```

var mql = window.matchMedia("(-ms-view-state: snapped)");
mql.addListener(styleForSnapped);

function styleForSnapped() {
    if (mql.matches) {
        //...
    }
}

// Set up listeners for other view states: full-screen, fill, and device-portrait
// or send all media queries to the same handler and check the current state therein.

```

You can see that the media query strings you pass to `window.matchMedia` are the same as used in CSS directly, and in the handler you can, of course, perform whatever actions you need from JavaScript.

Tip Be sure to test your view states on the `resuming` event, as display characteristics might have changed, such as plugging in a different monitor or going to the Settings charm > Change PC Settings > Ease of Access and toggling Make Everything on the Screen Bigger. That is, it's possible to bring your app from the background (suspended state) directly into snapped view, and screen dimensions might also have changed while you're suspended. So test your layout when `resuming` into snapped view and when resuming into different screen dimensions.

When handling view states (or `window.onresize` events), you can obtain exact dimensions of your app window through the `window.innerWidth` and `window.innerHeight` properties. The `document.body.clientWidth` and `document.body.clientHeight` properties will be the same, as will be the `clientWidth` and `clientHeight` properties of any element (like a `div`) that occupies 100% of the document body. Within the `resize` event, the `args.view.outerWidth` and `args.view.outerHeight` properties are also available.

In CSS there are also variables for the viewport height and viewport width: `vh` and `vw`. You can prefix these with a percentage number, such that 100vh is 100% of the viewport height, and 3.5vw is 3.5% of the viewport width. These variables can also be used in CSS `calc` expressions.

The current view state is available through the `Windows.UI.ViewManagement.ApplicationView.value` property. This value comes from the `Windows.UI.ViewManagement.ApplicationViewState` enumeration as shown in the earlier table. We've seen a few uses of this in earlier chapters. For instance, page controls (discussed in Chapter 3, "App Anatomy and Page Navigation") typically check the view state within their `ready` method and directly receive those states within their `updateLayout` method. In fact, every method of the `groupedItems` page control in the Grid App project template is sensitive to the view state. Take a look at the code in `pages/groupedItems/groupedItems.js`:

```

// A few lines and comments are omitted
var appView = Windows.UI.ViewManagement.ApplicationView;
var appViewState = Windows.UI.ViewManagement.ApplicationViewState;
var nav = WinJS.Navigation;
var ui = WinJS.UI;

ui.Pages.define("/pages/groupedItems/groupedItems.html", {

```

```

initializeLayout: function (listView, viewState) {
    if (viewState === appViewState.snapped) {
        listView.itemDataSource = Data.groups.dataSource;
        listView.groupDataSource = null;
        listView.layout = new ui.ListLayout();
    } else {
        listView.itemDataSource = Data.items.dataSource;
        listView.groupDataSource = Data.groups.dataSource;
        listView.layout = new ui.GridLayout({ groupHeaderPosition: "top" });
    }
},

itemInvoked: function (args) {
    if (appView.value === appViewState.snapped) {
        // If the page is snapped, the user invoked a group.
        var group = Data.groups.getAt(args.detail.itemIndex);
        nav.navigate("/pages/groupDetail/groupDetail.html", { groupKey: group.key });
    } else {
        // If the page is not snapped, the user invoked an item.
        var item = Data.items.getAt(args.detail.itemIndex);
        nav.navigate("/pages/itemDetail/itemDetail.html",
            { item: Data.getItemReference(item) });
    }
},

ready: function (element, options) {
    // ...
    this.initializeLayout(listView, appView.value);
    // ...
},

// This function updates the page layout in response to viewState changes.
updateLayout: function (element, viewState, lastViewState) {
    var listView = element.querySelector(".groupeditemslist").winControl;
    if (lastViewState !== viewState) {
        if (lastViewState === appViewState.snapped ||
            viewState === appViewState.snapped) {
            var handler = function (e) {
                listView.removeEventListener("contentanimating", handler, false);
                e.preventDefault();
            }
            listView.addEventListener("contentanimating", handler, false);
            this.initializeLayout(listView, viewState);
        }
    }
}
});

```

First, the `initializeLayout` method that's called from both `ready` and `updateLayout` checks the current view state and adjusts the `ListView` control accordingly. If you remember from Chapter 5, it's perfectly allowable to change a `ListView`'s layout and data source properties on the fly; here we use a `ListLayout` with a list of groups for snapped view and a `GridLayout` with grouped items in all others. This demonstrates how we're showing the same content but in a more concise manner by hiding the

individual items in snapped view. Because of this, `itemInvoked` also has to check the view state because the list items are groups in snapped view and should navigate to a group details page instead of an item page.

As for `updateLayout`, this is invoked from a `window.onresize` event handler in the `PageControlNavigator` code (see `js/navigator.js` in the Grid App project template). That handler passes the new and previous view states to `updateLayout`. If that function detects that we're switching to or from snapped state, it resets the `ListView` through `initializeLayout`. And because we're changing the `ListView`'s data source, there's no need to play entrance or transition animations. The little trick that's played with the `contentanimating` event here simply suppresses those.

Sidebar: Physical Display Orientations

The fullscreen-landscape and fullscreen-portrait view states suggest something of how a device is actually oriented in physical space, but such information is more accurately derived from properties of the `Windows.Graphics.Display.DisplayProperties` object. Specifically, the `currentOrientation` property contains a value from `Windows.Graphics.Display.DisplayOrientations` that indicates how the device is rotated in relation to its `nativeOrientation` (and an `orientationchanged` event fires when needed). This can tell you, for example, whether the device is being held upside-down against the sky, which would be useful for any kind of augmented reality app such as a star chart.

Similarly, the APIs in `Windows.Devices.Sensors`, specifically the `SimpleOrientationSensor` and `OrientationSensor` classes can provide more information from the hardware itself. These are covered in Chapter 9.

Screen Size, Pixel Density, and Scaling

I don't know about you, but when I first read that the snapped area was *always* 320 pixels—real pixels, not a percentage of the screen width—it really set me wondering. Wouldn't that give a significantly different user experience on different monitors? The answer is actually no. 320 pixels is about 25% of the baseline 1366x768 target display, which means that the remaining 75% of the screen is a familiar 1024x768. And on a 10-inch screen, it means that snap area is about the 2.5 physical inches wide. So far so good.

With a large monitor, on the other hand, let's say a 2560x1440 monster, those 320 pixels would only be 12.5% of the width, so the layout of the whole screen looks quite different. However, given that such monitors are in the 24-inch range, those 320 pixels still end up being about 2.5 physical inches wide, meaning that the snapped area gives essentially the same visual experience as before, just now with much more vertical space to play with and much more remaining screen space.

This now brings up the question of *pixel density*—what happens if your app ends up on a really small screen that also has a really high resolution? Obviously, 320 pixels on the latter display would be little more than an inch wide. Anyone got a magnifying glass?

Fortunately, this isn't anything a Store app has to worry about...almost. The main user benefit for such displays is greater sharpness, not greater density of information. Touch targets need to be the same size on any size display no matter how many pixels it occupies, because human fingers don't change with technology! To accommodate this, Windows automatically scales down the effective resolution that's reported to apps, which is to say that whatever coordinates you use within your app (in HTML, CSS, and JavaScript) are automatically scaled up to the necessary device resolution when the UI is actually rendered. This happens at within the low-level HTML/CSS rendering engine in the app host so that everything is drawn directly against native device pixels for maximum sharpness.

As for the "almost" above, the one place where you do need to care about pixel density is with raster graphics, as we discussed in Chapter 3 for your splash screen and tiles. We'll return to this shortly in the "Graphics that Scale Well" section below.

Display sizes and pixel densities can both be tested again using the Visual Studio simulator or the Device tab in Blend. The latter, shown in Figure 6-5, indicates the applicable DPI and scaling factor. 100% scale means the device resolution is reported directly to an app. 140% and 180%, on the other hand, indicate that scaling is taking place. With the 10.6" 2560x1440 setting with 180%, for example, the app will see dimensions of 1422x800 (2560/1.8 by 1440/1.8), which is very close to the standard 1366x768 display; similarly, the 10.6: 1920x1080 setting with 140% scaling will appear to the app as 1371x771 (1920/1.4 by 1080/1.4). In both cases, a layout designed for 1366x768 is completely sufficient though you can certainly be as precise as you want.

Tip If you have an app with a *fixed layout* (see "Fixed Layouts and the ViewBox Control" later on), you can address pixel density issues by simply using graphical assets that are scaled to 200% of your standard design. This is because a fixed layout can be scaled to arbitrary dimensions, so a 200% image scales well in all cases. Such an app does not need to provide 100%, 140%, and 180% variants of its images.

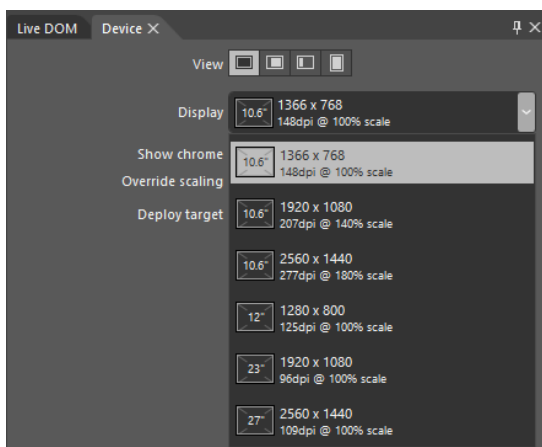


FIGURE 6-5 Options for display sizes and pixel densities in Blend’s Device tab.

As noted earlier with view states, you can programmatically determine the exact size of your app window through the `window.innerWidth` and `window.innerHeight` properties, the `document.body.clientWidth` and `document.body.clientHeight` properties, and the `clientWidth` and `clientHeight` properties of any element that occupies 100% of the body. Within `window.onresize`, you can use these (or the `args.view.outerWidth` and `args.view.outerHeight` properties) to adjust the app’s layout for changes in the overall display. Of course, if you’re using something like the CSS grid with fractional rows and columns to do your layout, much of that will be handled automatically.

In all cases, these dimensions will already reflect automatic scaling for pixel densities, so they are the dimensions against which you want to determine layout. If you want to know the physical *display* dimensions, on the other hand, you’ll find these in the `window.screen.width` and `window.screen.height` properties. Other aspects of the display can be found in the [Windows.Graphics.Display.DisplayProperties](#) object, such as the `logicalDPI` and the current `resolutionScale`. The latter is a value from the [Windows.Graphics.Display.ResolutionScale](#) enumeration, one of `scale100Percent`, `scale140Percent`, and `scale180Percent`. The actual values of these identifiers are 100, 140, and 180 so that you can use `resolutionScale` directly in calculations.

Sidebar: A Good Opportunity for Remote Debugging

Working with different device capabilities provides a great opportunity to work with remote debugging as described on [Running apps on a remote machine](#). This will help you test your app on different displays without needing to set up Visual Studio on each one, and it also gives you the benefit of multimonitor debugging. You only need to install and run the remote debugging tools on the target machine and make sure it’s connected with a cable to the same network as your development machine. (You might need to buy a small USB-Ethernet adapter if your device doesn’t have a suitable port—remote debugging doesn’t work over the Internet, and it doesn’t work over wireless networks.) The Remote Debugging Monitor running on the remote machine will announce itself to Visual Studio running on your development machine. Note that the first

time you run the app remotely, you'll be prompted to obtain a developer license for that machine, so it will need to be connected to the Internet during that time.

Graphics That Scale Well

Variable screen sizes and pixel densities can present a bit of a challenge to apps, not just in layout but also in making sure that graphical assets always look their best. You can certainly draw graphics directly with the HTML5 `canvas`; what I want to specifically address are predrawn assets.

HTML5 scalable vector graphics (SVGs) are very handy here. You include inline SVGs in your HTML (including page fragments), or you can keep them in separate files and refer to them in an `img.src` attribute. One of the easiest ways to use an SVG is to place an `img` element inside a proportionally sized cell of a CSS grid and set the element's `width` and `height` styles to 100%. The SVG will then automatically scale to fill the cell, and since the cell will resize with its container, everything is handled automatically.

One caveat with this approach is that the SVG will be scaled to the aspect ratio of the containing grid cell, which isn't always what you want. To control this behavior, make sure the SVG has `viewBox` and `preserveAspectRatio` attributes where the `viewBox` aspect ratio matches that defined by the SVG's `width` and `height` properties:

```
<svg
  xmlns:svg="http://www.w3.org/2000/svg"
  xmlns="http://www.w3.org/2000/svg"
  xmlns:xlink="http://www.w3.org/1999/xlink"
  version="1.0"
  width="300"
  height="150"
  viewBox="0 0 300 150"
  preserveAspectRatio="xMidYMid meet">
```

Of course, you don't always have nice vector graphics. Bitmaps that you include in your app package, pictures you load from files, and raster images you obtain from a service won't be so easily scalable. In these cases, you'll need to be aware of and apply the current scaling factor appropriately.

For assets in your app package, we already saw how to work with varying pixel densities in Chapter 3 through the `.scale-100`, `.scale-140`, and `.scale-180` file name suffixes. These work for any and all graphics in your app, just as they do for the splash screen, tile images, and the other graphics referenced by the manifest. So if you have a raster graphic named `banner.png`, you'll create three graphics in your app package called `banner.scale-100.png`, `banner.scale-140.png`, and `banner.scale-180.png`. You can then just refer to the base name in an element or in CSS, as in `` and `background-image: url('images/banner.png')`, and the Windows resource loader will magically load the appropriately scaled graphic automatically. (If files with `.scale-*` suffixes aren't found, it will look for `banner.png` directly.) We'll see even more such magic in Chapter 17, "Apps for Everyone," when we also include variants for different languages and contrast settings that introduce additional suffixes of their own.

If your developer sensibilities object to this file-naming scheme, know that you can also use similarly named folders instead. That is, create `scale-100`, `scale-140`, and `scale-180` folders in your `images` folder and place appropriate files with unadorned names (like `banner.png`) therein.

In CSS you can also use media queries with `max-resolution` and `min-resolution` settings to control which images get loaded. Remember, however, that CSS will see the logical DPI, not the physical DPI, so the cutoffs for each scaling factor are as follows (the DPI values here are slightly different from those given in documentation because they come from empirical tests; the docs suggest 134, 135, and 174 dpi, respectively):

```
@media all and (max-resolution: 133dpi) {  
    /* 100% scaling */  
}  
  
@media all and (min-resolution: 134dpi) {  
    /* 140% scaling */  
}  
  
@media all and (min-resolution: 172dpi) {  
    /* 180% scaling */  
}
```

As explained in the [Guidelines for scaling to pixel density](#), such media queries are especially useful for images you obtain from a remote source, where you might need to amend the specific URI or the URI query string. See Chapter 13, “Tiles, Notifications, the Lock Screen, and Background Tasks,” in the section “Using Local and Web Images” for how tile updates handle this for scale, contrast, and language.

Programmatically, you can again obtain `logicalDpi` and `resolutionScale` properties from the `Windows.Graphics.Display.DisplayProperties` object. Its `logicaldpichanged` event (a WinRT event) can also be used to check for changes in the `resolutionScale`, since the two are always coupled. Usage of these APIs is demonstrated in the [Scaling according to DPI sample](#).

If your app manages a cache of graphical assets, by the way, especially those downloaded from a service, organize them according to the `resolutionScale` for which that graphic was obtained. This way you can obtain a better image if and when necessary, or you can scale down a higher resolution image that you already obtained. It’s also something to be aware of with any app settings you might roam, because the pixel density and screen size may vary between a user’s devices.

Adaptive and Fixed Layouts for Display Size

Just as every page of your app needs to be prepared for different view states, it should also be prepared for different screen sizes. On this subject, I recommend you read the [Guidelines for scaling to screens](#), which has good information on the kinds of display sizes your app might encounter. From this we can

conclude that the smallest snapped view you'll ever encounter is 320x768, the minimum filled view is 1024x768, and the minimum full-screen views (portrait and landscape) are 1280x800 and 1366x768. These are your basic design targets.

From there, displays only get larger, so the question becomes "What do you do with more space?" The first part of the answer is "Fill the screen!" Nothing looks more silly than an app running on a 27" monitor that was designed and implemented with only 1366x768 in mind, because it will only occupy a quarter to half of the screen at best. As I've said a number of times, imagine the kinds of reviews and ratings your app might be given in the Windows Store if you don't pay attention to details like this!

The second part of the answer depends on your app's content. If you have only fixed content, which is common with games, then you'll want to use a fixed layout that scales to fit. If you have variable content, meaning that you should show more when there's more screen space, then you want to use an adaptive layout. Let's look at both of these in turn.

Sidebar: The Make Everything on Your Screen Bigger Setting

In PC Settings (Settings charm > Change PC Settings in the lower-right corner), there is an option within Ease of Access to "Make everything on your screen bigger." Turning this on effectively enlarges the display by about 40%, meaning that the system will report a screen size to the app that's about 30% smaller than the current scaled resolution (similar to the 140% scaling level). Fortunately, this setting is disabled if it would mean reporting a size smaller than 1024x768, which always remains the minimum screen size your app will encounter. In any case, when this setting is changed it will trigger a `Windows.Graphics.Display.DisplayProperties.-logicalDpiChanged` event.

Fixed Layouts and the ViewBox Control

A fixed layout is the best choice for apps that aren't oriented around variable content, because there isn't more content to show on a larger screen. Such an app instead need to scale its output to fill the display as best it can, depending on whether it needs to maintain an aspect ratio.

An app can certainly obtain the dimensions of its display window and redraw itself accordingly. Every coordinate in the app would be a variable in this case, and elements would be resized and laid out relative to one another. Such an approach is great when an app can adapt its aspect ratio to that of the screen, thereby filling 100% of the display.

You can do the same thing with a fixed aspect ratio by placing limits on your coordinates, perhaps by using an absolute coordinate system to which you then apply your own scaling factor.

Because this is the more common approach, WinJS provides a built-in layout control for exactly this purpose: [WinJS.UI.ViewBox](#) (not to be confused with the SVG `viewBox` attribute). Like all other WinJS controls, you can declare this using `data-win-control` in HTML as follows, where the ViewBox element can contain one and only one child element:

```

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

```

This is really all you ever see with the ViewBox as it has no other options or properties, no methods, and no events—very simple! Note also that because the ViewBox is just a control, you can use it for any fixed aspect-ratio content in an otherwise adaptive layout; it's not only for the layout of an entire page.

To set the reference size of the ViewBox—the dimensions against which you'll write the rest of your code—simply set the `width` and `height` styles of the child element in CSS. For example, to set a base size of 1024x768, we'd set those properties in the rule for the `fixedlayout` class:

```

.fixedlayout {
  width: 1024px;
  height: 768px;
}

```

Once instantiated, the ViewBox simply listens for `window.onresize` events, and it then applies a CSS 2D scaling transform to its child element based on the difference between the reference size and the actual size. This preserves the aspect ratio. This works to scale the contents up as well as down. Automatic letterboxing or sidepillars are also applied around the child element, and you can set the appearance of those areas (really any area not obscured by the child element) by using the `win-viewbox` class. As always, scope that selector to your specific control if you're using more than one ViewBox in your app, unless you want styles to be applied everywhere.

The basic structure above is what you get with a new app created from the Fixed Layout App project template in Visual Studio and Blend. As shown here, it creates a layout with a 1024x768 reference size, but you can use whatever dimensions you like.

The CSS for this project template reveals that the whole page itself is actually styled as a CSS flexbox to make sure the ViewBox is centered, and that the `fixedlayout` element is given a default grid:

```

html, body {
  height: 100%;
  margin: 0;
  padding: 0;
}

body {
  -ms-flex-align: center;
  -ms-flex-direction: column;
  -ms-flex-pack: center;
  display: -ms-flexbox;
}

.fixedlayout {
  -ms-grid-columns: 1fr;
  -ms-grid-rows: 1fr;
  display: -ms-grid;
}

```

```

    height: 768px;
    width: 1024px;
}

```

If you create a project with this template in Blend, add a border style to the `fixedlayout` rule (like `border: 2px solid Red;`), and fiddle with the view states and the display settings on the Device tab. Then you can see how the `ViewBox` provides all the scaling for free. To show this more obviously, the `FixedLayout` example for this chapter changes the child element of the `ViewBox` to a `canvas` on which it draws a 4x3 grid (to match the aspect ratio of 1024x768) of 256px squares containing circles. As shown in Figure 6-6 (after the sidebar), the squares and circles don't turn into rectangles and ovals as we move between view states and display sizes, and letterboxing is handled automatically (applying a `background-color` style to the `win-viewbox` class).

Sidebar: Raster Graphics and Fixed Layouts

If you use raster graphics within a `ViewBox`, size them according to the maximum 2560x1440 resolution so that they'll look good on the largest screens and they'll still scale down to smaller ones (rather than being stretched up). Alternately, you can use load different graphics (through different `img.src` URLs) that are better suited for the most common screen size.

Note that resolution scaling will still be applicable. If you're running on a high-density 10.6" 2560x1440 display (180% scale), the app and thus the `ViewBox` will still see smaller screen dimensions. But if you're supplying a graphic for the native device resolution, it will look sharp when rendered on the screen.

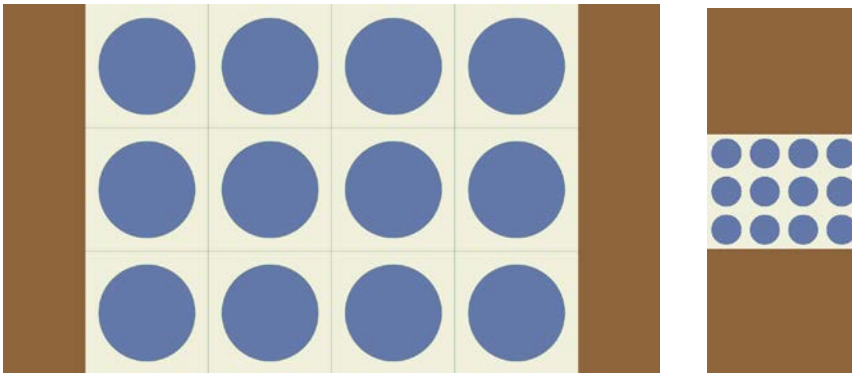


FIGURE 6-6 Fixed layout scaling with the `WinJS.UI.ViewBox` controls, showing letterboxing on a full-screen 1366x768 display (left) and in snapped view (right).

Adaptive Layouts

Adaptive layouts are those in which an app shows more content when more screen space is available.

Such a layout is most easily achieved with a CSS grid where proportional rows and columns will auto-matically scale up and down; elements within grid cells will then find themselves resized accordingly. This is demonstrated in the Visual Studio/Blend project templates, especially the Grid App project template. On a typical 1366x768 display you'll see a few items on a screen, as shown at the top of Figure 6-7. Switch over to a 27" 2560x1440 and you'll see a lot more, as you can see at the bottom of the figure.



FIGURE 6-7 Adaptive layout in the Grid App project template shown for a 1366x768 display (top) and a 2560x1440 display (bottom).

To be honest, the Grid App project template doesn't do anything different for display size than it already does for view states. Because it uses CSS grids and proportional cell sizes, the cell containing the ListView control automatically becomes bigger. The ListView control is listening for `window.onresize` on its own, so we don't need to separately instruct it to update its layout.

The overall strategy for an adaptive layout, then, is straightforward:

- Use a CSS grid where possible to handle adaptive layout automatically.
- Listen for `window.onresize` as necessary to reposition and resize elements manually, such as an HTML `canvas` element.
- Have controls listen to `window.onresize` to adapt themselves directly. This is especially important for collection controls like `ListView`.

As another reference point, refer to the [Adaptive layout with CSS sample](#), which really takes the same approach as the Grid App project template, relying on controls to resize themselves. In the sample, you will see that the app isn't doing any direct calculations based on window size.

Hint If you have an adaptive layout and want a background image specified in CSS to scale to its container (rather than being repeated), style `background-size` to either `contain` or `100% 100%`.

It should be also clear to you as a developer that how an app handles different screen sizes is also a design matter. The strategy above is what you use to implement a design, but the design still needs to think about how everything should look. The following considerations, which I only summarize here, are described on [Guidelines for scaling to screens](#):

- Which regions are fixed and which are adaptive?
- How do adaptive regions makes use of available space, including the directions in which that region adapts?
- How do adaptive and fixed regions relate in the wireframe?
- How does the app's layout overall makes use of space—that is, how does whitespace itself expand so that content doesn't become too dense?
- How does the app make use of multicolumn text?

Answering these sorts of questions will help you understand how the layout should adapt.

Using the CSS Grid

Starting back in Chapter 2, we've already been employing CSS grids for many purposes. Personally, I love the grid model because it so effortlessly allows for relative placement of elements and scaling easily to different screen sizes.

Because the focus of this book is on the specifics of Windows 8, I'll leave it to the W3C specs on <http://www.w3.org/TR/css3-grid-layout/> and <http://dev.w3.org/csswg/css3-grid-align/> to explain all the details. These specs are essential references for understanding how rows and columns are sized, especially when some are declared with fixed sizes, some are sized to content, and others are declared such that they fill the remaining space. The nuances are many!

Because the specs themselves are still in the draft stages as of this writing, it's good to know exactly which parts of those specs are actually supported by the HTML/CSS engine used for Store apps.

For the element containing the grid, the supported styles are simple. First use the `-ms-grid` and `-ms-inline-grid` display models (the `display:` style). We'll come back to `-ms-inline-grid` later.

Second, use `-ms-grid-columns` and `-ms-grid-rows` on the grid element to define its arrangement. If left unspecified, the default is one column and one row. The repeat syntax such as `-ms-grid-columns: (1fr)[3];` is supported, which is most useful when you have repeated series of rows or columns, which appear inside the parentheses. As examples, all the following are equivalent:

```
-ms-grid-rows:10px 10px 10px 20px 10px 20px 10px;  
-ms-grid-rows:(10px)[3] (20px 10px)[2];  
-ms-grid-rows:(10px)[3] (20px 10px) 20px 10px;  
-ms-grid-rows:(10px)[2] (10px 20px)[2] 10px;
```

How you define your rows and columns is the really interesting part, because you can make some fixed, some flexible, and some sized to the content using the following values. Again, see the specs for the nuances involving `max-content`, `min-content`, `minmax`, `auto`, and `fit-content` specifiers, along with values specified in units of `px`, `em`, `%`, and `fr`. Windows Store apps can also use `vh` (viewport height) and `vw` (viewport width) as units.

Within the grid now, child elements are placed in specific rows and columns, with specific alignment, spanning, and layering characteristics using the following styles:

- `-ms-grid-column` identifies the 1-based column of the child in the grid.
- `-ms-grid-row` identifies the 1-based row of the child in the grid.
- `-ms-grid-column-align` and `-ms-grid-row-align` specify where the child is placed in the grid cell. Allowed values are `start`, `end`, `center`, and `stretch` (default).
- `-ms-grid-column-span` and `-ms-grid-row-span` indicate that a child spans one or more rows/columns.
- `-ms-grid-layer` controls how grid items overlap. This is similar to the `z-index` style as used for positional element. Since grid children are not positioned directly with CSS and are instead positioned according to the grid, `-ms-grid-layer` allows for separate control.

Be very aware that row and column styles are 1-based, not 0-based. Really re-program your JavaScript-oriented mind to remember this, as you'll need to do a little translation if you track child elements in a 0-based array.

Also, when referring to any of these `-ms-grid*` styles as properties in JavaScript, drop the hyphens and switch to camel case, as in `msGrid`, `msGridColumn`, `msGridRowAlign`, `msGridLayer`, and so on.

Overall, grids are fairly straightforward to work with, especially within Blend where you can immediately see how the grid is taking shape. Let's now take a look at a few tips and tricks that you might find useful.

Overflowing a Grid Cell

One of the great features of the grid, depending on your point of view, is that overflowing content in a grid cell doesn't break the layout at all—it just overflows. (This is very different from tables!) What this means is that you can, if necessary, offset a child element within a grid cell so that it overlaps an adjacent cell (or cells). Besides not breaking the layout, this makes it possible to animate elements moving between cells in the grid, if desired.

A quick example of content that extends outside its containing grid cell can be found in the GridOverflow example with this chapter's companion content. For the most part, it creates a 4x4 grid of rectangles, but this code at the end of the `doLayout` function (`js/default.js`), places the first rectangle well outside its cell:

```
children[0].style.width = "350px";
children[0].style.marginLeft = "150px";
children[0].style.background = "#fbb";
```

This makes the first element in the grid wider and moves it to the right, thereby making it appear inside the second element's cell (the background is changed to make this obvious). Yet the overall layout of the grid remains untouched.

I'll cast a little doubt on this being a great feature because you might not want this behavior at times, hoping instead that the grid would resize to the content. For that behavior, try using an HTML table.

Centering Content Vertically

Somewhere in your own experience with CSS, you've probably made the bittersweet acquaintance with the `vertical-align` style in an attempt to place a piece of text in the middle of a `div`, or at the bottom. Unfortunately, it doesn't work: this particular style works only for table cells and for inline content (to determine how text and images, for instance, are aligned in that flow).

As a result, various methods have been developed to do this, such as those discussed in <http://blog.themeforest.net/tutorials/vertical-centering-with-css/>. Unfortunately, just about every technique depends on fixed heights—something that can work for a website but doesn't work well for the adaptive layout needs of a Store app. And the one method that doesn't use fixed heights uses an embedded table. Urk.

Fortunately, both the CSS grid and the flexbox (see "Item Layout" later on) easily solve this problem. With the grid, you can just create a parent `div` with a 1x1 grid and use the `-ms-grid-row-align: center` style for a child `div` (which defaults to cell 1, 1):

```
<!-- In HTML -->
<div id="divMain">
  <div id="divChild">
    <p>Centered Text</p>
  </div>
```



```

</div>

/* In CSS */
#divMain {
    width: 100%;
    height: 100%;
    display: -ms-grid;
    -ms-grid-rows: 1fr;
    -ms-grid-columns: 1fr;
}

#divChild {
    -ms-grid-row-align: center;
    -ms-grid-column-align: center;

    /* Horizontal alignment of text also work with the following */
    /* text-align: center; */
}

```

The solution is even simpler with the `flexbox` layout, where `flex-align: center` handles vertical centering, `flex-pack: center` handles the horizontal, and a child `div` isn't needed at all. This is the same styling that's used in the Fixed Layout App project template to center the `ViewBox`:

```

<!-- In HTML -->
<div id="divMain">
    <p>Centered Text</p>
</div>

/* In CSS */
#divMain {
    width: 100%;
    height: 100%;
    display: -ms-flexbox;
    -ms-flex-align: center;
    -ms-flex-direction: column;
    -ms-flex-pack: center;
}

```

Code for both these methods can be found in the `CenteredText` example for this chapter. (This example is also used to demonstrate the use of ellipsis later on, so it's not exactly as it appears above.)

Scaling Font Size

One particularly troublesome area with HTML is figuring out how to scale a font size with an adaptive layout. I'm not suggesting you do this with the standard typography recommended by Windows app design as we saw earlier in this chapter. It's more a consideration when you need to use fonts in some other aspect of your app such as large letters on a tile in a game.

With an adaptive layout, you typically want certain font sizes to be proportional to the dimensions of its parent element. (It's not a concern if the parent element is a fixed size, because then you can fix the

size of the font.) Unfortunately, percentage values used in the `font-size` style in CSS are based on the default font size (1em), not the size of the parent element as happens with `height` and `width`. What you'd love to be able to do is something like `font-size: calc(height * .4)`, but, well, the value of other CSS styles on the same element are just not available to `calc`.

One exception to this is the `vh` value (which can be used with `calc`). If you know, for instance, that the text you want to scale is contained within a grid cell that is always going to be 10% of the viewport height, and if you want the font size to be half of that, then you can just use `font-size: 5vh` (5% of viewport height).

Another method is to use an SVG for the text, wherein you can set a `viewBox` attribute and a `font-size` relative to that `viewBox`. Then scaling the SVG to a grid cell will effectively scale the font:

```
<svg viewBox="0 0 600 400" preserveAspectRatio="xMaxYMax">
  <text x="0" y="150" font-size="200" font-family="Verdana">
    Big SVG Text
  </text>
</svg>
```

You can also use JavaScript to calculate the desired font size programmatically based on the `clientHeight` property of the parent element. If that element is in a grid cell, the font size (and line height) can be some percentage of that cell's height, thereby allowing the font to scale with the cell.

You can also try using the `WinJS.UI.ViewBox` control. If you want content like text to take up 50% of the containing element, wrap the `ViewBox` in a `div` that is styled to be 50% of the container and style the child element of the `ViewBox` with `position: absolute`. Try dropping the following code into `default.html` of a new Blank app project for a demonstration:

```
<div style="height:50%;">
  <div data-win-control="WinJS.UI.ViewBox">
    <p style="position:absolute;">Big text!</p>
  </div>
</div>
```

Item Layout

So far in this chapter we've explored page-level layout, which is to say, how top-level items are positioned on a page, typically with a CSS grid. Of course, it's all just HTML and CSS, so you can use tables, line breaks, and anything else supported by the rendering engine so long as you adapt well to view states and display sizes.

It's also important to work with item layout in the flexible areas of your page. That is, if you set up a top-level grid to have a number of fixed-size areas (for headings, title graphics, control bars, etc.), the remaining area can vary greatly in size as the window size changes. In this section, then, let's look at some of the tools we have within those specific regions: CSS transforms, flexbox, nested and inline grids, multicolumn text, CSS figures, and CSS connected frames. A general reference for these and all other

CSS styles that are supported for Windows Store apps (such as background, borders, and gradients) can be found on the [Cascading Style Sheets](#) topic.

CSS 2D and 3D Transforms

It's really quite impossible to think about layout for elements without taking CSS transforms into consideration. Transforms are very powerful because they make it possible to change the display of an element without actually affecting the document flow or the overall layout. This is very useful for animations and transitions; transforms are used heavily in the WinJS animations library that provides the Windows 8 look and feel for all the built-in controls. As we'll explore in Chapter 11, "Purposeful Animations," you can make direct use of this library as well.

CSS transforms can be used directly, of course, anytime you need to translate, scale, or rotate an element. Both 2D and 3D transforms (<http://dev.w3.org/csswg/css3-2d-transforms/> and <http://www.w3.org/TR/css3-3d-transforms/>) are supported for Windows Store apps, specifically these styles:³⁷

CSS Style	JavaScript Property (element.style.)
<code>backface-visibility</code>	<code>backfaceVisibility</code>
<code>perspective</code> , <code>perspective-origin</code>	<code>perspective</code> , <code>perspectiveOrigin</code>
<code>transform</code> , <code>transform-origin</code> , and <code>transform-style</code>	<code>transform</code> , <code>transformOrigin</code> , and <code>transformStyle</code>

Full details can be found on the [Transforms](#) reference. Know also that because the app host uses the same underlying engines as Internet Explorer, transforms enjoy all the performance benefits of hardware acceleration.

Flexbox

Just as the grid is magnificent for solving many long-standing problems with page layout, the CSS flexbox module, documented at <http://www.w3.org/TR/css3-flexbox/>, is excellent for handling variable-sized areas wherein the content wants to "flex" with the available space. To quote the W3C specification:

In [this] box model, the children of a box are laid out either horizontally or vertically, and unused space can be assigned to a particular child or distributed among the children by assignment of 'flex' to the children that should expand. Nesting of these boxes (horizontal inside vertical, or vertical inside horizontal) can be used to build layouts in two dimensions.

As the flexbox spec is presently in draft form, the specific display styles for Store apps are `display: -ms-flexbox` (block level) and `display: -ms-inline-flexbox` (inline).³⁸ For a complete reference of the

³⁷ At the time of writing, the `-ms-*` prefixes on these styles were no longer needed but are still supported.

³⁸ If you're accustomed to the `-ms-box*` styles for flexbox, Microsoft has since aligned to the W3C specifications that are expected to be the last major revision before the standard is finalized. As the new syntax replaces the old, the old will not

other supported properties, see the [Flexible Box \("Flexbox"\) Layout](#) documentation:

CSS Style	JavaScript Property (<code>element.style.</code>)	Values
<code>-ms-flex-align</code>	<code>msFlexAlign</code>	<code>start</code> <code>end</code> <code>center</code> <code>baseline</code> <code>stretch</code>
<code>-ms-flex-direction</code>	<code>msFlexDirection</code>	<code>row</code> <code>column</code> <code>row-reverse</code> <code>column-reverse</code> <code>inherit</code>
<code>-ms-flex-flow</code>	<code>msFlexFlow</code>	<code><direction> <pack></code> where <code><direction></code> is an <code>-ms-flex-direction</code> value and <code><pack></code> is an <code>-ms-flex-pack</code> value.
<code>-ms-flex-orient</code>	<code>msFlexOrient</code>	<code>horizontal</code> <code>vertical</code> <code>inline-axis</code> <code>block-axis</code> <code>inherit</code>
<code>-ms-flex-item-align</code>	<code>msFlexItemAlign</code>	<code>auto</code> <code>start</code> <code>end</code> <code>center</code> <code>baseline</code> <code>stretch</code>
<code>-ms-flex-line-pack</code>	<code>msFlexLinePack</code>	<code>start</code> <code>end</code> <code>center</code> <code>justify</code> <code>distribute</code> <code>stretch</code>
<code>-ms-flex-order</code>	<code>msFlexOrder</code>	<code><integer></code> (ordinal group)
<code>-ms-flex-pack</code>	<code>msFlexPack</code>	<code>start</code> <code>end</code> <code>center</code> <code>justify</code>
<code>-ms-flex-wrap</code>	<code>msFlexWrap</code>	<code>none</code> <code>wrap</code> <code>wrapreverse</code>

As with all styles, Blend is a great tool in which to experiment with different flexbox styles because you can see the effect immediately. It's also helpful to know that flexbox is used in a number of places around WinJS and in the project templates, as we saw with the Fixed Layout template earlier. The ListView control in particular takes advantage of it, allowing more items to appear when there's more space. The FlipView uses flexbox to center its items, and the Ratings, DatePicker, and TimePicker controls all arrange their inner elements using an inline flexbox. It's likely that your own custom controls will do the same.

Nested and Inline Grids

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

work in Windows Store apps nor Internet Explorer 10.

```

//Within the activated handler
document.getElementById("chkInline").addEventListener("click", function () {
    setGridStyle(document.getElementById("chkInline").checked);
});

setGridStyle(true);

//Elsewhere in default.js
function setGridStyle(inline) {
    var gridClass = inline ? "inline" : "block";

    document.getElementById("grid1").className = gridClass;
    document.getElementById("grid2").className = gridClass;
    document.getElementById("grid3").className = gridClass;
}

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

.block {
    display: -ms-grid;
}

```

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

Cell 1-1	Cell 1-2	
Cell 2-1	Cell 2-2	

Cell 1-1	Cell 1-2	
Row 2 (spanning columns)		

Cell 1-1	Cell 1-2	Column 3 (spanning rows)
Cell 2-1	Cell 2-2	

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

Cell 1-1	Cell 1-2
Cell 2-1	Cell 2-2

Cell 1-1	Cell 1-2
Row 2 (spanning columns)	

Cell 1-1	Cell 1-2	Column 3 (spanning rows)
Cell 2-1	Cell 2-2	

Fonts and Text Overflow

As discussed earlier, typography is an important design element for Store apps, and for the most part the standard font styles using Segoe UI are already defined in the default WinJS stylesheets. In the Windows SDK there is a very helpful [CSS typography sample](#) that compares the HTML header elements and the `win-type-*` styles, demonstrating font fallbacks and how to use bidirectional fonts (left to right and right to left directions).

Speaking of fonts, custom font resources using the `@font-face` rule in CSS are allowed in Store apps. For local context pages, the `src` property for the rule must refer to an in-package font file (that is, a URI that begins with `/` or `ms-appx:///`). Pages running in the web context can load fonts from remote sources.

Another piece of text and typography is dealing with text that overflows its assigned region. You can use the CSS `text-overflow: ellipsis` style to crop the text with a `...`, and the WinJS stylesheets contain the `win-type-ellipsis` class for this purpose. In addition to setting `text-overflow`, this class also adds `overflow: hidden` (to suppress scrollbars) and `white-space: nowrap`. It's basically a style you can add to any text element when you want the ellipsis behavior.

The W3C specification on text overflow, <http://dev.w3.org/csswg/css3-ui/#text-overflow>, is a helpful reference as to what can and cannot be done here. One of the limitations of the current spec is that multiline wrapping text doesn't work with ellipsis. That is, you can word-wrap with the `word-wrap: break-word` style, but it won't cooperate with `text-overflow: ellipsis` (`word-wrap` wins). I also investigated whether flowing text from a multiline CSS region (see next section) into a single-line region with ellipsis would work, but `text-overflow` doesn't apply to regions. So at present you'll need to shorten the text and insert ellipsis manually if it spans multiple lines.

For a demonstration of ellipsis and word-wrapping, see the CenteredText example for this chapter.

Multicolumn Elements and Regions

Translating the multicolumn flow of content that we’re so accustomed to in print media has long been a difficult proposition for web developers. While it’s been easy enough to create elements for each column, there was no inherent relationship between the content in those columns. As a result, developers have had to programmatically determine what content could be placed in each element, accounting for variations like font size or changing the number of columns based on the screen width or changes in device orientation.

CSS3 provides for doing multicolumn layout within an element (see <http://www.w3.org/TR/css3-multicol>). With this, you can instruct a single element to lay out its contents in multiple columns, with specific control over many aspects of that layout. The specific styles supported for Windows Store apps (with no pesky little vendor prefixes!) are as follows:

CSS Styles	JavaScript Property (element.style.)
<code>column-width</code> and <code>column-count</code> (<code>columns</code> is the shorthand)	<code>columnWidth</code> , <code>columnCount</code> , and <code>columns</code>
<code>column-gap</code> , <code>column-fill</code> , and <code>column-span</code>	<code>columnGap</code> , <code>columnFill</code> , and <code>columnSpan</code>
<code>column-rule-color</code> , <code>column-rule-style</code> , and <code>column-rule-width</code> (<code>column-rule</code> is the shorthand for separators between columns)	<code>columnRuleColor</code> , <code>columnRuleStyle</code> , and <code>columnRuleWidth</code> (<code>columnRule</code> is the shorthand)
<code>break-before</code> , <code>break-inside</code> , and <code>break-after</code>	<code>breakBefore</code> , <code>breakInside</code> , and <code>breakAfter</code>
<code>overflow: scroll</code> (to display scrollbars in the container)	<code>overflow</code>

The reference documentation for these can be found on [Multi-column layout](#).

Again, Blend provides a great environment to explore how these different styles work. If you’re placing a multicolumn element within a variable-size grid cell, you can set `column-width` and let the layout engine add and remove columns as needed, or you can use media queries or JavaScript to set `column-count` directly.

CSS3 multicolumn again only applies to the contents of a single element. While highly useful, it does impose the limitation of a rectangular element and rectangular columns (spans aside). Certain apps like magazines need something more flexible, such as the ability to flow content across multiple elements with more arbitrary shapes, and columns that are offset from one another. These relationships are illustrated in Figure 6-8, where the box in the upper left might be a title, the inset box might contain an image, and the text content flows across two irregular columns.

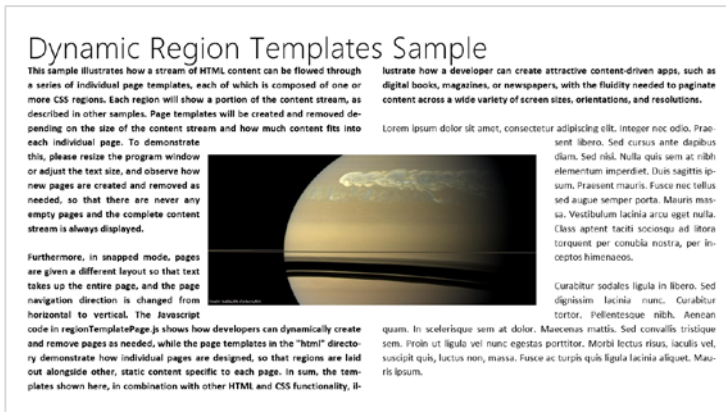


FIGURE 6-8 Using CSS regions to achieve a more complex layout with irregular text columns.

To support irregular columns, CSS Regions (see <http://dev.w3.org/csswg/css3-regions/>) are coming online and are supported in Store apps (see [Regions](#) reference). Regions allow arbitrarily (that is, absolutely) positioned elements to interact with inline content. In Figure 6-8, the image would be positioned absolutely on the page and the column content would flow around it.

The key style for a positioned element is the `float: -ms-positioned` style which should accompany `position: absolute`. Basically that's all you need to do: drop in the positioned element, and the layout engine does the rest. It should be noted that CSS Hyphenation, yet another module, relates closely to all this because doing dynamic layout on text immediately brings up such matters. Fortunately, Store apps support the `-ms-hyphens` and the `-ms-hyphenation-*` styles (and their equivalent JavaScript properties). The hyphenation spec is located at <http://www.w3.org/TR/css3-text/>; documentation for Store apps is found on the [Text styles reference](#).

The second part of the story consists of named flows and region chains (which are also part of the Regions spec). These provide the ability for content to flow across multiple container elements, as shown in Figure 6-9. Region chains can also allow the content to take on the styling of a particular container, rather than being defined at the source. Each container, in other words, gets to set its own styling and the content adapts to it, but commonly all the containers share similar styling for consistency.

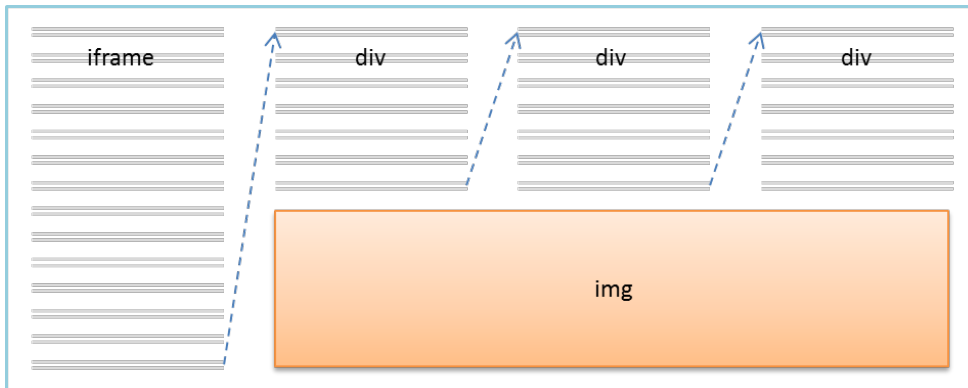


FIGURE 6-9 CSS region chains to flow content across multiple elements.

How this all works is that the source content is defined by an `iframe` that points to an HTML file (and the `iframe` can be in the web or local context, of course). It's then styled with `-ms-flow-into: <element>` (`msFlowInfo` in JavaScript) where `<element>` is the id of the first container:

```
<!-- HTML -->
<iframe id="s1-content-source" src="/html/content.html"></iframe>
<div class="s1-container"></div>
<div class="s1-container"></div>
<div class="s1-container"></div>

/* CSS */
#s1-content-source {
    -ms-flow-into: content;
}
```

Note that `-ms-flow-into` prevents the `iframe` content from displaying on its own.

Container elements can be any *nonreplaced* element—that is, any element whose appearance and dimensions are not defined by an external resource, such as `img`—and can contain content between its opening and closing tags, like a `div` (the most common) or `p`. Each container is styled with `-ms-flow-from: <element>` (`msFlowFrom` in JavaScript) where the `<element>` is the first container in the flow. The layout then happens in the order elements appear in the HTML (as above):

```
.s1-container {
    -ms-flow-from: content;
    /* Other styles */
}
```

This simple example was taken from the [Static CSS regions sample](#), which also provides a few other scenarios. There are two other applicable projects here as well, the [Dynamic CSS regions sample](#) and the [Dynamic CSS region templates sample](#), where the latter is the source for Figure 6-8 above. In all these cases, be aware that styling for regions is limited to properties that affect the container and not the

content—content styles are drawn from the `iframe` HTML source. This is why using `text-overflow: ellipsis` doesn't work, nor will `font-color` and so forth. But styles like `height` and `width`, along with borders, margin, padding, and other properties that don't affect the content can be applied.

What We've Just Learned

- Layout that is consistent with Windows 8 design principles—specifically the silhouette and typography—helps users focus immediately on content rather than having to figure out each specific app.
- The principle of “content before chrome” allows content to use 75% or more of the display space rather than 25% as is common with chrome-heavy desktop or web applications.
- In some cases, such as a home or hub page of an app with varied and content that does not come from a single collection, it's best to just use plain HTML/CSS layout rather than using a control.
- Pannable HTML sections can use snap points to automatically stop panning at particular intervals within the content.
- The CSS grid is a highly useful mechanism for adaptive page-level layout, and it can also be used inline. The CSS flexbox is most useful for inline content, though it has uses at the page level as well, as for centering content vertically and horizontally.
- Every page of an app (including the extended splash screen) can encounter all four view states, so an app design must show how those states are handled. Media queries and the Media Query Listener API can be used to handle the view states declaratively and programmatically.
- Apps can specify a preferred orientation in their manifest and also lock the orientation at run time.
- The `window.onresize` event is best for knowing when the window size has changed, due to view states and/or changes in screen size and pixel density.
- Handling varying screen sizes is accomplished either through a grid-based adaptive layout or a fixed layout utilizing the `WinJS.UI.ViewBox` control that does automatic scaling of its content.
- The chief concern with pixel density is providing graphics that scale well. This means either using vector graphics or providing scaled variants of each raster graphic.
- Windows Store apps can take advantage of a wide range of CSS 3 options, including the grid, flexbox, transforms, multicolumn text, and regions.

Chapter 7

Commanding UI

For consumers coming anew to Windows 8 and Windows Store apps, one of their first reactions might be “Where are the menus? Where is the ribbon? How do I tell this app to do something with the items I selected from a list?” This will be a natural response until users become more accustomed to where commands live, giving another meaning, albeit a mundane one, to the dictum “Blessed are those who have not seen, and yet believe!”

With the design principle of “content before chrome,” UI elements that exist solely to invoke actions and don’t otherwise contain meaningful content fall into the category of “chrome.” As such, they are generally kept out of sight until needed, as are system-level commands like the Charms bar. The user indicates his or her desire for those commands through an appropriate gesture. A swipe on the top or bottom edge of the display, a right mouse button click, or the Win+Z key combination brings up app-specific commands at the top and bottom. A swipe on the left edge of the display, a mouse click on the upper left corner, or Win+Tab allows for switching between apps. And a swipe on the right edge of the display, a mouse click on the upper-right or lower-right corner, or Win+C reveals the Charms bar. (Win+Q, Win+H, and Win+i open the Search, Share, and Settings charms directly.) An app responds to the different charms through particular contracts, as we’ll see in a number of the chapters that follow.

App-specific commands, for their part, are generally provided through an app bar control: [WinJS.UI.AppBar](#). In many ways, the app bar is the equivalent of a menu and ribbon for Windows Store apps, because you can create all sorts of UI within it and even show menu elements. Menus, supplied by the [WinJS.UI.Menu](#) control, can also pop up from specific points on the app’s main display, such as a menu attached to a header.

The app bar and menus are specific instances of the more generic [WinJS.UI.Flyout](#) control, which is used directly for messages or actions that the user can cancel or ignore; such flyouts are dismissed simply by clicking or tapping outside the flyout’s window. (This is like pressing a Cancel button.) For important messages that require action—that is, where the user must choose between a set of options—apps employ [WinJS.UI.MessageDialog](#). Dialog boxes are a familiar concept from the world of desktop applications and have long been used for collecting all kinds of information and adjusting app settings. In Windows Store app design, however, dialog boxes are used only to ask a question and get a simple answer, or just to inform the user of some condition. Settings are specifically handled through the Settings charm, as we’ll see in Chapter 8, “State, Settings, Files, and Documents.”

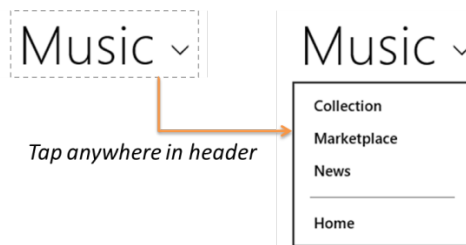
An important point with all of these command controls is that they don’t participate in page layout: they instead “fly out” and remain on top of the current page. This means we thankfully don’t need to worry about their impact on layout...with one small exception that I’ll keep secret for now.

To begin with, though, let's take a step back to think about an app's commands as a whole and where those commands are ideally placed.

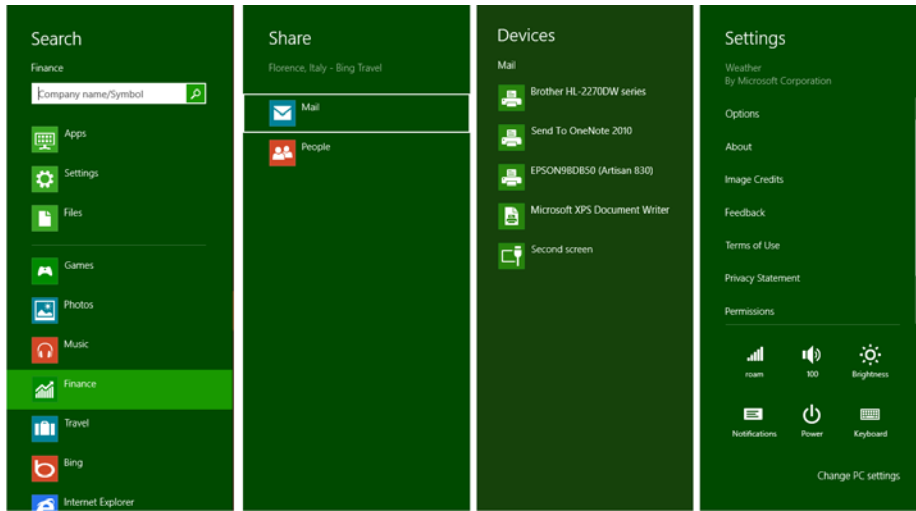
Where to Place Commands

The placement of commands is really quite central to app design. Unlike the guidelines—or lack thereof!—for desktop application commands, which has resulted in quite a jumble, the Windows Developer Center offers two rather extensive topics on this subject: [Commanding design](#) and [Laying out your UI](#). These are must-reads for any designer working on an app, because they describe the different kinds of commanding UI and how to gain the best smiling accolades from Windows 8 design pundits. These are also good topics for developers because they can give you some idea of what you might expect from your designers. Let's review that guidance, then, as an introductory tour to the various options:

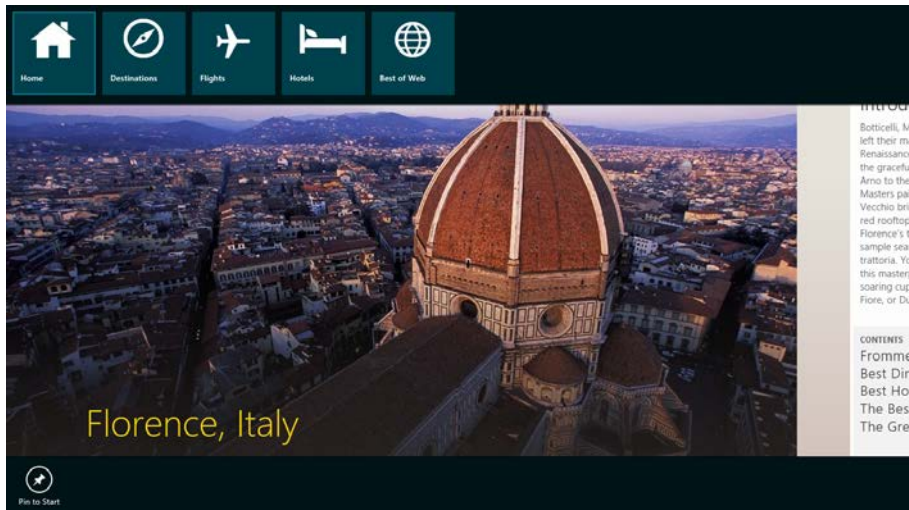
- The user should be able to complete their most important scenarios using just the *app canvas*, so commands that are essential to a workflow should appear directly on-screen. The overall purpose here is to minimize the distraction of unnecessary commands. Nonessential commands should be kept out of view, except for navigation options that can be placed in a drop-down header menu like this:



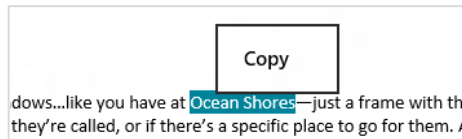
- Always use *Charms* for common app commands where possible. That is, instead of supplying your own search control, use the Search charm (except when the app has a much richer search UI with additional criteria beyond keywords). Instead of supplying individual commands to share with specific targets such as email apps, your contacts, social network apps, and the like, use the Share charm. Instead of supplying your own Print commands, rely on the Device charm. And instead of creating pages within your navigation hierarchy for app settings, help, About, permissions, license agreements, privacy statements, and login/account management, simplify your life and use the Settings charm! (Refer also to "Sidebar: Logins and License Agreements.") Examples of these are shown in the image below, which also illustrates that many app commands can leverage the Charms bar, which means less clutter in the rest of your commanding UI. Again, we'll cover how to respond to Charms events in later chapters.



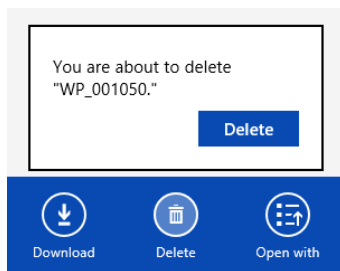
- Commands that can't be placed in Charms and don't need to be on the app canvas are then placed within the *app bar* as shown below in the Travel app; this is the closest analogy to a traditional menu:
 - The top app bar is reserved for navigation commands.
 - The bottom app bar contains all other commands that are sensitive to the context or selection, as well as global (nonselection) commands. Context and global commands are placed on different sides of the app bar.
 - App bar commands can display menus to group related commands and not clutter the app bar itself.



- *Context menus* can provide specific commands for particular content or a selection. For example, selected text typically provides a context menu for clipboard commands, as shown here in the Mail app.



- Confirmations and other questions (including collecting information) that you need to display *in response to a user action* should use a *flyout* control; see [Guidelines and checklist for Flyouts](#). Tapping or clicking outside the control (or pressing ESC) is the same as canceling. Here's an example from the SkyDrive app:



- For blocking events that are not related to a user command but that affect the whole app, use a *message dialog*. A message dialog effectively disables the rest of the app until you pay attention to it! A good example of this is a loss of network connectivity, where the user needs to be informed that some capabilities may not be available until connectivity is restored. User consent prompts for capabilities like geolocation, as shown below from the Maps app, is another place you see message dialogs. Note that a message dialog is used only when the app is in the foreground. Toast notifications, as we'll see in Chapter 13, "Tiles, Notifications, the Lock Screen, and Background Tasks," apply only to background apps.



- Finally, other errors that don't require user action can be displayed either inline (on the app canvas) or through flyouts. See [Laying out your UI: errors](#) for full details; we'll see some examples later on as well.

Where the bottom app bar is concerned, it's also important to organize your commands into sets, as this streamlines implementation as we'll see in the next section. For full guidance I recommend two additional topics in the documentation: [Guidelines and checklist for app bars](#) and [Commanding Design](#).

which provide many specifics on placement, spacing, and grouping. That guidance can be summarized as follows:

- First, make two groups of commands: one with those commands that appear throughout the entire app, regardless of context, and another with those that show only on certain pages. The app bar control is fairly simple to reconfigure at run time for different groups.
- Next, create command sets, such as those that are functionally related, those that toggle view types, and those that apply to selections. Remember that an app bar command can display a popup menu, as shown below, to provide a list of options and/or additional controls, including longer labels, drop-down lists, checkboxes, radiobuttons, and toggle switches. In this way you can combine closely related commands into a single one that gets more room to play than its little space on the app bar proper.



- For placement, put persistent commands on the right side of the app bar and the most common context-specific commands on the left. After that, begin to populate toward the middle. This recommendation comes from the ergonomic realities of human hands: fingers and thumbs—even on the largest hands of basketball players!—grow only so long and can reach only so far on the screen without having to move one's hand. The most commonly used commands are best placed nearest to where a person's thumbs will be when holding a device, as indicated in the image below (from the [Windows 8 Touch Posture](#) topic in the docs). Those spots are easier to reach (especially by those of us that can't grip a large ball with one hand!) and thus make the whole user experience more comfortable.



- The app bar is always available in all view states, including snapped. It's recommended in snapped view (and sometimes portrait) to limit the commands to 10 so that they can fit into one or two rows.
- Know too that the app bar is not limited to circular command buttons: you can create whatever custom layout you like, which is how top navigation bars are implemented. With any custom layout, make sure that your elements are appropriately sized for touch interaction. More on this—including a small graphic of the aforementioned finger of a basketball player—can again be found on [Guidelines and checklist for app bars](#) as well as [Touch interaction design](#) under "Windows 8 Touch targets."

Sidebar: Logins and License Agreements

As noted above, Microsoft recommends that login/account management and license agreements/terms-of-use pages are accessed through the Settings charm, where an app adds relevant commands to the Settings pane that first appears when the charm is invoked. These commands then invoke subsidiary pages with the necessary controls for each functions. Of course, sometimes logins and license agreements need some special handling. For example, if your app *requires* a login or license agreement on startup, such controls can be shown on the app's first page or provided through the Credential Picker UI (see Chapter 14, "Networking"). If the user provides a login and/or agrees to the terms of service, the app can continue to run. Otherwise, the app should show a page that indicates that a login or agreement is necessary to do something more interesting than stare at error messages.

If a login is *recommended* but not required, perhaps to enable additional features, you can place those controls directly on the canvas. When the user logs in, you can replace those controls with bits of profile information (user name and picture, for example, as on the Windows Start screen). If, on the other hand, a login is entirely optional, keep it within Settings.

In all cases, commands to view the license agreement, manage one's account or profile, and log in or out should still be available within Settings. Other app bar or on-canvas commands can invoke Settings programmatically, as we'll see in Chapter 8.

The App Bar

After placing essential commands on the app canvas, most of your app's commands will be placed in the app bar. Again, the app bar is automatically brought up in response to various user gestures, such as a top or bottom edge swipe, Win+Z, or a right mouse button click. Whenever you perform one of these gestures, Windows looks for app bar controls on the current page and invokes them—you don't need to process any input events yourself.

Tip To prevent the app bar from appearing, you can do one of two things. First, to prevent the appbar from appearing at all (for any gesture), set the app bar's element's `winControl.disabled` property to `true`. Second, if you want to prevent it for, say, a right-click on a particular element (such as a canvas), listen to the `contextmenu` (right click) event for that element and call `eventArgs.preventDefault()` within your handler.

For apps written in HTML and JavaScript, the app bar control is implemented as a WinJS control: [WinJS.UI.AppBar](#). As with all other WinJS controls, you declare an app bar in HTML and instantiate it with a call to `WinJS.UI.process` or `WinJS.UI.processAll`. For a first example, we don't need to look any farther than some of the Visual Studio/Blend project templates like the Grid App project, where a placeholder app bar is included in `default.html` (initially commented out):

```
<div id="appbar" data-win-control="WinJS.UI.AppBar">
  <button data-win-control="WinJS.UI.AppBarCommand"
    data-win-options="{id:'cmd', label:'Command', icon:'placeholder'}">
  </button>
</div>
```

The super-exciting result of this markup, using the `ui-dark.css` stylesheet, is as follows:



Because the app bar is declared in `default.html`, which is the container for all other page controls, *this same app bar will apply to all the pages in the app*. With this approach you can declare all your commands within a single app bar and assign different classes to the commands that allow you to easily show and hide command sets as appropriate for each page. This also centralizes those commands that appear on multiple pages, and you can wire up event handlers for them in your app's primary activation code (such as that in `default.js`).

Alternately, you can declare an app bar within the markup for individual page controls. Since an app bar will still be in the DOM, the Windows gestures will invoke it on each particular page. In the Grid App project, for example, you can move the markup above from `default.html` into `groupedItems.html`, `groupDetail.html`, and `itemDetail.html` with whatever modifications you like for each page. This might be especially useful if your app's pages don't share many commands in common.

In these cases, each page's `ready` method should take care of wiring up the commands on its particular app bar. Note also that you can add handlers within a page's `ready` method even for a central app bar; it's just a matter of calling `addEventListener` on the appropriate child element within that app bar.

Let's look now at how all this works through the [HTML AppBar control sample](#). (This chapter's companion content also has a modified version.) We'll start with the basics and the standard command-oriented configuration for app bars, look at how to display menus for some of those commands, and then see how to create custom layouts as is used for a top navigation bar.

Hint Technically speaking, you can declare as many app bars as you want in whatever pages you want, and they'll all be present in the DOM. However, the last one that gets processed in your markup will be the one that's topmost in the z-index by default and therefore the one to receive events. Windows does not make any attempt to combine app bars, so because page controls are inserted into the middle of a host page like default.html, an app bar in default.html that's declared after the page control host element will appear on top. At the same time, if the page control's app bar is larger than that in default.html, a portion of it might be visible. The bottom line: declare app bars *either* in the host page or in a page control, but not both.

App Bar Basics and Standard Commands

As I just mentioned, an app bar can be declared once for an app in a container page like default.html or can be declared separately for each individual page control. The HTML AppBar control sample does the latter, because it provides very distinct app bars for its various scenarios.

Scenario 1 of the sample (html/create-appbar.html) declares an app bar with four commands and a separator:

```
<div id="createAppBar" data-win-control="WinJS.UI.AppBar" data-win-options="">
  <button data-win-control="WinJS.UI.AppBarCommand" data-win-options="{id:'cmdAdd',
    label:'Add', icon:'add', section:'global', tooltip:'Add item'}">
  </button>
  <button data-win-control="WinJS.UI.AppBarCommand" data-win-options="{id:'cmdRemove',
    label:'Remove', icon:'remove', section:'global', tooltip:'Remove item'}">
  </button>
  <hr data-win-control="WinJS.UI.AppBarCommand" data-win-options="{type:'separator',
    section:'global'}" />
  <button data-win-control="WinJS.UI.AppBarCommand" data-win-options="{id:'cmdDelete',
    label:'Delete', icon:'delete', section:'global', tooltip:'Delete item'}">
  </button>
  <button data-win-control="WinJS.UI.AppBarCommand" data-win-options="{id:'cmdCamera',
    label:'Camera', icon:'camera', section:'selection', tooltip:'Take a picture'}">
  </button>
</div>
```

This appears in the app as follows, using the ui-light.css stylesheet, in which we can also see a tooltip, a focus rectangle, and a hover effect on the Add command (I placed my mouse over the command to see all this):



In the markup, the app bar control is declared like any other WinJS control (this is becoming a habit!) using some containing element (a `div`) with `data-win-control="WinJS.UI.AppBar"`. Each page in this sample is loaded with `WinJS.UI.Pages.render` that conveniently calls `WinJS.UI.processAll` to instantiate the app bar. (It is also allowable, as with other controls, to create an app bar programmatically using the `new` operator.)

This example doesn't provide any specific options for the app bar in its `data-win-options`, but there are a number of possibilities:

- `disabled`, if set to `true`, creates an initially disabled app bar; the default is `false`.
- `layout` can be `"commands"` (the default) or `"custom"`, as we'll see in the "Custom App Bars and Navigation Bars" section later.
- `placement` can be either `"top"` or `"bottom"` (the default). We'll use `"top"` for a navigation bar later.
- `sticky` changes the light-dismiss behavior of the app bar. With the default of `false`, the app bar will be dismissed when you click or tap outside of it. If this is set to `true`, the app bar will stay on the screen until either you change `sticky` to `false` and tap outside or you programmatically relieve the control from its duties with its `hide` method.

So, if you wanted a sticky navigation bar with a custom layout to appear at the top of the screen, you'd use markup like this:

```
<div id="navBar" data-win-control="WinJS.UI.AppBar"
    data-win-options="{layout:'custom', placement:'top', sticky: true}">
```

Note that having two app bars in a page with different `placement` values will not interfere with each other. Also, the `sticky` property for each placement operates independently. So if you want to implement an appwide top navigation bar, you could declare that within `default.html` (or whatever your top-level page happens to be), and declare bottom app bars in each page control. Again, they're all just elements in the DOM!

As you can see, an app bar control can contain any number of child elements for its commands, each of which *must* be a [WinJS.UI.AppBarCommand](#) control within a `button` or `hr` element or the app bar won't instantiate.

The properties and options of an app bar command are as follows:

- `id` The element identifier, which you can use with `document.getElementById` or the app bar's `getCommandById` method to wire up `click` handlers.
- `type` One of `"button"` (the default), `"separator"` (which creates a vertical bar), `"flyout"` (which triggers a popup specified with the `flyout` property; see "Command Menus" later), and `"toggle"` (which creates a button with on/off states). In the latter case, the `selected` property of a command can also be used to set the initial value and to retrieve the state at run time.
- `label` The text shown below for the command button. You always want to use this instead of providing text for the `button` element itself, because such text won't be aligned properly in the control. (Try it and you'll see!) Also, note that this property, along with `tooltip` below, is often localized using `data-win-res` attributes. We'll cover this in Chapter 17, "Apps for Everyone," but for the time being you can look at the `html/localize-appbar.html` file in the sample (Scenario 8) to see how it works.

- `tooltip` The (typically localized) tooltip text for the command, using the value of `label` as the default. Note that this is just text; using a full HTML-based `WinJS.UI.Tooltip` control here is not supported.
- `icon` Specifies the glyph that's shown in the command. Typically, this is one of the strings from the `WinJS.UI.AppBarIcon` enumeration, which contains 150 different options from the Segoe UI Symbol font. If you look in the `ui.strings.js` resource file of WinJS you can see how these are defined using codes like `\uE109`—the enumeration, in fact, simply provides friendly names for character codes `\uE100` through `\uE1E9`. But you're not limited by these. For one thing, you can use any other Unicode escape value `'\uXXXX'` you want from the Segoe UI Symbol font. (Note the single quotes.) You can also use a different font or use your own graphics as described in "Custom Icons" later.³⁹
- `section` Controls the placement of the command. For left-to-right languages (such as English), the default value of `"selection"` places the command on the left side of the app bar and `"global"` places it on the right. For right-to-left languages (such as Hebrew and Arabic), the sides are swapped. These simple choices encourage consistent placement of these two categories of commands (and using any other random value here defaults to `"selection"`). This trains users' eyes to look for the most constant commands on one side and selection-specific commands on the other. Note that the commands in each section are laid out left-to-right (or right-to-left) in the order they appear in your markup.
- `onclick` Can be used to declaratively specify a click handler; remember that any function named here in markup must be marked safe for processing. (See Chapter 4, "Controls, Control Styling, and Data Binding" in the "Strict Processing and processAll Functions" section.) Click handlers can also be assigned programmatically with `addEventListener`, in which case the mark is not needed.
- `disabled` Sets the disabled state of a command if `true`; the default is `false`.
- `extraClass` Specifies one or more CSS classes that are attached to the command. These can be used to individually style command controls as well as to create sets that you can easily show and hide, as explained in the "Showing, Hiding, Enabling, and Updating Commands" section later.

If you want to generate commands at run time, you can do so by setting the app bar's `commands` property with an array of JSON `AppBarCommand` descriptors any time the app bar isn't visible (that is, when its `hidden` property is `true`). An array of such descriptors for the Scenario 1 app bar in the sample would be as follows (this is provided in the modified sample included with this chapter; see `js/create_appbar.js`):

³⁹ Three notes: First, within `data-win-options` the Unicode escape sequence can also be in the HTML form of `&#xNNNN`; I prefer the JSON form because it has much less ceremony and is less prone to error. Second, you can use the Character Map desktop applet (`charmap.exe`) to examine all the symbols within any particular font. Third, if you need to localize an icon, you can specify the icon property in the `data-win-res` string since the `icon` property ultimately resolves to a string.

```

var appBar = document.getElementById("createAppBar").winControl;

//Set the app bar commands property to populate it
var commands = [
    { id: 'cmdAdd', label: 'Add', icon: 'add', section: 'global', tooltip: 'Add item' },
    { id: 'cmdRemove', label: 'Remove', icon: 'remove', section: 'global',
      tooltip: 'Remove item' },
    { type: 'separator', section: 'global' },
    { id: 'cmdDelete', label: 'Delete', icon: 'delete', section: 'global',
      tooltip: 'Delete item' },
    { id: 'cmdCamera', label: 'Camera', icon: 'camera', section: 'selection',
      tooltip: 'Take a picture' }
];

appBar.commands = commands;

```

When the app bar is created, it will iterate through the `commands` array and create `WinJS.UI.AppBar-Command` controls for each item. If `type` isn't specified or if it's set to `"button"`, `"flyout"`, or `"toggle"`, then the command is a `button` element. A `type` of `"separator"` creates an `hr` element. Note that you should localize the `label`, `tooltip`, and possibly `icon` fields in each command declaration rather than using explicit text as shown here.

You can also use such an array directly within declarative markup, but this form cannot be localized and is thus discouraged (though I include comments that show how in the modified sample). At the same time, because the value of `commands` in markup is just a string, you can assign its value through data binding with an attribute like this in the app bar element:

```
data-win-bind="{ winControl.commands: myData.commands }"
```

where `myData.commands` can refer to a localized data source. However, this does not work with the `data-win-res` attribute (as we'll see in Chapter 17 and which is also shown in Scenario 8 of the sample) because the resource string won't be converted to JSON as part of the resource lookup. Attempting to play such a trick would be more trouble than it's worth anyway, so it's best to use either the HTML declarative form or a localized commands array at run time.

Also, be aware that `commands` is a rare example of a *write-only* property: you can set it, but you cannot retrieve the array from an app bar. The app bar uses this array only to configure itself and the array is discarded once all the elements are created in the DOM. At run time, however, you can use the app bar's `getCommandById` method to retrieve a particular command element.

Command Events

Speaking of the command elements, an app bar's `AppBarCommand` controls (other than separators) are all just `button` elements and thus respond to the usual events. Because each command element is assigned the `id` you specify, you can use `getElementById` as usual as a prelude to `addEventListener`. In Scenario 1 of the HTML App Bar control sample, for instance, this code appears in the page's `ready` method:

```
document.getElementById("cmdAdd").addEventListener("click", doClickAdd, false);
document.getElementById("cmdRemove").addEventListener("click", doClickRemove, false);
document.getElementById("cmdDelete").addEventListener("click", doClickDelete, false);
document.getElementById("cmdCamera").addEventListener("click", doClickCamera, false);
```

Although this works, each call to `document.getElementById` traverses the entire DOM and is relatively inefficient. I would recommend that you use the app bar's `getCommandById` method instead, a change I've made throughout the modified sample included with this chapter:

```
//Using the app bar's getCommandById avoids traversing the entire DOM for each button
var appBar = document.getElementById("createAppBar").winControl;
appBar.getCommandById("cmdAdd").addEventListener("click", doClickAdd, false);
appBar.getCommandById("cmdRemove").addEventListener("click", doClickRemove, false);
appBar.getCommandById("cmdDelete").addEventListener("click", doClickDelete, false);
appBar.getCommandById("cmdCamera").addEventListener("click", doClickCamera, false);
```

Of course, if you specify a handler for each command's `onclick` property in your markup (with each one having its `supportedForProcessing` property `true`), you can avoid all of this entirely!

It should also be obvious that you can wire up events like this from anywhere in your app, and you can certainly listen to any other events you want to, especially when doing custom layouts with other UI. Also, know that the `click` event conveniently handles touch, mouse, and keyboard input alike, so you don't need to do any extra work there. In the case of the keyboard, by the way, the app bar lets you move between commands with the Tab key; Enter or Spacebar will invoke the `click` handler.

App Bar Events and Methods

In addition to the app bar's `getCommandById` method we just saw, the app bar has several other methods and a handful of events. First, the methods:

- `show` displays an app bar if its `disabled` property is `false`; otherwise, the call is ignored.
- `hide` dismisses the app bar.
- `showCommands`, `hideCommands`, and `showOnlyCommands` are used to manage command sets as described in the next section, "Showing, Hiding, Enabling, and Updating Commands."

As for events, there are a total of four that are common to the overlay-style UI controls in WinJS (that is, those that don't participate in layout):

- `beforeshow` occurs before a flyout becomes visible. For an app bar, this is a time when you could set the `commands` property depending on the state of the app at the moment or enable/disable specific commands.
- `aftershow` occurs immediately after a flyout becomes visible. For an app bar, if its `sticky` property is `true`, you can use this event to adjust the app's layout if you have a scrolling element that might be partially covered otherwise—see below.
- `beforehide` occurs before a flyout is hidden. For an app bar, you'd use this event to hide any supplemental UI created with the app bar and to readjust layout around a `sticky` app bar.

- `afterhide` occurs immediately after a flyout is hidden. For an app bar, this again could be a time to readjust the app's layout if necessary.

You can find an example of using the `show` method along with the `aftershow` and `beforehide` events in Scenario 4 of the HTML AppBar control sample.

The matter with app layout identified above (and what I kept secret in the introduction to this chapter) arises because an app bar overlays and obscures the bottom portion of the page. If that page contains a scrolling element, an app bar with `sticky` set to `true` will, for mouse users, partly cover a vertical scrollbar and will make a horizontal scrollbar wholly inaccessible. If you're using a sticky app bar with such a page, then—and because Windows Store policy does not look kindly upon discrimination against mouse users!—you should use `aftershow` to reduce the scrolling element's height by the `offsetHeight` or `clientHeight` value of the app bar control, thereby keeping the scrollbars accessible. When the app bar is hidden and `afterhide` fires, you can then readjust the layout. Always use a runtime value like `clientHeight` in these calculations as well, because it accommodates different resolution scales and because the height of an app bar can vary with commands and with view states.

To show this, Scenario 6 of the sample has a horizontally panning ListView control that normally occupies most of the page; a scrollbar will appear along the very bottom when the mouse is used. If you select an item, the app bar is made sticky and then shown (see the `doSelectedItem` function in `js/appbar-listview.js`):

```
appBar.sticky = true;
appBar.show();
```

The `show` method triggers both `beforeshow` and `aftershow` events. To adjust the layout, the appropriate event to use is `aftershow`, which makes sure the height of the app bar is valid. The sample handles this event in function called `doAppBarShow` (also in `js/appbar-listview.js`):

```
function doAppBarShow() {
    var listView = document.getElementById("scenarioListView");
    var appBar = document.getElementById("scenarioAppBar");
    var appBarHeight = appBar.offsetHeight;
    // Move the scrollbar into view if appbar is sticky
    if (appBar.winControl.sticky) {
        var listViewTargetHeight = "calc(100% - " + appBarHeight + "px)";
        var transition = {
            property: 'height',
            duration: 367,
            timing: "cubic-bezier(0.1, 0.9, 0.2, 0.1)",
            to: listViewTargetHeight
        };
        WinJS.UI.executeTransition(listView, transition);
    }
}
```

Note The sample on the Windows Developer Center uses `beforeshow` instead of `aftershow`, with the result that sometimes the app bar still has a zero height and the layout is not adjusted properly. To guarantee that the app bar has its proper height for such calculations, use the `aftershow` event as demonstrated in the modified sample included with this chapter's companion content.

Here you can see that the `appBar.offsetHeight` value is simply subtracted from the `ListView's height` with an animated transition. (See Chapter 11, "Purposeful Animations.") The operation is reversed in `doAppBarHide` where the `ListView height` is simply reset to 100% with a similar animation. In this case, the event handler doesn't depend on the app bar's height at all, so it can use either `beforehide` or `afterhide` events. If, on the other hand, you need to know the size of the app bar for your own layout, use the `beforehide` event.

As an exercise, run Scenario 7 of the SDK sample. Notice how the bottom part of the text region's vertical scrollbar is obscured by the sticky app bar. Try taking the code from Scenario 6 to handle `aftershow` and `beforehide` to adjust the text area's height to accommodate the app bar and keep the scrollbar completely visible. And no, I won't be grading you on this quiz: the solution is provided in the modified sample with this chapter.

Showing, Hiding, Enabling, and Updating Commands

In the previous section I mentioned using the `beforeshow` event to configure an app bar's `commands` property such that it contains those commands appropriate to the current page and the page state. This might include setting the `disabled` property for specific commands that are, for example, dependent on selection state. This can be done through the `commands` array, in markup, or again by using the app bar's `getCommandById` method:

```
appbar.getCommandById("cmdAdd").disabled = true;
```

Let me reiterate that the commands that appear on an app bar are specific to each page; it's not necessary to try to maintain a consistent app bar structure across pages. That is, if a command would always be disabled for a particular page, don't bother showing it at all. What's more important is that the app bar for a page is consistent, because it's a really bad idea to have commands appear and disappear depending on the state of the page. That would leave users guessing at how to get the page in the right state for certain commands to appear!

Speaking of changes, it is entirely allowable to modify or update a command at run time, which can eliminate the need to create multiple commands that you alternately show or hide. Since each command on the app bar is just a DOM element, you can really make any changes you want at any time. An example of this is shown in Scenario 3 of the sample where the app bar is initially created with a Play button (`html/custom-icons.html`):

```
<button data-win-control="WinJS.UI.AppBarCommand"
  data-win-options="{id:'cmdPlay', label:'Play', icon:'play', tooltip:'Play this song'}">
</button>
```


This button's click handler uses the `doClickPlay` function in `js/custom-icons.js` to toggle between states:

```
var isPaused = true;

function doClickPlay() {
  var cmd = document.getElementById('cmdPlay');

  if (!isPaused) {
    isPaused = true; // paused
    cmd.winControl.icon = 'play';
    cmd.winControl.label = 'Play';
    cmd.winControl.tooltip = 'Play this song';
  } else {
    isPaused = false; // playing
    cmd.winControl.icon = 'pause';
    cmd.winControl.label = 'Pause';
    cmd.winControl.tooltip = 'Pause this song';
  }
}
```

You can use something similar with a command to pin and unpin a secondary tile, as we'll see in Chapter 13. And again, the button is just an element in the DOM and updating any of its properties, including styles, will update the element on the screen once you return control to the UI thread.

Now using `beforeshow` for the purpose of adjusting your commands is certainly effective, but you can accomplish the same goal in other ways. The strategy you use depends on the architecture of your app as well as personal preference. From the user's point of view, so long as the appropriate commands are available at the right time, it doesn't really matter how the app gets them there!

Thinking through your approach is especially important when dealing with snapped view, because the recommendation is that you have ten commands or fewer so that the app bar fits on one or two rows. This means that you will want to think through how to adjust the app bar for different view states, perhaps combining multiple commands into a popup menu on a single button.

One approach is to have each page in the app declare and handle its own app bar, which includes pages that create app bars on the fly within their `ready` methods. This makes the relationship between the page content and the app bar very clear and local to the page. The downside is that common commands—those that appear on more than one page—end up being declared multiple times, making them more difficult to maintain and certainly inviting small inconsistencies like ants to sugar. Nevertheless, if you have very distinct content in your various pages and few common commands, this approach might be the right choice. It is also necessary if your app uses multiple top-level pages rather than one page with page controls, as we discussed in Chapter 3, “App Anatomy and Page Navigation,” because each top-level HTML page has to declare its own app bar anyway.

For apps using page controls, another approach is to declare a single app bar in the top-level page and set its `commands` property within each page control's `ready` method. The drawback here is that because `commands` is a write-only property, you can't declare your common commands in HTML and

append your page-specific commands later on, unless you go through the trouble of creating each individual `AppBarCommand` child element within each `ready` method. This kind of code is both tedious to write and to maintain.

Fortunately, there is a third approach that allows you to define a single app bar in your top-level page that contains *all* of your commands, for all of your pages, and then selectively show certain sets of those commands within each page's `ready` method. This is the purpose of the app bar's `showCommands`, `hideCommands`, and `showOnlyCommands` methods.

All three of these methods accept an array of commands, which can be either `AppBarCommand` objects or command id's. `showCommands` makes those commands visible and can be called multiple times with different sets for a cumulative result. On the opposite side, `hideCommands` hides the specified commands in the app bar, again with cumulative effects. The basic usage of these methods is demonstrated in Scenario 4 of the sample.

`showOnlyCommands` then combines the two, making specific commands visible while hiding all others. If you declare an app bar with all your commands, you can use `showOnlyCommands` within each page's `ready` method to quickly and easily adjust what's visible. The trick is obtaining the appropriate array to pass to the method. You can, of course, hard-code commands into specific arrays, as Scenario 4 of the sample does for `showCommands` and `hideCommands`. However, if you're thinking that this is A Classic Bad Idea, you're thinking like I'm thinking! Such arrays mean that any changes you make to app bar must happen in both HTML and JavaScript file, meaning that anyone having to maintain your code in the future will surely curse your name!

A better path to happiness and long life is thus to programmatically obtain the necessary arrays from the DOM, using the `extraClass` property on each command to effectively define command sets. This enables you to call `querySelectorAll` to retrieve those commands that belong to a particular set.

Consider the following app bar definition, where for the sake of brevity I've omitted properties like `label`, `icon`, and `section`, as well as any other styling classes:

```
<div id="appbar" data-win-control="WinJS.UI.AppBar" data-win-options="{
  commands:[
    {id:'home', extraClass: 'menuView gameView scoreView'},
    {id:'play', extraClass: 'menuView gameView scoreView'},
    {id:'rules', extraClass: 'menuView gameView scoreView'},
    {id:'scores', extraClass: 'menuView gameView scoreView'},
    {id:'newgame', extraClass: 'gameView gameSnapView'},
    {id:'resetgame', extraClass: 'gameView gameSnapView'},
    {id:'loadgame', extraClass: 'gameView gameSnapView'},
    {id:'savegame', extraClass: 'gameView gameSnapView'},
    {id:'hint', extraClass: 'gameView gameSnapView'},
    {id:'timer', extraClass: 'gameView gameSnapView'},
    {id:'pause', extraClass: 'gameView gameSnapView'},
    {id:'home2', extraClass: 'gameSnapView'},
    {id:'replaygame', extraClass: 'scoreView'},
    {id:'resetscores', extraClass: 'scoreView'}
  ]}>
</div>
```

In the `extraClass` properties we've defined four distinct sets: `menuView`, `gameView`, `gameSnapView`, and `scoreView`. With these in place, a simple call to `querySelectorAll` provides exactly the array we need for `showOnlyCommands`. A generic function like the following can then be used from within each page's `ready` method (or elsewhere) to activate commands for a particular view:

```
function updateAppBar(view) {
  var appBar = document.getElementById("appbar").winControl;
  var commands = appBar.element.querySelectorAll(view);
  appBar.showOnlyCommands(commands);
}
```

With this approach, credit for which belongs to my colleague Jesse McGatha, the app bar is wholly defined in a single location, making it very easy to manage and maintain.

App Bar Styling

The `extraClass` property for commands can, of course, be used for styling purposes as well as managing command sets. It's very simple: whatever classes you specify in `extraClass` are added to the `AppBarCommand` controls created for the app bar.

There are also seven WinJS style classes utilized by the app bar, as described in the following table, where the first two apply to the app bar as a whole and the other five to the individual commands:

CSS class (app bar)	Description
<code>win-appbar</code>	Styles the app bar container; typically this style is used as a root for more specific selectors.
<code>win-commandlayout</code>	Styles the app bar commands layout; apps generally don't modify this style at all.
CSS class (commands)	Description
<code>win-command</code>	Styles the entire <code>AppBarCommand</code> .
<code>win-commandicon</code>	Styles the icon box for the <code>AppBarCommand</code> .
<code>win-commandimage</code>	Styles the image for the <code>AppBarCommand</code> .
<code>win-commandring</code>	Styles the icon ring for the <code>AppBarCommand</code> .
<code>win-label</code>	Styles the label for the <code>AppBarCommand</code> .

Hint To help yourself styling an app bar in Blend, make it `sticky` or add a call to `show` in your page's `ready` method or your app's `activated` event. This makes sure that the app bar is visible and navigable in Blend; it can otherwise be difficult to get the app bar to show within the tool.

Generally speaking, you don't need to override the `win-appbar` or `win-commandlayout` styles directly; instead, you should create selectors for a custom class related to these and then style the particular pieces you need. This can include pseudo-selectors like `button:hover`, `button:active`, and so forth.

Scenario 2 of the HTML AppBar Control sample shows many such selectors in action, in this case to set the background of the app bar and its commands to blue and the foreground color to green (a somewhat hideous combination, but demonstrative nonetheless).

As a basis, Scenario 2 (`html/custom-color.html`) adds a CSS class *customColor* to the app bar:

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

In `css/custom-color.css` it then styles selectors based on `.win-appbar.customColor`. The following rules, for instance, set the overall background color, the label text color, and the color of the circle around the commands for the `:hover` and `:active` states:

```
.win-appbar.customColor {  
    background-color: rgb(20, 20, 90);  
}  
.win-appbar.customColor .win-label {  
    color: rgb(90, 200, 90);  
}  
.win-appbar.customColor button:hover .win-commandring,  
.win-appbar.customColor button:active .win-commandring {  
    background-color: rgba(90, 200, 90, 0.13);  
    border-color: rgb(90, 200, 90);  
}
```

All of this styling, by the way, applies only to the standard command-oriented layout. If you're using a custom layout, the app bar just contains whatever elements you want with whatever style classes you want, so you just handle styling as you would any other HTML.

Custom Icons

Earlier we saw that the `icon` property of an `AppBarCommand` typically comes from the Segoe UI Symbol font. Although this is suitable for most needs, you might want at times to use a character from a different font (some of us just can't get away from Wingdings!) or to provide custom graphics. The app bar supports both.

To use a different font for the whole app bar, simply add a class to the app bar and create a rule based on `win-appbar`:

```
win-appbar.customFont {  
    font-family: "Wingdings";  
}
```

To change the font of a specific command button, add a class to its `extraClass` property (such as *customButtonFont*) and create a rule with the following selector (as used in Scenario 1 of the modified sample):

```
button.customButtonFont .win-commandimage {  
    font-family: "Wingdings";  
}
```

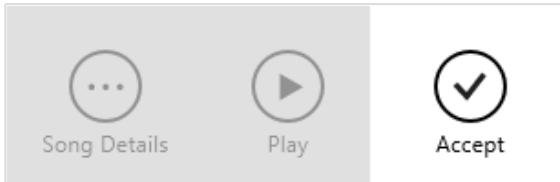
To provide graphics of your own, do the following for a 100% resolution scale:

- Create a 160x80 pixel png sprite image with a transparent background, saving the file with the *.scale-100* suffix in the filename.
- This sprite is divided into two rows of four columns—that is, 40x40 pixel cells. The top row is for the light styling theme, and the bottom is for the dark theme.
- Each row has four icons for the following button states, in this order from left to right: default (rest), hover, pressed (active), and disabled.
- Each image is centered in its respective 40x40 cell, but remember that a ring will be drawn around the icon, so generally keep the image in the 20–30 pixel range vertically and horizontally. It can be wider or taller in the middle areas, of course, where the ring is widest.

For other resolution scales, multiple the sizes by 1.4 (140%) and 1.8 (180%) and use the *.scale-140* and *.scale-180* suffixes in the image filename.

To use the custom icon, point the command's `icon` property to the base image URI (without the *.scale-1x0* suffixes)—for instance, `icon: 'url(images/icon.png)'`.

Scenario 3 of the HTML AppBar Control sample demonstrates custom icon graphics for an Accept button:



The icon comes from a file called `accept.png`, which appears something like this—I've adjusted the brightness and contrast and added a border so that you can see each cell clearly:



The declaration for the app bar button then appears as follows (some properties omitted for brevity):

```
<button data-win-control="WinJS.UI.AppBarCommand"
  data-win-options="{id:'cmdAccept', label:'Accept', icon:'url(images/accept.png)'}">
```

Note that although the sample doesn't have variations of the icon for resolution scales, it does provide variants for high contrast themes, an important accessibility consideration that we'll come back to in Chapter 17. For this reason, the `button` element includes `style="-ms-high-contrast-adjust:none"` to override automatic adjustments for high contrast.

Command Menu

The next aspect of an app bar we need to explore in a little more depth are those commands whose `type` property is set to `flyout`. In this case the command's `flyout` property must identify a `WinJS.UI.Flyout` object or a `WinJS.UI.Menu` control (which is a flyout). As noted before, flyout or popup menus like this are used when there are too many related commands cluttering up the basic app bar, or when you need other types of controls that aren't quite appropriate on the app bar itself. It's said, though, that if you're tempted to use a button labeled "More", "Advanced", or "Other Stuff" because you can't figure out how to organize the commands otherwise, it's a good sign that the app itself is too complex! Seek ways to simplify the app's purpose so that the app bar doesn't just become a repository for randomness.

We'll be covering flyouts more fully a little later in this chapter, but let's see how to use one in an app bar, as demonstrated in Scenario 6 of the [HTML flyout control sample](#) (not the app bar sample, mind you!):



In `html/appbar-flyout.html` of this sample we see the app bar button declared as follows:

```
<button data-win-control="WinJS.UI.AppBarCommand"
  data-win-options="{id:'respondButton', label:'Respond', icon:'edit', type:'flyout',
  flyout:'respondFlyout'}">
```

The `respondFlyout` element identified here is defined earlier in `html/appbar-flyout.html`; note that such an element must be declared prior to the app bar to make sure it's instantiated *before* the app bar is created:

```
<div id="respondFlyout" data-win-control="WinJS.UI.Menu">
  <button data-win-control="WinJS.UI.MenuCommand"
    data-win-options="{id:'alwaysSaveMenuItem',
      label:'Always save drafts', type:'toggle', selected:'true'}">
  </button>
  <hr data-win-control="WinJS.UI.MenuCommand"
    data-win-options="{id:'separator', type:'separator'}" />
  <button data-win-control="WinJS.UI.MenuCommand"
    data-win-options="{id:'replyMenuItem', label:'Reply'}">
  </button>
  <button data-win-control="WinJS.UI.MenuCommand"
    data-win-options="{id:'replyAllMenuItem', label:'Reply All'}">
```

```

</button>
<button data-win-control="WinJS.UI.MenuCommand"
        data-win-options="{id: 'forwardMenuItem', label: 'Forward'}">
</button>
</div>

```

It should come as no surprise by now that the menu is just another WinJS control, `WinJS.UI.Menu`, where its child elements define the menu's contents. As all these elements are, once again, just elements in the DOM; their `click` events are wired up in `js/appbar-flyout.js` with the ever-present `addEventListener`. (Again, the sample uses `document.getElementById` to obtain the elements in order to call `addEventListener`; it would be more efficient to use the app bar's `getCommandById` method instead as in the modified app bar sample.)

Each menu item, as you can see, is a `WinJS.UI.MenuCommand` object, and we'll come back to the details later—for the time being, you can see that those items have an `id`, a `label`, and a `type`, very similar to `WinJS.UI.AppBarCommand` objects.

That's pretty much all there is to it—the one added bit is that when a menu item is selected, you'll want to dismiss the menu and perhaps also the app bar (if it's not sticky). This is shown in the sample within `js/appbar-flyout.js` in a function called `hideFlyoutAndAppBar`:

```

function hideFlyoutAndAppBar() {
    document.getElementById("respondFlyout").winControl.hide();
    document.getElementById("appBar").winControl.hide();
}

```

Custom App Bars and Navigation Bars

All this time we've been looking at the *standard commands layout* of the app bar, which is of course the simplest way to use the control. There will be times, however, when the standard commands layout isn't sufficient. Perhaps you want to place more interesting controls on the app bar, especially custom controls (like a color selector). For this you set the app bar's `layout` property to `'custom'` and place whatever HTML you want within the app bar control, styling it with CSS, and wiring up whatever events you need in JavaScript.

A custom layout is also typically used to implement a top navigation bar—that is, the app bar with `placement` set to `'top'`—because command buttons aren't usually the UI you want. We saw an example earlier in the Weather app, and the navigation bar of Internet Explorer provides another:



Our good friend the [HTML AppBar control sample](#) again delivers an example of custom layout, in Scenario 5. In `html/custom-layout.html` we see the markup for a custom top app bar containing arbitrary elements:

```
<div id="customLayoutAppBar" data-win-control="WinJS.UI.AppBar" aria-label="Navigation Bar"
    data-win-options="{layout:'custom', placement:'top'}">
  <header aria-label="Navigation bar" role="banner">
    <button id="cmdBack" class="win-backbutton" aria-label="Back">
  </button>
    <div class="titleArea">
      <h1 class="win-type-xx-large" tabindex="0">
        Page Title</h1>
      </div>
    </header>
  </div>
```

Admittedly, the result of this example is a little odd—it creates a navigation bar with a typical page header with a back button where each control might have a focus rectangle. I don’t recommend following this design!



As mentioned in the “Tips and Tricks” section in Chapter 4 (under “Control Styling”), you can suppress the focus rectangle with a `<selector>:focus { outline: none; }` rule in CSS. To remove it from the back button, for example, you can add the style to the following rule in `css/custom-layout.css`:

```
.win-appbar header .win-backbutton {
  margin-left: 39px;
  margin-top: 59px;
  outline: none;
}
```

Notice again how this rule and the others in `css/custom-layout.css` all use the `win-appbar` class as a base selector but only because it’s styling other generic classes like `header` and `win-backbutton`. If you use specific classes in your app bar or navigation bar, you really don’t need the `win-appbar` selector at all.

To implement a navigation bar like that of Internet Explorer or the Weather app, you can certainly use a `ListView` control along with item templates or custom item rendering functions, where you’d wire up `itemInvoked` events to `WinJS.Navigation.navigate` and so forth. Again, there’s nothing particularly special or complicated here: with a custom layout, the app bar is really just a flyout container for other HTML elements.

Flyouts and Menus

Going back to our earlier discussion about where to place commands, a flyout control—`WinJS.UI.Flyout`—is used for confirmations, collecting information, and otherwise answering questions in response to a user action. The menu control—`WinJS.UI.Menu`—is then a particular kind of flyout that contains `WinJS.UI.MenuCommand` controls rather than arbitrary HTML. In fact, `WinJS.UI.Menu` is directly derived from `WinJS.UI.Flyout` using `WinJS.Class.define`, so they share much in common. As flyouts, they also share some feature in common with the app bar. (Both the app bar and the flyout classes are themselves derived from a `WinJS.UI._Overlay` base class that is internal to WinJS.)

Tip In addition to the `WinJS.UI.Flyout` object that you'll employ from an app, there is also a system flyout that appears in response to some API calls, such as creating or removing a secondary tile (see Chapter 13, specifically Figure 13-5 and the “Secondary Tiles” section). Although visually the same, the system flyout will trigger a `blur` event to the app whereas the WinJS flyout, being part of the app, does not. As a result, a system flyout will cause a non-sticky app bar to be dismissed. To prevent this, it's necessary to set the appbar's `sticky` property to `true` before calling APIs with system flyouts. This is demonstrated in Scenario 7 of the [Secondary tiles sample](#).

Before we look at the details, let's see a number of visual examples from the [HTML flyout control sample](#) in which we already saw a popup menu on an app bar command. The `WinJS.UI.Flyout` controls used in Scenarios 1–4 are shown in Figure 7-1 (on the next page). Notice the variance of content in the flyout itself and how the flyout is always positioned near the control that invoked it, such as the Buy, Login, and Format output text buttons and the Lorem ipsum hyperlink text. These examples illustrate that a flyout can contain a simple message with a button (Scenario 1, for warnings and confirmations), can contain fields for entering information or changing settings (Scenarios 2 and 3), and can have a title (Scenario 4). Scenario 5, for its part, contains the example of a popup header menu with `WinJS.UI.Menu` that we'll see a little later.

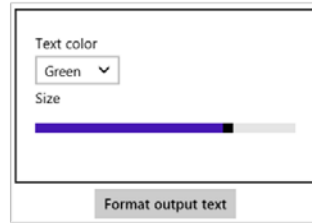
There are two key characteristics of flyout controls, including menus. One is that flyouts can be dismissed programmatically, like an app bar, when an appropriate control within the flyout is invoked. This is the case with the Complete Order button of Scenario 1 and the Login button of Scenario 2.

The second characteristic, also shared with the app bar, is the light dismiss behavior: clicking or tapping outside the control dismisses it, as does the ESC key, which means light dismiss is the equivalent of pressing a Cancel or Close button in a traditional dialog box. The benefit here is that we don't need a visible button for this purpose, which helps simplify the UI. At the same time, notice in Scenario 3 of Figure 7-1 that there is no OK button or other control to confirm changes you might make in the flyout. With this particular design, changes are immediately applied such that dismissing the flyout does not reverse or cancel them. If you don't want that kind of behavior, you can place something like an Apply button on the flyout and not make changes until that button is pressed. In this case, dismissing the flyout would cancel the changes.

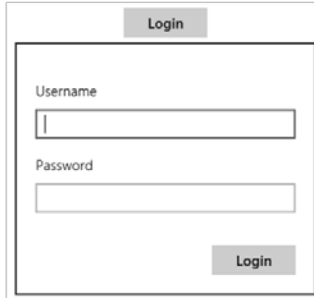
Scenario 1: confirmation flyout



Scenario 3: adjusting settings



Scenario 2: gathering information



Scenario 4: flyout with title

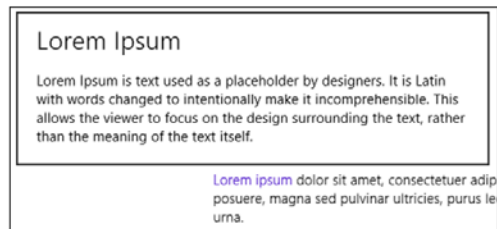


Figure 7-1 Examples of flyout controls from the HTML flyout control sample.

I'll again encourage you to read the [Guidelines and checklist for Flyouts](#) topic that goes into detail about how and when to use the different designs that are possible with this control. It also outlines when *not* to use the control: for example, to surface errors not related to user action (use a message dialog instead), for primary commands (use the app bar), for text selection context menus, and for UI that is part of a workflow and should be directly on the app canvas. These guidelines also suggest keeping a flyout small and focused (omitting unnecessary controls) and making sure a flyout is positioned close to the object that invoked it. Let's now see how that works in the code.

Note In addition to apps that display a `WinJS.UI.Flyout` on their own, some system APIs (such as that to create a secondary tile) create a system flyout. In these cases, the app will receive a `blur` event, which will cause any light dismiss app bars to be dismissed. To prevent this, set the app bar to `sticky` when using those APIs.

WinJS.UI.Flyout Properties, Methods, and Events

Most of the properties, methods, and events of the `WinJS.UI.Flyout` control are exactly the same as we've already seen for the app bar. The `show` and `hide` methods control its visibility, a `hidden` property indicates its visible state, and same the `beforeshow`, `aftershow`, `beforehide`, and `afterhide` events fire as appropriate. The `afterhide` event is typically used to detect dismissal of the flyout.

Like the app bar, the flyout also has a `placement` property, but it has different values that are only meaningful in the context of the flyout's `alignment` and `anchor` properties. In fact, all three properties are optional parameters to the `show` method because they determine where, exactly, the flyout appears on the screen; the default `placement` and `alignment` can also be set on the control itself because these

are optional with `show`. (Note also that if you don't specify an anchor in the `show` method; the `anchor` property must already be set on the control or `show` will throw an exception.)

The `anchor` property identifies the control that invokes the flyout or whatever other operation might bring up a flyout (as for confirmation). The `placement` property then indicates how the flyout should appear in relation to the `anchor`: `'top'`, `'bottom'`, `'left'`, `'right'`, or `'auto'` (the default). Typically, you use a specific `placement` only if you don't want the flyout to possibly obscure important content. Otherwise, you run the risk of the flyout element being shrunk down to fit the available space. The flyout's `content` will remain the same size, mind you, so it means that—ick!—you'll get scrollbars! So, unless you have a really good reason and a note from your doctor, stick with `'auto'` placement so that the control will be placed where it can be shown full size. Along these same lines, remember that in snapped view you have only 320 horizontal pixels to work with, meaning that flyouts you show in that view state should be that size or smaller.

The `alignment` property, for its part, when used with a `placement` of `'top'` or `'bottom'`, determines how the flyout aligns to the edge of the `anchor`: `'left'`, `'right'`, or `'center'` (the default). The content of the flyout itself is aligned through CSS as with any other HTML.

If you need to style the flyout control itself, you can set styles in the `win-flyout` class, like fonts, default alignments, margins, and so on. As with other WinJS style classes like this, use `win-flyout` as a basis for more specific selectors unless you really want to style every flyout in the app. Typically, in fact, you also exclude `win-menu` from the rule so that menu flyouts aren't affected by such styling. For example, most of the scenarios in the HTML flyout control sample, which we'll be looking at next, have rules like this:

```
.win-flyout:not(.win-menu) button,
.win-flyout:not(.win-menu) input[type="button"] {
    margin-top: 16px;
    margin-left: 20px;
    float: right;
}
```

Finally, if for some reason you need to know when a flyout is loaded, listen to the `DOMNodeInserted` method on `document.body`:

```
document.body.addEventListener("DOMNodeInserted", insertionHandler, false);
```

Flyout Examples

A flyout control is created like any other WinJS control with `data-win-control` and `data-win-options` attributes and processed by `WinJS.UI.process/processAll`. Flyouts with relatively fixed content will typically be declared in markup where you can use data binding on specific properties of the elements within the flyout. Flyouts that are very dynamic, on the other hand, can be created directly from code by using `new WinJS.UI.Flyout(<element>, <options>)`, and you can certainly change its child elements at any time. It's all just part of the DOM! (Am I repeating myself?)

Like I said before (apparently I am repeating myself), a `WinJS.UI.Flyout` control can contain arbitrary HTML, styled as always with CSS. The flyout for Scenario 1 in the sample appears as follows in `html/confirm-action.html` (condensed slightly):

```
<div id="confirmFlyout" data-win-control="WinJS.UI.Flyout" aria-label="{Confirm purchase flyout}">
  <div>Your account will be charged $252.</div>
  <button id="confirmButton">Complete Order</button>
</div>
```

The login flyout in Scenario 2 is similar, and it even employs an HTML form to attach the Login button to the Enter key (`html/collect-information.html`):

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

The flyout is displayed by calling its `show` method. In Scenario 1, for instance, the button's `click` event is wired to the `showConfirmFlyout` function (`js/confirm-action.js`), where the Buy button is given as the anchor element. Handling the Complete Order button just happens through a `click` handler attached to that element, and here we want to make sure to call `hide` to programmatically dismiss the flyout. Finally, the `afterhide` event is used to detect dismissal:

```
var bought;
```

```
var page = WinJS.UI.Pages.define("/html/confirm-action.html", {
  ready: function (element, options) {
    document.getElementById("buyButton").addEventListener("click",
      showConfirmFlyout, false);
    document.getElementById("confirmButton").addEventListener("click",
      confirmOrder, false);
    document.getElementById("confirmFlyout").addEventListener("afterhide",
      onDismiss, false);
  }
});

function showConfirmFlyout() {
  bought = false;
  var buyButton = document.getElementById("buyButton");
  document.getElementById("confirmFlyout").winControl.show(buyButton);
}
```

```
// When the Buy button is pressed, hide the flyout since the user is done with it.
```

```

function confirmOrder() {
    bought = true;
    document.getElementById("confirmFlyout").winControl.hide();
}

// On dismiss of the flyout, determine if it closed because the user pressed the buy button.
// If not, then the flyout was light dismissed.
function onDismiss() {
    if (!bought) {
        // (Sample displays a dismissal message on the canvas)
    }
}

```

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

As the flyout of Scenario 2 is a little larger, the default `placement` of `'auto'` on a 1366x768 display (as in the simulator) makes it appear below the button that invokes it. There isn't quite enough room above that button. So try setting `placement` to `'top'` in the call to `show`:

```

function showLoginFlyout() {
    // ...
    document.getElementById("loginFlyout").winControl.show(loginButton, "top");
}

```

Then you can see how the flyout gets scrollbars because the overall control element is too short:

What was that word I used before? “lck”?

To move on, Scenario 3 again declares a flyout in markup, where it contains some `label`, `select`, and `input` controls. In JavaScript, though, it listens for change events on the latter and applies those new values to the output element on the app canvas:

```
var page = WinJS.UI.Pages.define("/html/change-settings.html", {
  ready: function (element, options) {
    // ...
    document.getElementById("textColor").addEventListener("change", changeColor, false);
    document.getElementById("textSize").addEventListener("change", changeSize, false);
  }
});

// Change the text color
function changeColor() {
  document.getElementById("outputText").style.color =
    document.getElementById("textColor").value;
}

// Change the text size
function changeSize() {
  document.getElementById("outputText").style.fontSize =
    document.getElementById("textSize").value + "pt";
}
```

If this flyout had an Apply button rather than applying the changes immediately, its `click` handler would obtain the current selection and slider values and use them like `changeColor` and `changeSize` do.

Finally, in Scenario 4 we see a flyout with a title, which is just a piece of larger text in the markup; the flyout control itself doesn’t have a separate notion of a header:

```
<div id="moreInfoFlyout" data-win-control="WinJS.UI.Flyout" aria-label="{More info flyout}">
  <div class="win-type-x-large">Lorem Ipsum</div>
  <div>
    Lorem Ipsum is text used as a placeholder by designers...
  </div>
</div>
```

The point of this last example is to show that unlike traditional desktop dialog boxes, flyouts don't often need a title because they already have context within the app itself. Dialog boxes in desktop applications need titles because that's what appears in task-switching UI alongside other apps.

Hint If you find that `beforeshow`, `aftershow`, `beforehide`, or `afterhide` events triggered from a flyout are getting propagated to a containing app bar, which shares the same event names, include a call to `eventArgs.stopPropagation()` inside your flyout's handler.

Menus and Menu Commands

What distinguishes a `WinJS.UI.Menu` control from a more generic `WinJS.UI.Flyout` is that a menu expects that all its child elements are `WinJS.UI.MenuCommand` objects, similar to how the standard command layout of the app bar expects `AppBarCommand` objects (and won't instantiate if you declare something else). In fact, the menu control shares other characteristics with the app bar as well as the flyout, such as:

- `show` and `hide` methods.
- `getCommandById`, `showCommands`, `hideCommands`, and `showOnlyCommands`, along with the `commands` property, meaning that you can use the same strategies to manage commands as discussed in "Showing, Hiding, Enabling, and Updating Commands" in the app bar section, including specifying commands using a JSON array rather than discrete elements.
- `beforeshow`, `aftershow`, `beforehide`, and `afterhide` events.
- `anchor`, `alignment`, and `placement` properties.

The menu also has two styles for its appearance—`win-menu` and `win-command`—that you use to create more specific selectors, as we've seen, for the entire menu and for the individual text commands.

`MenuCommand` objects are also very similar to `AppBarCommand` objects. Both share many of the same properties: `id`, `label`, `type` ('button', 'toggle', 'flyout', and 'separator'), `disabled`, `extraClass`, `flyout`, `hidden`, `onclick`, and `selected`. Menu commands do not have icons, sections, and tooltips but you can see from `type` that menu items can be buttons (including just text items), checkable items, separators, and also another flyout. In the latter case, the secondary menu will replace the first rather than show up alongside, and to be honest, I've yet to see secondary menus used in a real app. Still, it's supported in the control!

We've already seen how to use a flyout menu from an app bar command, which is covered in Scenario 6 of the HTML flyout control sample (see the earlier "Command Menus" section). Another primary use case is to provide what looks like drop-down menu from a header element, covered Scenario 5. Here (see `html/header-menu.html`), the standard design is to place a down chevron symbol () at the end of the header:

```
<header aria-label="Header content" role="banner">
  <button class="win-backbutton" aria-label="Back"></button>
  <div class="titlearea win-type-ellipsis">
```

```

        <button class="titlecontainer">
            <h1>
                <span class="pagetitle">Music</span>
                <span class="chevron win-type-x-large">&#xe099</span>
            </h1>
        </button>
    </div>
</header>

```

Notice that the whole header is wrapped in a `button`, so its `click` handler can display the menu with `show`:

```

document.querySelector(".titlearea").addEventListener("click", showHeaderMenu, false);

function showHeaderMenu() {
    var title = document.querySelector("header .titlearea");
    var menu = document.getElementById("headerMenu").winControl;
    menu.anchor = title;
    menu.placement = "bottom";
    menu.alignment = "left";
    menu.show();
}

```

The flyout (defined as `headerMenu` in `html/header-menu.html`) appears when you click anywhere on the header (not just the chevron, as that's just a character in the header text):



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

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

```

<div id="secondaryMenu" data-win-control="WinJS.UI.Menu">
    <button data-win-control="WinJS.UI.MenuCommand"
        data-win-options="{id:'command1', label:'Command 1'}"></button>
    <button data-win-control="WinJS.UI.MenuCommand"
        data-win-options="{id:'command2', label:'Command 2'}"></button>
    <button data-win-control="WinJS.UI.MenuCommand"
        data-win-options="{id:'command3', label:'Command 3'}"></button>
</div>

```

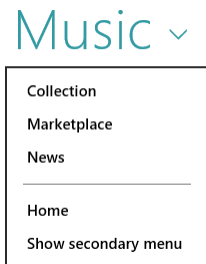


```

<div id="headerMenu" data-win-control="WinJS.UI.Menu">
  <!-- ... -->
  <button data-win-control="WinJS.UI.MenuCommand"
    data-win-options="{id:'showFlyout', label:'Show secondary menu',
      type:'flyout', flyout:'secondaryMenu'}">
  </button>
</div>

```

Also, go into `css/header-menu.css` and adjust the `width` style of `#headerMenu` to 200px. With these changes, the first menu will appear as follows where the color change in the header is the hover effect:



When you select *Show secondary menu*, the first menu will be dismissed and the secondary one will appear in its place:



Another example of a header flyout menu can be found in the [Adaptive layout with CSS sample](#) we saw in Chapter 6, “Layout.” It’s implemented the same way we see above, with the added detail that it actually changes the page contents in response to a selection.

Context Menus

Besides the flyout menu that we’ve seen so far, there are also context menus as described in [Guidelines and checklist for context menus](#). These are specifically used for commands that are directly relevant to a selection of some kind, like clipboard commands for text, and are invoked with a right mouse click on that selection, a tap, or the context menu key on the keyboard. Text and hyperlink controls already provide these by default. Context menus are also good for providing commands on objects that cannot be selected (like parts of an instant messaging conversation), as app bar commands can’t be context-ually sensitive to such items. They’re also recommended for actions that cannot be accomplished with a direct interaction of some kind. However, don’t use them on page backgrounds—that’s what the app bar is for because the app bar will automatically appear with a right-click gesture.

Hint If you process the right mouse button click event for an element, be aware that the default behavior to show the app bar will be suppressed over that element. Therefore, use the right-click event judiciously, because users will become accustomed to right-clicking around the app to bring up the app bar. Note also that you can programmatically invoke the app bar yourself using its [show](#) method.

The [Context menu sample](#) gives us some context here—I know, it’s a bad pun! In all cases, you need only listen to the HTML `contextmenu` event on the appropriate element; you don’t need to worry about mouse, touch, and keyboard separately. Scenario 1 of the sample, for instance, has a nonselectable `attachment` element on which it listens for the event (`html/scenario1.html`):

```
document.getElementById("attachment").addEventListener("contextmenu",
    attachmentHandler, false);
```

In the event handler, you then create a [Windows.UI.Popups.PopupMenu](#) object (which comes from WinRT, not WinJS!), populate it with [Windows.UI.Popups.UICommand](#) objects (that contain an item label and click handler) or [UICommandSeparator](#) objects, and then call the menu’s `showAsync` method (`js/scenario1.js`):

```
function attachmentHandler(e) {
    var menu = new Windows.UI.Popups.PopupMenu();
    menu.commands.append(new Windows.UI.Popups.UICommand("Open with", onOpenWith));
    menu.commands.append(new Windows.UI.Popups.UICommand("Save attachment",
        onSaveAttachment));

    menu.showAsync({ x: e.clientX, y: e.clientY }).done(function (invokedCommand) {
        if (invokedCommand === null) {
            // The command is null if no command was invoked.
        }
    });
}
```

Notice that the results of the `showAsync` method⁴⁰ is the `UICommand` object that was invoked; you can examine its `id` property to take further action. Also, the parameter you give to `showAsync` is a [Windows.Foundation.Point](#) object that indicates where the menu should appear relative to the mouse pointer or the touch point. The menu is placed above and centered on this point.

The `PopupMenu` object also supports a method called `showForSelectionAsync`, whose first argument is a [Windows.Foundation.Rect](#) that describes the applicable selection. Again, the menu is placed above and centered on this rectangle. This is demonstrated in Scenario 2 of the sample in `js/scenario2.js`:

⁴⁰ The sample actually calls `then` and not `done` here. If you’re wondering why such inconsistencies exist, it’s because the `done` method was introduced mid-way during the production of Windows 8 when it became clear that we needed a better mechanism for surfacing exceptions within chained promises. As a result, numerous SDK samples and code in the documentation still use `then` instead of `done` when handling the last promise in a chain. It still works; it’s just that exceptions in the chain will be swallowed, thus hiding possible errors.

```
//In the contextmenu handler
menu.showForSelectionAsync(getSelectionRect()).then(function (invokedCommand) { //... }
//...

function getSelectionRect() {
    var selectionRect = document.selection.createRange().getBoundingClientRect();

    var rect = {
        x: getClientCoordinates(selectionRect.left),
        y: getClientCoordinates(selectionRect.top),
        width: getClientCoordinates(selectionRect.width),
        height: getClientCoordinates(selectionRect.height)
    };
    return rect;
};
```

This scenario also demonstrates that you can use a `contextmenu` event handler on text to override the default commands that such controls otherwise provide.

A final note for context menus: because these are created with WinRT APIs and are not WinJS controls, the menus don't exist in the DOM and are not DOM-aware, which explains the use of other WinRT constructs like `Point` and `Rect`. Such is also true of message dialogs, which is our final subject for this chapter.

Message Dialogs

Our last piece of commanding UI for this chapter is the message dialog. Like the context menu, this flyout element comes not from WinJS but from WinRT via the [Windows.UI.Popups.MessageDialog](#) API. Again, this means that the message dialog simply appears on top of the current page and doesn't participate in the DOM. Message dialogs automatically dim the app's current page and block input events from the app until the user responds to the dialog.

The [Guidelines and checklist for message dialogs](#) topic explains the use cases for this UI:

- To display urgent information that the user must acknowledge to continue, especially conditions that are not related to a user command of some kind.
- Errors that apply to the overall app, as opposed to a workflow where the error is better surfaced inline on the app canvas. Loss of network connectivity is a good example of this.
- Questions that *require* user input and cannot be light dismissed like a flyout. That is, use a message dialog to block progress when user input is essential to continue.

The interface for message dialogs is very straightforward. You create the dialog object with a `new Windows.UI.Popups.MessageDialog`. The constructor accepts a required string with the message content and an optional second string containing a title. The dialog also has `content` and `title` properties that you can use independently. In all cases the strings support only plain text.

You then configure the dialog through its [commands](#), [options](#), [defaultCommandIndex](#) (the command tied to the Enter key), and [cancelCommandIndex](#) (the command tied to the ESC key).

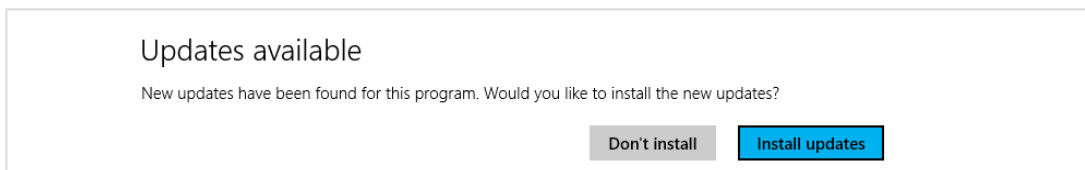
The [options](#) come from the [Windows.UI.Popups.MessageDialogOptions](#) enumeration where there are only two members: [none](#) (the default, for no special behavior) and [acceptUserInputAfterDelay](#) (which causes the message dialog to ignore user input for a short time to prevent possible clickjacking; this exists primarily for Internet browsers loading arbitrary web content and isn't typically needed for most apps).

The [commands](#) property then contains up to three [Windows.UI.Popups.UICommand](#) objects, the same ones used in context menus. Each command again contains an [id](#), a [label](#), and an [invoked](#) property to which you assign the handler for the command. Note that the [defaultCommandIndex](#) and [cancelCommandIndex](#) properties work on the indices of the [commands](#) array, not the [id](#) properties of those commands. Also, if you don't add any commands of your own, the message dialog will default to a single Close command.

Finally, once the dialog is configured, you display it with a call to its [showAsync](#) method. Like the context menu, the result is the selected [UICommand](#) object that's given to the completed handler you provide to the promise's [done](#) method. Typically, you don't need to obtain that result because the selected command will have triggered its associated [invoked](#) handler where you process those commands.

Note If the Search, Share, Devices, or Settings charm is invoked while a message dialog is active, or if an app is activated to service a contract, a message dialog will be dismissed without any command being selected. The completed handler for [showAsync](#) will be called, however, with the result set to the default command. Be aware of this if you're using the completed handler to process such commands.

The [Message dialog sample](#)—one of the simplest samples in the whole Windows SDK!—demonstrates various uses of this API. Scenario 1 displays a message dialog with a title and two command buttons, setting the second command (index 1) as the default. This appears as follows:



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



Scenario 3 is identical to Scenario 1 but uses the completed handler of the `showAsync().done` method to process the selected command. You can use this to see the effect of invoking a charm while the dialog is shown.

Finally, Scenario 4 assigns the first command to be the default and marks the second as the cancel command, so the message is dismissed with that command or the ESC key:



And that's really all there is to it!

Improving Error Handling in Here My Am!

To complete this chapter and bring together much of what we've discussed, let's make some changes to Here My Am!, last seen in Chapter 3, to improve its handling of various error conditions. As it stands right now, Here My Am! doesn't behave very well in a few areas:

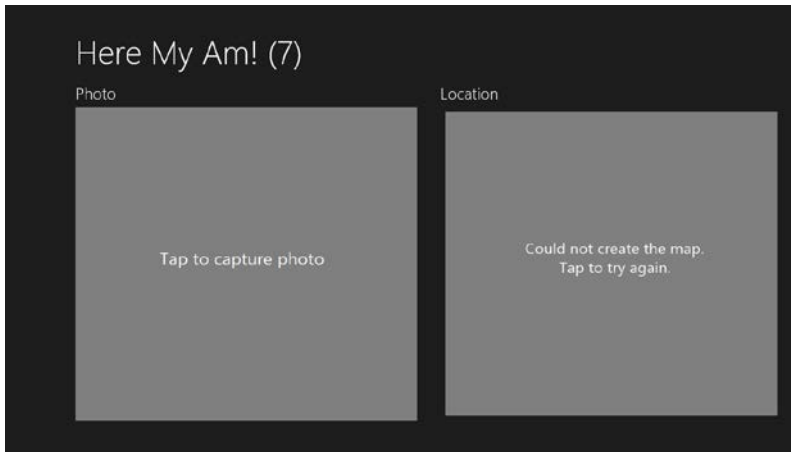
- If the Bing Maps control script fails to load from a remote source, the code in `html/map.html` just throws an exception and the app terminates.
- If we're using the app on a mobile device and have changed our location, there isn't a way to refresh the location on the map other than dragging the pin; that is, the geolocation API is used only on startup.
- When WinRT's geolocation API is trying to obtain a location without a network connection, a several-second timeout period occurs, during which the user doesn't have any idea what's happening.
- If our attempt to use WinRT's geolocation API fails, typically due to timeout or network connectivity problems, but also possibly due to a denial of user consent, there isn't any way to try again.

The Here My Am! (7) app for this chapter addresses these concerns. First, I've added an error image to the `html/map.html` file (the image is in `html/maperror.png`) so that a failure to load the Bing maps script will display a message in place of the map (the display style of `none` is removed in that case):

```

```

I've also added a `click` handler to the image that reloads the `iframe` contents with `document.location.reload(true)`. With this in place, I can remove the exceptions that were previously raised when the map couldn't be created, preventing the app from being terminated. Here's how it looks if the map can't be created:



To test this, you need to disconnect from the Internet, uninstall the app (to clear any cached map script; otherwise, it will continue to load!), and run the app again. It should hit the error case at the beginning of the `init` method in `html/map.html`, which shows the error image by removing the default `display: none` style and wiring up the `click` handler. Then reconnect the Internet and click the image, and the map should reload, but if there are continued issues the error message will again appear.

The second problem—adding the ability to refresh our location—is easily done with an app bar. I’ve added such a control to `default.html` with one command:

```
<div id="appbar" data-win-control="WinJS.UI.AppBar" data-win-options="">
  <button data-win-control="WinJS.UI.AppBarCommand"
    data-win-options="{id:'cmdRefreshLocation', label:'Refresh location',
      icon:'globe', section:'global', tooltip:'Refresh your location'}">
  </button>
</div>
```

This command is wired up within `pages/home/home.js` in the page control’s `ready` method:

```
var appbar = document.getElementById("appbar").winControl;
appbar.getCommandById("cmdRefreshLocation").addEventListener("click", this.tryRefresh.bind(this));
```

where the `tryRefresh` handler, also in the page control, hides the app bar and calls another new method, `refreshPosition`, where I moved the code that obtains the geolocation and updates the map:

```
tryRefresh: function () {
  //Hide the app bar and retry
  var appbar = document.getElementById("appbar").winControl.hide();
  this.refreshPosition();
},
```

I also needed to tweak the `pinLocation` function within `html/map.html`. Without a location refresh command, this function was only ever called once on app startup. Since it can now be called multiple times, we need to remove any existing pin on the map before adding one for the new location. This is done with a call to `map.entities.pop` prior to the existing call to `map.entities.push` that pins the new location.

The app bar now appears as follows, and we can refresh the location as needed. (If you aren't on a mobile device in your car, try dragging the first pin to another location and then refreshing to see the pin return to your current location.)



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

```
<div id="retryFlyout" data-win-control="WinJS.UI.Flyout" aria-label="{Trying geolocation}"
    data-win-options="{anchor: 'map', placement: 'bottom', alignment: 'center'}">
  <div class="win-type-large">Attempting to obtain geolocation...</div>
</div>
```

The `refreshPosition` function that we just added to `pages/home/home.js` makes a great place to display the flyout just before calling `getGeopositionAsync`:

```
refreshPosition: function () {
    document.getElementById("retryFlyout").winControl.show();
    var gl = new Windows.Devices.Geolocation.Geolocator();

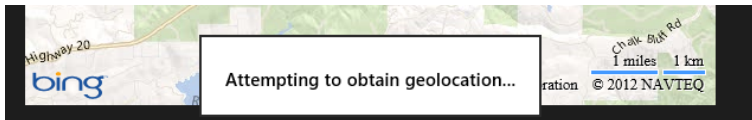
    gl.getGeopositionAsync().done(function (position) {
        //...

        //Always hide the flyout
        document.getElementById("retryFlyout").winControl.hide();

        //...
    }, function (error) {
        //...

        //Always hide the flyout
        document.getElementById("retryFlyout").winControl.hide();
    });
},
```

Note that we want to hide the flyout inside the completed and error handlers so that the message stays visible while the async operation is happening. If we placed a single call to hide outside these handlers, the message would flash only very briefly before being dismissed, which isn't what we want. As we've written it, the user will have enough time to see the notice along the bottom of the map (subject to light dismiss):



The last piece is to notify the user when obtaining geolocation fails. We could do this with another flyout with a Retry button, or with an inline message as below. We would *not* use a message dialog in this case, however, because the message could appear in response to a user-initiated refresh action. A message dialog might be allowable on startup, but with an inline message combined with the flyout we already added we have all the bases covered.

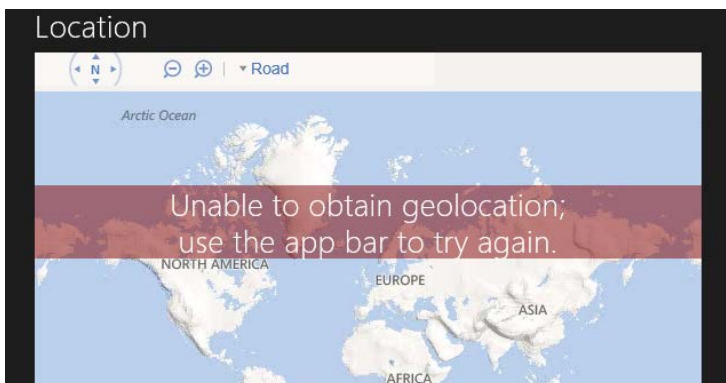
For an inline message, I've added a floating `div` that's positioned about a third of the way down on top of the map. It's defined in `pages/home/home.html` as follows, as a sibling of the map `iframe`:

```
<div id="locationSection" class="subsection" aria-label="Location section">
  <h2 class="group-title" role="heading">Location</h2>
  <iframe id="map" class="graphic" src="ms-appx-web:///html/map.html"
    aria-label="Map"></iframe>
  <div id="floatingError" class="win-type-x-large">Unable to obtain geolocation;<br />
    use the app bar to try again.</div>
</div>
```

The styles for the `#floatingError` rule in `pages/home/home.css` provide for its placement and appearance:

```
#floatingError {
  display: none;
  float: left;
  -ms-grid-column: 1;
  -ms-grid-row: 2;
  -ms-grid-row-align: start;
  width: 100%;
  text-align: center;
  background-color: rgba(128, 0, 0, 0.5);
  margin-top: 20%;
}
```

Because this is placed in the same grid cell as the map with `float` style, it appears as a nice overlay:



This message will appear if the user denies geolocation consent at startup or allows it but later uses the Settings charm to deny the capability. You can use these variations to test the appearance of the message. It's also possible, if you run the app the first time without network connectivity, for this message to appear on top of the map error image; this is why I've positioned the geolocation error toward the top so that it doesn't obscure the message in the image. But if you've successfully run the app once and then lose connectivity, the map should still get created because the Bing maps script will have been cached.

With `display: none` in the CSS, the error message is initially hidden, as it should be. If we get to the error handler for `getGeolocationAsync`, we set `style.display` to `block`, which reveals the element:

```
document.getElementById("floatingError").style.display = "block";
```

We again hide the message within the `tryRefresh` function, which is again invoked from the app bar command, so that the message stays hidden unless the error persists:

```
tryRefresh: function () {  
    document.getElementById("floatingError").style.display = "none";  
    //...  
},
```

One more piece we could add is a message dialog if we detect that we lost connectivity and thus couldn't update our position. As we'll see in Chapter 14, this could be done with the [Windows.-Networking.NetworkInformation.onnetworkstatuschanged](#) event. This is a case where a message dialog is appropriate because such a condition does not arise from direct user action.

Also, it's worth noting that if we used the Bing Maps SDK control in this app, the script we're normally loading from a remote source would instead exist in our app package, thereby eliminating the first error case altogether. We'll make this change in the next revision of the app.

What We've Just Learned

- In Windows Store app design, commands that are essential to a workflow should appear on the app canvas or on a popup menu from an element like a header. Those that can be placed on the Setting charm should also go there; doing so greatly simplifies the overall app implementation. Those commands that remain typically appear on an app bar or navigation bar, which can contain flyout menus for some commands. Context menus ([Windows.UI.Popups.PopupMenu](#)) can also be used for specific commands on content.
- The `WinJS.UI.Flyout` control is used for confirmations and other questions in response to user action; they can also just display a message, collect additional information, or provide controls to change settings for some part of the page. Flyouts are light-dismissed, meaning that clicking outside the control or pressing ESC will dismiss it, which is the equivalent of canceling the question.

- Message dialogs ([Windows.UI.Popups.MessageDialog](#)) are used to ask questions that the user must answer or acknowledge before the app can proceed; a message dialog disables the rest of the app. Message dialogs are best used for errors or conditions that affect the whole app; error messages that are specific to page content should appear inline.
- The app bar is a WinJS control on which you can place standard commands, using the commands layout, or any HTML of your choice, using the custom layout. Custom icons are also possible, using different fonts or custom graphics. An app can have both a top and a bottom app bar, where the top is typically used for navigation and employs a custom layout. App bars can be sticky to keep them visible instead of being light-dismissed.
- The app bar's [showCommands](#), [hideCommands](#), and [showOnlyCommands](#) methods, along with the [extraClass](#) property of commands, make it easy to define an app bar in a single location in the app and to selectively show specific command sets by using [querySelectorAll](#) with a class that represents that set.
- Command menus, as appear from an app bar command or an on-canvas control of some kind, are implemented with the [WinJS.UI.Menu](#) control.
- As an example of using many of these features, the Here My Am! app is updated in this chapter to greatly improve its handling of various error conditions.

Chapter 8

State, Settings, Files, and Documents

It would be very interesting when you travel if every hotel room you stayed in was automatically configured exactly as how you like it—the right pillows and sheets, the right chairs, the right food in the minibar rather than atrociously expensive and obscenely small snack tins. If you're sufficiently wealthy, of course, you can send people ahead of you to arrange such things, but such luxury remains naught but a dream for most of us.

Software isn't bound by such limitations, fortunately. Sending agents on ahead doesn't involve booking airfare for them, providing for their income and healthcare, and contributing to their retirement plans. All it takes is a little connectivity, some cloud services, and voila! All of your settings can automatically travel with you—that is, between the different devices you're using.

This roaming experience, as it's called, is built right into Windows 8 for systemwide settings such as your profile picture, start screen preferences, Internet favorites, your desktop theme, saved credentials, and so forth. When you use a Microsoft account to log into Windows on a trusted PC, these settings are securely stored in the cloud and automatically transferred to other trusted Windows 8 devices where you use the same account. I was pleasantly surprised during the development of Windows 8 that I no longer needed to manually transfer all this data when I updated my machine from one release preview to another!

With such an experience in place for system settings, users will expect similar behavior from apps: they will expect that app-specific settings on one device will appropriately roam to the same app installed on other devices. I say "appropriately" because some settings don't make sense to roam, especially those that are particular to the hardware in the device. On the other hand, if I configure email accounts in an app on one machine, I would certainly hope those show up on others! (I can't tell you how many times I've had to set up my four active email accounts in Outlook.) In short, as a user I'll expect that my transition between devices—on the system level and the app level—is both transparent and seamless.

This means, then, that each app must do its part to manage its state, deciding what information is local to a device, what data roams between devices (including roaming documents and other user data through services like SkyDrive), and even what kinds of caching can help improve performance and provide an good offline experience. As I've said with many such functional aspects, the effort you invest in these can make a real difference in how users perceive your app and the ratings and reviews they'll give it in the Windows Store.

Many such settings will be completely internal to an app, but others can and should be directly configurable by the user. In the past, this has given rise to an oft-bewildering array of nested dialog boxes with multiple tabs, each of which is adorned with buttons, popup menus, and long hierarchies of

check boxes and radio buttons. As a result, there's been little consistency in how apps are configured. Even a simple matter of where such options are located has varied between Tools/Options, Edit/Preferences, and File/Info commands, among others!

Fortunately, the designers of Windows 8 recognized that most apps have settings of some kind, so they included Settings on the Charms bar alongside the other near-ubiquitous search, share, and device functions. For one thing, this eliminates the need for users to remember where a particular app's settings are located, and apps don't need to wonder how, exactly, to integrate settings into their overall content flow and navigation hierarchy. That is, by being placed in the Settings charm, settings are effectively removed from an app's content structure, thereby simplifying the app's overall design. The app needs only to provide distinct pages that are displayed when the user invokes the charm.

Clearly, then, an app's state and its Settings UI are intimately connected, as we will see in this chapter. We'll also have the opportunity to look at the storage and file APIs in WinRT, along with some of the WinJS file I/O helpers and other storage options like IndexedDB.

Of course, app data—settings and internal state—is only one part of the story. *User data*—such as documents, pictures, music/audio, and videos—is equally important. For these we'll look at the various capabilities in the manifest that allow an app to work with document and media libraries, as well as removable storage, how to enumerate folder contents with queries, and how the file picker lets the user give consent to other safe areas of the file system (but not system areas and the app data folders of other apps).

Here, too, Windows 8 actually takes us beyond the local file system. The vast majority of data to which a user has access today is not local to their machine but lives online. The problem here has been that such data is typically buried behind the API of a web service, meaning that the user has to manually use a web app to browse data, download and save it to the local file system, and then import it into another app. Seeing this pattern, the Windows 8 designers found another opportunity to introduce a new level of integration and consistency, allowing apps to surface back-end data such that it appears as part of the local file system to other apps. This happens through the file picker contracts, bringing users a seamless experience across local and online data. Here we'll see the consumer side of the story, saving the provider side for Chapter 12, "Contracts."

In short, managing state and providing access to user data, wherever it's located, is one of the most valuable features that apps can provide, and it goes a long way to helping consumers feel that your app is treating them like they truly matter.

The Story of State

To continue the analogy started in this chapter's introduction, when we travel to new places and stay in hotels, most of us accept that we'll spend a little time upon arrival unpacking our things and setting up the room to our tastes. On the other hand, we expect the complete opposite from our homes: we expect continuity or *statefulness*. Having moved twice in the last year myself (once to a temporary home

while our permanent home was being completed), I can very much appreciate the virtues of statefulness. Imagine that everything in your home got repacked into boxes every time you left, such that you had to spend hours, days, or weeks unpacking it all again! No, home is the place where we expect things to stay put, even if we do leave for a time. I think this exactly why many people enjoy traveling in a motor home!

Windows Store apps are intended to be similarly stateful, meaning that they maintain a sense of continuity between sessions, even if the app is suspended and terminated along the way. In this way, apps feel more like a home than a temporary resting place; they become a place where users come to relax with the content they care about. So, the less work they need to do to start enjoying that experience, the better.

An app's state is central to this experience because it has a much longer lifetime than the app itself. State persists, as it should, when an app isn't running and can also persist between different versions of the app. The state version is, in fact, managed separately from the app version.

To clearly understand app state, let's first briefly revisit user data again. User data like documents, pictures, music, videos, playlists, and other such data are created and consumed by an app but not dependent on the app itself. User data implies that any number of apps might be able to load and manipulate such data, and such data always remains on a system irrespective of the existence of apps. For this reason, user data itself isn't really part of an app's state. That is, while the *paths* of documents and other files might be remembered as the current file, in the user's favorites, or in a recently used list, the actual *contents* of those files aren't part of that state. User data, then, doesn't have a strong relationship to app lifecycle events either. It's either saved explicitly through a user-invoked command or implicitly on events like `visibilitychange` rather than `suspending`. Again, the app might remember which file is currently loaded as part of its session state during `suspending`, but the file contents itself should be saved outside of this event since you have only five seconds to complete whatever work is necessary.

In contrast to user data, app data is comprised of everything an app needs to run and maintain its statefulness. App data is also maintained on a per-user basis, is completely tied to the existence of a specific app, and is accessible by that app exclusively. As we've seen earlier in this book, app data is stored in user-specific folders that are wholly removed from the file system when an app is uninstalled. For this reason, never store anything in app data that the user might want outside your app. It also makes sense to avoid using document and media libraries to store state that wouldn't continue to be meaningful to the user if the app is uninstalled.

App data is used to manage the following kinds of state:

- **Session state** The state that an app saves when being suspended to restore it after a possible termination. This includes form data, the page navigation history, and so forth. As we saw in Chapter 3, "App Anatomy and Page Navigation," being restarted after being suspended and terminated is the *only* case in which an app restores session state. Session state is typically saved incrementally (as the state changes) or within the `suspending` or `checkpoint` event.

- **Local app state** Those settings that are typically loaded when an app is launched. App state includes cached data, saved searches, lists of recently viewed items, and various behavioral settings that appear in the Settings panel like display units, preferred video formats, device-specific configurations, and so on. Local app state is typically saved when it's changed since it's not directly tied to lifecycle events.
- **Roaming app state** App state that is shared between the same app running on multiple Windows 8 devices where the same user is logged in, such as favorites, viewing position within videos, account configurations, URLs for important files on cloud storage locations, perhaps some saved searches or queries, etc. Like local app state, these might be manipulated through the Settings panel. Roaming state is also best saved when values are changed; we'll see more details on how this works later.

There are two other components of app state that are actually managed separately from app data folders and settings containers. One is the list of those files originally obtained through file pickers to which the app would like to have programmatic access in the future. For these files you cannot just save the pathname—you must also save the fact that the user granted permission through the file picker. This is the purpose of the [Windows.Storage.AccessCache](#) APIs, and is essentially part of local state.

The other part is credentials that you've collected from a user and would like to retrieve in the future. Because this is a critical security concern, an app should never directly save credentials in its own app data. Instead, use the credential locker API in [Windows.Security.Credentials.PasswordVault](#). The contents of the locker will be roamed between a user's trusted PCs, so this constitutes part of roaming app state. We'll see more of this API in Chapter 14, "Networking."

Settings and State

App state may or may not be surfaced directly to the user. Many bits of state are tracked internally within the app or, like a navigation history, might reflect user activity but aren't otherwise presented directly to the user. Other pieces of state, like preferences, accounts, profile pictures, and so forth, can and should be directly exposed to the user, which is the purpose of the Settings charm.

What appears in the Settings charm for an app should be those settings that affect behavior of the app as a whole and are adjusted only occasionally. State that applies only to particular pages or workflows should not appear in Settings: they should be placed directly on the page (the app canvas) or in the app bar, as we've seen in Chapter 7, "Commanding UI." All of these things still comprise app state and are managed as such, but not everything is appropriate for Settings.

Some examples of good candidates for the Settings charm are as follows:

- Display preferences like units, color themes, alignment grids, and defaults.
- Roaming preferences that allow the user to customize the app's overall roaming experience, such as to keep configurations for personal and work machines separate.

- Account and profile configurations, along with commands to log in, log out, and otherwise manage those accounts and profiles. Passwords should never be stored directly or roamed, however; use the Credential Locker instead.
- Behavioral settings like online/offline mode, auto-refresh, refresh intervals, preferred video/audio streaming quality, whether to transfer data over metered network connections, the location from which the app should draw data, and so forth.
- A feedback link where you can gather specific information from the user.
- Additional information about the app, such as Help, About, a copyright page, a privacy statement, license agreements, and terms of use. Oftentimes these commands will take the user to a separate website, which is perfectly fine.

I highly recommend that you run the apps that are built into Windows and explore their use of the Settings charm. You're welcome to explore how Settings are used by other apps in the Store as well, but those might not always follow the design guidelines as consistently—and consistency is essential to settings!

Speaking of which, Windows automatically provides commands called Permissions and Rate and Review for all apps. Rate and Review takes the user to the product page in the Windows Store where he or she can, of course, provide a rating and write a review. Permissions, for its part, allows the user to control the app's access to sensitive capabilities like geolocation, the camera, the microphone, and so forth. What appears here is driven by the capabilities declared in the app manifest, and it's where the user can go to revoke or grant consent for those capabilities. Of course, if the app uses no such capabilities, Permissions doesn't appear.

You might have noticed that I've made no mention of showing app updates within Settings. This is specifically discouraged because update notices are provided through the Windows Store directly. This is another way of reducing the kinds of noise with which users have had to contend with in the past, with each app presenting its updates in different ways (and sometimes far too often!).

App Data Locations

Now that we understand what kinds of information make up app state, the next question is, Where is it stored? You might remember from Chapter 1, "The Life Story of an App," that when Windows installs an app for a user (and all Windows Store apps are accessible to only the user who installed them), it automatically creates LocalState, TempState, and RoamingState folders within the current user's AppData folder, which are the same ones that get deleted when you uninstall an app. On the file system, if you point Windows Explorer to `%localappdata%\packages`, you'll see a bunch of folders for the different apps on your system. If you navigate to any of these, you'll see these folders along with one called "Settings," as shown in Figure 8-1 for the built-in Sports app. The figure also shows the varied contents of these folders.

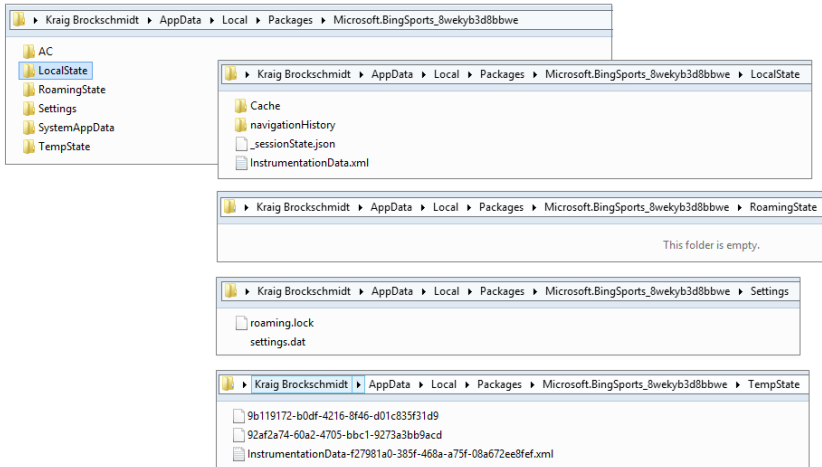


FIGURE 8-1 The Sports app's AppData folders and their contents.

In the LocalState folder of Figure 8-1 you can see a file named `_sessionState.json`. This is the file where WinJS saves and loads the contents of the `WinJS.Application.sessionState` object as we saw in Chapter 3. Since it's just a text file in JSON format, you can easily open it in Notepad or some other JSON viewer to examine its contents. In fact, if you look open this file for the Sports app, as is shown in the figure, you'll see a value like `{"lastSuspendTime":1340057531501}`. The Sports app (along with News, Weather, etc.) show time-sensitive content, so they save when they were suspended and check elapsed time when they're resumed. If that time exceeds their refresh intervals, they can go get new data from their associated service. In the case of the Sports app, one of its Settings specifically lets the user set the refresh period.

If your app uses any of the HTML5 storage APIs, like local storage, IndexedDB, and app cache, their data will also appear within the LocalState folder.

Note If you look carefully at Figure 8-1, you'll see that all the app data-related folders, including roaming, are in the user's overall AppData/Local folder. There is also a sibling AppData/Roaming folder, but this applies only to roaming user account settings on intranets, such as when a domain-joined user logs in to another machine on a corporate network. This AppData/Roaming folder has no relationship to the AppData/Local.../RoamingState folder for Windows Store apps.

Programmatically, you can refer to these locations in several ways. First, you can use the `ms-appdata:///` URI scheme as we saw in Chapter 3, where `ms-appdata:///local`, `ms-appdata:///roaming`, and `ms-appdata:///temp` refer to the individual folders and their contents. (Note the triple slashes, which is a shorthand allowing you to omit the package name.) You can also use the object returned from the `Windows.Storage.ApplicationData.current` method, which contains all the APIs you need to work with state.

By the way, you might have some read-only state directly in your app package. With URIs, you can just use relative paths that start with `/`. If you want to open and read file contents directly, you can use the `StorageFolder` object from the `Windows.ApplicationModel.Package.current.installedLocation` property. We'll come back to the `StorageFolder` class shortly.

AppData APIs (WinRT and WinJS)

When you ask Windows for the `Windows.Storage.ApplicationData.current` property, what you get is a `Windows.Storage.ApplicationData` object that is completely configured for your particular app. This object contains the following:

- `localFolder`, `temporaryFolder`, and `roamingFolder` Each of these properties is a `Windows.Storage.StorageFolder` object that allows you to create whatever files and additional folder structures you want in these locations (but note the `roamingStorageQuota` below).
- `localSettings` and `roamingSettings` These properties are `Windows.Storage.ApplicationDataContainer` objects that provide for managing a hierarchy of key-value settings pairs or composite groups of such pairs. All these settings are stored in the appdata Settings folder in the settings.dat file.
- `roamingStorageQuota` This property contains the number of kilobytes that Windows will automatically roam for the app (typically 100); if the total data stored in `roamingFolder` and `roamingSettings` exceeds this amount, roaming will be suspended until the amount falls below the quota. You have to track how much data you store yourself if you think you're near the limit.
- `dataChanged` An event indicating the contents of the `roamingFolder` or `roamingSettings` have been synchronized from the cloud; an app should re-read roaming state in this case. This is a WinRT event for which you need to use `removeEventListener` as described in Chapter 3 in the "WinRT Events and `removeEventListener`" section.
- `signalDataChanged` A method that triggers a `dataChanged` event. This allows you to consolidate local and roaming updates in a single handler for the `dataChanged` event.
- `version` property and `setVersionAsync` method These provide for managing the version stamp on your app data. This version applies to the whole of your app data, local, temp, and roaming together; there are not separate versions for each.
- `clearAsync` A method that clears out the contents of all AppData folders and settings containers. Use this when you want to reinitialize your default state, which can be especially helpful if you've restarted the app because of corrupt state.

- `clearAsync(<locality>)` A variant of `clearAsync` that is limited to one locality (local, temp, and roaming). The locality is identified with a value from `Windows.Storage.ApplicationDataLocality`, such as `Windows.Storage.ApplicationDataLocality.local`. In the case of local and roaming, the contents of both the folders and settings containers are cleared; temp affects only the TempState folder.

Let's now see how to use the API here to manage the different kinds of app data, which includes a number of WinJS helpers for the same purpose.

Hint The APIs that work with app state will generate events in the Event Viewer if you've enabled the channel as described in Chapter 3 in the "Debug Output, Error Reports, and the Event Viewer" section. Again, make sure that View > Show Analytics and Debug Logs is checked on the menu. Then navigate to Application and Services Log, and expand Microsoft/Windows/AppModel-State, where you'll find Debug and Diagnostic groups.

Settings Containers

For starters, let's look at the `localSettings` and `roamingSettings` properties, which are typically referred to as *settings containers*. You work with these through the `ApplicationDataContainer` API, which is relatively simple. Each container has four read-only properties: a `name` (a string), a `locality` (again from `Windows.Storage.ApplicationDataLocality`, with `local` and `roaming` being the only values here), and collections called `values` and `containers`.

The top-level settings containers have empty names; the property will be set for child containers that you create with the `createContainer` method (and remove with `deleteContainer`). Those child containers can have other containers as well, allowing you to create a whole settings hierarchy. That said, these settings containers are intended to be used for small amounts of data, typically user settings; any individual setting is limited to 8K and any composite setting (see below) to 64K. With these limits, going beyond about a megabyte of settings implies a somewhat complex hierarchy, which will be difficult to manage and will certainly slow to access. So don't be tempted to think of app data settings as a kind of database; other mechanisms like IndexedDB and SQLite are much better suited for that purpose, and you can write however much data you like as files in the various AppData folders (remembering the roaming limit when you write to `roamingFolder`).

For whatever container you have in hand, its `containers` collection is an `IMapView` object through which you can enumerate its contents. The `values` collection, on the other hand, is just an array (technically an `IPropertySet` object in WinRT, which is projected into JavaScript as an array with `IPropertySet` methods). So, although the `values` property in any container is itself read-only, meaning that you can't assign some other arbitrary array to it, you can manipulate the contents of the array however you like.

We can see this in the [Application data sample](#), which is a good reference for many of the core app data operations. Scenario 2, for example (js/settings.js), shows the simply use of the `localSettings.values` array:

```

var localSettings = Windows.Storage.ApplicationData.current.localSettings;
var settingName = "exampleSetting";
var settingValue = "Hello World";

function settingsWriteSetting() {
    localSettings.values[settingName] = settingValue;
}

function settingsDeleteSetting() {
    localSettings.values.remove(settingName);
}

```

Many settings, like that shown above, are just simple key-value pairs, but other settings will be objects with multiple properties. This presents a particular challenge: although you can certainly write and read the individual properties of that object within the `values` array, what happens if a failure occurs with one of them? That would cause your state to become corrupt.

To guard against this, the app data APIs provide for *composite settings*, which are groups of individual properties that are guaranteed to be managed as a single unit. (Again, each composite has a 64K limit.) It's like the perfect group consciousness: either we all succeed or we all fail, with nothing in between! That is, if there's an error reading or writing any part of the composite, the whole composite fails; with roaming, either the whole composite roams or none of it roams.

A composite object is created using [Windows.Storage.ApplicationDataCompositeValue](#), as shown in Scenario 4 of the Application data sample (`js/compositeSettings.js`):

```

var roamingSettings = Windows.Storage.ApplicationData.current.roamingSettings;
var settingName = "exampleCompositeSetting";
var settingName1 = "one";
var settingName2 = "hello";

function compositeSettingsWriteCompositeSetting() {
    var composite = new Windows.Storage.ApplicationDataCompositeValue();
    composite[settingName1] = 1; // example value
    composite[settingName2] = "world"; // example value
    roamingSettings.values[settingName] = composite;
}

function compositeSettingsDeleteCompositeSetting() {
    roamingSettings.values.remove(settingName);
}

function compositeSettingsDisplayOutput() {
    var composite = roamingSettings.values[settingName];
    // ...
}

```

The `ApplicationDataCompositeValue` object has, as you can see in the documentation, some additional methods and events to help you manage it such as `clear`, `insert`, and `mapchanged`.

Composites are, in many ways, like their own kind of settings container, just that they cannot contain additional containers. It's important to not confuse the two. Child containers within settings are used only to create a hierarchy (refer to Scenario 3 in the sample). Composites, on the other hand, specifically exist to create more complex groups of settings that act like a single unit, a behavior that is not guaranteed for settings containers themselves.

As noted earlier, these settings are all written to the settings.dat file in your app data Settings folder. It's also good to know that changes you make to settings containers are automatically saved, though there is some built-in batching to prevent excessive disk activity when you change a number of values all in a row. In any case, you really don't need to worry about *when* you save settings; the system will manage those details.

Versioning App State

From Windows' point of view, local, temp, and roaming state are all parts of the same whole and all share the same version. That version number is set with [Windows.Storage.ApplicationData.-setVersionAsync](#), the value of which you can retrieve through [Windows.Storage.ApplicationData.version](#) (a read-only property). If you like, you can maintain your own versioning system within particular files or settings. I would recommend, however, that you avoid doing this with roaming data because it's hard to predict how Windows will manage synchronizing slightly different structures. Even with local state, trying to play complex versioning games is, well, rather complex, and probably best avoided altogether.

The version of your app data is also a different matter than the version of your *app*; in fact, there is really no inherent relationship between the two. While the app data version is set with [setVersion-Async](#), the app version is managed through the Packaging section of the app manifest. You can have versions 1.0.0.0 through 4.3.9.3 of the app use version 1.0.0.0 of app data, or maybe version 1.2.1.9 of the app shifts to version 1.0.1.0 of the app data, and version 2.1.1.3 moves to 1.2.0.0 of the app data. It doesn't really matter, so long as you keep it all straight and can migrate old versions of the app data to new versions!

Migration happens as part of the [setVersionAsync](#) call, whose second parameter is a function to handle the conversion. That function receives a [SetVersionRequest](#) object that contains [currentVersion](#) and [desiredVersion](#) properties, thereby instructing your function as to what kind of conversion is actually needed. In response, you should go through all your app data and migrate the individual settings and files accordingly. Once you return from the conversion handler, Windows will assume the migration is complete. Of course, because the process will often involve asynchronous file I/O operations, you can use a deferral mechanism like that we've seen with activation. Call the [SetVersionRequest.getDeferral](#) method to obtain the deferral object (a [SetVersionDeferral](#)), and call its [complete](#) method when all your async operations are done. Examples of this can be found in Scenario 9 of the [Application data sample](#).

It is also possible to migrate app data when a new app update has been installed. For this you use a background task for the [servicingComplete](#) trigger. See Chapter 13, “Tiles, Notifications, the Lock Screen, and Background Tasks,” specifically the “Background Tasks and Lock Screen Apps” section toward the end.

Storage Folders and Storage Files

As it is often highly convenient to save app data directly in file, it’s high time we start looking more closely at the File I/O APIs for Windows Store apps.

First, however, other APIs like [URL.createObjectURL](#)—working with what are known as *blobs*—make it possible to do many things in an app without having descend to the level of file I/O at all! We’ve already seen how to use this to set the [src](#) of an [img](#) element, and the same works for other elements like [audio](#), [video](#), and [iframe](#). The file I/O operations involved with such elements is encapsulated within [createObjectURL](#). There are other ways to use a blob as well, such as converting a [canvas](#) element with [canvas.msToBlob](#) into something you can assign to an [img](#) element, and obtaining a binary blob from [WinJS.xhr](#), saving it to a file, and then sourcing an [img](#) from that. We’ll see some more of this in Chapter 10, “Media,” and you can refer to the [Using a blob to save and load content sample](#) for more.

For working directly with files, now, let’s get a bearing on what we have at our disposal, with concrete examples supplied by the [File access sample](#).

The core WinRT APIs for files live within the [Windows.Storage](#) namespace. The key players are the [StorageFolder](#) and [StorageFile](#) classes. These are sometimes referred to generically as “storage items” because they are both derived from [IStorageItem](#) and share properties like [name](#), [path](#), [dateCreated](#), and [attributes](#) properties along with the methods [deleteAsync](#) and [renameAsync](#).

File I/O in WinRT almost always starts by obtaining a [StorageFolder](#) object through one of the methods below. In a few cases you can also get to a [StorageFile](#) directly:

- [Windows.ApplicationModel.Package.current.installedLocation](#) gets a [StorageFolder](#) through which you can load data from files in your package (all files therein are read-only).
- [Windows.Storage.ApplicationData.current.localFolder](#), [roamingFolder](#), or [temporaryFolder](#) provides [StorageFolder](#) objects for your app data locations (read-write).
- An app can allow the user to select a folder or file directly using the file pickers invoked through [Windows.Storage.Pickers.FolderPicker](#) plus [FileOpenPicker](#) and [FileSavePicker](#). This is the preferred way for apps that don’t need to enumerate contents of a library (see next bullet). This is also the only means through which app can access safe (nonsystem) areas of the file system without additional declarations in the manifest.
- [Windows.Storage.KnownFolders](#) provides [StorageFolder](#) objects for the Pictures, Music, Videos, and Documents libraries, as well as Removable Storage. Given the appropriate capabilities in your manifest, you can work with the contents of these folders. (Attempting to obtain a folder without the correct capability will throw an Access Denied exception.)

- The [Windows.Storage.DownloadsFolder](#) object provides a [createFolderAsync](#) method through which you can obtain a [StorageFolder](#) in that location. It also provides a [createFileAsync](#) method to create a [StorageFile](#) directly. You would use this API if your app manages downloaded files directly. Note that [DownloadsFolder](#) itself provides only these two methods; it is not a [StorageFolder](#) in its own right.
- The static method [Windows.Storage.StorageFolder.getFolderFromPathAsync](#) returns a [StorageFolder](#) for a given pathname *if and only if* your app already has permissions to access it; otherwise, you'll get an Access Denied exception. A similar static method exists for files called [Windows.Storage.StorageFile.getFileFromPathAsync](#), with the same restrictions; [Windows.Storage.StorageFile.getFileFromApplicationUriAsync](#) opens files with [ms-appx:///](#) (package) and [ms-appdata:///](#) URIs. Other schemas are not supported.
- Once a folder or file object is obtained, it can be stored in the [Windows.Storage.-AccessCache](#) that allows an app to retrieve it sometime in the future with the same programmatic permissions. This is primarily needed for folders or files selected through the pickers because permission to access the storage item is granted only for the lifetime of that in-memory object. You should always use this API, as demonstrated in Scenario 6 of the File access sample, where you'd normally think to save a file path. Again, [StorageFolder.getFolder-FromPathAsync](#) and [StorageFile.getFileFromPathAsync](#) will throw Access Denied exceptions if they refer to any locations where you don't already have permissions. Pathnames also will not work for files provided by another app through the file picker, because the [StorageFile](#) object might not, in fact, refer to anything that actually exists on the file system.

Once you have a [StorageFolder](#) in hand, you can do the kinds of operations you'd expect: obtain folder properties (including a thumbnail), create and/or open files and folders, and enumerate the folder's contents. With the latter, the API provides for obtaining a list folder contents, of course (see the [getItemsAsync](#) method), but what you want more often is a partial list of those contents according to certain criteria, along with thumbnails and other indexed file metadata (music album and track info, picture titles and tags, etc.) that you can use to group and organize the files. This is the purpose of file, folder, and item (file + folder) *queries*, which you manage through the methods [createFileQuery-\[WithOptions\]](#), [createFolderQuery\[WithOptions\]](#), and [createItemQuery\[WithOptions\]](#). We already saw a little of this with the FlipView app we built using the Pictures Library in Chapter 5, "Collection and Collection Controls" and we'll return to the subject in the context of user data, the primary scenario for queries, at the end of this chapter.⁴¹

⁴¹ Obtaining folder properties happens through a storage item's [getBasicPropertiesAsync](#) method (linked here for [StorageFolder](#) but also available on [StorageFile](#).) This provides a [Windows.Storage.FileProperties.-BasicPropertiesClass](#), which then has a [retrievePropertiesAsync](#) method. Through this you can retrieve any number of [Windows.properties](#). A property like [System.FreeSpace](#) will actually give you the free space on the drive where the [StorageFolder](#) lives.

Tip There are some file extensions that are reserved by the system and won't be enumerated, such as .lnk, .url, and others; a complete list is found on the [How to handle file activation](#) topic. Also note that the ability to access UNC pathnames requires the *Private Networks (Client & Server)* and *Enterprise Authentication* capabilities in the manifest along with declarations of the file types you wish to access.

With any given [StorageFolder](#), especially for your app data locations, you can create whatever folder structures you like through its [createFolderAsync/getFolderAsync](#) methods, which give you more [StorageFolder](#) objects. Within any of those folders you then use the [createFileAsync/getFileAsync](#) methods to access individual files, each of which you again see as a [StorageFile](#) object.

Each [StorageFile](#) provides relevant properties like [name](#), [path](#), [dateCreated](#), [fileType](#), [contentType](#), and [attributes](#), of course, along with methods like [getThumbnailAsync](#), [copyAsync](#), [deleteAsync](#), [moveAsync](#), [moveAndReplaceAsync](#), and [renameAsync](#) for file management purposes. A file can be opened in a number of ways depending on the kind of access you need, using these methods:

- [openAsync](#) and [openReadAsync](#) provide random-access byte-reader/writer streams. The streams are [IRandomAccessStream](#) and [IRandomAccessStreamWithContentType](#) objects, respectively, both in the [Windows.Storage.Streams](#) namespace. The first of these works with a pure binary stream; the second works with data+type information, as would be needed with an http response that prepends a content type to a data stream.
- [openSequentialReadAsync](#) provides a read-only [Windows.Storage.Streams.-IInputStream](#) object through which you can read file contents in blocks of bytes but cannot skip back to previous locations. You should always use this method when you simply need to consume the stream as it has better performance than a random access stream (the source can optimize for sequential reads).
- [openTransactedWriteAsync](#) provides a [Windows.Storage.StorageStreamTransaction](#) that's basically a helper object around an [IRandomAccessStream](#) with [commitAsync](#) and [close](#) methods to handle the transactioning. This is necessary when saving complex data to make sure that the whole write operation happens atomically and won't result in corrupted files if interrupted. Scenario 4 of the File access sample shows this.

The [StorageFile](#) class also provides two static methods, [createStreamedFileAsync](#) and [createStreamedFileFromUriAsync](#). These provide a [StorageFile](#) that you typically pass to other apps through contracts as we'll see more of in Chapter 12. The utility of these methods is that the underlying file isn't accessed at all until data is first requested from it, *if* such a request ever happens at all.

Pulling all this together now, here's a little piece of code using the raw API we've seen thus far to create and open a "data.tmp" file in our temporary AppData folder, and write a given string to it. This bit of code is in the RawFileWrite example for this chapter. Let me be clear that what's shown here utilizes the lowest-level API in WinRT for this purpose and isn't what you typically use, as we'll see in the next section. It's instructive nonetheless as there are times you need to use something similar:

```

var fileContents = "Congratulations, you're written data to a temp file!";
writeTempFileRaw("data.tmp", fileContents);

function writeTempFileRaw(filename, contents) {
    var tempFolder = Windows.Storage.ApplicationData.current.temporaryFolder;
    var outputStream;

    //Egads!
    tempFolder.createFileAsync(filename,
        Windows.Storage.CreationCollisionOption.replaceExisting)
        .then(function (file) {
            return file.openAsync(Windows.Storage.FileAccessMode.readWrite);
        }).then(function (stream) {
            outputStream = stream.getOutputStreamAt(0);
            var writer = new Windows.Storage.Streams.DataWriter(outputStream);
            writer.writeString(contents);
            return writer.storeAsync();
        }).done();
}

```

Good thing we learned about chained async operations a while back! First we create or open the specified file in our app data's temporaryFolder ([createFileAsync](#)), and then we obtain an output stream for that file ([openAsync](#) and [getOutputStreamAt](#)). We then create a [DataWriter](#) around that stream, write our contents into it ([writeString](#)), and make sure it's stored in the file ([storeAsync](#)).

But, you're saying, "You've got to be kidding me! Four chained async operations just to write a simple string to a file! Who designed this API?" Indeed, when we started building the very first Store apps within Microsoft, this is all we had, and we asked these questions ourselves! After all, doing some hopefully simple file I/O is typically the first thing you add to a Hello World app, and this was anything but simple. To make matters worse, at that time we didn't yet have promises for async operations in JavaScript, so we had to write the whole thing with raw nested operations. Such were the days.

Fortunately, simpler APIs were already available and more came along shortly thereafter. These are the APIs you'll typically use when working with files as we'll see in the next section. It is nevertheless important to understand the structure of the low-level code above because the [Window.Storage.-Streams.DataWriter](#) class shown in that code, along with its [DataReader](#) sibling, are very important mechanisms for working with a variety of different I/O streams and are essential for data encoding processes. Having control over the fine details also supports scenarios such as having different components in your app that are all contributing to the file structure. So it's good to take a look at their reference documentation along with the [Reading and writing data sample](#) just so that you're familiar with the capabilities.

Sidebar: Closing Streams vs. Closing Files

Developers who have worked with file I/O APIs in the past sometimes ask why the `StorageFile` object doesn't have some kind of `close` method on it. The reason for this is because the `StorageFile` itself represents a file entity, not a data stream through which you can access its contents. It's when you call methods like `StorageFile.openAsync` to obtain a stream that the file is actually open, and the file is only closed when you close the stream through *its* particular `close` method.

You don't see a call to that method in the code above, however, because the `DataReader` and `DataWriter` classes both take care of that detail for you when they are discarded. However, if you separate a stream from these objects through their `detachStream` methods, you're responsible for calling the stream's `close` method.

When developing apps that write to files, if you see errors indicating that the file is still open, check whether you've properly closed the streams involved.

The FileIO, PathIO, and WinJS helper classes (plus FileReader)

Simplicity is a good thing where File I/O is concerned, and the designers of WinRT made sure that the most common scenarios didn't require a long chain of async operations like we saw in the previous section. The `Windows.Storage.FileIO` and `PathIO` classes provide such a streamlined interface, with the only difference between the two being that the `FileIO` methods take a `StorageFile` parameter whereas the `PathIO` methods take a filename string. Otherwise they offer the same methods called `[read | write]BufferAsync` (these work with byte arrays), `[append | read | write]LinesAsync` (these work with arrays of strings), and `[append | read | write]TextAsync` (these work with singular strings). In the latter case the `WinJS.IOHelper` class provides an even simpler interface through its `readText` and `writeText` methods.

Let's see how those work, starting with a few examples from the [File access sample](#). Scenario 2 shows writing a text string from a control to a file (this code is simplified from the sample for clarity):

```
var userContent = textArea.innerText;

//sampleFile created on startup from Windows.Storage.KnownFolders.documentsLibrary.getFileAsync
Windows.Storage.FileIO.writeTextAsync(sampleFile, userContent).done(function () {
    outputDiv.innerHTML = "The following text was written to '" + sampleFile.name
        + "':<br /><br />" + userContent;
});
```

To compare to the example in the previous section, we can replace all the business with streams and `DataWriters` with one line of code:

```
tempFolder.createFileAsync(filename, Windows.Storage.CreationCollisionOption.replaceExisting)
    .then(function (file) {
        Windows.Storage.FileIO.writeTextAsync(file, contents).done();
    })
```

To make it even simpler, the `WinJS.Application.temp` object ([WinJS.Application.IOHelper](#)) reduces even this down to a single line (which is an async call and returns a promise):

```
WinJS.Application.temp.writeText(file, contents);
```

Reading text through the async `readText` method is equally simple, and WinJS provides the same interface for the `local` and `roaming` folders along with two other methods, `exists` and `remove`.⁴² That said, these WinJS helpers are available *only* for your AppData folders and not for the file system more broadly; for that you should be using the `FileIO` and `PathIO` classes.

You also have the HTML5 `FileReader` class available for use in Windows Store apps, which is part of the [W3C File API specification](#). As its name implies, it's suited only for reading files and cannot write them, but one of its strengths is that it can work both with files and blobs. Some examples of this are found in the [Using a blob to save and load content sample](#).

Encryption and Compression

WinRT provides two capabilities that might be very helpful to your state management: encryption and compression.

Encryption is provided through the [Windows.Security.Cryptography](#) and [Windows.-Security.Cryptography.Core](#) API. The former contains methods for basic encoding and decoding (base64, hex, and text formats); the latter handles actual encryption according to various algorithms. As demonstrated in the [Secret saver encryption sample](#), you typically encode data in some manner with the `Windows.Security.Cryptography.CryptographicBuffer.convertStringToBinary` method and then create or obtain an algorithm and pass that with the data buffer to `Windows.-Security.Cryptography.Core.CryptographicEngine.encrypt`. Methods like `decrypt` and `convertBinaryToString` perform the reverse.

Compression is a little simpler in that its only purpose is to provide a built-in API through which you can make your data smaller (say, to decrease the size of your roaming data). The API for this in [Windows.Storage.Compression](#) is composed of `Compressor` and `Decompressor` classes, both of which are demonstrated in the [Compression sample](#). Although this API can employ different compression algorithms, including one called MSZIP, it does *not* provide a means to manage .ZIP *files* and the contents therein. For this purpose you'll need to employ either a third-party JavaScript library or you can write a WinRT component in C# or Visual Basic that can use the `System.IO.Compression` APIs (see Chapter 16, "WinRT Components.")

⁴² If you're curious as to why async methods like `readText` and `writeText` don't have `Async` in their names, this was a conscious choice on the part of the WinJS designers to follow existing JavaScript conventions where such a suffix isn't typically used. The WinRT API, on the other hand, is language-independent and thus has its own convention with the `Async` suffix.

Using App Data APIs for State Management

Now that we've seen the nature of the APIs, let's see how they're used for different kinds of app data and any special considerations that come into play.

Session State

As described before, session state is whatever an app saves when being suspended so that it can restore itself to that state if it's terminated by the system and later restarted. Being terminated by the system is again the only time this happens, so what you include in session state should always be scoped to giving the user the illusion that the app was running the whole time. In some cases, as described in Chapter 3, you might not in fact restore this state, especially if it's been a long time since the app was suspended and it's unlikely the user would really remember where they left off. That's a decision you need to make for your own app and the experience you want to provide for your customers.

Session state should be saved within the appdata `localFolder` or the `localSettings` object. It should not be saved in a temp area since the user could run the disk cleanup tool while your app is suspended or terminated in which case session state would be deleted (see next section).

The WinJS `sessionState` object internally creates a file called `_sessionState.json` within the `localFolder`. The file is just JSON text, so you can examine it any time. You can and should write session state variables to the `sessionState` object whenever they change, using `sessionState` essentially as a namespace for those session variables. This way those values get saved and reloaded automatically without needing to manage variables somewhere else.

If you need to store additional values within `sessionState` before its written, do that in your handler for `WinJS.Application.oncheckpoint`. A good example of such data is the navigation stack for your page controls, which is available in `WinJS.Navigation.history`; you could also copy this data to `sessionState` within the `PageControlNavigator.navigated` method (in `navigator.js` as provided by the project templates). In any case, WinJS has its own `checkpoint` handler that is always called last (after your handler) to make sure that any changes you make to `sessionState` in response to that event are saved.

If you don't use the WinJS `sessionState` object and just use the WinRT appdata APIs directly, you can write your session state whenever you like (including within `checkpoint`), and you'll need to restore it directly within your activated event for `previousExecutionState == terminated`.

It's also a good practice to build some resilience into your handling of session state: if what gets loaded doesn't seem consistent or has some other problem, revert to default session values. Remember too that you can use the `localSettings` container with composite settings to guarantee that groups of values will be stored and retrieved as a unit. You might also find it helpful during development to give yourself a simple command to clear your app state in case things get really fouled up, but just uninstalling your app will clear all that out as well. At the same time, it's not necessary to provide your users with a command to clear session state: if your app fails to launch after being terminated, the `previousExecutionState` flag will be `notRunning` the next time the user tries, in which case you won't attempt to restore the state.

Similarly, it's not necessary to include a version number in session state. If the user installs an update during the time your app has been suspended and terminated, and the appdata version changes, the `previousExecutionState` value will be reset. If for some reason you don't actually change the appdata version—for instance, if your update is only very minor—then your session state can carry forward. But in this case it's essentially the same app, so versioning the state isn't an issue.

Sidebar: Using HTML5 `sessionStorage` and `localStorage`

If you prefer, you can use HTML5 `localStorage` object for both session and other local app data; its contents get persisted to the app data `localFolder`. The contents of `localStorage` are not loaded until first accessed and are limited to 10MB per app; the WinRT and WinJS APIs, on the other hand, are limited only by the capacity of the file system.

As for the HTML5 `sessionStorage` object, it's not really needed when you're using page controls and maintaining your script context between app pages—your in-memory variables already do the job. However, if you're actually changing page contexts by using `<a>` links or setting `document.location`, `sessionStorage` can still be useful. You can also encode information into URLs as commonly done with web apps.

Both `sessionStorage` and `localStorage` are also useful within `iframe` pages running in the web context, since the WinRT APIs are not available. At the same time, you can load WinJS into a web context page (this is supported) and the `WinJS.Application.local`, `roaming`, and `temp` objects still work using in-memory buffers instead of the file system.

Local and Temporary State

Unlike session state that is restored only in particular circumstances, local app state is composed of those settings and other values that are *always applied* when an app is launched. Anything that the user can set directly falls into this category, unless it's also part of the roaming experience in which case it is still loaded on app startup. Any other cached data, saved searches, recently used items, display units, preferred media formats, and device-specific configurations also fall into this bucket. In short, if it's not pure session state and not part of your app's roaming experience, it's local or temporary state. (Remember again that credentials should be stored in the Credential Locker instead of your appdata.)

All the same APIs we've seen work for this form of state, including all the WinRT APIs, the `WinJS.Application.local` and `temp` objects, and HTML `localStorage`. You can also use the HTML5 IndexedDB APIs, SQLite, and the HTML App Cache—these are just other forms of local app data.

It's very important to version-stamp your local and temp app data because it will always be preserved across an app update (unless temp state has been cleaned up in the meantime). With any app update, be prepared to load old versions of your state and make the necessary updates, or simply decide that a version is too old and purge it (`Windows.Storage.ApplicationData.current.clearAsync`)

before setting up new defaults. As mentioned before, it's also possible to migrate state from a background task. (See Chapter 13.)

Generally speaking, local and temp app data are the same—they have the same APIs and are stored in parallel folders. Temp, however, doesn't support settings and settings containers. The other difference is that the contents of your temp folder (along with the HTML5 app cache) are subject to the Windows Disk Cleanup tool. This means that your temp data could disappear at any time when the user wants to free up some disk space. You could also employ a background task with a maintenance trigger for doing cleanup on your own (again see Chapter 13, in the section "Tasks for Maintenance Triggers.")

For these reasons, temp should be used for storage that optimizes your apps performance but not for anything that's critical to its operation. For example, if you have a JSON file in your package that you parse or decompress on first startup such that the app runs more quickly afterwards, and you don't make any changes to that data from the app, you might elect to store that in temp. The same is true for graphical resources that you might have fine-tuned for the specific device you're running on; you can always repeat that process from the original resources, so it's another good candidate for temp data. Similarly, if you've acquired data from an online service as an optimization (that is, so that you can just update the local copy incrementally), you can always reacquire it. This is especially helpful for providing an offline experience for your app, though in some cases you might want to let the user choose to save it in local instead of temp (an option that would appear in Settings along with the ability to clear the cache).

Sidebar: HTML5 App Cache

Store apps can employ the HTML 5 app cache as part of an offline/caching strategy. It is most useful in `iframe` web context elements where it can be used for any kind of content. For example, an app that reads online books can show such content in an `iframe`, and if those pages include app cache tags, they'll be saved and available offline. In the local context, the app cache works for nonexecutable resources like images, audio, and video, but not for HTML or JavaScript.

IndexedDB and Other Database Options

Many forms of local app data are well suited to being managed in a database. In Windows Store apps, the IndexedDB API is available through the `window.indexedDB` and `worker.indexedDB` objects. For complete details on using this feature, I'll refer you to the [W3C specifications](#), the [Indexed Database API reference](#) for Store apps, and the [IndexedDB sample](#).

Although an IndexedDB database is stored within your app's local app data, be aware that there are some limitations because there isn't a means through which the app or the system can shrink a database file and reclaim unused space:

- IndexedDB has a 250MB limit per app and an overall system limit of 375MB on drives smaller than 32GB, or 4% (to a maximum 20GB) for drives over 32GB. So it could be true that your app might not have much room to work with anyway, in which case you need to make sure you have a fallback mechanism. (When the limit is exceeded the APIs will throw a Quota Exceeded exception.)
- IndexedDB on Windows 8 has no complex key paths—that is, it does not presently support multiple values for a key or index (multientry).
- By default, access to IndexedDB is given only to HTML pages that are part of the app package and those declared as content URLs. (See the “Local and Web Contexts within the App Host” section at the beginning of Chapter 3.) Random web pages you might host in an `iframe` will not be given access, primarily to preserve space within the 250MB limit for those pages you really care about in your app. However, you can grant access to arbitrary pages by including the following tag in your home page and not setting the `iframe src` attribute until the `DOMContentLoaded` or `activated` event has fired:

```
<meta name="ms-enable-external-database-usage" content="true"/>
```

Beyond IndexedDB there are a few other database options for Store apps. For a local relational database, try SQLite. This is an API that’s suited well for apps written in a language like C#, as described in [Tim Heuer’s blog on the subject](#), but fortunately, there is a version called SQLjs, which is [SQLite compiled to JavaScript via Emscripten](#). Very cool! There might also be other JavaScript solutions available in the community.

If the storage limits for IndexedDB are a concern, you might use the [Win32 “Jet” or Extensible Storage Engine \(ESE\) APIs](#) (on which the IndexedDB implementation is built). For this you’ll need to write a WinRT Component wrapper in C# or C++ (the general process for which is in Chapter 16, “WinRT Components”), since JavaScript cannot get to those APIs directly.

The same is true for other third-party database APIs. So long as that engine uses only the Win32 APIs allowable for Store apps (listed on the [Win32 and COM for Windows Store apps](#) page), they’ll work just fine.

It’s also worth noting that the [OData Library for JavaScript](#) also works great for Store apps to access online SQL Servers, because the OData protocol itself just works via REST.

Finally, another option for searchable file-backed data is to use the *system index* by creating a folder named “indexed” in your local AppData folder. The contents of the files in this folder will be indexed by the system indexer and can be queried using Advanced Query Syntax (AQS) with the APIs explained later in “Rich Enumeration with File Queries.” You can also do property-based searched for [Windows properties](#), making this approach a simple alternative to database solutions.

Roaming State

The automatic roaming of app state between a user's devices is one of the most interesting and enabling features of Windows 8. There are few areas where a small piece of technology like this has so greatly reduced the burden on app developers!

It works very simply. First, your app data `roamingFolder` and your `roamingSettings` container behave exactly like their local counterparts. So long as their combined size is less than `Windows.Storage.-ApplicationData.current.roamingStorageQuota`, Windows will copy that data to other devices where the same user is logged in has the same app installed; in fact, when an app is installed, Windows attempts to copy roaming data so that it's there when the app is first launched.

If the app is running simultaneously on multiple devices, the last writer of any particular file or setting always wins. When data has been roamed the other apps will receive the `Windows.Storage.-ApplicationData.ondatachanged` event. So your app will always read the appropriate roaming state on startup and refresh that state as needed within `datachanged`. You should always employ this strategy too in case Windows cannot bring down roaming state for a newly installed app right away (such as when the user installed the app and lost connectivity). As soon as the roaming state appears, you'll receive the `datachanged` event. Scenario 5 of the [Application data sample](#) provides a basic demonstration of this.

Deciding what your roaming experience really looks like is really a design question more than a development question. It's a matter of taking all app settings that are not specific to the device hardware (such as settings that are related to screen size, video capabilities, or the presence of particular peripherals or sensors), and thinking through whether it makes sense for each setting to be roamed. A user's favorites, for example, are appropriate to roam *if* they refer to data that isn't local to the device. That is, favorite URLs or locations on a cloud storage service like SkyDrive, FaceBook, or Flickr are appropriate to roam; favorites and recently used files in a user's local libraries are not. The viewing position within a cloud-based video, like a streaming movie, would be appropriate to roam, as would be the last reading position in a magazine or book. But again, if that content is local, then maybe not. Account configurations like email settings are often good candidates, so the user doesn't have to configure the app again on other devices.

At the same time, you might not be able to predict whether the user will really want to roam certain settings. In this case, the right choice is to give the user a choice! That is, include options in your Settings UI to allow the user to customize the roaming experience to their liking, especially as a user might have devices for both home and work where they want the same app to behave differently. For instance, with an RSS Reader the user might not want notifications on their work machine whenever new articles arrive, but would want real-time updates at home. The set of feeds itself, on the other hand, would probably always be roamed, but then again the user might want to keep separate lists.

To minimize the size of your roaming state and stay below the quota, you might employ the `Windows.Storage.Compression` API for file-based data. For this same reason, never use roaming state for *user data*. Instead, use an online service like SkyDrive to store user data in the cloud, and then roam URIs to those files as part of the roaming experience. More details on using SkyDrive through its REST API can

be found on the [SkyDrive reference](#), on the [Skydrive core reference](#) (which includes a list of supported file types), and in the [PhotoSky sample](#). A backgrounder on this and other Windows Live services can also be found on the Building Windows 8 blog post entitled [Extending "Windows 8" apps to the cloud with SkyDrive](#).

By now you probably have a number of other questions forming in your mind about how roaming actually works: "How often is data synchronized?" "How do I manage different versions?" "What else should I know?" These are good questions, and fortunately there are good answers!

- Assuming there's network connectivity, an app's roaming state is roamed within 30 minutes on an active machine. It's also roamed immediately when the user logs on or locks the machine. Locking the machine is always the best way to force a sync to the cloud. Note that if the cloud service is only aware of the user (that is, a Microsoft account) having only one device, synchronization with the cloud service happens only about once per day. When the service is aware that the user has multiple machines, it begins synchronizing within the 30-minute period; if the app is uninstalled on all but one machine, synchronization reverts to the longer period.
- When saving roaming state, you can write values whenever you like, such as when those settings are changed. You don't need to worry about writing settings as a group because Windows has a built-in debounce period to combine changes together and reduce overall network traffic.
- If you have a group of settings that really must be roamed together, manage these as a composite setting in your `roamingSettings` container.
- With files you create within the `roamingFolder`, these will not be roamed so long as you have the file open for writing (that is, as long as you have an open stream). It's a good idea to make sure that all streams are closed when the app is suspended.
- Windows allows each app to have up to 8K worth of "high priority" settings that will be roamed within one minute, thereby allowing apps on multiple devices to stay much more closely in sync. To use this, create a single or composite setting in the root of your `roamingSettings` with the name *HighPriority*—that is, `roamingSettings.values["HighPriority"]` (a *container* with this name will roam normally). So long as you keep the size of this setting below 8K, it will be roamed within a minute of being changed; if you exceed that size, it will be roamed with normal priority. See Scenario 6 of the Application data sample for a demonstration.
- On a trusted PC, systemwide user settings like the Start page configuration are automatically roamed independent of app. This also includes encrypted credentials saved by apps in the credential locker; apps should never attempt to roam passwords. Apps that create secondary tiles (as we'll see in Chapter 13) can indicate whether such tiles should be copied to a new device when the app is installed.
- When there are multiple app data versions in use by the same app (with multiple app versions, of course), Windows will manage each version of the app data separately, meaning that newer app data won't be roamed to devices with apps that use older app data versions. In light of this, it's a

good idea to not be too aggressive in versioning your app data since it will break the roaming connection between apps.

- The cloud service will retain multiple versions of roaming app data so long as there are multiple versions in use by the same Microsoft account. Only when all instances of the app have been updated will older versions of the roaming state be eligible for deletion.
- When an updated app encounters an older version of roaming state, it should load it according to the old version but save it as the new version and call `setVersionAsync`.
- Avoid using secondary versioning schemes within roaming state such that you introduce structural differences without changing the appdata version through `setVersionAsync`. Because the cloud service is managing the roaming state by this version number, and because the last writer always wins, some version of an app that expects to see some extra bit of data, and in fact saved it there, might find that it's been removed because a slightly older version of the app didn't write it.
- Even if all apps are uninstalled from a user's devices, the cloud service retains roaming data for "a reasonable time" (maybe 30 days) so that if a user reinstalls the app within that time period they'll find that their settings are still intact. To avoid this retention and explicitly clear roaming state from the cloud, use the `clearAsync` method.

Settings Pane and UI

We've now seen all the different APIs that an app can use to manage its state where storage is concerned. This is all you need for settings and other app data that are managed internally within the app. The question now is how to manage user-configurable settings, and for that we turn to the Settings charm.

When the user invokes the Settings charm (which can also be done directly with the Win+i key), Windows displays the Settings pane, a piece of UI that is populated with various settings commands as well as system functions along the bottom. Apps can add their own commands but are not obligated to do so. Windows guarantees that something always shows up for the app in this pane: it automatically displays the app name and publisher, a Rate and Review command that takes you to the Windows Store page for the app, an Update command if an update is available from the Store, and a Permissions command if the app has declared any capabilities in its manifest. (Note that Rate and Review won't appear for apps you run from Visual Studio since they weren't acquired from the Store.)

The Settings charm is always available no matter where you are in the app, so you don't need to think about having such a command on your app bar, nor do you ever need a settings command on your app canvas. That said, you can invoke the Settings charm programmatically, such as when you detect that a certain capability is turned off and you prompt the user about that condition. You might ask something like "Do you want to turn on geolocation for this app?" and if the user says Yes, you can

invoke the Settings charm. This is done through the settings pane object returned from [Windows.UI.ApplicationSettings.SettingPane.getForCurrentView](#), whose `show` method display the UI (or throws a kindly exception if the app is in snapped view or doesn't have the focus, so don't invoke it under those conditions!). The `edge` property of the settings pane object also tells you if it's on the left or right side of the screen, depending on the left-to-right or right-to-left orientation of the system as a whole (a regional variance).

And with that we've covered all the methods and properties of this object! Yet the most interesting part is how we add our own commands to the settings pane. But let's first look at the guidelines for using Settings.

Design Guidelines for Settings

Beyond the commands that Windows automatically adds to the settings pane, the app can provide up to eight others, typically around four; anything more than eight will throw an exception. Because settings are global to an app, the commands you add are always the same: they are not sensitive to context. To say it another way, the *only* commands that should appear on the settings pane are those that are global to the app; commands that apply only to certain pages or contexts within a page should appear on the app bar or on the app canvas. Some examples of commands on the top-level settings pane are shown in Figure 8-2.

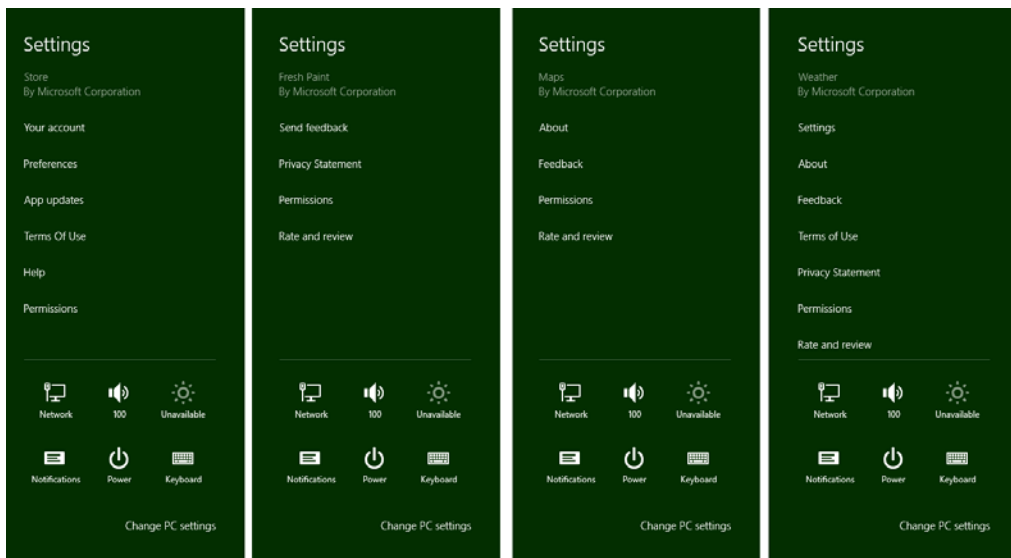


FIGURE 8-2 Examples of commands on the top-level settings pane. Notice that the lower section of the pane always has system settings and the app name and publisher always appear at the top. Permissions and Rate and Review are added automatically.

Each app-supplied command can do one of two things. First, a command can simply be a hyperlink to a web page. Some apps use links for their Help, Privacy Statement, Terms of Use, License Agreements, and so on, which will open the linked pages in a browser. The other option is to have the command invoke a secondary flyout panel with more specific settings controls or simply an `iframe` to display web-based content. You can provide Help, Terms of Use, and other textual content in both these ways rather than switch to the browser.

Note As stated in the [Windows 8 app certification requirements](#), section 4.1, apps that collect personal information in any way must have a privacy policy or statement. This must be included on the app's product description page in the Store as a minimum. Though not required, it is suggested that you also include a command for this in your Settings pane.

Secondary flyouts are created with the `WinJS.UI.SettingsFlyout` control; some examples are shown in Figure 8-3. Notice that the secondary settings panes come in two sizes: narrow (346px) and wide (646px). The design guidelines suggest that all secondary panes for an app are the same size—that is, don't make some narrow and some wide. You'll only have a couple of these panes anyway, so that shouldn't be a problem. Also note that the Permissions flyout, shown on the left of Figure 8-3, is provided by Windows automatically and is configured according to capabilities declared in your manifest. Some capabilities like geolocation are controlled in this pane; other capabilities like Internet and library access are simply listed because the user is not allowed to turn them on or off.

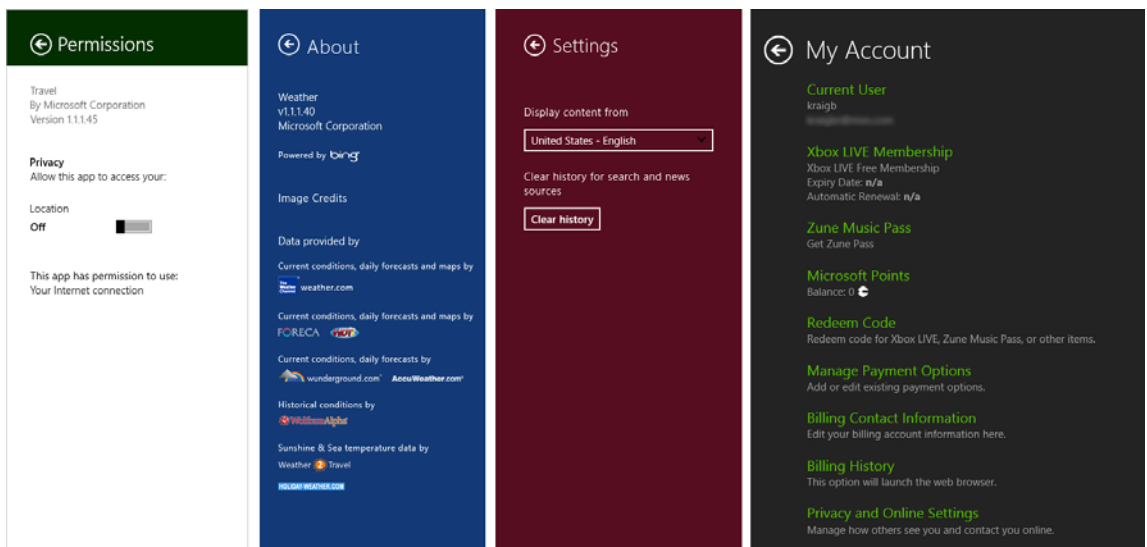


FIGURE 8-3 Examples of secondary settings panes in the Travel, Weather, News, and Music apps of Windows 8. The first three are the narrow size; the fourth is wide. Notice that each app-provided pane is appropriately branded and provides a back button to return to the main Settings pane. The Permissions pane is provided by the system and thus reflects the system theme; it cannot be customized.

A common group of settings are those that allow the user to configure their roaming experience—that is, a group of settings that determine what state is roamed (you see this on PC Settings > Sync Your Settings). It is also recommended that you include account/profile management commands within Settings, as well as login/logout functionality. As noted in Chapter 7, logins and license agreements that are necessary to run the app at all should be shown upon launch. For ongoing login-related functions, and to review license agreements and such, create the necessary commands and panes within Settings. Refer to [Guidelines and checklist for login controls](#) for more information on this subject. Guidelines for a Help command can also be found on [Adding app help](#).

Behaviorally, settings panes are light-dismiss but also have a header with a back button to return to the primary settings pane with all the commands. Because of the light-dismiss behavior, changing a setting on a pane applies the setting immediately: there is no OK or Apply button or other such UI. If the user wants to revert a change, she should just restore the original setting.

For this reason it's a good idea to use simple controls that are easy to switch back, rather than complex sets of controls that would be difficult to undo. The recommendation is to use toggle switches for on/off values (rather than check boxes), a button to apply an action (but without closing the settings UI), hyperlinks (to open the browser), text input boxes (which should be set to the appropriate type such as email address, password, etc.), radio buttons for groups of up to five mutually exclusive items, and a listbox (select) control for four to six text-only items.

In all your settings, think in terms of “less is more.” Avoid having all kinds of different settings, because if the user is never going to find them, you probably don't need to surface them in the first place! Also, while a settings pane can scroll vertically, try to limit the overall size such that the user has to pan down only once or twice, if at all.

Some other things to avoid with Settings:

- Don't use Settings for workflow-related commands. Those belong on the app bar or on the app canvas, as discussed in Chapter 7.
- Don't use a top-level command in the Settings pane to perform an action other than linking to another app (like the browser). That is, top-level commands should never execute an action *within* the app.
- Don't use settings commands to navigate within the app.
- Don't use `WinJS.UI.SettingsFlyout` as a general-purpose control.

And on that note, let's now look at the steps to use Settings and the `SettingsFlyout` properly!

Populating Commands

The first part of working with Settings is to provide your specific commands when the Settings charm is invoked. Unlike app bar commands, these should always be the same no matter the state of the app; if you have context-sensitive settings, place commands for those in the app bar.

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

To know when the charm is invoked through WinRT, obtain the settings pane object through [Windows.UI.ApplicationSettings.SettingsPane.getForCurrentView](#) and add a listener for its `commandsrequested` event (this is a WinRT event, so be sure to remove the listener if necessary):

```
// The n variable here is a convenient shorthand
var n = Windows.UI.ApplicationSettings;
var settingsPane = n.SettingsPane.getForCurrentView();
settingsPane.addEventListener("commandsrequested", onCommandsRequested);
```

Within your event handler, create [Windows.UI.ApplicationSettings.SettingsCommand](#) objects for each command, where each command has an `id`, a `label`, and an `invoked` function that's called when the command is tapped or clicked. These can all be specified in the constructor as shown below:

```
function onCommandsRequested(e) {
    // n is still the shortcut variable to Windows.UI.ApplicationSettings
    var commandHelp = new n.SettingsCommand("help", "Help", helpCommandInvoked);
    e.request.applicationCommands.append(commandHelp);
}
```

The second line of code is where you then add these commands to the settings pane itself. You do this by appending them to the [e.request.applicationCommands](#) object. This object is a WinRT construct called a *vector* that manages a collection with commands like `append` and `insertAt`. In this case we have a vector of `SettingsCommand` objects, and as you can see above, it's easy enough to append a command. You'd make such a call for each command, or you can pass an array of such commands to the `replaceAll` method instead of `append`. What then happens within the `invoked` handler for each command is the interesting part, and we'll come back to that in the next section.

You can also prepopulate the `applicationCommands` vector outside of the `commandsrequested` event; this is perfectly fine because your settings commands should be constant for the app. The [Quickstart: add app help](#) topic shows an example of this, which I've modified here to show the use of `replaceAll`:

```
var n = Windows.UI.ApplicationSettings;
var settingsPane = n.SettingsPane.getForCurrentView();
var vector = settingsPane.applicationCommands;

//Ensure no settings commands are currently specified in the settings charm
vector.clear();

var commands = [ new settingsSample.SettingsCommand("Custom.Help", "Help", OnHelp),
                  new n.SettingsCommand("Custom.Parameters", "Parameters", OnParameters)];
vector.replaceAll(commands);
```

This way, you don't actually need to register for or handle `commandsrequested` directly.

Now because most apps will likely use settings in some capacity, WinJS provides some shortcuts to this whole process. First, instead of listening for the WinRT event, simply assign a handler to `WinJS.Application.onsettings` (which is a wrapper for `commandsrequested`):

```
WinJS.Application.onsettings = function (e) {  
    // ...  
};
```

In your handler, create a JSON object describing your commands and store that object in `e.detail.applicationcommands`. Mind you, this is *different* from the WinRT object—just setting this property accomplishes nothing. What comes next is passing the now-modified event object to `WinJS.UI.SettingsFlyout.populateSettings` as follows (taken from Scenario 2 of the [App settings sample](#)):

```
WinJS.Application.onsettings = function (e) {  
    e.detail.applicationcommands =  
        { "help": { title: "Help", href: "/html/2-SettingsFlyout-Help.html" } };  
    WinJS.UI.SettingsFlyout.populateSettings(e);  
};
```

The `populateSettings` method traverses the `e.details.applicationcommands` object and calls the WinRT `applicationCommands.append` method for each item. This gives you a more compact method to accomplish what you'd do with WinRT, and it also simplifies the implementation of settings commands, as we'll see next.

Note The WinJS helpers are specifically designed for invoking `SettingsFlyout` controls that are populated with the HTML file you indicate in the `href` property. That property must refer to in-package contents; it cannot be used to create settings commands that launch a URI (commonly used for Terms of Service and Privacy Statement commands). In such cases you must use the WinRT API directly alongside `WinJS.UI.SettingsFlyout.populateSettings`. Then again, it's a simple matter to bring web content directly into a settings flyout with an `iframe`, which keeps the Settings experience within the app.

Implementing Commands: Links and Settings Flyouts

Technically speaking, within the `invoked` function for a settings command you can really do anything. Truly! Of course, as described in the design guidelines earlier, there are recommendations for how to use settings and how not to use them. For example, settings commands shouldn't act like app bar commands that affect content, nor should they navigate within the app itself. Ideally, a settings command does one of two things: either launch a hyperlink (to open a browser) or display a secondary settings pane.

In the base WinRT model for settings, launching a hyperlink uses the [Windows.System.-Launcher.launchUriAsync](#) API as follows:

```
function helpCommandInvoked(e) {
    var uri = new Windows.Foundation.Uri("http://example.domain.com/help.html");
    Windows.System.Launcher.LaunchUriAsync(uri).done();
}
```

In the second case, secondary panes are implemented with the [WinJS.UI.SettingsFlyout](#) control. Again, technically speaking, you're not required to use this control: you can display any UI you want within the `invoked` handler. The [SettingsFlyout](#) control, however, provides for the recommended narrow and wide sizes, supplies enter and exit animations, fires events like `[before | after][show | hide]`⁴³ and other such features. And since you can place any HTML you want within the control, including other controls, and the flyout will automatically handle vertical scrolling, there's really no reason *not* to use it.

As a WinJS control, you can declare a [SettingsFlyout](#) for each one of your commands in markup (making sure [WinJS.UI.process/processAll](#) is called, which handles any other controls in the flyout). For example, Scenario 2 of the [App settings sample](#) has the following flyout for help (omitting the text content and reformatting a bit), the result of which is shown in Figure 8-4:

```
<div data-win-control="WinJS.UI.SettingsFlyout" aria-label="Help settings flyout"
    data-win-options="{settingsCommandId:'help', width:'wide'}">
    <!-- Use either 'win-ui-light' or 'win-ui-dark' depending on the contrast between
         the header title and background color; background color reflects app's
         personality -->
    <div class="win-ui-dark win-header" style="background-color:#00b2f0">
        <button type="button" onclick="WinJS.UI.SettingsFlyout.show()"
            class="win-backbutton"></button>
        <div class="win-label">Help</div>
        
    </div>
    <div class="win-content ">
        <div class="win-settings-section">
            <h3>Settings charm usage guidelines summary</h3>
            <!-- Other content omitted -->
            <li>For more in-depth usage guidance, refer to the
                <a href="http://msdn.microsoft.com/en-us/library/windows/apps/hh770544">
                    App settings UX guide</a>.</li>
        </div>
    </div>
</div>
```

As always, there are options for this control as well as a few applicable `win-*` style classes. The only two options are `settingsCommandId`, for obvious purpose, and `width`, which can be `'narrow'` or `'wide'`. We see these both in the example above. The styles that apply here are `win-settingsflyout`, which styles the whole control (typically not used except for scoping other style rules), and `win-ui-light` and `win-ui-dark`, which apply a light or dark theme to the parts of the flyout. In this example, we use the dark theme for the header while the rest of the flyout uses the default light theme.

⁴³ How's that for a terse combination of four event names? It's also worth noting that the `document.body.-DOMNodeInserted` event will also fire when a flyout appears.

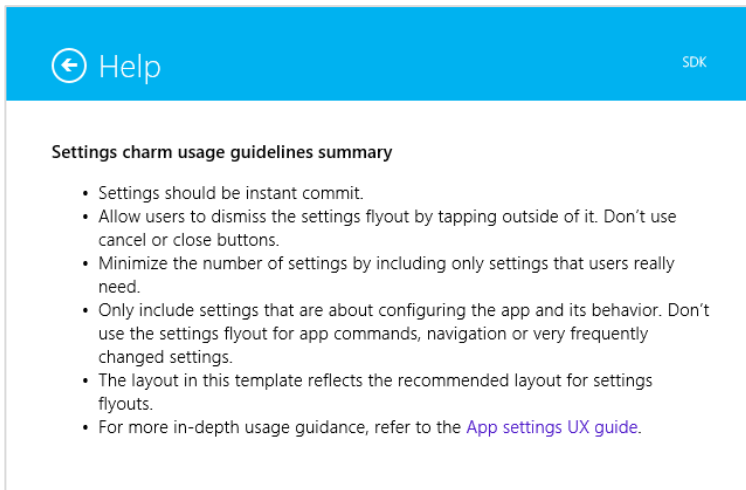


FIGURE 8-4 The Help settings flyout (truncated vertically) from Scenario 2 of the App settings sample. Notice the hyperlink on the lower right.

In any case, you can see that everything within the control is just markup for the flyout contents, nothing more, and you can wire up events to controls in the markup or in code. You're free to use hyperlinks here, such as to launch the browser to open a fuller Help page. You can also use an `iframe` to directly host web content within a settings flyout, as demonstrated in Scenario 3 of the same sample.

So how do we get this flyout to show when a command is invoked on the top-level settings pane? The easy way is to let WinJS take care of the details using the information you provide to `WinJS.UI.SettingsFlyout.populateSettings`. Here's the example again from Scenario 2, as we saw in the previous section:

```
WinJS.Application.onsettings = function (e) {
    e.detail.applicationcommands =
        { "help": { title: "Help", href: "/html/2-SettingsFlyout-Help.html" } };
    WinJS.UI.SettingsFlyout.populateSettings(e);
};
```

In the JSON you assign to `applicationCommands`, each object identifies both a command and its associated flyout. The name of the object is the flyout id ("help"), its `title` property provides the command label for the top-level settings pane ("Help" in the above), and its `href` property identifies the HTML page where the flyout with that id is declared ("/html/2-SettingsFlyout-Help.html").

With this information, WinJS can both populate the top-level settings pane and provide automatic invocation of the desired flyout (calling `WinJS.UI.process` all along the way) without you having to write any other code. This is why in most of the scenarios of the sample you don't see any explicit calls to `showSettings`, just a call to `populateSettings`.

Programmatically Invoking Settings Flyouts

Let's now see what's going on under the covers. In addition to being a control that you use to define a specific flyout, `WinJS.UI.SettingsFlyout` has a couple of other static methods besides `populateSettings`: `show` and `showSettings`. The `show` method specifically brings out the top-level Windows settings pane—that is, `Windows.UI.ApplicationSettings.SettingsPane`. This is why you see the back button's `click` event in the above markup wired directly to `show`, because the back button should return to that top-level UI.

Note While it's possible to programmatically invoke your own settings panes, you cannot do so with system-provided commands like Permissions and Rate and Review. If you have a condition for which you need the user to change a permission, such as enabling geolocation, the recommendation is to display an error message that instructs the user to do so.

The `showSettings` method, on the other hand, shows a *specific* settings flyout that you define somewhere in your app. The signature of the method is `showSettings(<id> [, <page>])` where `<id>` identifies the flyout you're looking for and the optional `<page>` parameter identifies an HTML document to look in if a flyout with `<id>` isn't found in the current document. That is, `showSettings` will always start by looking in the current document for a `WinJS.UI.SettingsFlyout` element that has a matching `settingsCommandId` property or a matching HTML `id` attribute. If such a flyout is found, that UI is shown.

If the markup in the previous section (with Figure 8-4) was contained in the same HTML page that's currently loaded in the app, the following line of code will show that flyout:

```
WinJS.UI.SettingsFlyout.showSettings("help");
```

In this case you could also omit the `href` part of the JSON object passed to `populateCommands`, but only again if the flyout is contained within the current HTML document already.

The `<page>` parameter, for its part, allows you to separate your settings flyouts from the rest of your markup; its value is a relative URI within your app package. The App settings sample uses this to place the flyout for each scenario into a separate HTML file. You can also place all your flyouts in one HTML file, so long as they have unique ids. Either way, if you provide a `<page>`, `showSettings` will load that HTML into the current page using `WinJS.UI.Pages.load` (which calls `WinJS.UI.processAll`), scans that DOM tree for a matching flyout with the given `<id>`, and shows it. Failure to locate the flyout will cause an exception.

Scenario 5 of the sample shows this form of programmatic invocation. This is also a good example (see Figure 8-5) of a vertically scrolling flyout:

```
WinJS.UI.SettingsFlyout.showSettings("defaults", "/html/5-SettingsFlyout-Settings.html");
```

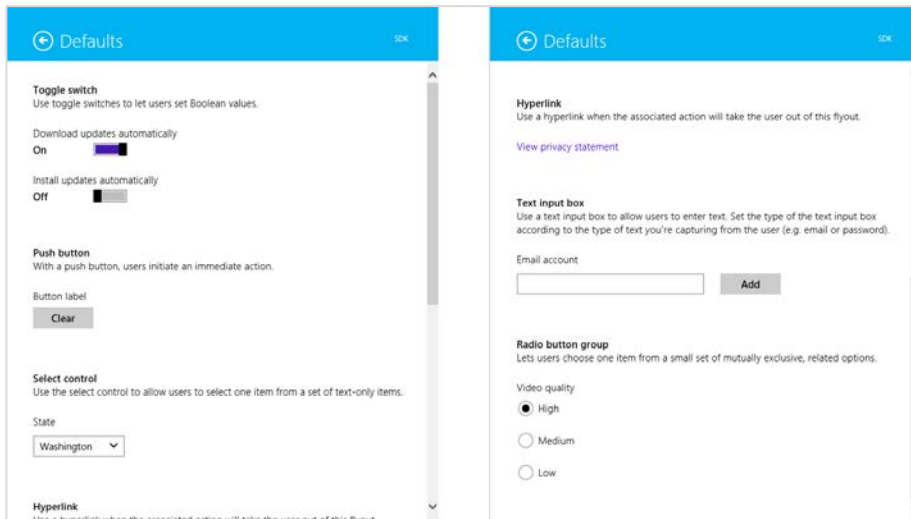


FIGURE 8-5 The settings flyout from Scenario 5 of the App settings sample, showing how a flyout supports vertical scrolling; note the scrollbar positions for the top portion (left) and the bottom portion (right).

A call to `showSettings` is thus exactly what you use within any particular command's `invoked` handler and is what WinJS sets up within `populateCommands`. But it also means you can call `showSettings` from anywhere else in your code when you want to display a particular settings pane. For example, if you encounter an error condition in the app that could be rectified by changing an app setting, you can provide a button in the message dialog of notification flyout that calls `showSettings` to open that particular pane. And for what it's worth, the `hide` method of that flyout will dismiss it; it doesn't affect the top-level settings pane for which you must use `Windows.UI.Application-Settings.SettingsPane.getForCurrentView.hide`.

You might use `showSettings` and `hide` together, in fact, if you need to navigate to a third-level settings pane. That is, one of your own settings flyouts could contain a command that calls `hide` on the current flyout and then calls `showSettings` to invoke another. The back button of that subsidiary flyout (and it should always have a back button) would similarly call `hide` on the current flyout and `showSettings` to make its second-level parent reappear. That said, we don't recommend making your settings so complex that third-level flyouts are necessary, but the capability is there if you have a particular scenario that demands it.

Knowing how `showSettings` tries to find a flyout is also helpful if you want to create a `WinJS.UI.SettingsFlyout` programmatically. So long as such a control is in the DOM when you call `showSettings` with its id, WinJS will be able to find it and display it like any other. It would also work, though I haven't tried this and it's not in the sample, to use a kind of hybrid approach. Because `showSettings` loads the HTML page you specify as a page control with `WinJS.UI.Pages.load`, that page can also include its own script wherein you define a page control object with methods like `processed` and `ready`. Within those methods you could then make specific customizations to the settings flyout defined in the markup.

Sidebar: Changes to Permissions

A common question along these lines is whether an app can receive events when the user changes settings within the Permissions pane. The answer is no, which means that you discover whether access is disallowed only by handling Access Denied exceptions when you try to use the capability. To be fair, though, you always have to handle denial of a capability gracefully because the user can always deny access the first time you use the API. When that happens, you again display a message about the disabled permission (as shown with the Here My Am! app from Chapter 7) and provide some UI to reattempt the operation. But the user still needs to invoke the Permissions settings manually. Refer also to the [Guidelines for devices that access personal data](#) for more details.

User Data: Libraries, File Pickers, and File Queries

Now that we've thoroughly explored app data and app settings, we're ready to look at the other part of state: user data. User data, again, is all the good stuff an app might use or generate that isn't specifically tied to the app. Multiple apps might be able to work with the same files, such as pictures and music, and user data always stays on a device regardless of what apps are present.

Our first concern with user data is where to put it and where to access it, which involves the various user data libraries, removable storage, and the file pickers. Using the access cache is also important to remember the fact that a user once granted access to a file or folder that we're normally not allowed to touch programmatically. The good thing about all such files and folders is that working with them happens through the same `StorageFolder` and `StorageFile` classes we've already seen. The other main topic we'll explore is that of file queries, a richer way to enumerate the contents of folders and libraries that lend very well to visual representations within controls like a `ListView`.

As we've seen, a Windows Store app, by default, has access only to its package and its `AppData` folders. This means that, by default, it doesn't actually have any access to typical locations for user data! There are then two ways that such access is acquired:

- Declare a library capability in the manifest.
- Let the user choose a location through the File Picker.

We'll look first at the File Picker, because in many cases it's all you really need in an app! But there are other scenarios—such as gallery-style apps—where you need direct access, so there are five capabilities in the manifest for this purpose, as shown in Figure 8-6 (left side). Three of them—*Music Library*, *Pictures Library*, and *Videos Library*—grant full read-write access to the user's Music, Pictures, and Videos folders. These appear on the app's product page in the Windows Store and on the Permissions settings pane, but they are not subject to user consent at run time. Of course, if it's not obvious why you're declaring these capabilities, be sure to explain yourself on your product page. And

as for *Documents Library* and *Removable Storage*, simply declaring the capability isn't sufficient: you also need to declare specific file type associations to which you're then limited. (The *Documents Library* capability is intended only for apps that need to open embedded content in another document.)

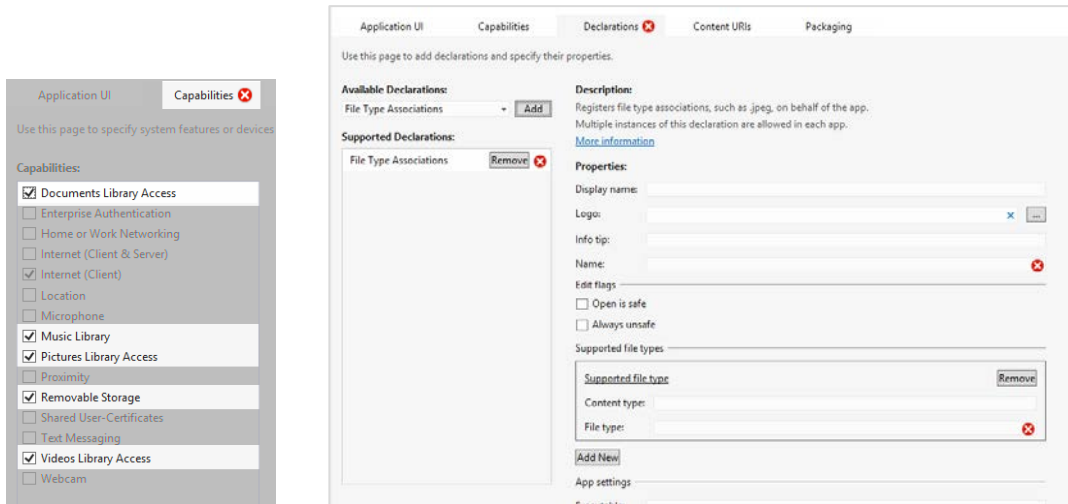


FIGURE 8-6 Capabilities related to user data in the manifest editor (left) and the file type association editor (right). Notice that the red X appears on Capabilities when additional declarations are needed in conjunction with this capability. The red X on Declarations indicates that the information is not yet complete.

Sidebar: The Background Transfer API

A topic that's relevant to user data, but one that we won't cover in detail until Chapter 14 is the [Windows.Networking.BackgroundTransfer](#) API of WinRT. This API allows you to run downloads and uploads independently of app lifetime—that is, while the app is running, suspended, or not running at all. This API is provided because transfer of large files to and from online resources is a common need for apps but one that doesn't really need the apps themselves to run in the background and consume power. Instead, apps set up transfer operations with the system that will continue if the app is shut down. When the app is relaunched, it can then check on the status of those transfers.

Using the File Picker

Although the File Picker doesn't sound all that glamorous, it's actually, to my mind, one of the coolest features in Windows 8. "Wait a minute!" you say, "How can a UI to pick a file or folder be, well, *cool*!" The reason is that this is *the* place where the users can browse and select from their entire world of data. That world includes not only what's on their local file system or the local network, but also any data that's made available by what are called *file picker providers*. These are apps that specifically take a library of data that's otherwise buried behind a web service, within an app's own database, or even generated on the fly, and makes it appear as if it's part of the local file system.

Think about this for a moment (as I invited you to do way back in Chapter 1). When you want to work with an image from a photo service like Flickr or Picasa, for example, what do you typically have to do? First step is to download that file to the local file system within some app that gives you an interface to that service (which might be a web app). Then you can make whatever edits and modifications you want, after which you typically need to upload the file back to the service. Well, that's not so bad, except that it's time consuming, forces you to switch between multiple apps, and eventually litters your system with a bunch of temporary files, the relationship of which to your online files is quickly forgotten.

Having a file picker provider that can surface such data directly, both for reading and writing, eliminates all those intermediate steps, and eliminates the need to switch apps. This means that a provider for a photo service makes it possible for other apps to load, edit, and save online content as if it all existed on the local file system. Consuming apps don't need to know anything about those other services, and they automatically have access to more services as more provider apps are installed. What's more, providers can also make data that isn't normally stored as files appear as though they are. For example, the Windows 8 Camera app is a file picker provider that lets you can activate your camera, take a picture, and have it returned as if you loaded it from the file system. All of this gives users a very natural means to flow in and out of data no matter where it's stored. Like I said, I think this is a very cool feature!

We'll look more at the question of providers in Chapter 12. Our more immediate concern is how we make use of these file pickers to obtain a [StorageFile](#) or [StorageFolder](#) object.

The File Picker UI

Before looking at the code, let's familiarize ourselves with the file picker UI itself. When invoked, you'll see a full-screen view like that in Figure 8-7, which shows the picker in single-selection mode with a "thumbnail" view. In such a view, items are shown as images in a ListView with a rich tooltip control appearing when you hover over an item. In a way, the file picker itself is like an app that's invoked for this purpose, and it's designed to be beautiful and immersive just like other Windows Store apps.

In Figure 8-7, the Pictures heading shows the current location of the picker. The Sort By Name drop-down list lets you choose other sorting criteria, and the drop-down list next to the Files header lets you choose other locations, as shown in Figure 8-8. These locations include other areas of the file system (though never protected areas like the Windows folder or Program Files), network locations, *and* other provider apps.

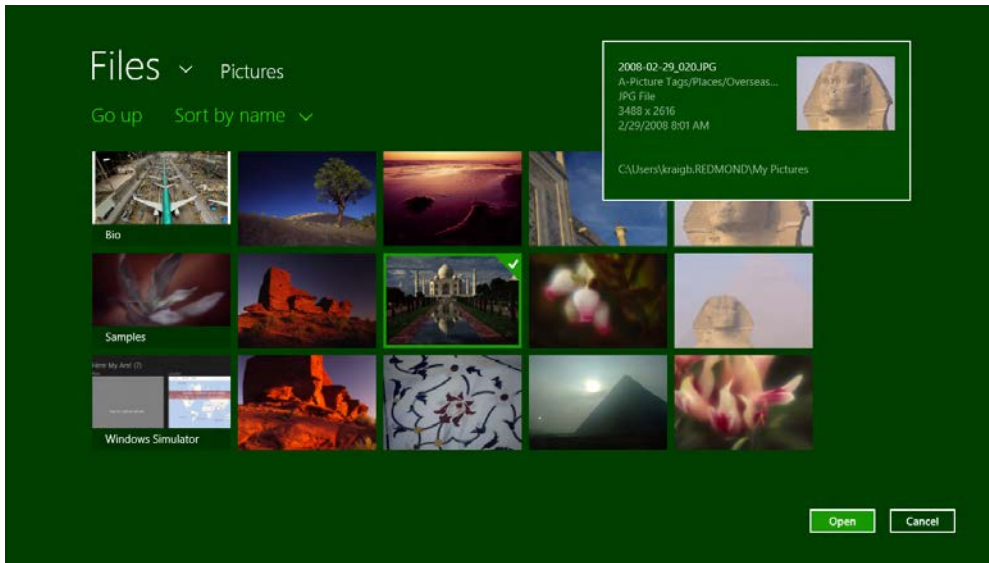


FIGURE 8-7 A single-selection file picker on the Pictures library in thumbnail view mode, with a hover tooltip showing for one of the items (the head of the Sphinx) and the selection frame showing on another (the Taj Mahal).

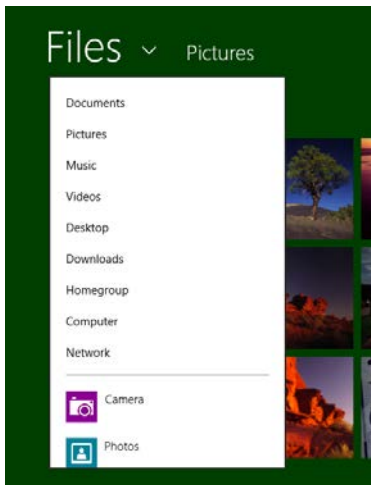


FIGURE 8-8 Selecting other locations in which to browse files; notice that apps are listed along with file system locations.

Choosing another file system location navigates there, of course, from which you can browse into other folders. Selecting an app, on the other hand, launches that app through the file picker provider contract. In this case it appears within a system-provided (but app-branded) UI like that shown in Figure 8-9. Here the drop-down list next to the heading lets you switch back to other picker locations, and the Open and Cancel buttons act as they do for other picker selections. In short, a provider app really is just an extension to the File Picker UI, but a very powerful one at that. And ultimately such an app just

returns an appropriate [StorageFile](#) object that makes its way back to the original app. It's quite a lot happening with just a single call to the file picker API!

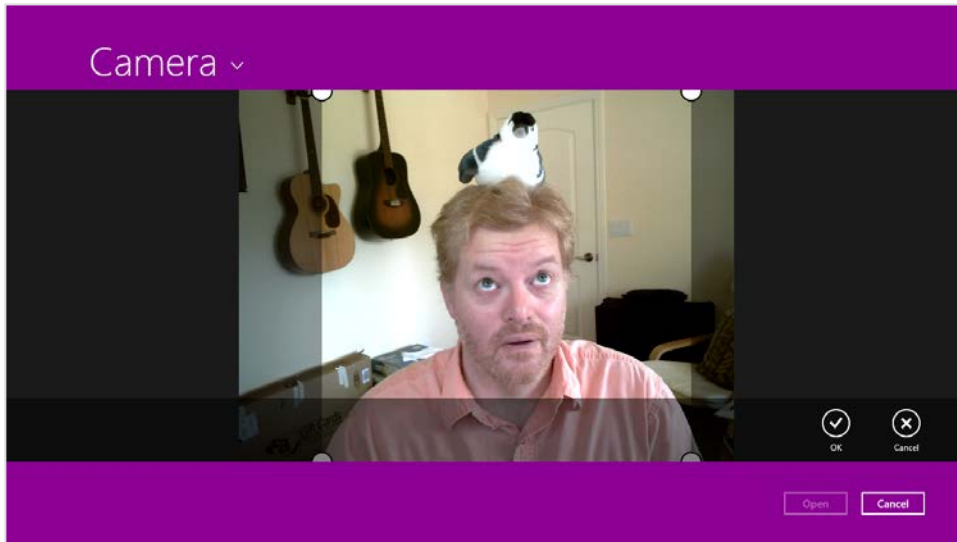


FIGURE 8-9 The camera app invoked through the file picker provider contract. Where did that nuthatch come from?

The file picker has a couple of other modes. One is the ability to select multiple files—even from different apps!—as shown in Figure 8-10, where all the selections are placed into what's called the *basket* on the bottom of the screen. The picker can also be used to select a folder, as shown in Figure 8-11 (provider apps aren't shown in this case), or a save location and filename, as shown in Figure 8-12.

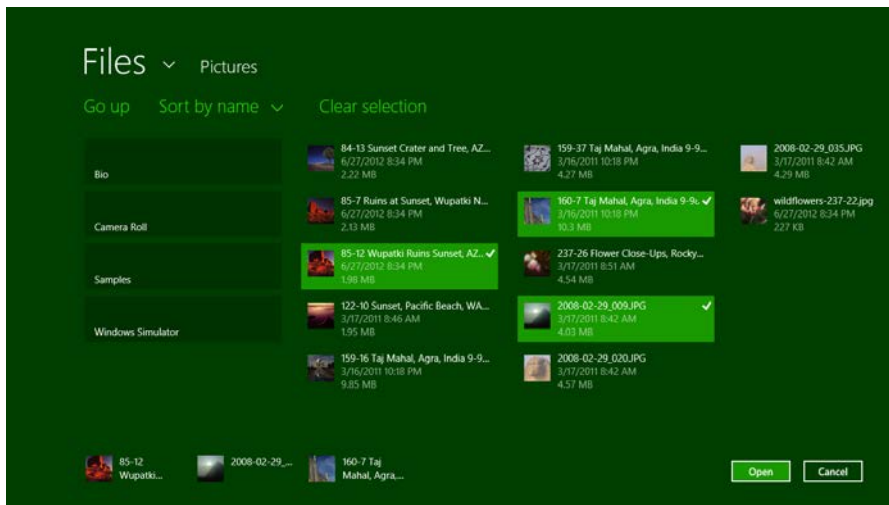


FIGURE 8-10 The file picker in multiselect mode with the selection basket at the bottom. What shown here is also the “list” view mode that’s set independently from the selection mode.

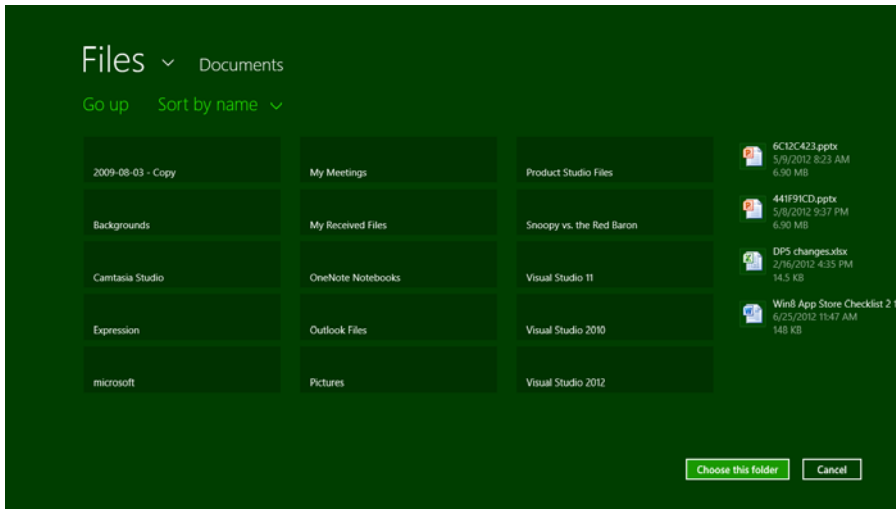


FIGURE 8-11 The file picker used to select a folder—notice that the button text changed and a preview of the folder contents appear on the right.

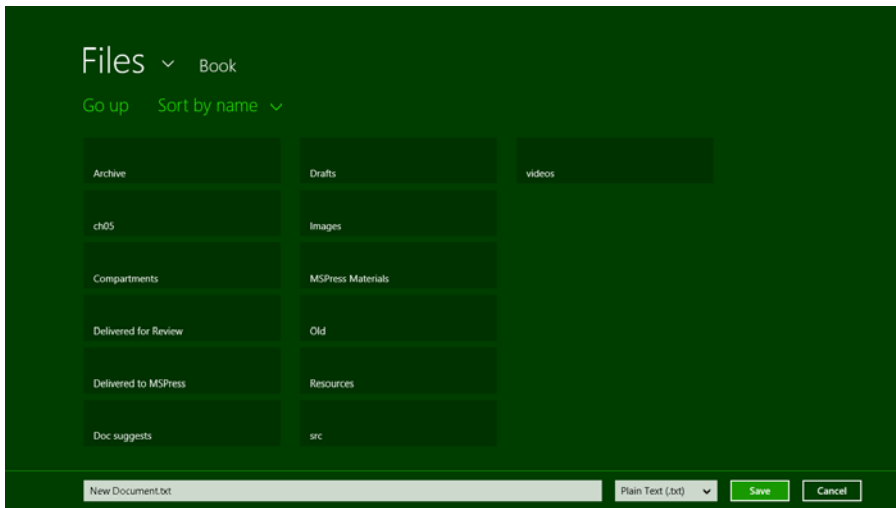


FIGURE 8-12 The file picker used to select a save location and filename.

The File Picker API (and a Few Friends)

Now that we've seen the visual results of the file picker, let's see how we invoke it from our app code through the API in [Windows.Storage.Pickers](#). All the images we just saw came from the [File picker sample](#), so we'll also use that as the source of our code.

For starters, Scenario 1 in its `pickSinglePhoto` function (`js/scenario1.js`) uses the picker to obtain a single `StorageFile` for opening (reading and writing):


```

function pickSinglePhoto() {
    // Verify that we are currently not snapped, or that we can unsnap to open the picker
    var currentState = Windows.UI.ViewManagement.ApplicationView.value;
    if (currentState === Windows.UI.ViewManagement.ApplicationViewState.snapped &&
        !Windows.UI.ViewManagement.ApplicationView.tryUnsnap()) {
        // Fail silently if we can't unsnap
        return;
    }

    // Create the picker object and set options
    var openPicker = new Windows.Storage.Pickers.FileOpenPicker();
    openPicker.viewMode = Windows.Storage.Pickers.PickerViewMode.thumbnail;
    openPicker.suggestedStartLocation =
        Windows.Storage.Pickers.PickerLocationId.picturesLibrary;

    // Users expect to have a filtered view of their folders depending on the scenario.
    // For example, when choosing a documents folder, restrict the filetypes to documents
    // for your application.
    openPicker.fileTypeFilter.replaceAll([".png", ".jpg", ".jpeg"]);

    // Open the picker for the user to pick a file
    openPicker.pickSingleFileAsync().done(function (file) {
        if (file) {
            // Application now has read/write access to the picked file
        } else {
            // The picker was dismissed with no selected file
        }
    });
}

```

As you can see, you should not try to invoke the File Picker when in snapped view; this will, like the Settings Pane, cause an exception. You can check for such a condition ahead of time, as shown here, or you can add an error handler within the `done` at the end.⁴⁴ In any case, to invoke the picker we create an instance of `Windows.Storage.Pickers.FileOpenPicker`, configure it, and then call its `pickSingleFileAsync` method. The result of `pickSingleFileAsync` is the `file` argument given to the completed handler, which will be either a `StorageFile` object for the selected file or `null` if the user canceled. This is why you must always check that the picker's result is not `null`.

With the configuration, here we're setting the picker's `viewMode` to `thumbnail` (from the enumeration `Windows.Storage.Pickers.PickerViewMode`), resulting in the view of Figure 8-7. The other possibility here is `list`, which gives a view like Figure 8-10.

We also set the `suggestedStartLocation` to the `picturesLibrary`, which is a value from the `Windows.Storage.Pickers.PickerLocationId` enumeration; other possibilities are `documentsLibrary`, `computerFolder`, `desktop`, `downloads`, `homeGroup`, `musicLibrary`, and `videosLibrary`, basically all the other locations you see in Figure 8-8. Note that using these locations does *not* require you to declare capabilities in your manifest because by using the picker, the user is giving consent for you to access

⁴⁴ The sample, it should be noted, uses `then` instead of `done` on that last async call; while `then` works, it should actually be `done` especially if you're going to handle exceptions there.

those files. If you check the manifest in this sample, you'll see that no capabilities are declared at all.

The one other property we set is the `fileTypeFilter` (a [FileExtensionVector](#) object) to indicate the type of files we're interested in (PNG and JPEG). Beyond that, the `FileOpenPicker` also has a `commitButtonText` property that sets the label of the primary button in the UI (the one that's not Cancel), and `settingsIdentifier`, a means to essentially remember different contexts of the file picker. For example, an app might use one identifier for selecting pictures, where the starting location is set to the pictures library and the view mode to thumbnails, and another id for selecting documents with a different location and perhaps a list view mode.

This sample, as you can also see, doesn't actually do anything with the file once it's obtained, but it's quite easy to see what we might do. We can, for instance, simply pass the `StorageFile` to `URL.createObjectURL` and assign the result to an `img.src` property for display. The same thing could be done with audio and video, possibilities that are all demonstrated in Scenario 1 of the [Using a blob to save and load content sample](#) I mentioned earlier in this chapter. That sample also shows reading the file contents through the HTML `FileReader` API alongside the other WinRT and WinJS APIs we've seen. You could also transcode an image (or other media) in the `StorageFile` to another format (as we'll see in Chapter 10), retrieve thumbnails as shown in the [File and folder thumbnail sample](#), or use the `StorageFile` methods to make a copy in another location, rename the file, and so forth. But from the file picker's point of view, its particular job was well done!

Returning now to the file picker sample, picking multiple files is pretty much the same story as shown in the `pickMultipleFiles` function of `js/scenario2.js`. Here we're using the `list` view mode and starting off in the `documentsLibrary`. Again, these start locations don't require capability declarations in the manifest:

```
function pickMultipleFiles() {
    // Verify that we are currently not snapped, etc... (some code omitted)

    // Create the picker object and set options
    var openPicker = new Windows.Storage.Pickers.FileOpenPicker();
    openPicker.viewMode = Windows.Storage.Pickers.PickerViewMode.list;
    openPicker.suggestedStartLocation =
        Windows.Storage.Pickers.PickerLocationId.documentsLibrary;
    openPicker.fileTypeFilter.replaceAll(["*"]);

    // Open the picker for the user to pick a file
    openPicker.pickMultipleFilesAsync().done(function (files) {
        if (files.size > 0) {
            // Application now has read/write access to the picked file(s)
        } else {
            // The picker was dismissed with no selected file
        }
    });
}
```

When picking multiple files, the result of `pickMultipleFilesAsync` is a [FilePickerSelected-FilesArray](#) object, which you can access like any other array using `[]` (though it has limited methods otherwise).

Scenario 3 of the sample shows a call to `pickSingleFolderAsync`, where the result of the operation is a `StorageFolder`. Here you must indicate a `fileTypeFilter` that helps users pick an appropriate location where some files of that type exist, or where they can create a new one (`js/scenario3.js`):

```
function pickFolder() {
    // Verify that we are currently not snapped... (some code omitted)

    // Create the picker object and set options
    var folderPicker = new Windows.Storage.Pickers.FolderPicker;
    folderPicker.suggestedStartLocation = Windows.Storage.Pickers.PickerLocationId.desktop;
    folderPicker.fileTypeFilter.replaceAll([".docx", ".xlsx", ".pptx"]);

    folderPicker.pickSingleFolderAsync().then(function (folder) {
        if (folder) {
            // Cache folder so the contents can be accessed at a later time
            Windows.Storage.AccessCache.StorageApplicationPermissions.futureAccessList
                .addOrReplace("PickedFolderToken", folder);
        } else {
            // The picker was dismissed with no selected file
        }
    });
}
```

In this example we also see how to save that selected `StorageFolder` in the [Windows.Storage.-AccessCache](#) for future use. Again, by selecting this folder the user has granted the app programmatic access to its contents, but only for the current session. To maintain that access, the app must save the storage item in the `futureAccessList` of the cache, where it can be later retrieved using the `futureAccessList.getFolderAsync`, `getItemAsync`, or `getFileAsync` methods. As before, refer to Scenario 6 of the [File access sample](#) for more on this feature, and note that the `AccessCache` API also provides for recently used items as well. The key thing to remember here is that for any location outside of your package, app data, or libraries for which you've declared access, you must use the `AccessCache` to maintain access in the future. It won't work to save a pathname to such locations and attempt to open files again later.

For the final file picker use case, Scenario 4 of the file picker sample creates a [FileSavePicker](#) object and calls its `pickSaveFileAsync` method, resulting in the UI of Figure 8-12:

```
function saveFile() {
    // Verify that we are currently not snapped... (some code omitted)

    // Create the picker object and set options
    var savePicker = new Windows.Storage.Pickers.FileSavePicker();
    savePicker.suggestedStartLocation =
        Windows.Storage.Pickers.PickerLocationId.documentsLibrary;
    // Dropdown of file types the user can save the file as
    savePicker.fileTypeChoices.insert("Plain Text", [".txt"]);
}
```

```

// Default file name if the user does not type one in or select a file to replace
savePicker.suggestedFileName = "New Document";

savePicker.pickSaveFileAsync().done(function (file) {
    if (file) {
        // Prevent updates to the remote version of the file until we finish making
        // changes and call CompleteUpdatesAsync.
        Windows.Storage.CachedFileManager.deferUpdates(file);

        // write to file
        Windows.Storage.FileIO.writeTextAsync(file, file.name).done(function () {
            // Let Windows know that we're finished changing the file so the other app
            // can update the remote version of the file.
            // Completing updates might require Windows to ask for user input.
            Windows.Storage.CachedFileManager.completeUpdatesAsync(file)
                .done(function (updateStatus) {
                    if (updateStatus ===
                        Windows.Storage.Provider.FileUpdateStatus.complete) {
                    } else {
                        // ...
                    }
                })
            });
        } else {
            // The picker was dismissed
        }
    }
});
}

```

The `FileSavePicker` has many of the same properties as the `FileOpenPicker`, but it replaces `fileTypeFilter` with `fileTypeChoices` (to populate the drop-down list) and includes a `suggestedFileName` (a string), `suggestedSaveFile` (a `StorageFile`), and `defaultFileExtension` (a string). What's interesting in the code above are the interactions with the `Windows.Storage.CachedFileManager`. This object helps file picker providers know when they should synchronize local and remote files, which would be necessary when a file consumer saves new content as we see here. From the consumer side, what we see here is a typical pattern for files obtained from the file picker (or the access cache if saved there in a previous session): we simply need to let the `CachedFileManager` know that we're writing to the file and tell it when we're done. Of course, this isn't needed when working with files that you know are local, such as those in your AppData folders. We'll learn more about this mechanism in Chapter 12 when we look at the provider side.

Media Libraries

Now that we've seen understood the capabilities of the file picker, we can turn our attention to the other libraries. But before you start checking off capabilities in your manifest, pause for a moment to ask this: are those capabilities actually needed? The file pickers provide very extensive access to all these libraries without needing any capabilities at all. Through the pickers you can have the user select one or more files to open, manipulate, and save; the user can give you access to a folder; and the user can

indicate a new filename in which to save user data.

You only need specific library access if you're going to work within any of these libraries outside of the file picker. For example, if you want to enumerate the contents of the Pictures or Music folder to display a list in a ListView or FlipView control, as we did in Chapter 5, you do need to declare a capability.

To be even more specific, without going through the file picker there is only *one* way to gain programmatic access to a media library: obtaining a [StorageFolder](#) from the [Windows.Storage.KnownFolders](#) object. For media, the applicable properties here are [picturesLibrary](#), [musicLibrary](#), and [videosLibrary](#). Without the appropriate capability, trying to retrieve one of these will throw an access denied exception.

If you don't need to access [KnownFolders](#), then, you don't need to declare the capabilities! And remember that since all your declared capabilities are listed for your app in the Windows Store and might make consumers think twice about installing your app, fewer is definitely better.

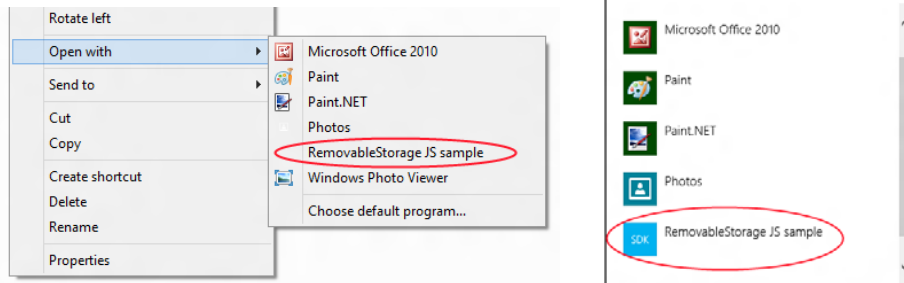
That said, if you do decide to access media libraries directly, your work there involves [StorageFolder](#) and [StorageFile](#) objects pretty much like any other storage location. One difference, however, is that you can work with the metadata often included with media files; we'll see a little more of this in Chapter 10.

Documents and Removable Storage

As with the media libraries, programmatic access to the user's documents folder as well as removable storage devices are controlled by capability declarations. It's important once again to understand that both of these capabilities also require you to declare file type associations, meaning that you cannot simply enumerate the contents of these folders directly or write whatever files you want therein. Put another way, going directly to these folders—through [Windows.Storage.KnownFolders.documentsLibrary](#) and [removableDevices](#), both of which are [StorageFolder](#) objects—just for a limited set of file types is useful only in very particular scenarios. For the documents library, in fact, the documentation states that “the only acceptable use of the [capability] is to support the opening of embedded content within another document.” The Windows Store will also ask you to justify your declaration of the capability directly and also requires that you have a “company account” on the Windows Store, not just an individual account.

It's more likely that you simply want to allow the user to open or save a file in these locations, or enumerate their contents, in which case all you need is the file picker. This way you don't need to associate your app with specific tile types, which also says that your app is available to service such files at any time.

For example, the [Removable storage sample](#), in order to demonstrate access to removable devices, declares associations with .gif, .jpg, and .png files. As a result, it shows up in Open With lists like the context menu of Windows Explorer on the desktop and the default program selector:



The same is also true for documents (see the [File access sample](#) again), so unless your app is again positioned to service those file types, you probably don't need these capabilities.

How exactly to declare file type associations is a form of contracts, so we'll cover the details in Chapter 12.

Rich Enumeration with File Queries

To enumerate files within a particular location (capabilities aside), you employ what's called a *file query* or, simply, a search. A file query is exactly what Windows Explorer on the desktop uses to search the file system and can involve file contents as well as any number of other properties/metadata. These queries involve what are known as [Advanced Query Syntax \(AQS\)](#) strings that are capable of identifying and describing as many specific criteria you desire. As such, the whole topic is somewhat beyond the scope of this book, but we can at least touch on how WinRT makes the power of file queries available to apps through the [Windows.Storage.Search](#). One foot in the deep rabbit hole will be enough!

Indeed, there are literally thousands of options you can use in an AQS string, because they are built from any number of [Windows properties](#) such as *System.ItemDate*, *System.Author*, *System.Keywords*, *System.Photo.LightSource*, and so on.⁴⁵ Each property can contain a target value such as *System.-Author(Patrick or Bob)* and *System.ItemType: "mp3"*, and terms can be combined with AND, OR, and NOT operators. We'll see many more examples in Chapter 10, where we can use queries to retrieve collections of files in many different "shapes" such as a flat list, a hierarchy, and various sort orders, including those oriented around media properties. In addition, file queries also provide for obtaining thumbnails as well as automatic retrieval of album art for music.

Here, let's concentrate on understanding how file queries work, starting with the basics that are demonstrated in the FileQuery example included with this chapter's companion content. This example is a copy of the [Programmatic file search sample](#) in the Windows SDK, which itself has only one

⁴⁵ Contrary to any examples in the docs, queries should *always* use the full name of Windows properties such as *System.ItemDate*: rather than the user-friendly shorthand *date*: because the latter will not work on localized builds of Windows.

scenario oriented on the music library that lets you enter in an AQS string directly. However, this isn't always what you'll be using in an app, so I wanted to show many other variations.

Queries always start with a [StorageFolder](#), whose [createFileQuery\[WithOptions\]](#), [createFolderQuery\[WithOptions\]](#), and [createItemQuery\[WithOptions\]](#) methods (6 total) provide for enumerating files, folders, or both, within whatever folder the [StorageFolder](#) is attached to. The simplest queries are created with the base [StorageFolder.create*](#) methods and no parameters:

```
folder.createFileQuery();
folder.createFolderQuery();
folder.createItemQuery();
```

The first two methods here are actually just shortcuts for the one-parameter variants with the same names, where that parameter is a value from the [Windows.Storage.Search.CommonFileQuery](#) or [CommonFolderQuery](#) enumerations. These shortcut versions just use the [defaultQuery](#) value for a simple alphabetical, shallow enumeration of the folder contents. [createItemQuery](#), for its part, has only this one form.

Creating a query itself doesn't actually enumerate anything until you ask it to through an async method: for file queries the method is [getFilesAsync](#), for folders it's [getFoldersAsync](#), and for items it's [getItemsAsync](#). (See a pattern here?) So, in Scenario 2 of the FileQuery example, I have these three functions attached to buttons:

```
function fileQuery() {
    var query = picturesLibrary.createFileQuery();
    SdkSample.showResults(query.getFilesAsync());
}

function folderQuery() {
    var query = picturesLibrary.createFolderQuery();
    SdkSample.showResults(query.getFoldersAsync());
}

function itemQuery() {
    var query = picturesLibrary.createItemQuery();
    SdkSample.showResults(query.getItemsAsync());
}
```

where the [SdkSample.showResults](#) function in `js/default.js` just creates a listing of the items in the collection. Running this sample, you'll see a list of those files and/or folders in your Pictures library.

Tip The actual object types returned by these [create*Query](#) APIs are [StorageFileQueryResult](#), [StorageFolderQueryResult](#), and [StorageItemQueryResult](#), all in the [Windows.Storage.Search](#) namespace. These all provide some additional properties like [folder](#), methods like [findStartIndexAsync](#) and [getItemCountAsync](#), and events like [optionschanged](#) and [contentschanged](#) (both WinRT events). The latter event especially is something you can use to monitor changes to the file system that affect query results.

Beyond this shallow default behavior, file and folder queries have many other possibilities as expressed in the [CommonFileQuery](#) and [CommonFolderQuery](#) enumerations:

- [CommonFileQuery](#): [orderByName](#), [orderByTitle](#), [orderByDate](#), [orderByMusic-Properties](#), and [orderBySearchRank](#).
- [CommonFolderQuery](#): [groupByType](#), [groupByTag](#), [groupByAuthor](#), [groupByYear](#), [groupByMonth](#), [groupByArtist](#), [groupByComposer](#), [groupByGenre](#), [groupByPublishedYear](#), and [groupByRating](#).

Clearly, the effect of these choices depends on the queried items actually containing metadata that supports the ordering or grouping, but it is allowable to query all folders for all types of files and folders. To demonstrate this, Scenario 3 of the [FileQuery](#) example lets you choose the music, pictures, or videos library. Then you can choose whether to query files or folders, choose the common query to apply, and run a search to see the results. Note that using [orderBySearchRank](#) with files isn't really meaningful in this context because it's meant to work with AQS-based searches. We'll see this a little later. (Also—call me a slacker!—the results of a grouped folder query isn't very interesting when one doesn't group the display output, but for an example of that you can refer to Scenario 2 of the [Folder enumeration sample](#).)

The code in `js/scenario3.js` is pretty much just the mechanics of mapping your UI selections to either [createFileQuery](#) or [createFolderQuery](#) with the right parameters, so there's no need to look at most of it here. One important piece is the use of the [isCommonFileQuerySupported](#) and [isCommonFolderQuerySupported](#) methods of [StorageFolder](#). These are used to test whether the current folder will actually support the particular query you want to try:

```
if (folder.isCommonFileQuerySupported(selectedQuery)) {  
    query = folder.createFileQuery(selectedQuery);  
    if (query) {  
        promise = query.GetFilesAsync();  
    }  
}
```

You'll find that when running the sample in the media libraries, at least, all the common file and folder queries are supported, but that might not be true for all [StorageFolder](#) objects you might encounter. Remember, for example, that the folder picker might give you a [StorageFolder](#) from a provider whose data is off in some online service or a database, in which case certain queries might not work.

A similar method, [StorageFolder.areQueryOptionsSupported](#), also exists to tests support for custom queries beyond the common ones. A custom query is described by a [Windows.Storage.-Search.QueryOptions](#) object (the common queries are just prepopulated instances of these) and is created by passing such an object to the [createFileQueryWithOptions](#), [createFolderQuery-WithOptions](#), and [createItemQueryWithOptions](#).

A `QueryOptions` is generally created from scratch using the `new` operator, after which you populate its properties. You can also use `new QueryOptions(<CommonFolderQuery>)` to retrieve the object for one of the common folder queries, and `new QueryOptions(<CommonFileQuery> [, <file type filter>])` to do the same for common file queries. In this latter case, an optional array of file types can also be given; this is a shortcut to quickly customize a common query with a specific set of file types. Without it, the filter is set to `"*"` by default. That is, if you wanted to just find `.mp3` files in your music library ordered by title, you would use this kind of code (see `js/scenario4.js` in the `FileQuery` example):

```
var musicLibrary = Windows.Storage.KnownFolders.musicLibrary;
var options = new Windows.Storage.Search.QueryOptions(
    Windows.Storage.Search.CommonFileQuery.orderByTitle, [".mp3"]);

if (musicLibrary.areQueryOptionsSupported(options)) {
    var query = musicLibrary.createFileQueryWithOptions(options);
    SdkSample.showResults(query.GetFilesAsync());
}
```

If you create a `QueryOptions` from scratch, you can set a number of options. The more general or basic ones are as follows:⁴⁶

- `fileTypeFilter` An vector of strings that describe the desired file type extensions, as in `".mp3"`. The default is an empty list (no filtering).
- `folderDepth` Either `Windows.Storage.Search.FolderDepth.shallow` (the default) or `deep`.
- `indexerOption` A value from `Windows.Storage.Search.IndexerOption`, which is one of `useIndexerWhenAvailable`, `onlyUseIndexer` (limit the search to indexed content only), and `doNotUseIndexer` (query the file system directly bypassing the indexer). As the latter is the default, you'll typically want to explicitly set this property to `useIndexerWhenAvailable`.
- `sortOrder` A vector of `Windows.Storage.Search.SortEntry` structures that each contain a Boolean named `ascendingOrder` (false for descending order) and a `propertyName` string. Each entry in the vector defines a sort criterion; these are applied in the order they appear in the vector. An example of this will be given a little later.

Three of the `QueryOptions` properties then apply to searches with AQS strings:

- `applicationSearchFilter` An AQS string.
- `userSearchFilter` Another AQS string.
- `language` A string containing the BCP-47 language tag associated with the AQS strings.

⁴⁶ Another property, `dateStackOption` (a value from `Windows.Storage.Search.DateStackOption`), is read-only within this structure but can be set when creating a `QueryOptions` from a `CommonFolderQuery`.

When the query is built through a method like `createFileQueryWithOptions`, the application and user filter strings here are combined. What this means is that you can separately manage any filter you want to apply generally for your app (`applicationSearchFilter`) from user-supplied search terms (`userSearchFilter`). This way you can enforce some search filters without requiring the user to type them in, and without always having to combine strings yourself.

As noted before, the `CommonFileQuery.orderBySearchRank` query is meaningful only when combined with an AQS string, which is to say that keyword-based searches return ranked results for which this common file query would apply. Returning to Scenario 1 of the Programmatic file search sample, then, we see how it uses this ordering along with the `userSearchFilter` property:

```
var musicLibrary = Windows.Storage.KnownFolders.musicLibrary;
var options = new Windows.Storage.Search.QueryOptions(
    Windows.Storage.Search.CommonFileQuery.orderBySearchRank, ["*"]);
options.userSearchFilter = searchFilter;
var fileQuery = musicLibrary.createFileQueryWithOptions(options);
```

On my machine, where I have a number of songs with "Nightingale" in the title, as well as an album called "Nightingale Lullaby," a search using the string `"Nightingale" System.ItemType: "mp3"` in the above code gives me results that look like this in the sample:

24 files found

```
04 Song Of The Nightingale2.mp3
16 Song Of The Nightingale (Instrumental).mp3
01 Song of the Nightingale.mp3
04 Song of the Nightingale.mp3
12 Song Of The Nightingale.mp3
- 18 - Twinkle Twinkle.mp3
- 17 - Tum-Balalayka.mp3
- 16 - Tina and Gina Bobina.mp3
- 15 - Shayna's Lullaby.mp3
- 14 - Rockabye Baby.mp3
- 12 - Lullaby for Carol.mp3
```

This shows that the search ranking favors songs with "Nightingale" directly in the title, but also includes those from an album with that name.

My search string here, by the way, shows how you might use the `applicationSearchFilter` and `userSearchFilter` properties together. If my app was capable of working only with mp3 or some other formats, I could store `"System.ItemType: 'mp3'"` in `applicationSearchFilter` and store user-provided terms like `"Nightingale"` in `userSearchFilter`. This way I avoid having to join them manually in my code.

Beyond the properties that you set within a `QueryOptions` object, it also has some information and capabilities of its own. The `groupPropertyName`, for one, is a string property that indicates the type of property that the query is being grouped by. You can also retrieve the query options as a string using the `saveToString` method and recreate the object from a string using `loadFromString` (that is, the analog of `JSON.stringify` and `JSON.parse`).

The `setPropertyPrefetch` method goes even deeper still, allowing you to indicate a group of file properties that you want to optimize for fast retrieval—they're accessed through the same APIs as file properties in general, but they come back faster, meaning that if you're displaying a collection of files in a `ListView` using a custom data source with certain properties from enumerated files, you'd want to set those up for prefetch so that the control renders faster. (The `WinJS.UI.StorageDataSource` does this already.) Similarly, `setThumbnailPrefetch` tells Windows what kinds of thumbnails you want to include in the enumeration—again, you can ask for these without setting the prefetch, but they come back faster when you do. This helps you optimize the display of a file collection.⁴⁷

We briefly saw similar usage of thumbnail properties back in Chapter 5, when we took advantage of a shortcut to the pictures library with `WinJS.UI.StorageDataSource` and could specify a thumbnail size option:

```
myFlipView.itemDataSource = new WinJS.UI.StorageDataSource("Pictures",
    { requestedThumbnailSize: 480 });
```

A more general example that also includes the `QueryOptions.sortOrder` vector can be found in the [StorageDataSource and GetVirtualizedFilesVector sample](#), which got a footnote in Chapter 5. In its `js/scenario2.js` we see the creation of a `QueryOptions` from scratch, setting up two `sortOrder` criteria, and setting up thumbnail options in the data source:

```
function loadListViewControl() {
    // Build datasource from the pictures library
    var library = Windows.Storage.KnownFolders.picturesLibrary;
    var queryOptions = new Windows.Storage.Search.QueryOptions;
    // Shallow query to get the file hierarchy
    queryOptions.folderDepth = Windows.Storage.Search.FolderDepth.shallow;
    queryOptions.sortOrder.clear();
    // Order items by type so folders come first
    queryOptions.sortOrder.append({ascendingOrder: false, propertyName: "System.IsFolder"});
    queryOptions.sortOrder.append({ascendingOrder: true, propertyName: "System.ItemName"});
    queryOptions.indexerOption =
        Windows.Storage.Search.IndexerOption.useIndexerWhenAvailable;

    var fileQuery = library.createItemQueryWithOptions(queryOptions);
    var dataSourceOptions = {
        mode: Windows.Storage.FileProperties.ThumbnailMode.picturesView,
        requestedThumbnailSize: 190,
        thumbnailOptions: Windows.Storage.FileProperties.ThumbnailOptions.none
    };

    var dataSource = new WinJS.UI.StorageDataSource(fileQuery, dataSourceOptions);

    // Create the ListView...
};
```

If you're really interested in digging deeper here, you can look at how `StorageDataSource` sets up file

⁴⁷ See [What's Changed for App Developers since the Consumer Preview](#) on the Windows 8 Developer Blog for a few more details on these.

queries; just search for this class in the ui.js file of WinJS and you'll find it. Along the way, you'll run into one more set of WinRT APIs—perhaps the bottom of the hole!—that I wanted to mention before wrapping up this subject: [Windows.Storage.BulkAccess](#). These actually exist solely for use by [StorageDataSource](#) and are not intended for direct use in apps. Even if you create your own data source or collection control, it's best to just use the enumeration and prefetch APIs we've already discussed, as they give identical performance.

Here My Am! Update

To bring together some of the topics we've covered in this chapter, the companion content includes another revision of the Here My Am! app with the following changes and additions (mostly to `pages/home/home.js` unless noted):

- It now incorporates the [Bing Maps SDK](#) so that the control is part of the package rather than loaded from a remote source. This eliminates the `iframe` we've been using to host the map, so all the code from `html/map.html` can move into `js/default.js`. Note that to run this sample in Visual Studio you need to download and install the SDK yourself.
- Instead of copying pictures taken with the camera to app data, those are now copied to a HereMyAm folder in the Pictures library. The *Pictures Library* capability has been declared.
- Instead of saving a pathname to the last captured image file, which is used when the app is terminated and restarted, the [StorageFile](#) is saved in [Windows.Storage.AccessCache](#) to guarantee future programmatic access.
- An added appbar command allows you to use the File Picker to select an image to load instead of relying solely on the camera. This also allows you to use a camera app, if desired. Note that we use a particular [settingsIdentifier](#) with the picker in this case to distinguish from the picker for recent images.
- Another appbar command allows you to choose from recent pictures from the camera. This defaults to our folder in the Pictures library and uses a different [settingsIdentifier](#).
- Additional commands for About, Help, and a Privacy Statement are included on the Settings pane using the [WinJS.Application.onsettings](#) event (see `js/default.js`). The first two display content from within the app whereas the third pulls down web content in an `iframe`; all the settings pages are found in the `html` folder of the project.

What We've Just Learned

- Statefulness is important to Windows Store apps, to maintain a sense of continuity between sessions even if the app is suspended and terminated.
- App data is session, local, temporary, and roaming state that is tied to the existence of an app; it is accessible only by that app.
- User data is stored in locations other than app data (such as the user's music, pictures, videos, and documents libraries, along with removable storage) and persists independent of any given app, and multiple apps might be able to open and manipulate user files.
- App data is accessed through the [Windows.Storage.ApplicationData](#) API and accommodates both structured settings containers as well as file-based data. Additional APIs like IndexedDB and HTML5 [localStorage](#) are also available.
- It is important to version app state, especially where roaming is concerned, as versioning is how the roaming service manages what app state gets roamed to which devices based on what version apps are looking for.
- The size of roaming state is limited to a quota (provided by an API), otherwise Windows will not roam the data. Services like SkyDrive can be used to roam larger files, including user data.
- The typical roaming period is 30 minutes or less. A single setting or composite named "HighPriority," so long as it's under 8K, will be roamed within a minute.
- The [StorageFolder](#) and [StorageFile](#) classes in WinRT are the core objects for working with folders and files. All programmatic access to the file system begins, in fact, with a [StorageFolder](#). Otherwise, the user can point to files and folders through the file picker API, which is really the first choice for file access.
- Blobs are useful aids in working with files, as are the WinRT APIs in the [Windows.Storage.FileIO](#) and [PathIO](#) classes. WinJS offers some simplified methods for reading and writing text files (especially in conjunction with app state), and the HTML5 [FileReader](#) is supported.
- WinRT offers encryption services through [Windows.Security.Cryptography](#), as well as a built-in compression mechanism in [Windows.Storage.Compression](#).
- To use the Settings pane, an app populates the top-level pane provided by Windows with specific commands. Those commands map to handlers that either open a hyperlink (in a browser) or display a settings flyout using the [WinJS.UI.SettingsFlyout](#) control. Those flyouts can contain any HTML desired, including [iframe](#) elements that load remote content.
- Access to user data folders, such as media libraries, documents, and removable storage, is controlled by manifest capabilities. Such capabilities need be declared only if the app needs to access the file system in some way other than using the file picker.

- The file picker is the way that users can select files from any safe location in the file system, as well as files that are provided by other apps (where those files might be remote, stored in a database, or otherwise not present as file entities on the local file system). The ability to select files directly from other apps—including files that another app might generate on demand—is one of the most convenient and powerful features of Windows Store apps.
- [StorageFolder](#) objects provide a very rich and extensive capability to query and search its contents through file queries. These queries can be simple to complex and can employ Advanced Query Syntax (AQS) search strings.

Chapter 9

Input and Sensors

Touch is clearly one of the most exciting means of interacting with a computer that has finally come of age. Sure, we've had touch-sensitive devices for many years: I remember working with a touch-enabled screen in my college days, which I have to admit is almost an embarrassingly long time ago now! In that case, the touch sensor was a series of transparent wires embedded in a plastic sheet over the screen, with an overall touch resolution of around 60 wide by 40 high...and, to really date myself, the monitor itself was only a text terminal!

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

Great touch interaction is thus now a fundamental feature of great apps, and designing for touch means in many ways thinking through UI concerns anew. In your layout, for example, it means making hit targets a size that's suitable for a variety of fingers. In your content navigation, it means utilizing direct gestures such as swipes and pinches rather than relying on only item selection and navigation controls. Similarly, designing for touch means thinking through how gestures might enrich the user experience—and also how to provide for discoverability and user feedback that has generally relied on mouse-only events like hover.

All in all, approach your design as if touch was the *only* means of interaction that your users might have. At the same time, it's very important to remember that new methods of input seldom obsolete existing ones. Sure, punch cards did eventually disappear, but the introduction of the mouse did not obsolete keyboards. The availability of speech recognition or handwriting has obsoleted neither mouse nor keyboard. I think the same is true for touch: it's really a complementary input method that has its own particular virtues but is unlikely to wholly supplant the others. As Bill Buxton of Microsoft Research has said, "Every modality, including touch, is best for something and worst for something else." I expect, in time, we'll see ourselves using keyboard, mouse, and touch together, just as we learned to integrate the mouse in what was once a keyboard-only reality.

Windows is designed to work well with all forms of input—to work great with touch, to work great with mice, to work great with keyboards, and, well, to just work great on diverse hardware! (And Windows Store certification requires this for apps as well.) For this reason, Windows provides a unified pointer-based input model wherein you can differentiate the different inputs if you really need to but can otherwise treat them equally. You can also focus more on higher-level gestures as well, which can arise from any input source, and not worry about raw pointer events at all. Indeed, the very fact that we haven't even brought this subject up until now, midway through this book, gives testimony to just how natural it is to work with all kinds of pointer input without having to think about it: the controls and

other UI elements we've been using have done all that work for us. Handling such events ourselves thus arises primarily when creating your own controls or otherwise doing direct manipulation of noncontrol objects.

The keyboard also remains an important consideration, and this means both hardware keyboards and the on-screen "soft" keyboard. The latter has gotten more attention in recent years for touch-only devices but actually has been around for some time for accessibility purposes. In Windows, too, the soft keyboard includes a handwriting recognizer—something apps just get for free. And when an app wants to work more closely with raw handwriting input—known as ink—those capabilities are present as well.

The other topic we'll cover in this chapter is sensors. It might seem an incongruous subject to place alongside input until you come to see that sensors, like touch screens themselves, *are* another form of input! Sensors tell an app what's happening to the device in its relationship to the physical world: how its positioned in space (relative to a number of reference points), how it's moving through space, how it's being held relative to its "normal" orientation, and even how much light is shining on it. Thinking of sensors in this light (pun intended), we begin to see opportunities for apps to directly integrate with the world around a device rather than requiring users to tell the app about those relationships in some more abstract way. And just to warn you, once you see just how easy it is to use the WinRT APIs for sensors, you might be shopping for a new piece of well-equipped hardware!

Touch, Mouse, and Stylus Input

Where pointer-based input is concerned—which includes touch, mouse, and pen/stylus input—the singular message from Microsoft has been and remains, "Design for touch and get mouse and stylus for free." This is very much the case, as we shall see, but we've also found that a phrase like "touch-first design" that sounds great to a consumer can be a terrifying proposition for developers! With all the attention around touch, consumer expectations are often very demanding, and meeting such expectations seems like it will take a lot of work.

Fortunately, Windows 8 provides a unified framework for handling pointer input—from all sources—such that you don't actually need to think about the differences until there's a specific reason to do so. In this way, touch-first design really *is* a design issue more than an implementation issue.

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

- First, *use templates and standard controls* and you get lots of touch support for free, along with mouse, pen, stylus, and keyboard support. If you build up your UI with standard controls, set appropriate `tabindex` attributes for keyboard users, and handle standard DOM events like `click`, you're pretty much covered. Controls like semantic zoom already handle different kinds of input (as we saw in Chapter 5, "Collections and Collection Controls"), and other CSS styles like snap points and content zooming automatically handle various interaction gestures.

- Second, when you need to handle gestures yourself, as with custom controls or other elements with which the user will interact directly, *use the gesture events* like [MSGestureTap](#) and [MSGestureHold](#) along with event sequences for inertial gestures ([MSGestureStart](#), [MSGestureChange](#), and [MSGestureEnd](#)). The benefit here is that gestures are essentially higher-order interpretations of lower-level pointer events, meaning that you don't have to do such interpretation yourself. For example, a pointer down followed by a pointer up within a certain movement threshold (to account for wiggling fingers) becomes a single tap gesture. A pointer down followed by a short drag followed by a pointer up becomes a swipe that triggers a series of events, possibly including inertial events (ones that continue to fire even after the pointer, like a touch point, is physically released). Note that if you want to capture and save pointer input directly without concern for gestures, there is also built-in support for *inking*, as we'll see later on.
- Third, if you need to handle pointer events directly, *use the unified pointer events* like [MSPointerDown](#), [MSPointerMove](#), and so forth. These are lower-level events than gestures, and they are primarily appropriate for apps that don't necessarily need gesture interpretation. For example, a drawing app simply needs to trace different pointers with on-screen feedback, where concepts like swipe and inertia aren't meaningful. Pointer events also provide more specialized device data such as pressure, rotation, and tilt, which is surfaced through the pointer events. Still, it is possible to implement gestures directly with pointer events, as a number of the built-in controls do.
- Finally, an app can *work directly with the gesture recognizer* to provide its own interpretations of pointer events into gestures.



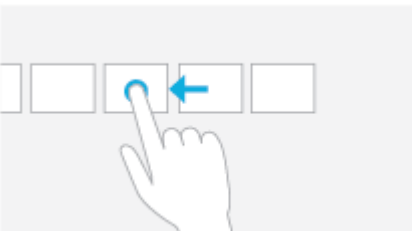
So, what about legacy DOM events that we already know and love, beyond [click](#)? Can you still work with the likes of [mousedown](#), [mouseup](#), [mouseover](#), [mousemove](#), [mouseout](#), and [mousewheel](#)? The answer is yes, because pointer events from all input sources will be automatically translated into these legacy events. This can be useful when you're porting code from a web app into a Windows Store app, for example. This translation takes a little extra processing time, however, so for new code you'll generally realize better responsiveness by using the gesture and pointer events directly. Legacy mouse events also assume a single pointer and will be generated only for the primary touch point (the one with the [isPrimary](#) property). As much as possible, use the gesture and pointer events in your code.

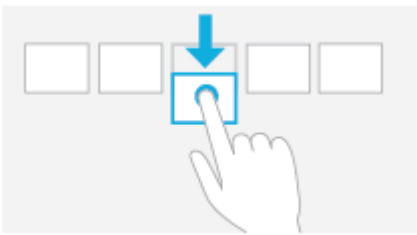
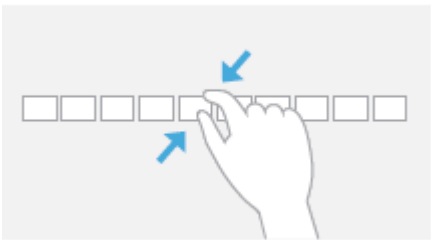



Note Visual feedback for touch input is one of the [Windows 8 app certification requirements](#) (section 3.5) and applies to everything in your app as well as any web content you might display in an [iframe](#) element. Providing feedback means executing small animations that acknowledge the touch. For this you can use the WinJS animations library or straight CSS animations and transitions, as discussed in Chapter 11, "Purposeful Animations."

The Touch Language, Its Translations, and Mouse/Keyboard Equivalents

On the Windows Developer Center, the rather extensive article on [Touch interaction design](#) is helpful for designers and developers alike. It discusses various ergonomic considerations, has some great diagrams on the sizes of human fingers, provides clear guidance on the proper size for touch targets given that human reality, and outlines key design principles such as providing direct feedback for touch interaction (animation) and having content follow your finger.

Most importantly, the design guidance also describes the Windows 8 Touch Language, which contains the eight core gestures that are baked into the system and the controls. The table below shows and describes the gestures and indicates what events appear in the app for them.

Gesture	Meaning and Gesture Events	Description
<p>One finger touches the screen and lifts up.</p> 	<p>Tap for primary action (commanding); appears as <code>click</code> and <code>MSGestureTap</code> events on the element.</p>	<p>Tapping on an element invokes its primary action, typically executing a command, checking a box, setting a rating, positioning a cursor, etc.</p>
<p>One finger touches the screen and stays in place.</p> 	<p>Press and hold to learn; appears as <code>contextmenu</code> and <code>MSGestureHold</code> events on the element.</p>	<p>This touch interaction displays detailed information or teaching visuals (for example, a tooltip or context menu) without a commitment to an action. Anything displayed this way should not prevent users from panning if they begin sliding their finger.</p>
<p>One or more fingers touch the screen and move in the same direction.</p> 	<p>Slide to pan (can be horizontal or vertical); appears as scrolling events as well as a gesture series (<code>MSGestureStart</code>, <code>MSGestureChange</code>, <code>MSGestureEnd</code>, possibly with inertial gesture events signaled by <code>MSInertiaStart</code>, plus <code>MSPointer*</code> events).</p>	<p>Slide is used primarily for panning interactions but can also be used for moving, drawing, or writing. Slide can also be used to target small, densely packed elements by scrubbing (sliding the finger over related objects such as radio buttons).</p>

<p>One or more fingers touch the screen and move a short distance in the same direction.</p> 	<p>Swipe to select, command, and move (can be horizontal or vertical)—also called <i>cross-slide</i>; appears as a gesture series (<code>MSGestureStart</code>, <code>MSGestureChange</code>, <code>MSGestureEnd</code>, as well as <code>MSPointer*</code> events). The gesture recognizer doesn't distinguish this from vertical panning, however, so an app or control needs to implement that interpretation directly (a good reason to use controls like the <code>ListView</code>!).</p>	<p>Sliding the finger a short distance, perpendicular to the panning direction, selects objects in a list or grid; also implies displaying commands in an app bar relevant to the selection.</p>
<p>Two or more fingers touch the screen and move closer together or farther apart.</p> 	<p>Pinch and stretch to zoom; appears as a gesture series (<code>MSGestureStart</code>, <code>MSGestureChange</code>, <code>MSGestureEnd</code>), but apps can use the <code>-ms-content-zooming: zoom</code> and <code>-ms-touch-action: pinch-zoom</code> CSS styles to enable touch zooming automatically.</p>	<p>Can be used for optical zoom or resizing, as well as for semantic zoom where applicable.</p>
<p>Two or more fingers touch the screen and move in a clockwise or counter-clockwise arc.</p> 	<p>Turn to rotate; appears as a gesture series (<code>MSGestureStart</code>, <code>MSGestureChange</code>, <code>MSGestureEnd</code>).</p>	<p>Rotates an object or a view.</p>
	<p>Swipe from top or bottom edge for app commands; handled automatically through the <code>AppBar</code> control, though an app can also detect these events directly through <code>Windows.UI.Input.EdgeGesture</code>.</p>	<p>The bottom app bar contains app commands for the current page context; the top app bar provides for navigation, if applicable.</p>
	<p>Swipe from edge for system commands; handled automatically by the system with the app receiving events related to the selected charm, when applicable, as well as <code>focus</code> and <code>blur</code> events if the foreground app is changed when swiping from the left edge.</p>	<p>Swiping from the right displays the Charms bar; swiping from the left cycles through currently running apps; swiping from the top edge to the bottom closes the current app; swiping from the top edge to the left or right snaps the current app to one side of the screen.</p>

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

You might notice in the table above that many of the gestures in the touch language don't actually have a single event associated with them (like pinch or rotate) but are instead represented by a series of gesture or pointer events. The reason for this is that these gestures, when used with touch, typically involve animation of the affected content while the gesture is happening. Swipes, for example, show linear movement of the object being panned or selected. A pinch or stretch movement will often be actively zooming the content. (Semantic Zoom is an exception, but then you just let the control handle the details.) And a rotate gesture should definitely give visual feedback. In short, handling these gestures with touch, in particular, means dealing with a series of events rather than just a single one.

This is one reason that it's so helpful (and time-saving!) to use the built-in controls as much as possible, because they already handle all the gesture details for you. The ListView control, for example, contains all the pointer/gesture logic to handling pans and swipes, along with taps. The Semantic Zoom control, like I said, implements pinch and stretch by watching `MSPointer*` events. If you look at the source code for these controls within WinJS, you'll start to appreciate just how much they do for you (and what it will look like to implement a rich custom control of your own, using the gesture recognizer!).

You can also save yourself a lot of trouble with the `-ms-touch-action` CSS properties described under "CSS Styles That Affect Input." Using this has the added benefit of processing the touch input on a non-UI thread, thereby providing much smoother manipulation than could be achieved by handling pointer or gesture events.

On the theme of "write for touch and get other input for free," all of these gestures also have mouse and keyboard equivalents, which the built-in controls also implement for you. It's also helpful to know what those equivalents are, as shown in the table below. The "Standard Keystrokes" section later in this chapter also lists many other command-related keystrokes.

Touch	Keyboard	Mouse	Pen/Stylus
Press and hold (or tap on text selection)	Right-click button	Right button click	Press and hold
Tap	Enter	Left button click	Tap
Slide (short distance)	Arrow keys	Left button click and drag, click on scrollbar arrows, drag the scrollbar thumb, use the mouse wheel	Tap on scrollbar arrows, drag scrollbar thumb, tap and drag
Slide + inertia (long distance)	Page Up/Page Down	Left button click and drag, click on scrollbar track, drag the scrollbar thumb, use the mouse wheel	Tap on scrollbar track, drag scrollbar thumb, tap and drag
Swipe to select	Right-click button or spacebar	Right button click	Tap and drag
Pinch/Stretch	Ctrl+ and Ctrl-	Ctrl+mouse wheel or UI command	UI command or other hardware feature
Swipe from edge	Win+Z, Win+Tab, Win+C or Win+Shift+C	Clicking on corners of the screen; right-click shows app bar	Drag in from edge
Rotate	Ctrl+, and Ctrl+.	Ctrl+Shift+mouse wheel	UI command or other hardware feature

You might notice a conspicuous absence of double-click and/or double-tap gestures in this list. Does that surprise you? In early builds of Windows 8 we actually did have a double-tap gesture, but it turned out to not be all that useful, conflicted with the zoom gesture, and sometimes very difficult for users to perform. I can say from watching friends over the years that double-clicking with the mouse isn't even all it's cracked up to be. People with not-entirely-stable hands will often move the mouse quite a ways between clicks, just as they might move their finger between taps. As a result, the reliability of a double-tap ends up being pretty low, and since it wasn't really needed in the touch language, it was simply dropped altogether.

Sidebar: Creating Completely New Gestures?

While the Windows 8 touch language provides a simple yet fairly comprehensive set of gestures, it's not too hard to imagine other possibilities. The question is, when is it appropriate to introduce a new kind of gesture or manipulation?

First, it makes sense that apps don't generally introduce new ways to do the same things, such as additional gestures that just swipe, zoom, etc. It's better to simply get more creative in how the app interprets an existing gesture. For example, a swipe gesture might pan a scrollable region but can also just move an object on the screen—no need to invent a new gesture.

Second, if you have controls placed on the screen where you want the user to give input, there's no need to think in terms of gestures at all: just apply the input from those controls appropriately.

Third, even when you do think a custom gesture is needed, the bottom-line recommendation is to make those interactions feel natural, rather than something you just invent for the sake of invention. We also recommend that gestures behave consistently with the number of pointers, velocity/time, and so on. For example, separating an element into three pieces with a three-finger stretch and into two pieces with a two-finger stretch is fine; having a three-finger stretch enlarge an element while a two-finger stretch zooms the canvas is a bad idea, because it's not very discoverable. Similarly, the speed of a horizontal or vertical flick can affect the velocity of an element's movement, but having a fast flick switch to another page while a slow flick highlights text is a bad idea. In this case, having different functions based on speed creates a difficult UI for your customers because they'll all have different ideas about what "fast" and "slow" mean and might be limited by their physical abilities.

Finally, with any custom gesture, recognize that you are potentially introducing an inconsistency between apps. When a user starts interacting with a certain kind of app in a new way, he or she might start to expect that of other apps and might become confused (or upset) when those apps don't behave in the same way, especially if those apps use a similar gesture for a completely different purpose! Complex gestures, too, might be difficult for some, if not many, people to perform; might be limited by the kind of hardware in the device (number of touch points, responsiveness, etc.); and are generally not very discoverable. In most cases it's probably simpler to add an appbar command on a button on your app canvas to achieve the same goal.

Edge Gestures

As we saw in Chapter 7, “Commanding UI,” you don’t need to do anything special for commands on the app bar or navigation bar to appear: Windows automatically handles the edge swipe from the top and bottom of your app, along with right-click, Win+Z, and the context menu key on the keyboard. That said, you can detect when these events happen directly by listening for the **starting**, **completed**, and **canceled** events on the [Windows.UI.Input.EdgeGesture](#) object:⁴⁸

```
var edgeGesture = Windows.UI.Input.EdgeGesture.getForCurrentView();
edgeGesture.addEventListener("starting", onStarting);
edgeGesture.addEventListener("completed", onCompleted);
edgeGesture.addEventListener("canceled", onCancel);
```

With these, **completed** fires for all input types; the **starting** and **canceled** events occur only for touch. Within these events, the `eventArgs.kind` property contains a value from the `EdgeGestureKind` enumeration that indicates the kind of input that invoked the event. The **starting** and **canceled** events will always have the kind of **touch**, obviously, whereas **completed** can be any **touch**, **keyboard**, or **mouse**:

```
function onCompleted(e) {
    // Determine whether it was touch or keyboard invocation
    if (e.kind === Windows.UI.Input.EdgeGestureKind.touch) {
        id("ScenarioOutput").innerText = "Invoked with touch.";
    }
    else if (e.kind === Windows.UI.Input.EdgeGestureKind.mouse) {
        id("ScenarioOutput").innerText = "Invoked with right-click.";
    }
    else if (e.kind === Windows.UI.Input.EdgeGestureKind.keyboard) {
        id("ScenarioOutput").innerText = "Invoked with keyboard.";
    }
}
```

The code above is taken from Scenario 1 of the [Edge gesture invocation sample](#). In Scenario 2, the sample also shows that you can prevent the edge gesture event from occurring for a particular element if you handle the `contextmenu` event for that element and call `eventArgs.preventDefault` in your handler. It does this for one element on the screen, such that right-clicking that element with the mouse or pressing the context menu key when that element has the focus will prevent the edge gesture events:

```
document.getElementById("handleContextMenuDiv").addEventListener("contextmenu", onContextMenu);

function onContextMenu(e) {
    e.preventDefault();
    id("ScenarioOutput").innerText =
        "The ContextMenu event was handled. The EdgeGesture event will not fire.";
}
```

⁴⁸ As WinRT object events, these are subject to the considerations in “WinRT Events and removeEventListener” in Chapter 3.

Note that this method has no effect on edge gestures via touch and does not affect the Win+Z key combination that normally invokes the app bar. It's primarily to show that if you need to handle the `contextmenu` event specifically, you usually want to prevent the edge gesture.

CSS Styles That Affect Input

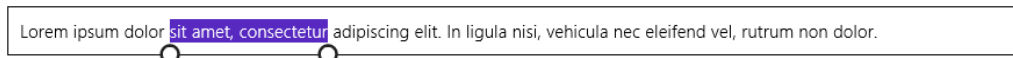
While we're on the subject of input, it's a good time to mention a number of CSS styles that affect the input an app might receive.

One style is `-ms-user-select`, which we've encountered a few times already in Chapter 3, "App Anatomy and Page Navigation," and Chapter 4, "Controls, Control Styling, and Data Binding." This style can be set to one of the following:

- `none` disables direct selection, though the element as a whole can be selected if its parent is selectable.
- `inherit` sets the selection behavior of an element to match its parent.
- `text` will enable selection for text even if the parent is set to `none`.
- `element` enables selection for an arbitrary element.
- `auto` (the default) may or may not enable selection depending on the control type and the styling of the parent. For an element that is not a text control and does not have `contenteditable="true"`, it won't be selectable unless it's contained within a selectable parent.

If you want to play around with the variations, refer to the [Unselectable content areas with -ms-user-select CSS attribute sample](#), which wins the prize for the longest JavaScript sample name in the entire Windows SDK!

A related style, but one not shown in the sample, is `-ms-touch-select`, which can be either `none` or `grippers`, the latter being the style that enables the selection control circles for touch:



Selectable text elements automatically get this style, as do other textual elements with `contenteditable = "true"`—you can thus use `-ms-touch-select` to turn them off. To see the effect, try this with some of the elements in Scenario 1 of the aforementioned sample with the really long name!

In Chapter 6, "Layout," we introduced the idea of snap points for panning, with the `-ms-scroll-snap*` styles. Along these same lines, listed on the [Touch: Zooming and Panning](#) styles reference, are others for content zooming, such as `-ms-content-zooming` and the `-ms-content-zoom*` styles that provide snap points for zoom operations as well. The important thing is that `-ms-content-zooming: zoom` (as opposed to the default, `none`) enables automatic zooming with touch and the mouse wheel, provided that the element in question allows for overflow in both x and y dimensions. There are quite a number of variations here for panning and zooming, and how those

gestures interact with WinJS controls. I'll leave it to the [HTML scrolling, panning, and zooming sample](#) to explain the details.

Finally, the `-ms-touch-action` style provides for a number of options on an element:⁴⁹

- `none` Disables touch on the element.
- `auto` Enables usual touch behaviors.
- `pan-x/pan-y` The element permits horizontal/vertical touch panning, which is performed on the nearest ancestor that is horizontally/vertically scrollable, such as a parent `div`.
- `pinch-zoom` Enables pinch-zoom on the element, performed on the nearest ancestor that has `-ms-content-zooming: zoom` and overflow capability. For example, an `img` element by itself won't respond to the gesture with this style, but if you place it in a parent `div` with `overflow` set, it will.
- `manipulation` Shorthand equivalent of `pan-x pan-y pinch-zoom`.

For an example of panning and zooming, try creating a simple app with markup like this (use whatever image you'd like):

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

and style the container as follows:

```
#imageContainer {
  overflow: auto;
  -ms-content-zooming: zoom;
  -ms-touch-action: manipulation;
}
```

What Input Capabilities Are Present?

The WinRT API in the [Windows.Devices.Input](#) namespace provides all the information you need about the input capabilities that are available on the current device, specifically through these three objects:

- [MouseCapabilities](#) Properties are `mousePresent` (0 or 1), `horizontalWheelPresent` (0 or 1), `verticalWheelPresent` (0 or 1), `numberOfButtons` (a number), and `swapButtons` (0 or 1).
- [KeyboardCapabilities](#) Contains only a single property: `keyboardPresent` (0 or 1). Note that this does not indicate the presence of the on-screen keyboard, which is always available; `keyboardPresent` specifically indicates a physical keyboard device.

⁴⁹ `double-tap-zoom` is not supported for Windows Store apps.

- [TouchCapabilities](#) Properties are `touchPresent` (0 or 1) and `contacts` (a number). Note that where touch is concerned, you might also be interested in the [Windows.UI.-ViewManagement.-UISettings.handPreference](#) property, which indicates the user's right- or left-handedness.

To check whether touch is available, then, you can use a bit of code like this:

```
var tc = new Windows.Devices.Input.TouchCapabilities();
var touchPoints = 0;

if (tc.touchPresent) {
    touchPoints = tc.contacts;
}
```

Note In the web context where WinRT is not available, some information about capabilities can be obtained through the [msPointerEnabled](#), [msManipulationViewsEnabled](#), and [msMax-TouchPoints](#) properties that are hanging off DOM elements. These also work in the local context.

You'll notice that the capabilities above don't say anything about a stylus or pen. For these and for more extensive information about all pointer devices, including touch and mouse, we have the [Windows.Devices.Input.PointerDevice.getPointerDevices](#) method. This returns an array of [PointerDevice](#) objects, each of which has these properties:

- `pointerDeviceType` A value from [Windows.Devices.Input.PointerDeviceType](#) that can be `touch`, `pen`, or `mouse`.
- `maxContacts` The maximum number of contact points that the device can support—typically 1 for mouse and stylus and any other number for touch.
- `isIntegrated` `true` indicates that the device is built into the machine so that its presence can be depended upon; `false` indicates a peripheral that the user could disconnect.
- `physicalDeviceRect` This [Windows.Foundation.Rect](#) object provides the bounding rectangle as the device sees itself. Oftentimes, a touch screen's input resolution won't actually match the screen pixels, meaning that the input device isn't capable of hitting exactly one pixel. On one of my touch-capable laptops, for example, this resolution is reported as 968x548 for a 1366x768 pixel screen (as reported in [screenRect](#) below). A mouse, on the other hand, typically does match screen pixels one-for-one. This could be important for a drawing app that works with a stylus, where an input resolution smaller than the screen would mean there will be some inaccuracy when translating input coordinates to screen pixels.
- `screenRect` This [Windows.Foundation.Rect](#) object provides the bounding rectangle for the device on the screen, which is to say, the minimum and maximum coordinates that you should encounter with events from the device. This rectangle will take multimonitor systems into account, and it's adjusted for resolution scaling.

- **supportedUsages** An array of [Windows.Devices.Input.PointerDeviceUsage](#) structures that supply what's called HID (human interface device) usage information. This subject is beyond the scope of this book, so I'll refer you to the [HID Usages](#) page on MSDN for starters.

The [Input Device capabilities sample](#) in the Windows SDK retrieves this information and displays it to the screen through the code in `js/pointer.js`. I won't show that code here because it's just a matter of iterating through the array and building a big HTML string to dump into the DOM. In the simulator, the output appears as follows—notice that the simulator reports the presence of touch and mouse both in this case.

```
Output
(1) Pointer Device Type Touch
(1) Is Integrated true
(1) Max Contacts 5
(1) Physical Device Rect 0,0,491.3385925292969,907.0866088867187
(1) Screen Rect 0,0,1366,768
(2) Pointer Device Type Mouse
(2) Is Integrated false
(2) Max Contacts 1
(2) Physical Device Rect 0,0,1366,768
(2) Screen Rect 0,0,1366,768
```

Curious Forge? Interestingly, I ran this same sample in Visual Studio's Local Machine debugger on a laptop that is definitely not touch-enabled, and yet a touch device was still reported as in the image above! Why was that? It's because I still had the Visual Studio simulator running, which adds a virtual touch device to the hardware profile. After closing the simulator completely (not just minimizing it), I got an accurate report for my laptop's capabilities. So be mindful of this if you're writing code to test for specific capabilities.

Tried remote debugging yet? Speaking of debugging, as mentioned in a sidebar in Chapter 6, "Layout," testing an app against different device capabilities is a great opportunity to use remote debugging in Visual Studio. If you haven't done so already, it takes only a few minutes to set up and makes it far easier to test apps on multiple machines. For details, see [Running Windows 8 apps on a remote machine](#).

Unified Pointer Events

For any situation where you want to directly work with touch, mouse, and stylus input, perhaps to implement parts of the touch language in this way, use the `MSPointer*` events. Most art/drawing apps, for example, will use these events to track and respond to screen interaction. Remember again that pointers are a lower-level way of looking at input than gestures, which we'll see in the next section. Which input model you use depends on the kind of events you're really looking to work with.

Tip Pointer events won't fire if the system is trying to do a manipulation like panning or zooming. To disable manipulations on an element, set the `-ms-content-zooming: none` or `-ms-touch-action: none`, and avoid using `-ms-touch-action` styles of `pan-x`, `pan-y`, `pinch-zoom`, and `manipulation`.

As with other events, you can listen to `MSPointer*` events on whatever elements are relevant to you, remembering again that these are translated into legacy mouse events, so you should not listen to both. The specific events are described as follows, given in the order of their typical sequencing:

- [`MSPointerOver`](#) Pointer moved into the bounds of the element from outside.
- [`MSPointerHover`](#) A pointer is hovering over the element (generally only for pen or mouse).
- [`MSPointerDown`](#) Pointer down occurred on the element.
- [`MSPointerMove`](#) Pointer moved across the element.
- [`MSPointerUp`](#) Pointer was released over the element. (If an element previously captured the touch, `msReleasePointerCapture` is called automatically.) Note that if a pointer is moved outside of an element and released, it will receive `MSPointerOut` but not `MSPointerUp`.
- [`MSPointerCancel`](#) The system canceled a pointer event.
- [`MSPointerOut`](#) Pointer moved out of the bounds of the element, which also occurs with an up event. This is the last pointer event an element will receive.
- [`MSGotPointerCapture`](#) The pointer is captured by the element.
- [`MSLostPointerCapture`](#) The pointer capture has been lost for the element.

These are the names you use with `addEventListener`; the equivalent property names are of the form `onmspointerdown`, as usual. It should be obvious that some of these events might not occur with all pointer types—touch screens, for instance, generally don't provide hover events, though some that can detect the proximity of a finger are so capable.

Tip If for some reason you want to prevent the translation of an `MSPointer*` event into a legacy mouse event, call the `eventArgs.preventDefault` method within the appropriate event handler.

The PointerEvents example provided with this chapter's companion content and shown in Figure 9-1 lets you see what's going on with all the mouse, pointer, and gesture events, selectively showing groups of events in the display.

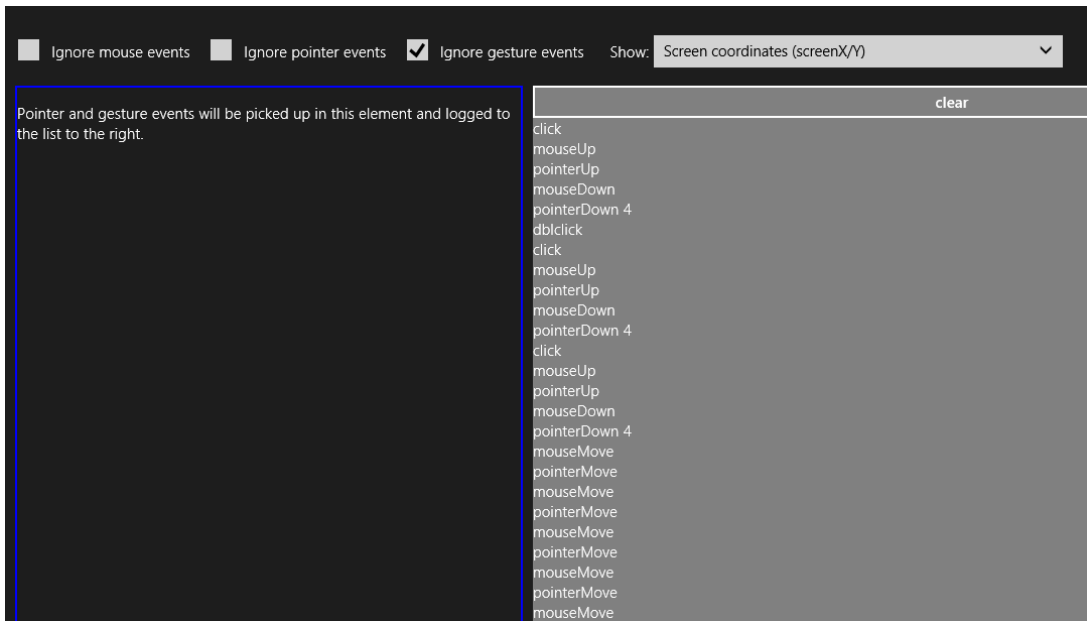


FIGURE 9-1 The PointerEvents example display (screen shot cropped a bit to show detail).

Within the handlers for all of the [MSPointer*](#) events, the [eventArgs](#) object contains a whole roster of properties. One of them, [pointerType](#), identifies the type of input: touch (2), pen (3), and mouse (4). This property lets you implement different behaviors for different input methods, if desired. Each event object also contains a unique [pointerId](#) value that identifies a stroke or a path for a specific contact point, allowing you to correlate an initial [MSPointerDown](#) event with subsequent events. When we look at gestures in the next section, we'll also see how we use the [pointerId](#) of [MSPointerDown](#) to associate a gesture with a pointer.

The complete roster of properties that come with the event is actually far too much to show here, as it contains many of the usual [DOM properties](#) along with many pointer-related ones from an object type called [MSPointerEvent](#). The best way to see what shows up is to run some code like the [Input DOM pointer event handling sample](#) (a [canvas](#) drawing app), set a breakpoint within a handler for one of the events, and examine the event object. The table on the following page describes some of the properties (and a few methods) relevant to our discussion here.

Properties	Description
<code>currentPoint</code>	A Windows.UI.Input.PointerPoint object. This contains many other properties such as <code>pointerDevice</code> (a Windows.Input.Device.PointerDevice object, as described in “What Input Capabilities Are Present” earlier in this chapter) and one just called <code>properties</code> , which is a Windows.UI.Input.PointerPointProperties .
<code>pointerType</code>	The source of the event could be touch or pen or mouse: <code>MSPOINTER_TYPE_TOUCH</code> (2), <code>MSPOINTER_TYPE_PEN</code> (3), and <code>MSPOINTER_TYPE_MOUSE</code> (4). You can use this to make adjustments according to input type, if necessary.
<code>pointerId</code>	The unique identifier of the contact. This remains the same throughout the lifetime of the pointer. If desired, you can call Windows.Devices.Input.getPointerDevice with this id to obtain a PointerDevice that describes the input device's capabilities, as described earlier in “What Input Capabilities are Present?”
<code>type</code>	The name of the event, as in “ <code>MSPointerDown</code> ”.
<code>x, screenX, y, screenY</code>	The x- and y-coordinates of the pointer's center point position relative to the screen.
<code>clientX, clientY</code>	The x- and y-coordinates of the pointer's center point position relative to the client area of the app.
<code>offsetX, offsetY</code>	The x- and y-coordinates of the pointer's center point position relative to the element.
<code>button</code>	Determines the button pressed by the user (on mice and other input devices with buttons). The left is 0, middle is 1, and right is 2; these values can be combined with bitwise the OR operator for <i>chord</i> presses (multiple buttons).
<code>ctrlKey, altKey, shiftKey</code>	Indicates whether certain keys were depressed when the pointer event occurred.
<code>hwTimestamp</code>	The timestamp (in microseconds) at which the event was received from the hardware.
<code>relatedTarget</code>	Provides the element related to the current event, e.g., the <code>MSPointerOut</code> event will provide the element to which the touch is moving. This can be <code>null</code> .
<code>isPrimary</code>	Indicates if this pointer is the primary one in a multitouch scenario (such as the pointer that the mouse would control).
<i>Properties surfaced depending on hardware support (if not supported, these values will be 0)</i>	
<code>width, height</code>	The contact width and height of the touch point specified by <code>pointerId</code> .
<code>pressure</code>	Pen pressure normalized in a range of 0 to 255.
<code>rotation</code>	Clockwise rotation of the cursor around its own major axis in a range of 0 to 359.
<code>tiltX</code>	The left-right tilt away from the normal of a transducer (typically perpendicular to the surface) in a range of -90 (left) to 90 (right).
<code>tiltY</code>	The forward-back tilt away from the normal of a transducer (typically perpendicular to the surface) in a range of -90 (forward/away from user) to 90 (back/toward user).
Methods	
<code>currentPoint</code> <code>getCurrentPoint</code>	Provides the Windows.UI.Input.PointerPoint object for the current pointer relevant to the target element (<code>currentPoint</code>) or a given element (<code>getCurrentPoint</code>)
<code>intermediatePoints</code> <code>getIntermediatePoints</code>	Provides the PointerPoint history for the current pointer relative to the target element (<code>intermediatePoint</code>) or a given element (<code>getIntermediatePoints</code>)

It's very instructive to run the Input DOM pointer event handling sample on a multitouch device, because it tracks each `pointerId` separately allowing you to draw with multiple fingers simultaneously.

Pointer Capture

It's common with down and up events for an element to set and release a capture on the pointer. To support these operations, the following methods are available on each element in the DOM and apply to each `pointerId` separately:

Method	Description
<code>msSetPointerCapture</code>	Captures the <code>pointerId</code> for the element so that pointer events come to it and are not raised for other elements (even if you move outside the first element and into another). <code>MSGotPointerCapture</code> will be fired on the element as well.
<code>msReleasePointerCapture</code>	Ends capture, triggering an <code>MSLostPointerCapture</code> event.
<code>msGetPointerCapture</code>	Returns the element with the capture, if any (otherwise <code>null</code>).

We see this in the Input DOM pointer event handling sample, where it sets capture within its `MSPointerDown` handler and releases it in `MSPointerUp`:

```
this.MSPointerDown = function (evt) {
    canvas.msSetPointerCapture(evt.pointerId);
    // ...
};

this.MSPointerUp = function (evt) {
    canvas.msReleasePointerCapture(evt.pointerId);
    // ...
};
```

Gesture Events

The first thing to know about all `MSGesture*` events is that they don't just fire automatically like `click` and `MSPointer*` events, and you don't just add a listener and be done with it (that's what `click` is for!). Instead, you need to do a little bit of configuration first to tell the system how exactly you want gestures to occur, and you need to use `MSPointerDown` to associate the gesture configurations with a particular `pointerId`. This small added bit of complexity makes it possible for apps to work with multiple concurrent gestures and keep them all independent just as you can do with pointer events. Imagine, for example, a jigsaw puzzle app (as presented in a small way in one of the samples in "The Gesture Samples" below) that allows multiple people sitting around a table-size touch screen to work with individual pieces as they will. Using gestures, each person can be manipulating an individual piece (or two!), moving it around, rotating it, perhaps zooming in to see a larger view, and, of course, testing out placement. For Windows Store apps written in JavaScript, it's also helpful that manipulation deltas for configured elements—which include translation, rotation, and scaling—are given in the coordinate space of the parent element, meaning that it's fairly straightforward to translate the manipulation into CSS transforms and such to make the manipulation visible. In short, there is a great deal of flexibility here when you need it; if you don't, you can use gestures in a simple manner as well. Let's see how it all works.

Tip If you're observant, you'll notice that everything in this section has no dependency on WinRT APIs. As a result, the gesture events described here work in both the local and web contexts.

The first step to receiving gesture events is to create an `MSGesture` object and associate it with the element for which you're interested in receiving events. In the `PointerEvents` example, that element is named `divElement`; you need to store that element in the gesture's `target` property and store the gesture object in the element's `gestureObject` property for use by `MSPointerDown`:

```
var gestureObject = new MSGesture();
gestureObject.target = divElement;
divElement.gestureObject = gestureObject;
```

With this association, you can then just add event listeners as usual. The example shows the full roster of the six gesture events:

```
divElement.addEventListener("MSGestureTap", gestureTap);
divElement.addEventListener("MSGestureHold", gestureHold);

divElement.addEventListener("MSGestureStart", gestureStart);
divElement.addEventListener("MSGestureChange", gestureChange);
divElement.addEventListener("MSGestureEnd", gestureEnd);
divElement.addEventListener("MSInertiaStart", inertiaStart);
```

We're not quite done yet, however. If this is all you do in your code, you still won't receive any of the events because each gesture has to be associated with a pointer. You do this within the `MSPointerDown` event handler:

```
function pointerDown(e) {
    //Associate this pointer with the target's gesture
    e.target.gestureObject.addPointer(e.pointerId);
}
```

To enable rotation and pinch-stretch gestures with the mouse wheel (which you should do), add an event handler for the `wheel` event, set the `pointerId` for that event to 1 (a fixed value for the mouse wheel), and send it on to your `MSPointerDown` handler:

```
divElement.addEventListener("wheel", function (e) {
    e.pointerId = 1; // Fixed pointerId for MouseWheel
    pointerDown(e);
});
```

Now gesture events will start to come in *for that element*. (Remember that mouse wheel by itself is translate, Ctrl+wheel is zoom, and Shift+Ctrl+wheel is rotate.) What's more, if additional `MSPointerDown` events occur for the same element with different `pointerId` values, the `addPointer` method will include that new pointer in the gesture. This automatically enables pinch-stretch and rotation gestures that rely on multiple points.

If you run the `PointerEvents` example (checking Ignore Mouse Events and Ignore Pointer Events) and start doing taps, tap-holds, and short drags (with touch or mouse), you'll see output like that shown in Figure 9-2.

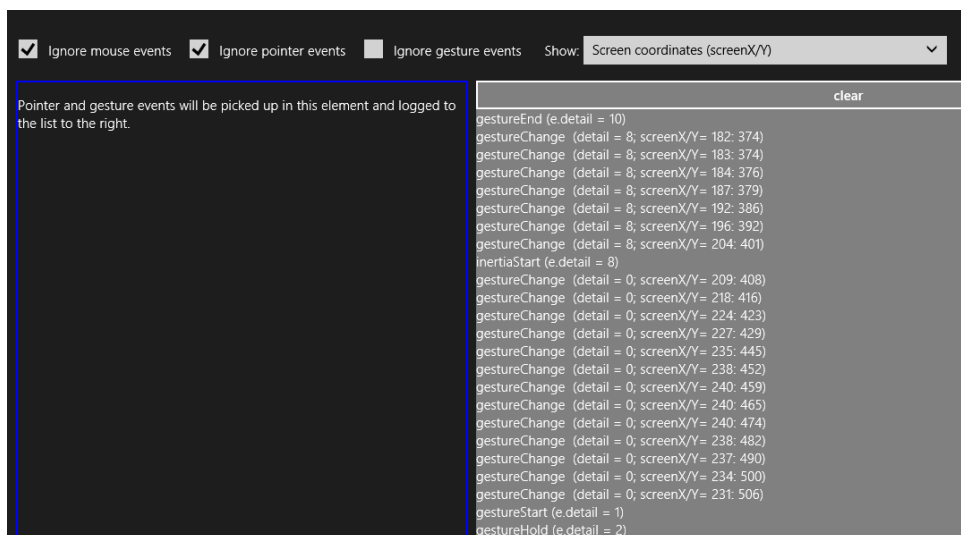
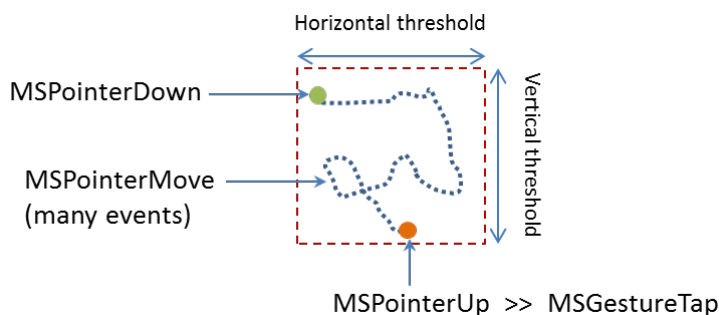


FIGURE 9-2 The PointerEvents example output for gesture events (screen shot cropped a bit to emphasize detail).

Again, gesture events are fired in response to a series of pointer events, offering higher-level interpretations of the lower-level pointer events. It's the process of interpretation that differentiates the tap/hold events from the start/change/end events, how and when the `MSInertiaStart` event kicks off, and what the gesture recognizer does when the `MSGesture` object is given multiple points.

Starting with a single pointer gesture, the first aspect of differentiation is a *pointer movement threshold*. When the gesture recognizer sees an `MSPointerDown` event, it starts to watch the `MSPointerMove` events to see whether they stay inside that threshold, which is the effective boundary for tap and hold events. This accounts for and effectively ignores small amounts of jiggle in a mouse or a touch point as illustrated (or shall I say, exaggerated!) below, where a pointer down, a little movement, and a pointer up generates an `MSGestureTap`:



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

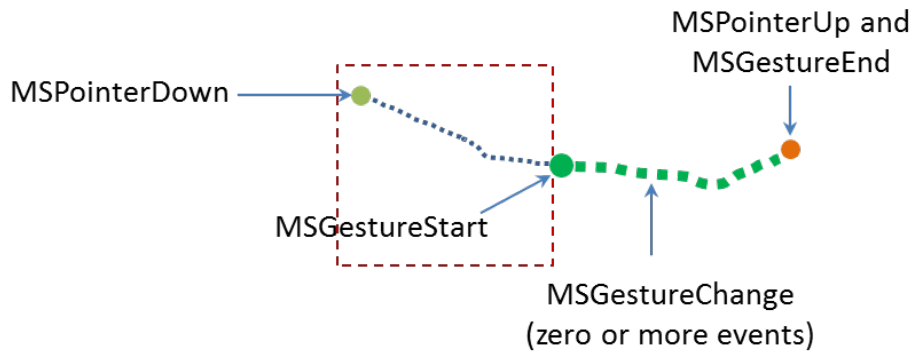
- `MSGestureTap` occurs when `MSPointerDown` is followed by `MSPointerUp` within the time threshold.

- `MSGestureHold` occurs when `MSPointerDown` is followed by `MSPointerUp` outside the time threshold. `MSGestureHold` then fires once when the time threshold is passed with `eventArgs.detail` set to 1 (`MSGESTURE_FLAG_BEGIN`). Provided that the pointer is still within the movement threshold, `MSGestureHold` fires then again when `MSPointerUp` occurs, with `eventArgs.detail` set to 2 (`MSGESTURE_FLAG_END`). You can see this detail included in the first two events of Figure 9-2 above.

The gesture flags in `eventArgs.detail` value is accompanied by many other positional and movement properties in the `eventArgs` object as shown in the following table:

Properties	Description
<code>screenX, screenY</code>	The x- and y-coordinates of the gesture center point relative to the screen.
<code>clientX, clientY</code>	The x- and y-coordinates of the gesture center point relative to the client area of the app.
<code>offsetX, offsetY</code>	The x- and y-coordinates of the gesture center point relative to the element.
<code>translationX, translationY</code>	Translation along the x- and y-axes.
<code>velocityX, velocityY</code>	Velocity of movement along x- and y-axes.
<code>scale</code>	Scale factor for zoom (percentage change in the scale).
<code>expansion</code>	Diameter of the manipulation area (absolute change in size, in pixels).
<code>velocityExpansion</code>	Velocity of expanding manipulation area.
<code>rotation</code>	Rotation angle in radians.
<code>velocityAngular</code>	Angular velocity in radians.
<code>detail</code>	<p>Contains the gesture flags that describe the gesture state of the event; these flags are defined as values in <code>eventArgs</code> itself:</p> <p><code>eventArgs.MSGESTURE_FLAG_NONE</code> (0): Indicates ongoing gesture such as <code>MSGestureChange</code> where there is change in the coordinates.</p> <p><code>eventArgs.MSGESTURE_FLAG_BEGIN</code> (1): The beginning of the gesture sequence. If the interaction contains single event such as <code>MSGestureTap</code>, both <code>MSGESTURE_FLAG_BEGIN</code> and <code>MSGESTURE_FLAG_END</code> flags will be set (detail will be 3).</p> <p><code>eventArgs.MSGESTURE_FLAG_END</code> (2): The end of the gesture sequence. Again, if the interaction contains single event such as <code>MSGestureTap</code>, both <code>MSGESTURE_FLAG_BEGIN</code> and <code>MSGESTURE_FLAG_END</code> flags will be set (detail will be 3).</p> <p><code>eventArgs.MSGESTURE_FLAG_CANCEL</code> (4): The gesture was cancelled. Always comes paired with <code>MSGESTURE_FLAG_END</code>, (detail will be 6).</p> <p><code>eventArgs.MSGESTURE_FLAG_INERTIA</code> (8): The gesture is in an inertia state. The <code>MSGestureChange</code> event can be distinguished from direct interaction and timer driven inertia through this flag.</p>
<code>hwTimestamp</code>	The timestamp of the pointer assigned by the system when the input was received from the hardware.

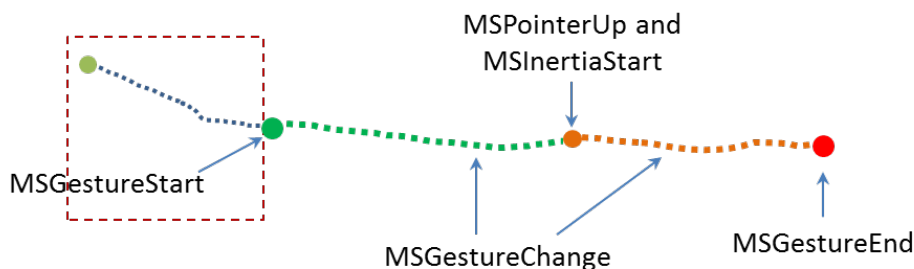
Many of these properties become much more interesting when a pointer moves *outside* the movement threshold, after which time you'll no longer see the tap or hold events. Instead, as soon as the pointer leaves the threshold area, `MSGestureStart` is fired, followed by zero or more `MSGestureChange` events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



Note that if a pointer has been held within the movement threshold long enough for the first `MSGestureHold` to fire with `MSGESTURE_FLAG_BEGIN`, but then the pointer is moved out of the threshold area, `MSGestureHold` will be fired a second time with `MSGESTURE_FLAG_CANCEL | MSGESTURE_FLAG_END` in `eventArgs.detail` (a value of 6), followed by `MSGestureStart` with `MSGESTURE_FLAG_BEGIN`. This series is how you differentiate a hold from a slide or drag gesture even if the user holds the item in place for a while.

Together, the `MSGestureStart`, `MSGestureChange`, and `MSGestureEnd` events define a *manipulation* of the element to which the gesture is attached, where the pointer remains in contact with the element throughout the manipulation. Technically this means that the pointer was no longer moving when it was released.

If the pointer *was* moving when released, then we switch from a manipulation to an *inertial* motion. In this case, an `MSInertiaStart` event gets fired in to indicate that the pointer effectively continues to move even though contact was released or lifted. That is, you'll continue to receive `MSGestureChange` events until the movement is complete:



Conceptually, you can see the difference between a manipulation and an inertial motion as illustrated in Figure 9-3; the curves shown here are not necessarily representative of actual changes between messages. If the pointer is moved along the green line such that it's no longer moving when released, we see the series of gesture that define a manipulation. If the pointer is released while moving, we see `MSInertiaStart` in the midst of `MSGestureChange` events and the event sequence follows the orange line.

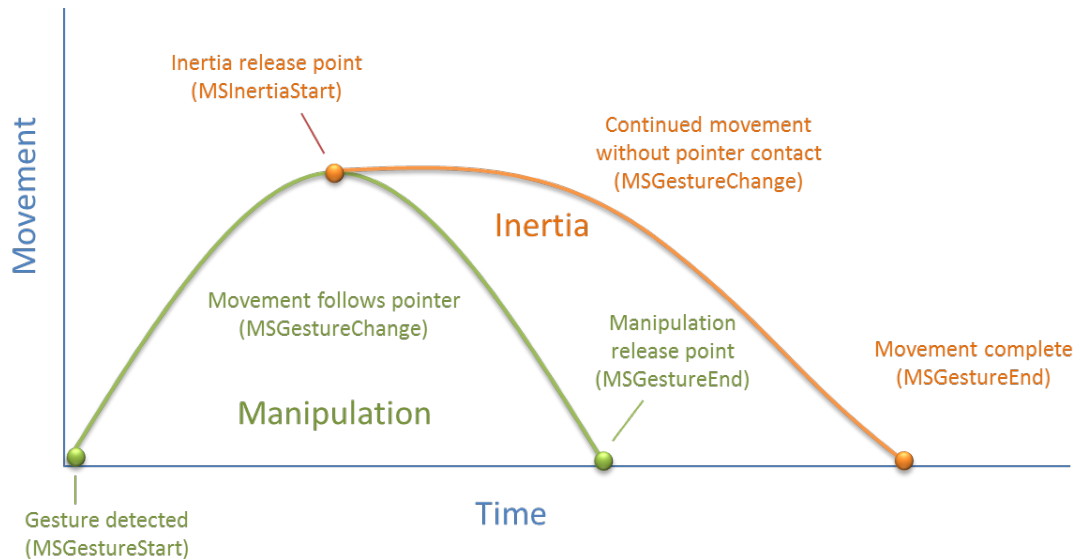


FIGURE 9-3 A conceptual representation of manipulation (green) and inertial (orange) motions.

Referring back to Figure 9-2, when the Show drop-down list (as shown!) is set to Velocity, the output for `MSGestureChange` events includes the `eventArgs.velocity*` values. During a manipulation, the velocity can change at any rate depending on how the pointer is moving. Once an inertial motion begins, however, the velocity will gradually diminish down to zero at which point `MSGestureEnd` occurs. The number of change events depends on how long it takes for the movement to slow down and come to a stop, of course, but if you're just moving an element on the display with these change events, the user will see a nice fluid animation. You can play with this in the PointerEvents example, using the Show drop-down list to also look at how the other positional properties are affected by different manipulations and inertial gestures.

Multipoint Gestures

What we've discussed so far has focused on a single point gesture, but the same is also true for multi-point gestures. When an `MSGesture` object is given multiple pointers through its `addPointer` event, it will also fire `MSGestureStart`, `MSGestureChange`, `MSGestureEnd` for rotations and pinch-stretch gestures, along with `MSInertiaStart`. In these cases, the `scale`, `rotation`, `velocityAngular`, `expansion`, and `velocity-Expansion` properties in the `eventArgs` object become meaningful.

You can selectively view these properties for `MSGestureChange` events through the upper-right drop-down list in the PointerEvents example. One thing you might notice is that if you do multipoint gestures in the Visual Studio simulator, you'll never see `MSGestureTap` events for the individual points. This is because the gesture recognizer can see that multiple `MSPointerDown` events are happening almost simultaneously (which is where the `hwTimestamp` property comes into play) and combines them into an `MSGestureStart` right away (for example, starting a pinch-stretch or rotation gesture).

Now I'm sure you're asking some important questions. While I've been speaking of pinch-stretch, rotation, and translation gestures as different things, how does one, in fact, differentiate these gestures when they're all coming into the app through the same `MSGestureChange` event? Doesn't that just make everything confusing? What's the strategy for translation, rotation, and scaling gestures?

Well, the answer is—you don't have to separate them! If you think about it for a moment, how you handle `MSGestureChange` events and the data each one contains depends on the kinds of manipulations you actually support in your UI:

- If you're supporting only translation of an element, you'll simply never pay any attention to properties like `scale` and `rotation` and apply only those like `translationX` and `translationY`. This would be the expected behavior for selecting an item in a collection control, for example (or a control that allowed drag-and-drop of items to rearrange them).
- If you support only zooming, you'll ignore all the positional properties and work with `scale`, `expansion`, and/or `velocityExpansion`. This would be the sort of behavior you'd expect for a control that supported optical or semantic zoom.
- If you're interested in only rotation, the `rotation` and `velocityAngular` properties are your friends.

Of course, if you want to support multiple kinds of manipulations, you can simply apply all of these properties together, feeding them into CSS transforms, for instance. This would be expected of an app that allowed arbitrary manipulation of on-screen objects, and it's exactly what one of the gesture samples of the Windows SDK demonstrates.

The Input Instantiable Gesture Sample

While the `PointerEvents` example included with this chapter gives us a raw view of pointer and gesture events, what really matters to apps is how to apply these events to real manipulation of on-screen objects, which is to say, implementing parts of touch language such as pinch/stretch and rotation. For these we can turn to the [Input Instantiable gestures sample](#).

This sample primarily demonstrates how to use gesture events on multiple elements simultaneously. In Scenarios 1 and 2, the app simulates a simple example of a puzzle app, as mentioned earlier. Each colored box can be manipulated separately, using drag to move (with or without inertia), pinch-stretch gestures to zoom, and rotation gestures to rotate, as shown in Figure 9-4.

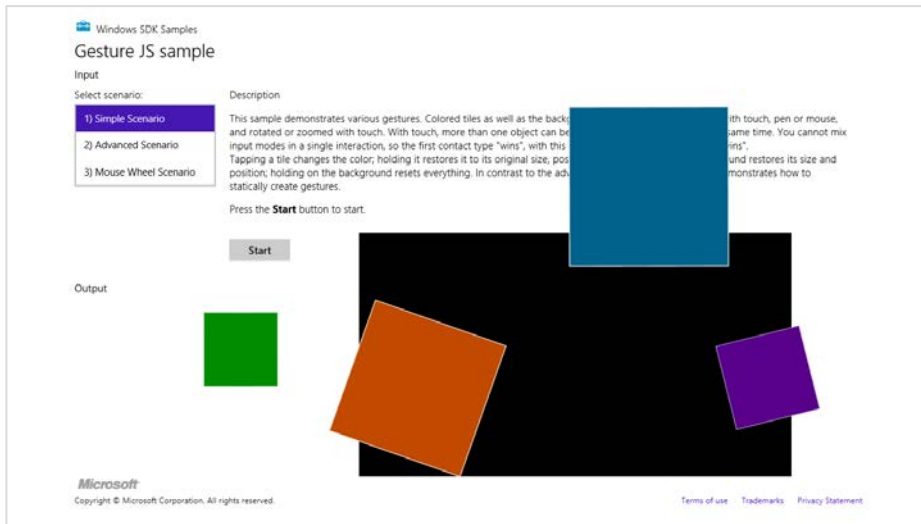


FIGURE 9-4 The Input Instantiable Gestures Sample after playing around a bit. The “instantiable” word comes from the need to instantiate an `MSGesture` object to receive gesture events.

In Scenario 1 (`js/instantiableGesture.js`), an `MSGesture` object is created for each screen element along with one for the black background “table top” element during initialization (the `initialize` function). This is the same as we’ve already seen. Similarly, the `MSPointerDown` handler (`onPointerDown`) adds pointers to the gesture object for each element, adding a little more processing to manage z-index. This avoids having simultaneous touch, mouse and stylus pointers working on the same element (which would be odd!):

```
function onPointerDown(e) {
    if (e.target.gesture.pointerType === null) { // First contact
        e.target.gesture.addPointer(e.pointerId); // Attaches pointer to element
        e.target.gesture.pointerType = e.pointerType;
    }
    else if (e.target.gesture.pointerType === e.pointerType) { // Contacts of similar type
        e.target.gesture.addPointer(e.pointerId); // Attaches pointer to element
    }

    // ZIndex Changes on pointer down. Element on which pointer comes down becomes topmost
    var zOrderCurr = e.target.style.zIndex;
    var elts = document.getElementsByClassName("GestureElement");
    for (var i = 0; i < elts.length; i++) {
        if (elts[i].style.zIndex === 3) {
            elts[i].style.zIndex = zOrderCurr;
        }
    }
    e.target.style.zIndex = 3;
}
}
```

The `MSGestureChange` handler for each individual piece (`onGestureChange`) then takes all the translation, rotation, and scaling data in the `eventArgs` object and applies them with CSS. This shows how convenient it is that all those properties are already reported in the coordinate space we need:

```
function onGestureChange(e) {
    var elt = e.target;
    var m = new MSCSSMatrix(elt.style.msTransform);

    elt.style.msTransform = m.
        translate(e.offsetX, e.offsetY).
        translate(e.translationX, e.translationY).
        rotate(e.rotation * 180 / Math.PI).
        scale(e.scale).
        translate(-e.offsetX, -e.offsetY);
}
```

There's a little more going on in the sample, but what we've shown here are the important parts. Clearly, if you didn't want to support certain kinds of manipulations, you'd again simply ignore certain properties in the event args object.

Scenario 2 of this sample has the same output but is implemented a little differently. As you can see in its `initialize` function (`js/gesture.js`), the only events that are initially registered apply to the entire "table top" that contains the black background and a surrounding border. Gesture objects for the individual pieces are created and attached to a pointer within the `MSPointerDown` event (`onTableTopPointerDown`). This approach is much more efficient and scalable to a puzzle app that has hundreds or even thousands of pieces, as gesture objects are held only for as long as a particular piece is being manipulated. Those manipulations are also like those of Scenario 1, where all the `MSGestureChange` properties are applied through a CSS transform. For further details, refer to the code comments in `js/gesture.js`, as they are quite extensive.

Scenario 3 of this sample provides another demonstration of performing translate, pinch-stretch, and rotate gestures using the mouse wheel. As shown in the `PointerEvents` example, the only thing you need to do here is process the `wheel` event, set `eventArgs.pointerId` to 1, and pass that onto your `MSPointerDown` handler that then adds the pointer to the gesture object:

```
elt.addEventListener("wheel", onMouseWheel, false);

function onMouseWheel(e) {
    e.pointerId = 1; // Fixed pointerId for MouseWheel
    onPointerDown(e);
}
```

Again, that's all there is to it. (I love it when it's so simple!) As an exercise, you might try adding this little bit of code to Scenarios 1 and 2 as well.

The Gesture Recognizer

With inertial gestures, which continue to send some number of `MSGestureChange` events after pointers are released, you might be asking this question: What, exactly, controls those events? That is, there is

obviously a specific deceleration model built into those events, namely the one around which the Windows look and feel is built. But what if you want a different behavior? And what if you want to interpret pointer events in different way altogether?

The agent that interprets pointer events into gesture events is called the gesture recognizer, which you can get to directly through the [Windows.UI.Input.GestureRecognizer](#) object. After instantiating this object with `new`, you then set its [gestureSettings](#) properties for the kinds of manipulations and gestures you're interested in. The documentation for [Windows.UI.Input.GestureSettings](#) gives all the options here, which include `tap`, `doubleTap`, `hold`, `holdWithMouse`, `rightTap`, `drag`, translations, rotations, scaling, inertia motions, and `crossSlide` (swipe). For example, in the [Input manipulations and gestures sample](#) (ballineight.js) we can see how it configures a recognizer for tap, rotate, translate, and scale (with inertia):

```
gr = new Windows.UI.Input.GestureRecognizer();

// Configuring GestureRecognizer to detect manipulation rotation, translation, scaling,
// + inertia for those three components of manipulation + the tap gesture
gr.gestureSettings =
    Windows.UI.Input.GestureSettings.manipulationRotate |
    Windows.UI.Input.GestureSettings.manipulationTranslateX |
    Windows.UI.Input.GestureSettings.manipulationTranslateY |
    Windows.UI.Input.GestureSettings.manipulationScale |
    Windows.UI.Input.GestureSettings.manipulationRotateInertia |
    Windows.UI.Input.GestureSettings.manipulationScaleInertia |
    Windows.UI.Input.GestureSettings.manipulationTranslateInertia |
    Windows.UI.Input.GestureSettings.tap;

// Turn off UI feedback for gestures (we'll still see UI feedback for PointerPoints)
gr.showGestureFeedback = false;
```

The [GestureRecognizer](#) also has a number of properties to configure those specific events. With cross-slides, for example, you can set the [crossSlideThresholds](#), [crossSlideExact](#), and [crossSlideHorizontally](#) properties. You can set the deceleration rates (in pixels/ms²) through [inertiaExpansionDeceleration](#), [inertiaRotationDeceleration](#), and [inertiaTranslation-Deceleration](#).

Once configured, you then start passing [MSPointer*](#) events to the recognizer object, specific to its methods named [processDownEvent](#), [processMoveEvents](#), and [processUpEvent](#) (also [processMouseWheelEvent](#), and [processInertia](#), if needed). In response, depending on the configuration, the recognizer will then fire a number of its own events. First, there are discrete events like [crossSliding](#), [dragging](#), [holding](#), [rightTapped](#), and [tapped](#). For all others it will fire a series of [manipuationStarted](#), [manipulationUpdated](#), [manipulationInertiaStarting](#), and [manipulationCompleted](#) events. Note that all of these come from WinRT to be sure to call [removeEventListener](#) as needed.

When you're using the recognizer directly, in other words, you'll be listening for [MSPointer*](#) events, feeding them to the recognizer, and then listening for and acting on the recognizer's specific events as above rather than the [MSGesture*](#) events that come out of the default recognizer configured by the [MSGesture](#) object.

Again, refer to the documentation on [Windows.UI.Input.GestureRecognizer](#) for all the details and refer to the sample for some bits of code. As one extra example, here's a snippet to capture a small horizontal motion using the `manipulationTranslateX` setting:

```
var recognizer = new Windows.UI.Input.GestureRecognizer();
recognizer.gestureSettings = Windows.UI.Input.GestureSettings.manipulationTranslateX;
var DELTA = 10;

myElement.addEventListener('MSPointerDown', function (e) {
    recognizer.processDownEvent(e.getCurrentPoint(e.pointerId));
});
myElement.addEventListener('MSPointerUp', function (e) {
    recognizer.processUpEvent(e.getCurrentPoint(e.pointerId));
});
myElement.addEventListener('MSPointerMove', function (e) {
    recognizer.processMoveEvents(e.getIntermediatePoints(e.pointerId));
});

// Remember removeEventListener as needed for this event
recognizer.addEventListener('manipulationcompleted', function (args) {
    var pt = args.cumulative.translation;
    if (pt.x < -DELTA) {
        // move right
    }
    else if (pt.x > DELTA) {
        // move left
    }
});
```

Beyond the recognizer, do note that you can always go the low-level route and do your own processing of `MSPointer*` events however you want, completely bypassing the gesture recognizer. This would be necessary if the configurations allowed by the recognizer object don't accommodate your specific need. At the same time, now is a good opportunity to re-read "Sidebar: Creating Completely New Gesture?" at the end of the earlier section on the touch language. It addresses a few of the questions about when and if custom gestures are really needed.

Keyboard Input and the Soft Keyboard

After everything to do with touch and other forms of input, it seems almost anticlimactic to consider the humble keyboard. Yet of course the keyboard remains utterly important for textual input, whether it's a physical key-board or the on-screen "soft" keyboard. It is especially important for accessibility as well, as some users are physically unable to use a mouse or other devices. In fact, the [Windows 8 app certification requirements](#) (section 3.5) make keyboard input mandatory.

Fortunately, there is nothing special about handling keyboard input in a Windows Store app, but a little goes a long way. Drawing from [Implementing keyboard accessibility](#), here's a summary:

- Process `keydown`, `keyup`, and `keypress` events as you already know how to do, especially for implementing special key combinations. See “Standard Keystrokes” later in this section for a quick run-down of typical mappings.
- Have `tabindex` attributes on interactive elements that should be tab stops. Avoid adding `tabindex` to noninteractive elements as this will interfere with screen readers.
- Have `accesskey` attributes on those elements that should have keyboard shortcuts.
- Call the DOM focus API on whatever element should be the default.
- Take advantage of the keyboard support that already exists in built-in controls, such as the App Bar.

As an example, the Here My Am! we’ve been working with in this book has been updated in this chapter’s companion content with full keyboard support. This was mostly a matter of adding `tabindex` to a few elements, setting focus to the image area, and picking up `keydown` events on the `img` elements for the Enter key and spacebar where we’ve already been handling `click`. Within those `keydown` events, note that it’s helpful to use the `WinJS.Utilities.Key` enumeration for comparing key codes:

```
var Key = WinJS.Utilities.Key;
var image = document.getElementById("photo");

image.addEventListener("keydown", function (e) {
    if (e.keyCode == Key.enter || e.keyCode==Key.space) {
        image.click();
    }
});
```

All this works for both the physical keyboard as well as the soft keyboard. Case closed? Well, not entirely. There are two special concerns with the soft keyboard: how to make it appear, and the effect of its appearance on app layout. At the end of this section I’ll also provide a quick run-down of standard keystrokes for app commands.

Soft Keyboard Appearance and Configuration

The appearance of the soft keyboard happens for one reason and one reason only: the user *touches* a text input element or an element with the `contenteditable="true"` attribute (such as a `div` or `canvas`). There isn’t an API to make the keyboard appear, nor will it appear when you click in such an element with the mouse or a stylus, or tab to it with a physical keyboard.

The configuration of the keyboard is also sensitive to the type of input control. We can see this through Scenario 2 of the [Input Touch keyboard text input sample](#), where `html/ScopedViews.html` contains a bunch of `input` controls (surrounding table markup omitted), which appear as shown in Figure 9-5:

```

<input type="url" name="url" id="url" size="50" />
<input type="email" name="email" id="email" size="50" />
<input type="password" name="password" id="password" size="50" />
<input type="text" name="text" id="text" size="50" />
<input type="number" name="number" id="number" />
<input type="search" name="search" id="search" size="50" />
<input type="tel" name="tel" id="tel" size="50" />

```

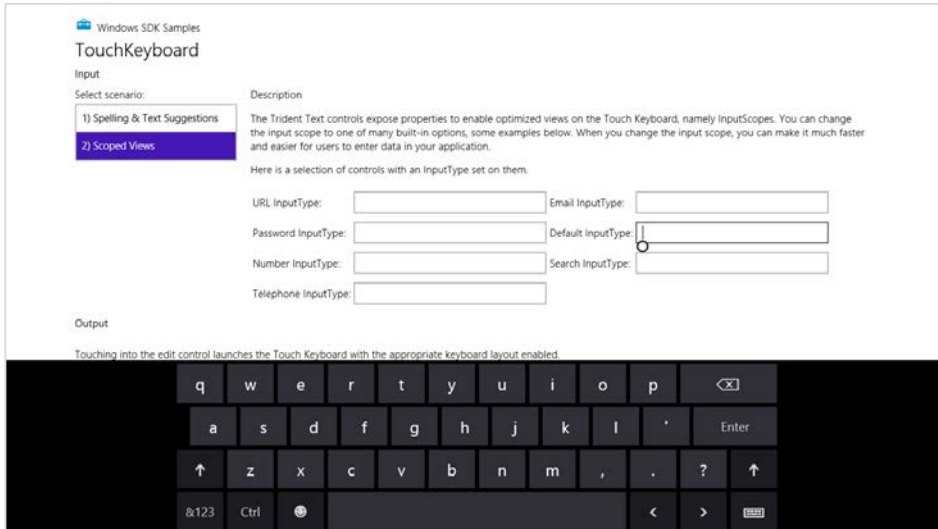


FIGURE 9-5 The soft keyboard appears when you touch an input field, as shown in the Input Touch keyboard text input sample.

What's shown in Figure 9-5 is the default keyboard. If you tap in the Search field, you get pretty much the same view except the Enter key turns into Search. For the Email field, it's much like the default view except you get @ and .com keys to either side of the spacebar:



The URL keyboard is the same except the @ key is dropped and Enter turns into Go:



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



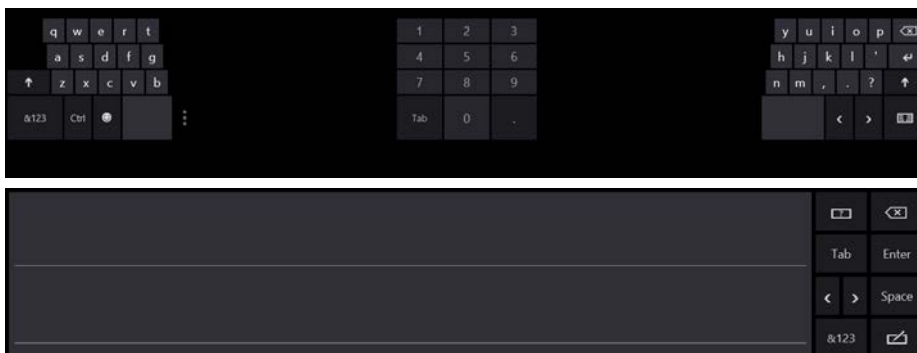
And finally, the Number and Telephone fields bring up a number-oriented view:



In all of these cases, the key on the lower right (whose icon looks a bit like a keyboard), lets you switch to other keyboard layouts:

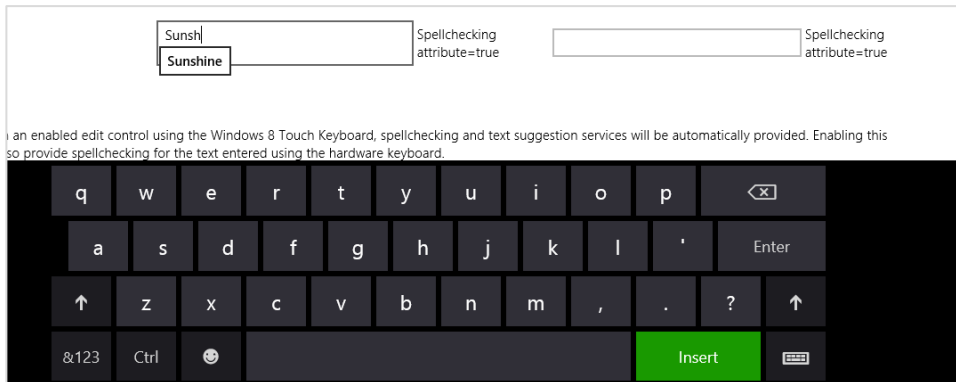


The options here are the normal (wide) keyboard, the split keyboard, a handwriting recognition panel, and a key to dismiss the soft keyboard entirely. Here’s what the default split keyboard and handwriting panels look like:



This handwriting panel for input is simply another mode of the soft keyboard: you can switch between the two, and your selection sticks across invocations. (For this reason, Windows does not automatically invoke the handwriting panel for a pen pointer, because the user may prefer to use the soft keyboard even with the stylus.)

The keyboard will also adjust its appearance with text input controls to provide text suggestions; specifically, a highlighted Insert key appears. This is demonstrated in Scenario 1 of the sample and shown below:



Adjusting Layout for the Soft Keyboard

The second concern with the soft keyboard (no, I didn't forget!) is handling layout when the keyboard might obscure the input field with the focus.

When the soft keyboard or handwriting panel appears, the system will try to make sure the input field is visible by scrolling the page content if it can. This means that it just sets a negative vertical offset to your entire page equal to the height of the soft keyboard. For example, if I add (as a total hack!) a bunch of `
` elements at the top of `html/ScopedView.html` in the sample, such that the input controls are at the bottom of the page, and then I touch one of them, the whole page is slid up, as shown in Figure 9-6.



FIGURE 9-6 When the soft keyboard appears, Windows will automatically slide the app page up to make sure the input field isn't obscured.

Although this can be the easiest solution to this particular concern, it's not always ideal. Fortunately, you can do something more intelligent if you'd like by listening to the `hiding` and `showing` events of the `Windows.UI.ViewManagement.InputPane` object and adjust your layout directly. Code for doing this can be found in the—are you ready for this one?—[Responding to the appearance of the on-screen](#)

[keyboard sample](#).⁵⁰ Adding listeners for these events is simple (see the bottom of `js/keyboardPage.js`, which also removes the listeners properly):

```
var inputPane = Windows.UI.ViewManagement.InputPane.getForCurrentView();
inputPane.addEventListener("showing", showingHandler, false);
inputPane.addEventListener("hiding", hidingHandler, false);
```

Within the `showing` event handler, the `eventArgs.occludedRect` object (a `Windows.Foundation.Rect`) gives you the coordinates and dimensions of the area that the soft keyboard is covering. In response, you can adjust whatever layout properties are applicable and set the `eventArgs.ensuredFocusedElementInView` property to `true`. This tells Windows to bypass its automatic offset behavior:

```
function showingHandler(e) {
    if (document.activeElement.id === "customHandling") {
        keyboardShowing(e.occludedRect);

        // Be careful with this property. Once it has been set, the framework will
        // do nothing to help you keep the focused element in view.
        e.ensuredFocusedElementInView = true;
    }
}
```

The sample shows both cases. If you tap on the aqua-colored *defaultHandling* element on the bottom left of the app, as shown in Figure 9-7, this `showingHandler` does nothing, so the default behavior occurs.

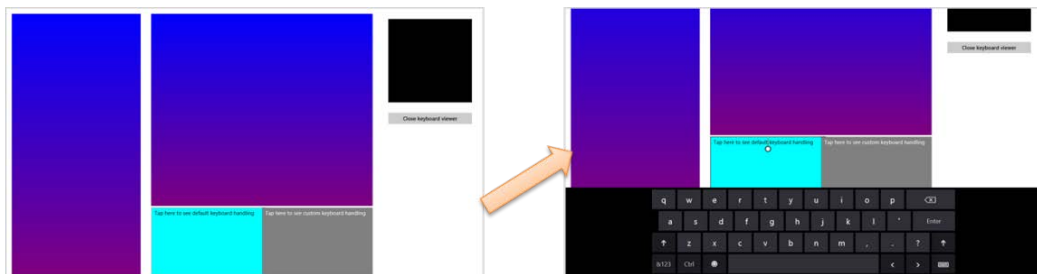


FIGURE 9-7 Tapping on the left *defaultHandling* element at the bottom shows the default behavior when the keyboard appears, which offsets other page content vertically.

If you tap the *customHandling* element (on the right), it calls its `keyboardShowing` routine to do layout adjustment:

```
function keyboardShowing(keyboardRect) {
    // Some code omitted...
```

⁵⁰ And while you might think this is the second longest JavaScript sample name in the Windows SDK, it actually gets only the bronze medal. The [Unselectable content areas with -ms-user-select CSS attribute sample](#), as we've seen, gets the gold by seven characters. [Using requestAnimationFrame for power efficient animations sample](#) wins the silver by 4. I don't mind such long names, however—I'm delighted that there we have such an extensive set of great samples to draw from!

```

var elementToAnimate = document.getElementById("middleContainer");
var elementToResize = document.getElementById("appView");
var elementToScroll = document.getElementById("middleList");

// Cache the amount things are moved by. It makes the math easier
displacement = keyboardRect.height;
var displacementString = -displacement + "px";

// Figure out what the last visible things in the list are
var bottomOfList = elementToScroll.scrollTop + elementToScroll.clientHeight;

// Animate
showingAnimation = KeyboardEventsSample.Animations.inputPaneShowing(elementToAnimate,
    { top: displacementString, left: "0px" }).then(function () {

    // After animation, layout in a smaller viewport above the keyboard
    elementToResize.style.height = keyboardRect.y + "px";

    // Scroll the list into the right spot so that the list does not appear to scroll
    elementToScroll.scrollTop = bottomOfList - elementToScroll.clientHeight;
    showingAnimation = null;
});
}

```

The code here is a little involved because it's animating the movement of the various page elements. The short of it is that the layout of affected elements—namely the one that is tapped—is adjusted to make space for the keyboard. Other elements on the page are otherwise unaffected. The result is shown in Figure 9-8.

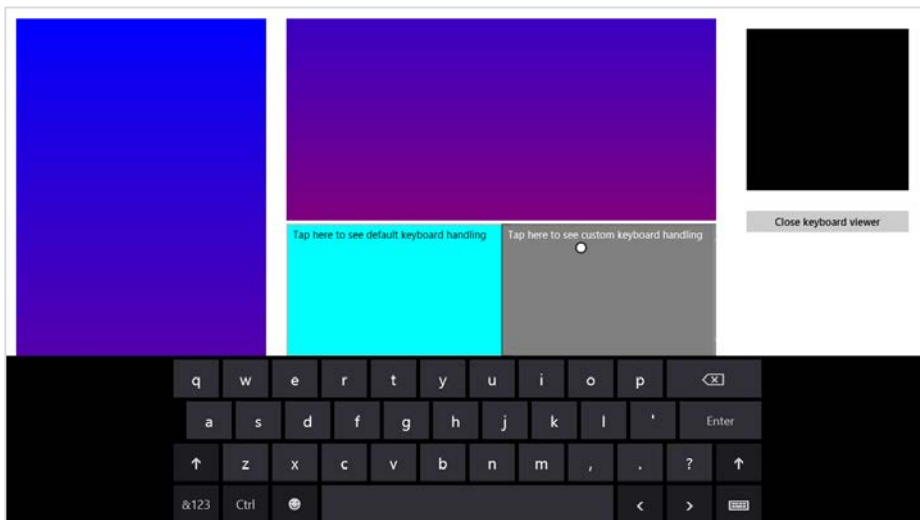


FIGURE 9-8 Tapping the gray *customHandling* element on the right shows custom handling for the keyboard's appearance.

Standard Keystrokes

The last piece I wanted to include on the subject of the keyboard is a table of command keystrokes you might support in your app. These are in addition to the touch language equivalents, and you're probably accustomed to using many of them already. They're good to review because again, apps should be fully usable with just the keyboard, and implementing keystrokes like these goes a long way toward fulfilling that requirement and enabling more efficient use of your app by keyboard users.

Action or Command	Keystroke
Move focus	Tab
Back (navigation)	Back button on special keyboards; backspace if not in a text field; Alt+left arrow
Forward (navigation)	Alt+right arrow
Up	Alt+up arrow
Cancel/Escape from mode	ESC
Walk through items in a list	Arrow keys (plus Tab)
Jump through items in a list to next group if selection doesn't automatically follow focus	Ctrl+arrow keys
Zoom (semantic and optical)	Ctrl+ and Ctrl-
Jump to something in a named collection	Start typing
Jump far	Page up/down (should work in panning UI, in either horizontal or vertical directions)
Next tab or group	Ctrl+Tab
Previous tab or group	Ctrl+Shift+Tab
Nth tab or group	Ctrl+N (1-9)
Open app bar (Windows handles this automatically)	Win+Z
Context menu	Context menu key
Open additional flyout/select menu item	Enter
Navigate into/activate	Enter (on a selection)
Select	Space
Select contiguous	Shift+arrow keys
Pin this	Ctrl+Shift+!
Save	Ctrl+S
Find	Ctrl+F
Print	Ctrl+P (call <code>Windows.Graphics.Printing.PrintManager.showPrintUIAsync</code>)
Copy	Ctrl+C
Cut	Ctrl+X
Paste	Ctrl+V
New Item	Ctrl+N
Open address	Ctrl+L or Alt+D
Rotate	Ctrl+, and Ctrl+.
Play/Pause	Ctrl+P (media apps only)
Next item	Ctrl+F (conflict with Find)
Previous item	Ctrl+B
Rewind	Ctrl+Shift+B
Fast forward	Ctrl+Shift+F

Inking

Beyond the built-in soft keyboard/handwriting pane, an app might also want to provide a surface on which it can directly accept pointer input as *ink*. By this I mean more than just having a [canvas](#) element and processing [MSPointer*](#) events to draw on it to produce a raster bitmap. Ink is a data structure that maintains the actual input data (including pressure, angle, and velocity if the hardware supports it) which allows for handwriting recognition and other higher-level processing that isn't possible with raster data. Ink, in other words, remembers how an image was drawn, not just the final image itself, and it works with all types of pointer input.

Ink support in WinRT is found in the [Windows.UI.Input.Inking](#) namespace. This API doesn't depend on any particular presentation framework, nor does it provide for rendering: it deals only with the managing data structures that an app can then render itself to a drawing surface such as a [canvas](#). Here's its basic function:

- Create an instance of the manager object with `new Windows.UI.Input.Inking.InkManager()`.
- Assign any drawing attributes by creating a [Windows.UI.Input.Inking.InkDrawing-Attributes](#) object and settings attributes like the ink [color](#), [fitToCurve](#) (as opposed to the default straight lines), [ignorePressure](#), [penTip](#) ([Windows.UI.Input.Inking.PenTipShape.circle](#) or [rectangle](#)), and [size](#) (a [Windows.Foundation.Size](#) object with [height](#) and [width](#)).
- For the input element, listen for the [MSPointerDown](#), [MSPointerMove](#), and [MSPointerUp](#) events, which you generally need to handle for display purposes already. The [eventArgs.currentPoint](#) is a [Windows.UI.Input.PointerPoint](#) object that contains a pointer id, point coordinates, and properties like pressure, tilt, and twist.
- Pass that [PointerPoint](#) object to the ink manager's [processPointerDown](#), [process-PointerUpdate](#), and [processPointerUp](#) methods, respectively.
- After [processPointerUp](#), the ink manager will create a [Windows.UI.Input.Inking.InkStroke](#) object for that path. Those strokes can then be obtained through the ink manager's [getStrokes](#) method and rendered as desired.
- Higher-order gestures can be also converted into [InkStroke](#) objects directly and given to the manager through its [addStroke](#) method. Stroke objects can also be deleted with [deleteStroke](#).

The ink manager also provides methods for performing handwriting recognition with its contained strokes, saving and loading the data, and handling different modes like draw and erase. For a complete demonstration, check out the [Input Ink sample](#) that is shown in Figure 9-9. This sample lets you see the full extent of inking capabilities, including handwriting recognition.



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

The SDK also includes the [Input Simplified ink sample](#) to demonstrate a more focused handwriting recognition scenario, as shown in Figure 9-10. You should know that this is one sample that *doesn't* support touch at all—it's strictly mouse and stylus and uses keystrokes for various commands instead of an app bar. Look at the [keydown](#) function in `simpleink.js` for a list of the Ctrl+key commands; the spacebar performs recognition of your strokes and the backspace key clears the canvas. As you can see in the figure, I think the handwriting recognition is quite good! (It tells me that the handwriting samples I gave to an engineering team at Microsoft somewhere in the mid-1990s must have made a valuable contribution.)

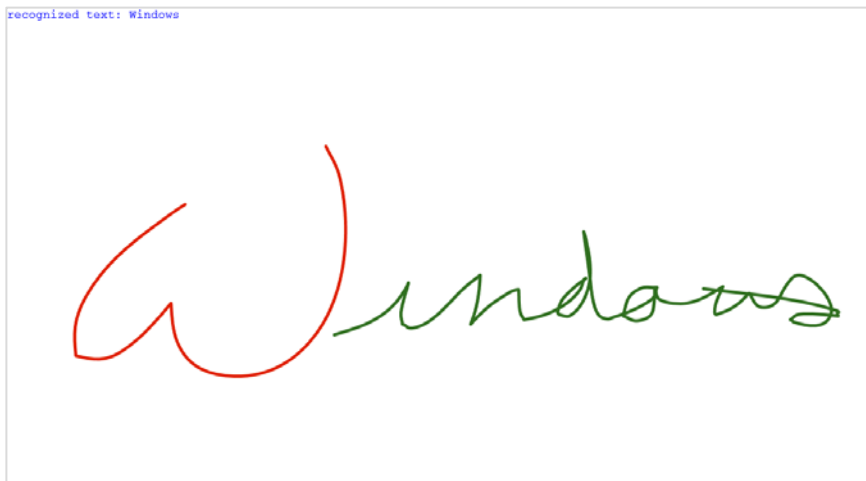


FIGURE 9-10 The Input Simplified Ink sample doing a great job recognizing my sloppy mouse-based handwriting.

Geolocation

Before we explore sensors more generally, I'll call out the geolocation capabilities for Windows Store apps separately because its API is structured differently from other sensors. We've already used this since Chapter 2, "Quickstart" in the Here My Am! app, but we need the more complete story of this highly useful capability.

Unlike all other sensors, in fact, geolocation is the *only* one that has an associated capability you must declare in the manifest. Where you are on the earth is an absolute measure, if you will, and is therefore classified as a piece of personal information. So, users must give their consent before an app can obtain that information, and your app must also provide a Privacy Statement in the Windows Store. Other sensor data, in contrast, is relative—you cannot, for example, really know anything about a *person* from how a device is tilted, how it's moving, or how much light is shining on it. Accordingly, you can use those other sensors without declaring any specific capabilities.

As you might know, geolocation can be obtained in two different ways. The primary and most precise way, of course, is to get a reading from an actual GPS radio that is talking to geosynchronous satellites some hundreds of miles up in orbit. The other reasonably useful means, though not always accurate, is to attempt to find one's position through the IP address of a wired network connection or to triangulate from the position of available WiFi hotspots. Whatever the case, WinRT will do its best to give you a decent reading.

To access geolocation readings, you must first create an instance of the WinRT geolocator, [Windows.Devices.Geolocation.Geolocator](#). With that in hand, you can then call its [getGeopositionAsync](#) method, whose results (delivered to your completed handler) is a [Geoposition](#) object (in the same [Windows.Devices.Geolocation](#) namespace, as everything here is unless noted). Here's the code as it appears in Here My Am!:

```
var gl = new Windows.Devices.Geolocation.Geolocator();

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

The [getGeopositionAsync](#) method also has a [variation](#) where you can specify two parameters: a maximum age for a cached reading (which is to say, how stale you can allow a reading to be) and a timeout value for how long you're willing to wait for a response. Both values are in milliseconds.

A [Geoposition](#) contains two properties. First, its [coordinate](#) property is a [Geocoordinate](#) object that provides [accuracy](#) (meters), [altitude](#) (meters), [altitudeAccuracy](#) (meters), [heading](#) (degrees relative to true north), [latitude](#) (degrees), [longitude](#) (degrees), [speed](#) (meters/sec), and a [timestamp](#) (Date). The second part of a [Geoposition](#) is a [CivicAddress](#) object named—what else!—[civicAddress](#), which might contain [city](#) (string), [country](#) (string, a two-letter ISO-3166 country code), [postalCode](#) (string), [state](#)

(string), and `timestamp` (Date) properties, if the geolocation provider supplies such data.⁵¹

You can indicate the accuracy you're looking for through the Geolocator's `desiredAccuracy` property, which is either `PositionAccuracy.default` or `PositionAccuracy.high`. The latter, mind you, will be much more radio or network intensive. This might incur higher costs on metered broadband connections and can shorten battery life, so set this to `high` only if it's essential to your user experience.

The Geolocator also provides a `locationStatus` property, which is a `PositionStatus` object containing `ready`, `initializing`, `noData`, `disabled`, `notInitialized`, and `notAvailable`. It should be obvious that you can't get data from a Geolocator that's in any state other than `ready`. To track this, you can listen to the Geolocator's `statusChanged` event, where `eventArgs.status` property in your handler will contain a `PositionStatus`; this is helpful when you find that a GPS device might take a couple seconds to provide a reading. For an example of using this event, see Scenario 1 of the [Geolocation sample](#) in the Windows SDK:

```
geolocator = new Windows.Devices.Geolocation.Geolocator();
geolocator.addEventListener("statuschanged", onStatusChanged); //Remember to remove later

function onStatusChanged(e) {
    switch (e.status) {
        // ...
    }
}
```

Note that `PositionStatus` and `statusChanged` reflect the readiness of the GPS *device*, and that readiness is not affected by the Location permission in the app's Settings pane. As demonstrated in *Here My Am!*, an app needs to check permissions by trying to obtain a setting, which is a different concern from device readiness.

The other two interesting properties of the Geolocator are `movementThreshold`, a distance in meters that the device can move before another reading is triggered (which can be used for geo-fencing scenarios), and `reportInterval`, which is the number of milliseconds between attempted readings. Be conservative with the latter, setting it to what you really need, because you again want to minimize network or radio activity. In any case, when the Geolocator takes another reading and finds that the device has moved beyond the `movementThreshold`, it will fire a `positionChanged` event, where the `eventArgs.position` property is a new `Geoposition` object. This is also shown in Scenario 1 of the Geolocation sample:

```
geolocator.addEventListener("positionchanged", onPositionChanged);

function onPositionChanged(e) {
    var coord = e.position.coordinate;

    document.getElementById("latitude").innerHTML = coord.latitude;
    document.getElementById("longitude").innerHTML = coord.longitude;
}
```

⁵¹ That is, the `civicAddress` property might not be available or might be empty. An alternate means to obtain it is to use the [Bing Maps API](#), specifically the `MapAddress` class, to convert coordinates into an address.

```

    document.getElementById("accuracy").innerHTML = coord.accuracy;
}

```

With `movementThreshold` and `reportInterval`, really think through what your app needs based on the accuracy and/or refresh intervals of the data you're using in relation to the location. For example, weather data is regional and might be updated only hourly. Therefore, `movementThreshold` might be set on the scale of miles or kilometers and `reportInterval` at 15, 30, 60 minutes, or longer. A mapping or real-time traffic app, on the other hand, works with data that is very location-sensitive and will thus have a much smaller threshold and a much shorter interval.

Where battery life is concerned, it's best to simply take a reading when the user wants one, rather than following the position at regular intervals. But this again depends on the app scenario, and you could also provide a setting that lets the user control geolocation activity.

It's also very important to note that apps won't get `positionChanged` or `statusChanged` events while suspended unless you register a time trigger background task for this purpose and the user adds the app to the lock screen. We'll talk more of this in Chapter 13, "Tiles, Notifications, the Lock Screen, and Background Tasks," and you can also see how this works in Scenario 3 of the Geolocation sample. If, however, you don't use a background task or the user doesn't place you on the lock screen and you still want to track the user's position, be sure to handle the `resuming` event and refresh the position there.

On the flip side, some geolocation scenarios, such as providing navigation, need to also keep the display active (preventing automatic screen shutoff) even when there's no user activity. For this purpose you can use the `Windows.System.Display.DisplayRequest` class, namely its `requestActive` and `releaseRelease` methods that you would call when starting and ending a navigation session. Of course, since keeping the display active consumes more battery power, only use this capability when necessary—as when specifically providing navigation—and avoid simply making the request when your app starts. Otherwise your app will probably gain a reputation in the Windows Store as being power hungry!

Sidebar: HTML5 Geolocation

An experienced HTML/JavaScript developer might wonder why WinRT provides a Geolocation API when HTML5 already has one: `window.navigator.geolocation` and its `getCurrentPosition` method that returns an object with coordinates. The reason for the overlap is that other languages like C#, Visual Basic, and C++ don't have another API to draw from, which leaves HTML/JavaScript developers a choice. Under the covers, the HTML5 API hooks into the same data as the WinRT API, requires the same manifest capability, *Location*, and is subject to the same user consent, so for the most part the two APIs are almost equivalent. I would give WinRT a slight edge due to the `movementThreshold` option, which helps an app cooperate with power management and enables easier geo-fencing. Doing the same with the HTML5 API would require more frequent polling and battery consumption. For many scenarios, however, you can use either one with equal results.

Like all other WinRT APIs, however, [Windows.Devices.Geolocation](#) is available only in local context pages in a Windows Store app. In web context pages you can use the HTML5 API.

Sensors

As I wrote in the chapter's introduction, I like to think of sensors as another form of input. It makes a lot of sense because every device that is now wholly integrated into our computer systems—such that we take them for granted—was at one point a kind of human-interface peripheral. In time, I suspect that many of the sensors that are new to us today will be standard equipment just about everywhere.

Sensors, again, are a way of understanding the relationship of a device to the physical world around it, and this constitutes input because you, as a human being, can affect that relationship primarily by moving the device around in physical space or otherwise changing its environment. Sensors can also be used as direct input to cause motion on the screen rather than relying on some form of abstract input like the keyboard or mouse. For example, instead of using keystrokes to abstractly tilt a game board, you can, with sensors, just tilt the device. Shaking, in fact, is becoming a well-known physical gesture that can be wired to a command of some kind like *Retry Now, darn you! Why aren't you doing what I want?* Haven't we for years been shaking or smacking our computers when they aren't behaving properly? Well, with sensors the computer can now actually respond!

Here, then, is what the various sensors tell us:

- **Location** The device's position on the earth (as we covered in the previous section).
- **Compass and orientation** The direction the device is pointing, relative to the earth's magnetic poles or relative to the device's inherent sense of position (both simple and complex orientation).
- **Inclinometer** The static pitch, roll, and yaw of the device in 3D space.
- **Gyrometer** The angular velocity/rotational motion of the device in 3D space.
- **Accelerometer** The linear G-force acceleration of the device within 3D space (x, y, z).
- **Ambient light** The amount of light surrounding the device.

These are the sensors that are represented in the WinRT API,⁵² some of which are created in software through *sensor fusion*. This means taking raw data from one or more hardware sensors and combining, interpreting, and presenting it all in a form that's more directly useful to apps. Just as with pointers, you can still get to raw data if you want it, but oftentimes it's unnecessary. For example, the Simple

⁵² There is also the proximity sensor for near-field communications (NFC) that tells us when devices are near one another or make contact, but this is more a networking handshake than a sensor like the others. We'll see this in Chapter 15, "Devices and Printing."

Orientation sensor provides a simple interpretation of how the device is oriented in relation to its default position, rounding everything off, as it were, to the nearest 90-degree quadrant. The full Orientation sensor, on the other hand, combines gyrometer, accelerometer, and compass data to provide an exact 3D orientation matrix that is much more precise but much more oriented (if I might make the pun!) to advanced scenarios than simply needing to know whether the device is upside-down or rightside-up.

Because all of these sensors are very similar in how they work (which is intentional, with the exception of the Simple Orientation sensor, which is intentionally dissimilar!), I want to show the general pattern of the sensor APIs rather than explicit examples for each. Such examples are readily available in these SDK samples: [Accelerometer](#), [Compass](#), [Gyrometer](#), [Inclinometer](#), [Light Sensor](#), and [OrientationSensor](#).

The usage pattern is as follows, with the particulars summarized in the table that follows:

- Obtain a sensor object via `Windows.Devices.Sensors.<sensor>.getDefault()`.
- Call that object's `getCurrentReading` to obtain a one-time reading.
- For ongoing readings, configure the object's `minimumReportInterval` and `reportInterval` properties (both in milliseconds) and listen to the object's `readingChanged` event. Your handler will receive a reading object of an appropriate type in response. As with geolocation, setting these values wisely will help optimize battery life by avoiding excess electrons flying through the sensors!

Sensor Name (Windows.Devices.Sensors.)	Added Members	Reading Type (Windows.Devices.Sensors)	Reading Properties (timestamp is a Date; all others are Numbers unless noted)
Accelerometer	Event: <code>shaken</code> (event args contains only a <code>timestamp</code> property)	AccelerometerReading	<code>accelerationX</code> (G's), <code>accelerationY</code> , <code>accelerationZ</code> , <code>timestamp</code>
Compass	n/a	CompassReading	<code>headingMagneticNorth</code> (degrees), <code>headingTrueNorth</code> , <code>timestamp</code>
Gyrometer	n/a	GyrometerReading	<code>angularVelocityX</code> (degrees/sec), <code>angularVelocityY</code> , <code>angularVelocityZ</code> , <code>timestamp</code>
Inclinometer	n/a	InclinometerReading	<code>pitchDegrees</code> (degrees), <code>rollDegrees</code> (degrees), <code>yawDegrees</code> (degrees), <code>timestamp</code>
LightSensor	n/a	LightSensorReading	<code>illuminanceInLux</code> (lux), <code>timestamp</code>
OrientationSensor	n/a	OrientationSensorReading	<code>quaternion</code> , (<code>SensorQuaternion</code> containing <code>w</code> , <code>x</code> , <code>y</code> , and <code>z</code> properties) <code>rotationMatrix</code> (<code>Sensor-RotationMatrix</code> containing <code>m11</code> , <code>m12</code> , <code>m13</code> , <code>m21</code> , <code>m22</code> , <code>m23</code> , <code>m31</code> , <code>m32</code> , <code>m33</code> properties), <code>timestamp</code>

Here's an example of such code from the Gyrometer sample (js/scenario1.js):

```
gyrometer = Windows.Devices.Sensors.Gyrometer.getDefault();

var minimumReportInterval = gyrometer.minimumReportInterval;
var reportInterval = minimumReportInterval > 16 ? minimumReportInterval : 16;
gyrometer.reportInterval = reportInterval;

gyrometer.addEventListener("readingchanged", onDataChanged); // Remember to remove as needed

function onDataChanged(e) {
    var reading = e.reading;

    document.getElementById("eventOutputX").innerHTML = reading.angularVelocityX.toFixed(2);
    document.getElementById("eventOutputY").innerHTML = reading.angularVelocityY.toFixed(2);
    document.getElementById("eventOutputZ").innerHTML = reading.angularVelocityZ.toFixed(2);
}
```

With the Orientation Sensor, a *quaternion* can be most easily understood as a rotation of a point [x,y,z] about a single arbitrary axis. This is different from a rotation matrix, which represents rotations around three axes. The mathematics behind quaternions is fairly exotic because it involves the geometric properties of complex numbers and mathematical properties of imaginary numbers, but working with them is simple and frameworks like DirectX support them. See the OrientationSensor sample for more.

Speaking of orientation, I mentioned that the [SimpleOrientationSensor](#) works a little differently. Its purpose is to supply *quadrant* orientation rather than exact orientation, which is perhaps all you need. For example, a star chart app would need to know if a slate device is upside-down so that it can adjust its display (along with a compass reading) to match the sky itself.

To summarize this sensor's usage:

- Call `Windows.Devices.Sensors.SimpleOrientation.getDefault` to obtain the object.
- Call the `getCurrentOrientation` to obtain a reading.
- The `orientationChanged` event provides for ongoing readings, where `EventArgs` contains `orientation` (a reading) and `timestamp` properties.
- The reading is a [SimpleOrientation](#) object that contains these properties:
 - `notRotated` ("portrait up"), `rotated90DegreesCounterclockwise` ("portrait left"), `rotated90-DegreesCounterclockwise` ("portrait down"), `rotated270Degrees-Counterclockwise` ("landscape right") Note that these are entirely different from view states like `fullscreen-landscape` and `fullscreen-portrait`.
 - `faceup`, `facedown` (slate devices only).

For a demonstration, see the [SimpleOrientationSensor sample](#).

What We've Just Learned

- “Design for touch, get mouse and stylus for free” is a message that holds true, because working with pointer and gesture input from a variety of input devices doesn’t require you to differentiate between the forms of input.
- Using built-in controls is the easiest way to handle input, but you can also handle `MSPointer*` events and `MSGesture*` events directly, when needed. You can also feed `MSPointer*` events into a custom gesture recognizer (that issues its own events).
- The Windows 8 touch language includes tap, press and hold, slide/pan, cross-slide (to select), pinch-stretch, rotate, and edge gestures (from top/bottom and from the sides). A tap is typically handled with a click `event`, whereas the others require the creation of an `MSGesture` object, association of that object with a pointer, and handling of `MSGesture*` event sequences which provide for manipulations and inertial motions together.
- The touch language also has mouse, stylus, and keyboard equivalents. For mouse and stylus, there is very little work an app needs to do (such as sending mouse `wheel` events to the gesture object). Keyboard support must be implemented separately, but simply uses the standard HTML/JavaScript events.
- Keyboard support also includes accommodating the soft (on-screen) keyboard, which appears automatically for text input fields and other content-editable elements. It automatically adjusts its appearance according to input type, and will slide the app contents up if necessary to avoid having the keyboard overlap the input control. An app can also handle visibility events directly to provide a better experience than the default.
- The Inking API provides apps with the means to record, save, and render an entire series of pointer activities, where the strokes can also be fed into a handwriting recognizer.
- The Geolocation API in WinRT, similar to the HTML5 geolocation API, provides apps with access to GPS data as well as events when the device has moved past a specified threshold.
- The WinRT API represents a number of sensors that can also be used as input to an app. In addition to geolocation, the sensors are compass, orientation, simple orientation (quadrant-based), inclinometer, gyrometer, accelerometer, and ambient light.
- Most sensors follow the same usage pattern: acquire the sensor object, get a current reading, and possibly listen to the `readingchanged` event. They are very easy to work with, leaving much of your energy to apply them creatively!

Chapter 10

Media

To say that media is important to apps—and to culture in general—is a gross understatement. Ever since the likes of Edison made it possible to record a performance for later enjoyment, and the likes of Marconi made it possible to widely broadcast and distribute such performances, humanity’s worldwide appetite for media—graphics, audio, and video—has probably outpaced the appetite for automobiles, electricity, and even junk food. In the early days of the Internet, graphics and images easily accounted for the bulk of network traffic. Today, streaming video even from a single source like Netflix holds top honors for pushing the capabilities of our broadband infrastructure! (It certainly holds true in my own household with my young son’s love of *Curious George*, *Bob the Builder*, *Dinosaur Train*, and other such shows.)

Incorporating some form of media is likely a central concern for most Windows Store apps. Simple ones, even, probably use at least a few graphics to brand the app and present an attractive UI, as we’ve already seen on a number of occasions. Many others, especially games, will certainly use graphics, video, and audio together. In the context of this book, all of this means using the `img`, `svg` (Scalable Vector Graphics), `canvas`, `audio`, and `video` elements of HTML5.

Of course, working with media goes well beyond just presentation because apps might also provide any of the following capabilities:

- Organize and edit media files, including those in the pictures, music, and videos media libraries.
- Transcode (convert) media files, possibly applying various filters and custom codecs.
- Organize and edit playlists.
- Capture audio and video from available devices.
- Stream media from a server to a device, or from a device to a PlayTo target, perhaps also applying DRM.

These capabilities, for which many WinRT APIs exist, along with the media elements of HTML5 and their particular capabilities within the Windows 8 environment, will be our focus for this chapter.

Note As is relevant to this chapter, a complete list of audio and video formats that are supported for WinRT apps can be found on [Supported audio and video formats](#).

Sidebar: Performance Tricks for Faster Apps

Some of the recommendations in this chapter come from a great talk by Jason Weber, the Performance Lead for Internet Explorer, called [50 Performance Tricks to Make Your Windows 8](#)

[Apps Using HTML5 Faster](#). While some of these tricks are specifically for web applications running in a browser, many of them are wholly applicable to Windows Store apps written in JavaScript as they run on top of the same infrastructure as Internet Explorer.

Creating Media Elements

Certainly the easiest means to incorporate media into an app is what we've already been doing for years: simply use the appropriate HTML element in your layout and *voila!* there you have it. With `img`, `audio`, and `video` elements, in fact, you're completely free to use content from just about any location. That is, the `src` attributes of these elements can be assigned URLs that point to in-package content (using relative paths, `ms-appx:///` URIs, or paths based on `Windows.ApplicationModel.Package.current.installedLocation` that you then pass to `URL.createObjectURL`), files in your app data folders (using `ms-appdata:///` URIs or paths based on `Windows.Storage.ApplicationData.current` again using `URL.createObjectURL`), and remote files with `http://` and other URIs. With the `img` element, this includes using SVG files as the source.

There are three ways to create a media element in a page or page control.

First is to include the element directly in declarative HTML. Here it's often useful to use the `preload="auto"` attribute for remote audio and video to increase the responsiveness of controls and other UI that depend on those elements. (Doing so isn't really important for local media files since they are, well, already local!) Oftentimes, media elements are placed near the top of the HTML file, in order of priority, so that downloading can begin while the rest of the document is being parsed.

On the flip side, if the user can wait a short time to start a video, use a preview image in place of the video and don't start the download until it's actually necessary. Code for this is shown later in this chapter in the "Video Playback and Deferred Loading" section.

Playback for a declarative element can be automatically started with the `autoplay` attribute, through the built-in UI if the element has the `controls` attribute, or by calling `<element>.play()` from JavaScript.

The second method is to create an HTML element in JavaScript via `document.createElement` and add it to the DOM with `<parent>.appendChild` and similar methods. Here's an example using media files in this chapter's companion content, though you'll need to drop the code into a new project of your own:

```
//Create elements and add to DOM, which will trigger layout
var picture = document.createElement("img");
picture.src = "media/wildflowers.jpg";
picture.width = 300;
picture.height = 450;
document.getElementById("divShow").appendChild(picture);

var movie = document.createElement("video");
movie.src = "media/ModelRocket1.mp4";
movie.autoplay = false;
```

```

movie.controls = true;
document.getElementById("divShow").appendChild(movie);

var sound = document.createElement("audio");
sound.src = "media/SpringyBoing.mp3";
sound.autoplay = true; //Play as soon as element is added to DOM
sound.controls = true; //If false, audio plays but does not affect layout

document.getElementById("divShow").appendChild(sound);

```

Unless otherwise hidden by styles, image and video elements, plus audio elements with the `controls` attribute, will trigger re-rendering of the document layout. An audio element *without* that attribute will not cause re-rendering. As with declarative HTML, setting `autoplay` to `true` will cause video and audio to start playing as soon as the element is added to the DOM.

Finally, for audio, apps can create an `Audio` object in JavaScript to play sounds or music without any effect on UI. More on this later. JavaScript also has the `Image` class, and the `Audio` class can be used to load video:

```

//Create objects (pre-loading), then set other DOM object sources accordingly
var picture = new Image(300, 450);
picture.src = "http://www.kraigbrockschmidt.com/downloads/media/wildflowers.jpg";
document.getElementById("image1").src = picture.src;

//Audio object can be used to pre-load (but not render) video
var movie = new Audio("http://www.kraigbrockschmidt.com/downloads/media/ModelRocket1.mp4");
document.getElementById("video1").src = movie.src;

var sound = new Audio("http://www.kraigbrockschmidt.com/downloads/media/SpringyBoing.mp3");
document.getElementById("audio1").src = sound.src;

```

Creating an `Image` or `Audio` object from code does not create elements in the DOM, which can be a useful trait. The `Image` object, for instance, has been used for years to preload an array of image sources for use with things like image rotators and popup menus. Preloading in this case only means that the images have been downloaded and cached. This way, assigning the same URI to the `src` attribute of an element that *is* in the DOM, as shown above, will have that image appear immediately. The same is true for preloading video and audio, but again, this is primarily helpful with remote media as files on the local file system will load relatively quickly as-is. Still, if you have large local images and want them to appear quickly when needed, preloading them into memory is a useful strategy.

Of course, you might want to load media only when it's needed, in which case the same type of code can be used with existing elements, or you can just create an element and add it to the DOM as shown earlier.

Graphics Elements: Img, Svg, and Canvas (and a Little CSS)

I know you're probably excited to get to sections of this chapter on video and audio, but we cannot forget that images have been the backbone of web applications since the beginning and remain a huge part of any app's user experience. Indeed, it's helpful to remember that video itself is conceptually just a series of static images sequenced over time! Fortunately, HTML5 has greatly expanded an app's ability to incorporate image data by adding SVG support and the `canvas` element to the tried-and-true `img` element. Furthermore, applying CSS animations and transitions (covered in detail in Chapter 11, "Purposeful Animations") to otherwise static image elements can make them appear very dynamic.

Speaking of CSS, it's worth noting that many graphical effects that once required the use of static images can be achieved with *just* CSS, especially CSS3:

- Borders, background colors, and background images
- Folder tabs, menus, and toolbars
- Rounded border corners, multiple backgrounds/borders, and image borders
- Transparency
- Embeddable fonts
- Box shadows
- Text shadows
- Gradients

In short, if you've ever used `img` elements to create small visual effects, create gradient backgrounds, use a nonstandard font, or provide some kind of graphical navigation structure, there's probably a way to do it in pure CSS. For details, see the great [overview of CSS3](#) by Smashing Magazine as well as the CSS specs at <http://w3.org/>. CSS also provides the ability to declaratively handle some events and visual states using pseudo-selectors of `hover`, `visited`, `active`, `focus`, `target`, `enabled`, `disabled`, and `checked`. For more, see <http://css-tricks.com/> as well as another Smashing Magazine [tutorial on pseudo-classes](#).

That said, let's review the three primary HTML5 elements for graphics:

- `img` is used for raster data. The PNG format generally preferred over other formats, especially for text and line art, though JPEG makes smaller files for photographs. GIF is generally considered outdated, as the primary scenarios where GIF produced a smaller file size can probably be achieved with CSS directly. Where scaling is concerned, Windows Store apps need to consider pixel density, as we saw in Chapter 6, "Layout," and provide separate image files for each scale the app might encounter. This is where the smaller size of JPEGs can reduce the overall size of your app package in the Windows Store.
- SVGs are best used for smooth scaling across display sizes and pixel densities. SVGs can be

declared inline, created dynamically in the DOM, or maintained as separate files and used as a source for an `img` element (in which case all the scaling characteristics are maintained). An `svg` file can also be used for an `iframe` source, which has the added benefit that the SVG's child elements are accessible in the DOM. As we saw in Chapter 6, preserving the aspect ratio of an SVG is often important, for which you employ the `viewBox` and `preserveAspectRatio` attributes of the `svg` tag.

- The `canvas` element provides a drawing surface and API for creating graphics with lines, rectangles, arcs, text, and so forth. The canvas ultimately generates raster data, which means that once created, a canvas scales like a bitmap. (An app, of course, will typically redraw a canvas with scaled coordinates when necessary to avoid pixelation.) The canvas is also very useful for performing pixel manipulation, even on individual frames of a video while it's playing.

Apps often use all three of these elements, drawing on their various strengths. I say this because when `canvas` first became available, developers seemed so enamored with it that they seemed to forget how to use `img` elements, and they ignored the fact that SVGs are often a better choice altogether! (And did I already say that CSS can accomplish a great deal by itself as well?)

In the end, it's helpful to think of all the HTML5 graphics elements as ultimately producing a bitmap that the app host simply renders to the display. You can, of course, programmatically animate the internal contents of these elements in JavaScript, as we'll see in Chapter 11, but for our purposes here it's helpful to simply think of these as essentially static.

What differs between the elements is how image data gets into the element to begin with. `Img` elements are loaded from a source file, `svg`'s are defined in markup, and `canvas` elements are filled through procedural code. But in the end, as demonstrated in Scenario 1 in the HTML Graphics example for this chapter and shown in Figure 10-1, each can produce identical results.

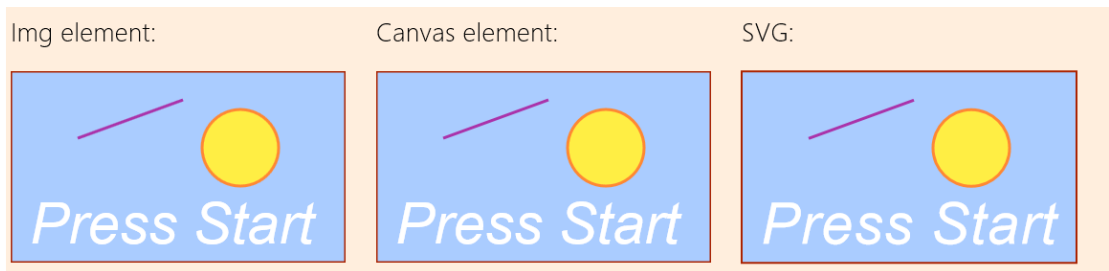


FIGURE 10-1 Image, canvas, and `svg` elements showing identical results.

In short, there are no fundamental differences as to what *can* be rendered through each type of element. However, they do have differences that become apparent when we begin to manipulate those elements as with CSS. Because each element is just a node in the DOM, plain and simple, and they are treated like all other nongraphic elements: CSS doesn't affect the internals of the element, just how it ultimately appears on the page. Individual parts of SVGs declared in markup can, in fact, be separately styled so long as they can be identified with a CSS selector. In any case, such styling only affects

presentation, so if new styles are applied, they are applied to the original contents of the element.

What's also true is that graphics elements can overlap with each other and with nongraphic elements (as well as video), and the rendering engine automatically manages transparency according to the [z-index](#) of those elements. Each graphic element can have clear or transparent areas, as is built into image formats like PNG. In a [canvas](#), any areas cleared with the [clearRect](#) method that aren't otherwise affected by other API calls will be transparent. Similarly, any area in an SVG's rectangle that's not affected by its individual parts will be transparent.

Scenario 2 in the HTML Graphics example allows you to toggle a few styles (with a check box) on the same elements shown earlier. In this case, I've left the background of the canvas element transparent so that we can see areas that show through. When the styles are applied, the [img](#) element gets rotated and transformed, the [canvas](#) gets scaled, and individual parts of the [svg](#) are styled with new colors, as shown in Figure 10-2.

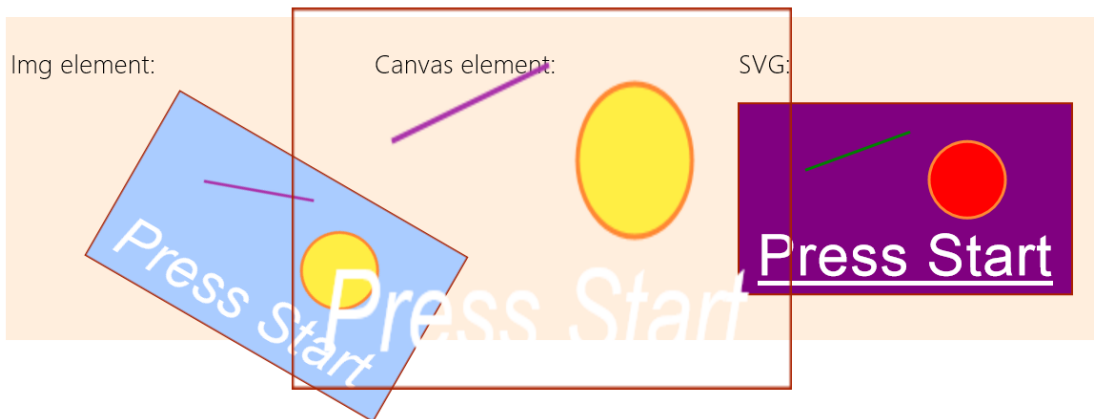


FIGURE 10-2 Styles applied to graphic elements; individual parts of the SVG can be styled if they are accessible through the DOM.

The styles in `css/scenario2.css` are simple:

```
.transformImage {  
  transform: rotate(30deg) translateX(120px);  
}  
  
.scaleCanvas {  
  transform: scale(1.5, 2);  
}
```

as is the code in `js/scenario2.js` that applies them:

```
function toggleStyles() {  
  var applyStyles = document.getElementById("check1").checked;  
  
  document.getElementById("image1").className = applyStyles ? "transformImage" : "";  
  document.getElementById("canvas1").className = applyStyles ? "scaleCanvas" : "";
```

```

document.getElementById("r").style.fill = applyStyles ? "purple" : "";
document.getElementById("l").style.stroke = applyStyles ? "green" : "";
document.getElementById("c").style.fill = applyStyles ? "red" : "";
document.getElementById("t").style.fontStyle = applyStyles ? "normal" : "";
document.getElementById("t").style.textDecoration = applyStyles ? "underline" : "";
}

```

The other thing you might have noticed when the styles are applied is that the scaled-up canvas looks rasterized, like a bitmap would typically be. This is expected behavior, as shown in the following table of scaling characteristics. These are demonstrated in Scenarios 3 and 4 of the HTML Graphics example.

Element	Scaling	Handling layout changes for best appearance
<code>img</code>	rasterized	Change <code>src</code> attribute for different scales (or just use an SVG file as a source).
<code>canvas</code>	rasterized	Redraw canvas using scaled dimensions; this is often best done by calling <code><context>.scale</code> according to the needed display size instead of changing the coordinates used in the code.
<code>svg</code>	smooth	Not needed. Use <code>viewBox</code> and <code>preserveAspectRatio</code> for proportional scaling.

Additional Characteristics of Graphics Elements

There are a few additional characteristics to be aware of with graphics elements. First, different kinds of operations will trigger a re-rendering of the element in the document. Second is the mode of operation of each element. Third are the relative strengths of each element. These are summarized in the following table:

Element	Trigger for re-rendering	Mode	Strengths
<code>img</code>	Change <code>src</code> attribute Change of styling via JavaScript	Pixel	Fast to render and transform Great for static elements and static/repeating backgrounds Sprite animation by changing <code>src</code> attribute
<code>canvas</code>	Calls to context API Change of styling via JavaScript Note: re-rendering happens only when code returns control to the host and unblocks the UI thread; there are no visible changes while the code is manipulating the canvas.	Immediate: API calls are rendered to pixels and forgotten	Fine-grained dynamic content Fast to render after being drawn Pixel-level manipulation Excellent for fine-grained dynamic/interactive content with frequent computation
<code>svg</code>	Change to element structure Change of styling via JavaScript	Retained: all shapes exist as DOM elements (unless used as <code>img src</code>)	Smooth scaling Fine-grained control over individual (retained) elements Shape-level manipulation Excellent for interactive graphics, detailed and scalable styling, and dynamic per-shape attributes

Sidebar: Using Media Queries to Show and Hide SVG Elements

Because SVGs generate elements in the DOM, those elements can be individually styled. You can use this fact with media queries to hide different parts of the SVG depending on its size. To do this, add different classes to those SVG elements. Then, in CSS, add or remove the `display: none` style for those classes within media queries like `@media (min-width:300px) and (max-width:499px)`. You may need to account for the size of the SVG relative to the app window, but it means that you can effectively remove detail from an SVG rather than allowing those parts to be rendered with too few pixels to be useful.

In the end, the reason why HTML5 has all three of these elements is because all three are really needed. All of them benefit from full hardware acceleration, just as they do in Internet Explorer, since apps written in HTML and JavaScript run on the same rendering engine as the browser.

The best practice in app design is to really explore the appropriate use of each type of elements. Each element can have transparent areas, so you can easily achieve some very fun effects. For example, if you have data that maps video timings to caption or other text, you can simply use an interval handler (with the interval set to the necessary granularity like a half-second) to take the video's `currentTime` property, retrieve the appropriate text for that segment, and render the text to an otherwise transparent canvas that sits on top of the video. Titles and credits can be done in a similar manner, eliminating the need to reencode the video.

Some Tips and Tricks

Working with the HTML graphics elements is generally straightforward, but knowing some details can help when working with them inside a Windows Store app.

Img Elements

- Use the `title` attribute of `img` for tooltips, not the `alt` attribute. You can also use a `WinJS.UI.Tooltip` control as described in Chapter 4, “Controls, Control Styling, and Data Binding.”
- To create an image from an in-memory stream, see [MSApp.createBlobFromRandom-AccessStream](#), the result of which can be then given to `URL.createObjectURL` to create an appropriate URI for a `src` attribute. We'll encounter this elsewhere in this chapter, and we'll need it when working with the Share contract in Chapter 12, “Contracts.” The same technique also works for audio and video streams.
- When loading images from `http://` or other remote sources, you run the risk of having the element show a red X placeholder image. To prevent this, catch the `img.onerror` event and supply your own placeholder:

```
var myImage = document.getElementById('image');  
myImage.onerror = function () { onImageError(this);}
```



```
function onImageError(source) {
  source.src = "placeholder.png";
  source.onerror = "";
}
```

Svg Elements

- `<script>` tags are not supported within `<svg>`.
- If you have an SVG file, you can load it into an `img` element by pointing at the file with the `src` attribute, but this doesn't let you traverse the SVG in the DOM. If you want the latter behavior, load the SVG in an `iframe` instead. The SVG contents will then be within that element's `contentDocument.documentElement` property:

```
<!-- in HTML-->
<iframe id="Mysvg" src="myFolder/mySVGFile.svg" />

// in JavaScript
var svg = document.getElementById("Mysvg").contentDocument.documentElement;
```

- PNGs and JPEGs generally perform better than SVGs, so if you don't technically need an SVG or have a high-performance scenario, consider using scaled raster graphics. Or you can dynamically create a scaled static image from an SVG so as to use the image for faster rendering later:

```
<!-- in HTML-->

<canvas id="canvas" style="display: none;" />

// in JavaScript
var c = document.getElementById("canvas").getContext("2d");
c.drawImage(document.getElementById("svg"),0,0);
var imageURLtoUse = document.getElementById("canvas").toDataURL();
```

- Two helpful SVG references (JavaScript examples): <http://www.carto.net/papers/svg/samples/> and <http://srufaculty.sru.edu/david.dailey/svg/>.

Canvas Elements

All the function names mentioned here are methods of a canvas's context object:

- Remember that a `canvas` element needs specific `width` and `height` attributes (in JavaScript, `canvas.width` and `canvas.height`), not styles. It does not accept px, em, %, or other units.
- Despite its name, the `closePath` method is *not* a direct complement to `beginPath`. `beginPath` is used to start a new path that can be stroked, clearing any previous path. `closePath`, on the other hand, simply connects the two endpoints of the current path, as if you did a `lineTo` between those points. It does *not* clear the path or start a new one. This seems to confuse programmers quite often, which is why you sometimes see a circle drawn with a line to the center!
- A call to `stroke` is necessary to render a path; until that time, think of paths as a pencil sketch of

something that's not been inked in. Note also that stroking implies a call to [beginPath](#).

- When animating on a canvas, doing [clearRect](#) on the entire canvas and redrawing every frame is generally easier to work with than clearing many small areas and redrawing individual parts of the canvas. The app host eventually has to render the entire canvas in its entirety with every frame anyway to manage transparency, so trying to optimize performance by clearing small rectangles isn't an effective strategy except when you're only doing a small number of API calls for each frame.
- Rendering canvas API calls is accomplished by converting them to the equivalent Direct2D calls in the GPU. This draws shapes with automatic antialiasing. As a result, drawing a shape like a circle in a color and drawing the same circle with the background color does *not* erase every pixel. To effectively erase a shape, use [clearRect](#) on an area that's slightly larger than the shape itself. This is one reason why clearing the entire canvas and redrawing every frame often ends up being easier.
- To set a background image in a canvas (so you don't have to draw each time), you can use the [canvas.style.backgroundImage](#) property with an appropriate URI to the image.
- Use the [msToBlob](#) method on a [canvas](#) object to obtain a [blob](#) for the canvas contents.
- When using [drawImage](#), you may need to wait for the source image to load using code such as

```
var img = new Image();
img.onload = function () { myContext.drawImage(myImg, 0, 0); }
myImg.src = "myImageFile.png";
```

- Although other graphics APIs see a circle as a special case of an ellipse (with x and y radii being the same), the canvas [arc](#) function works with circles only. Fortunately, a little use of scaling makes it easy to draw ellipses, as shown in the utility function below. Note that we use [save](#) and [restore](#) so that the [scale](#) call applies *only* to the [arc](#); it does not affect the [stroke](#) that's used from [main](#). This is important, because if the scaling factors are still in effect when you call [stroke](#), the line width will vary instead of remaining constant.

```
function arcEllipse(ctx, x, y, radiusX, radiusY, startAngle, endAngle, anticlockwise) {
    //Use the smaller radius as the basis and stretch the other
    var radius = Math.min(radiusX, radiusY);
    var scaleX = radiusX / radius;
    var scaleY = radiusY / radius;

    ctx.save();
    ctx.scale(scaleX, scaleY);

    //Note that centerpoint must take the scale into account
    ctx.arc(x / scaleX, y / scaleY, radius, startAngle, endAngle, anticlockwise);
    ctx.restore();
}
```

- By copying pixel data from a video, it's possible with the canvas to dynamically manipulate a video (without affecting the source, of course). This is a useful technique, even if it's processor-intensive; for this latter reason, though, it might not work well on low-power devices.

Here's an example of frame-by-frame video manipulation, the technique for which is nicely outlined in a Windows team blog post, [Canvas Direct Pixel Manipulation](#).⁵³ In the VideoEdit example for this chapter, default.html contains a `video` and `canvas` element in its main body:

```
<video id="video1" src="ModelRocket1.mp4" muted style="display: none"></video>
<canvas id="canvas1" width="640" height="480"></canvas>
```

In code (js/default.js), we call `startVideo` from within the activated handler. This function starts the video and uses `requestAnimationFrame` to do the pixel manipulation for every video frame:

```
var video1, canvas1, ctx;
var colorOffset = { red: 0, green: 1, blue: 2, alpha: 3 };

function startVideo() {
    video1 = document.getElementById("video1");
    canvas1 = document.getElementById("canvas1");
    ctx = canvas1.getContext("2d");

    video1.play();
    requestAnimationFrame(renderVideo);
}

function renderVideo() {
    //Copy a frame from the video to the canvas
    ctx.drawImage(video1, 0, 0, canvas1.width, canvas1.height);

    //Retrieve that frame as pixel data
    var imgData = ctx.getImageData(0, 0, canvas1.width, canvas1.height);
    var pixels = imgData.data;

    //Loop through the pixels, manipulate as needed
    var r, g, b;

    for (var i = 0; i < pixels.length; i += 4) {
        r = pixels[i + colorOffset.red];
        g = pixels[i + colorOffset.green];
        b = pixels[i + colorOffset.blue];

        //This creates a negative image
        pixels[i + colorOffset.red] = 255 - r;
        pixels[i + colorOffset.green] = 255 - g;
        pixels[i + colorOffset.blue] = 255 - b;
    }

    //Copy the manipulated pixels to the canvas
    ctx.putImageData(imgData, 0, 0);
}
```

⁵³ See also <http://beej.us/blog/2010/02/html5s-canvas-part-ii-pixel-manipulation/>.

```

    //Request the next frame
    requestAnimationFrame(renderVideo);
}

```

Here the page contains a hidden video element (`style="display: none"`) that is told to start playing once the document is loaded (`video1.play()`). In a `requestAnimationFrame` loop, the current frame of the video is copied to the canvas (`drawImage`) and the pixels for the frame are copied (`getImageData`) into the `imgData` buffer. We then go through that buffer and negate the color values, thereby producing a photographically negative image (an alternate formula to change to grayscale is also shown in the code comments, omitted above). We then copy those pixels back to the canvas (`putImageData`) so that when we return, those negated pixels are rendered to the display.

Again, this is processor-intensive as it's not generally a GPU-accelerated process and might perform poorly on lower-power devices (be sure, however, to run a Release build outside the debugger when evaluating erformance). It's much better to write a video effect DLL where possible as discussed in "Applying a Video Effect" later on. Nevertheless, it is a useful technique to know. What's really happening is that instead of drawing each frame with API calls, we're simply using the video as a data source. So we could, if we like, embellish the canvas in any other way we want before returning from the `renderVideo` function. An example of this that I really enjoy is shown in [Manipulating video using canvas](#) on Mozilla's developer site, which dynamically makes green-screen background pixels transparent so that an `img` element placed underneath the video shows through as a background. The same could even be used to layer two videos so that a background video is used instead of a static image. Again, be mindful of performance on low-power devices; you might consider providing a setting through which the user can disable such extra effects.

Video Playback and Deferred Loading

Let's now talk a little more about video playback itself. As we've already seen, simply including a `video` element in your HTML or creating an element on the fly, gives you playback ability. In the code below, the video is sourced from a local file, starts playing by itself, loops continually, and provides controls:

```
<video src="media/ModelRocket1.mp4" controls loop autoplay></video>
```

As we've been doing in this book, we're not going to rehash the details that are available in the W3C spec for the `video` and `audio` tags, found on <http://www.w3.org/TR/html5/video.html>. This spec will give you all the properties, methods, and events for these elements; especially note the event summary in [section 4.8.10.15](#), and that most of the properties and methods for both are found in [Media elements section 4.8.10](#). Note that the `track` element is supported for both `video` and `audio`; you can find an example of using it in Scenario 4 (demonstrating subtitles) of the [HTML media playback sample](#). We won't be covering it more here.

It's also helpful to understand that `video` and `audio` are closely related, since they're part of the same spec. In fact, if you want to just play the audio portion of a video, you can use the `Audio` object in JavaScript:

```
//Play just the audio of a video
var movieAudio = new Audio("http://www.kraigbrockschmidt.com/downloads/media/ModelRocket1.mp4");
movieAudio.load();
movieAudio.play();
```

For any given video element, you can set the width and height to control the playback size (as to 100% for full-screen). This is important when your app switches between view states, and you'll likely have CSS styles for video elements in your various media queries. Also, if you have a control to play full screen, simply make the video the size of the viewport (after also calling [Windows.UI.ViewManagement.ApplicationView.tryUnsnap](#) if you're in the snapped view). In addition, when you create a video element with the `controls` attribute, it will automatically have a full-screen control on the far right that does exactly what you expect within a Windows Store app:



In short, you don't need to do anything special to make this work. When the video is full screen, a similar button (or the ESC key) returns to the normal app view.

Note In case you're wondering, the audio and video elements don't provide any CSS pseudo-selectors for styling the controls bar. As my son's preschool teacher would say (in reference to handing out popsicles, but it works here too), "You get what you get and you don't throw a fit and you're happy with it." If you'd like to do something different with these controls, you'll need to turn off the defaults and provide controls of your own that would call the element methods appropriately.

When implementing your own controls, be sure to set a timeout to make the controls disappear (either hiding them or changing the z-index) when they're not being used. This is especially important if you have a full-screen button for video like the built-in controls, where you would basically resize the element to match the screen dimensions. When you do this, Windows will automatically detect this full-screen video state and do some performance optimizations, but not if any other element is front of the video. It's also a good idea to disable any animations you might be running and disable unnecessary background processes like web workers.

You can use the various events of the `video` element to know when the video is played and paused through the controls, among other things (though there is not an event for going full-screen), but you should also respond appropriately when hardware buttons for media control are used. For this purpose, listen for events coming from the [Windows.Media.MediaControl](#) object, such as `playpressed`, `pausepressed`, and so on. (These are WinRT object events, so call `removeEventListener` as needed.) Refer to the [Configure keys for media sample](#) for a demonstration, but adding the listeners generally looks like this:

```
mediaControl = Windows.Media.MediaControl;
mediaControl.addEventListener("soundlevelchanged", soundLevelChanged, false);
mediaControl.addEventListener("playpausetogglepressed", playpause, false);
mediaControl.addEventListener("playpressed", play, false);
mediaControl.addEventListener("stoppressed", stop, false);
mediaControl.addEventListener("pausepressed", pause, false);
```

I also mentioned that you might want to defer loading a video until it's needed and show a preview image in its place. This is accomplished with the `poster` attribute, whose value is the image to use:

```
<video id="video1" poster="media/rocket.png" width="640" height="480"></video>

var video1 = document.getElementById("video1");
var clickListener = video1.addEventListener("click", function () {
    video1.src = "http://www.kraigbrockschmidt.com/downloads/media/ModelRocket1.mp4";
    video1.load();

    //Remove listener to prevent interference with video controls
    video1.removeEventListener("click", clickListener);

    video1.addEventListener("click", function () {
        video1.controls = true;
        video1.play();
    });
});
```

In this case I'm not using `preload="true"` or even providing a `src` value so that nothing is transferred until the video is tapped. When a tap occurs, that listener is removed, the video's own controls are turned on, and playback is started. This, of course, is a more roundabout method; often you'll use `preload="true" controls src="..."` directly in the video element, as the `poster` attribute will handle the preview image.

Disabling Screen Savers and the Lock Screen During Playback

When playing video, especially full-screen, it's important to disable any automatic timeouts that would blank the display or lock the device. This is done through the [Windows.System.Display.Display-Request](#) object. Before starting playback, create an instance of this object and call its `requestActive` method.

```
var displayRequest = new Windows.System.Display.DisplayRequest();
displayRequest.requestActive();
```

If this call succeeds, you'll be guaranteed that the screen will stay active despite user inactivity. When the video is complete, be sure to call `requestRelease`. Note that Windows will automatically deactivate such requests when your app is moved to the background, and it will reactivate them when the user switches back.

Video Element Extension APIs

Beyond the HTML5 standards for [video](#) elements, some additional properties and methods are added to them in Windows 8, as shown in the following table and documented on the [video element](#) page. Also note the references to the [HTML media playback sample](#) where you can find some examples of using these.

Properties	Description
msHorizontalMirror	A Boolean that controls whether the playback is flipped horizontally. This is particularly useful when sourcing the video element from a camera to make sure the user sees the proper orientation. See the notes on the enclosureLocation property in “Selecting a Media Capture Device” later on.
msZoom	A Boolean that indicates whether to allow the video element to fit inside its display space by trimming the top/bottom or left/right (when true). This allows apps to give users control over videos whose aspect ratio differs from that of its given display area—that is, to remove letterboxing or sidepillars. For a demonstration, refer to Scenario 3 of the HTML media playback sample .
msIsLayoutOptimalForPlayback (onMSVideoOptimalLayoutChanged)	A Boolean that indicates whether a video will have the best playback based on its layout. When this changes the onMSVideoOptimalLayoutChanged event fires. For details, see How to optimize video rendering and Audio and Video Performance .
msIsStereo3D	A Boolean that indicates whether the system considers the video element’s source to be 3D (based on metadata in the video itself). Whether the <i>system</i> it itself capable can be determined through Windows.Graphics.Display.DisplayProperties.stereo-Enabled . Apps can also listen for Windows.Graphics.Display.DisplayProperties.-stereoEnabledChanged (a WinRT event) to know when the capabilities change. For details on this and other Stereo 3D concerns, refer to How to enable stereo video playback and Scenario 5 of the HTML media playback sample .
msStereo3DRenderMode	Can be mono (the default) or stereo so that apps can control playback. (See above for references.)
msStereo3DPackingMode	Can be none (2D default), topbottom , or sidebyside ; this is an adjustment available to apps when the video metadata doesn’t clearly indicate which orientation to use. (See above for references.)
msRealtime	Enables the media to reduce initial latency as much as possible for playback. This is important for two-way communication apps, for example, as well as gaming chat, but should be used carefully. For details, refer to How to enable low-latency playback and the Real-time communications sample .
msPlayToDisabled msPlayToPrimary msPlayToSource	Properties related to Windows’ PlayTo feature. See the “PlayTo” section at the end of this chapter. Note that these are available on img and audio elements as well.
msAudioTracks	An array of audio track descriptions to support additional languages or other tracks (e.g., commentary). Set msAudioTracks.selectedTrack to the desired index to change the playback audio. For details, refer to How to select audio tracks in different languages as well as Scenario 2 of the HTML media playback sample .
msAudioCategory	Identifies the kind of audio being played in the video; see “Playback Manager and Background Audio” later for the specific values. Note that setting this to “ Communications ” will also set the device type to “ Communications ” and force msRealtime to true .
msAudioDeviceType	Specified the output devices that audio will be sent to; see “Audio Element Extension APIs.”
Methods	Description
msFrameStep (onMSVideoFrameStepCompleted)	Steps the video by one frame forward or backward. The onMSVideoFrameStepCompleted event fires when the step is complete.

<code>msInsertVideoEffect</code> <code>msInsertAudioEffect</code> <code>msClearEffects</code>	Adds or removes effects during playback (see below). All are available on <code>video</code> ; <code>msInsertVideoEffect</code> is not available on <code>audio</code> elements.
<code>msSetMediaProtectionManager</code>	Used for DRM with both <code>audio</code> and <code>video</code> ; see “Streaming from a Server and Digital Rights Management (DRM)” toward the end of this chapter.
<code>msSetVideoRectangle</code>	Sets the dimension of a subrectangle within a video.
<code>onMSVideoFrameStepCompleted</code> (event)	Occurs when the video format changes.

The Source Attribute and Custom Codecs

Video (and audio) elements can use the HTML5 `source` attribute. In web applications, multiple source elements are used to provide alternate video formats in case a client system doesn’t have the necessary codec for the primary source. Given that the list of supported formats in Windows is well known (refer again to [Supported audio and video formats](#)), this isn’t much of a concern for Windows Store apps. However, `source` is still useful because it can identify the specific codecs for the source:

```
<video controls loop autoplay>
  <source src="video1.vp8" type="video/webm" />
</video>
```

This is important when you need to provide a custom codec for your app through `Windows.Media.MediaExtensionManager`, outlined in the “Custom Decoders/Encoders and Scheme Handlers” section later in this chapter, as the codec identifies the extension to load for decoding. I show WebM as an example here because it’s not directly available to Store apps (though it is in Internet Explorer). When the app host running a Store app encounters the `video` element above, it will look for a matching decoder for the specified `type`.

Applying a Video Effect

The earlier table shows that video elements have `msInsertVideoEffect` and `msInsertAudioEffect` methods on them. WinRT provides a built-in video stabilization effect that is easily applied to an element. This is demonstrated in Scenario 3 of the [Media extensions sample](#), which plays the same video with and without the effect, so the stabilized one is muted:

```
vidStab.msClearEffects();
vidStab.muted = true;
vidStab.msInsertVideoEffect(Windows.Media.VideoEffects.videoStabilization, true, null);
```

Custom effects, as demonstrated in Scenario 4 of the sample, are implemented as separate dynamic-link libraries (DLLs), typically written in C++ for best performance, and are included in the app package because a Store app can install a DLL only for its own use and not for systemwide access. With the sample you’ll find DLL projects for a grayscale, invert, and geometric effects, where the latter has three options for fisheye, pinch, and warp. In the `js/CustomEffect.js` file you can see how these are applied, with the first parameter to `msInsertVideoEffect` being a string that identifies the effect as exported by the DLL (see, for instance, the `InvertTransform.idl` file in the `InvertTransform` project):


```
vid.msInsertVideoEffect("GrayscaleTransform.GrayscaleEffect", true, null);

vid.msInsertVideoEffect("InvertTransform.InvertEffect", true, null);
```

The second parameter to `msInsertVideoEffect`, by the way, indicates whether the effect is required, so it's typically `true`. The third is a parameter called *config*, which just contains additional information to pass to the effect. In the case of the geometric effects in the sample, this parameter specifies the particular variation:

```
var effect = new Windows.Foundation.Collections.PropertySet();
effect["effect"] = effectName;
vid.msClearEffects();
vid.msInsertVideoEffect("PolarTransform.PolarEffect", true, effect);
```

where `effectName` will be either "Fisheye", "Pinch", or "Warp".

Audio effects, not shown in the sample, are applied the same way with `msInsertAudioEffect` (with the same parameters). Do note that each element can have at most two effects per media stream. A `video` element can have two video effects and two audio effects; an `audio` element can have two audio effects. If you try to add more, the methods will throw an exception. This is why it's a good idea to call `msClearEffects` before inserting any others.

For additional discussion on effects and other media extensions, see [Using media extensions](#).

Browsing Media Servers

Many households, including my own, have one or more media servers available on the local network from which apps can play media. Getting to these servers is the purpose of the one other property in `Windows.Storage.KnownFolders` that we haven't mentioned yet: `mediaServerDevices`. As with other known folders, this is simply a `StorageFolder` object through which you can then enumerate or query additional folders and files. In this case, if you call its `getFoldersAsync`, you'll receive back a list of available servers, each of which is represented by another `StorageFolder`. From there you can use file queries, as discussed in Chapter 8, "State, Settings, Libraries, and Documents," to search for the types of media you're interested in or apply user-provided search criteria. An example of this can be found in the [Media Server client sample](#).

Audio Playback and Mixing

As with video, the `audio` element provides its own playback abilities, including controls, looping, and autoplay:

```
<audio src="media/SpringyBoing.mp3" controls loop autoplay></audio>
```

Again, as described earlier, the same W3C spec applies to both `video` and `audio` elements. The same code to play just the audio portion of a video is exactly what we use to play an audio file:

```
var sound = new Audio("media/SpringyBoing.mp3");
sound1.msAudioCategory = "SoundEffect";
sound1.load(); //For pre-loading media
sound1.play(); //At any later time
```

As also mentioned before, creating an `Audio` object without controls and playing it has no effect on layout, so this is what's generally used for sound effects in games and other apps.

As with video, it's important for apps that do audio playback to respond appropriately to the events coming from the `Windows.Media.MediaControl` object, especially `playpressed`, `pausepressed`, `stoppressed`, and `playpausetogglepressed`. This lets the user control audio playback with hardware buttons, which you would use when playing music tracks, for instance. However, you would not apply these events to audio, such as game sounds.

Speaking of which, an interesting aspect of audio is how to mix multiple sounds together, as games generally require. Here it's important to understand that each `audio` element can be playing one sound: it only has one source file and one source file alone. However, multiple `audio` elements can be playing at the same time with automatic intermixing depending on their assigned categories. (See "Playback Manager and Background Audio" below.) In this example, some background music plays continually (`loop` is set to true, and the volume is halved) while another sound is played in response to taps (see the `AudioPlayback` example with this chapter's content):⁵⁴

```
var sound1 = new Audio("media/SpringyBoing.mp3");
sound1.load(); //For pre-loading media

//Background music
var sound2 = new Audio();
sound2.msAudioCategory = "ForegroundOnlyMedia"; //Set this before setting src
sound2.src = "http://www.kraigbrockschmidt.com/mp3/WhoIsSylvia_PortlandOR_5-06.mp3";
sound2.loop = true;
sound2.volume = 0.5; //50%;
sound2.play();

document.getElementById("btnSound").addEventListener("click", function () {
    //Reset position in case we're already playing
    sound1.currentTime = 0;
    sound1.play();
});
```

By loading the tap sound when the object is created, we know we can play it at any time. When initiating playback, it's a good idea to set the `currentTime` to 0 so that the sound always plays from the beginning.

The question with mixing, especially in games, really becomes how to manage many different sounds without knowing ahead of time how they will be combined. You may need, for instance, to overlap playback of the same sound with different starting times, but it's impractical to declare three audio

⁵⁴ And yes, I am playing the guitar and singing the lead part in this live recording, along with my friend Ted Cutler. The song, *Who is Sylvia?*, was composed by another friend, J. Donald Walters, using lyrics of Shakespeare.

elements with the same source. The technique that's emerged is to use "rotating channels" [as described on the Ajaxian website](#). To summarize:

1. Declare `audio` elements for each *sound* (with `preload="auto"`).
2. Create a pool (array) of `Audio` objects for however many simultaneous channels you need.
3. To play a sound:
 - a. Obtain an available `Audio` object from the pool.
 - b. Set its `src` attribute to one that matches a preloaded `audio element`.
 - c. Call that pool object's `play` method.

As sound designers in the movies have discovered, it is possible to have too much sound going on at the same time, because it gets really muddled. So you may not need more than a couple dozen channels at most.

Hint Need some sounds for your app? Check out <http://www.freesound.org>.

Audio Element Extension APIs

As with the `video` element, a few extensions are available on `audio` elements as well, namely those to do with effects (`msInsertAudioEffect`), DRM (`msSetMediaProtectionManager`), PlayTo (`msPlayToSource`, etc.), `msRealtime`, and `msAudioTracks`, as listed earlier in "Video Element Extension APIs." In fact, every extension API for `audio` exists on `video`, but two of them have primary importance for `audio`:

- `msAudioDeviceType` Allows an app to determine which output device audio will render to: "Console" (the default) and "Communications". This way an app that knows it's doing communication (like chat) doesn't interfere with media audio.
- `msAudioCategory` Identifies the type of audio being played (see table in the next section). This is very important for identifying whether audio should continue to play in the background (thereby preventing the app from being suspended), as described in the next section. Note that you should *always* set this property before setting the audio's `src` and that setting this to "Communications" will also set the device type to "Communications" and force `msRealtime` to `true`.

Do note that despite the similarities between the values in these properties, `msAudioDeviceType` is for selecting an output device whereas `msAudioCategory` identifies the *type* of audio that's being played through whatever device. A communications category audio could be playing through the console device, for instance, or a media category could be playing through the communications device. The two are separate concepts.

Playback Manager and Background Audio

To explore different kinds of audio playback, let's turn our attention to the [Playback Manager msAudioCategory sample](#). I won't show a screen shot of this because, doing nothing but audio, there isn't much to show! Instead, let me outline the behaviors of its different scenarios in the following table, as well as list those categories that aren't represented in the sample but that can be used in your own app. In each scenario you need to first select an audio file through the file picker.

Scenario	msAudioCategory	Description
1	BackgroundCapableMedia	Plays the selected audio when the app is both visible and in the background, including when the user is on the desktop, the Start screen, and the lock screen. The app will not be suspended when in the background, which you can confirm through Task Manager. This is typically used for playing local playlists, local or streaming media files, music videos, etc. Using this requires a declaration in the manifest and handlers for media control buttons.
2	Communications	Like BackgroundCapableMedia , this will also continue to play the selected audio when the app is in the background. Use this for peer-to-peer chat, VoIP, etc.
3	Other (the default for audio elements)	Plays the selected audio when the app is in the foreground, mixing with background audio; the audio is paused when the app is in the background.
4	ForegroundOnlyMedia	Plays the selected audio when the app is in the foreground; the audio is paused when the app is in the background. When audio of this category is played, background audio will be muted.
5	Alert	Plays the selected audio when the app is in the foreground and attenuates background audio. This is used for app notifications like ringtones as well as system alerts.
n/a	GameMedia	Used for ambient music in a game.
n/a	GameEffects	Used for game sound effects intended to mix with existing audio (all nonmusic sounds).
n/a	SoundEffects	Other sound effects (outside of games) intended to mix in with existing audio, such as brief dings, beeps, boinks, and blurps that indicate activity but don't otherwise qualify as alerts.

Where a single audio stream is concerned, there isn't always a lot of difference between some of these categories. Yet as the table indicates, different categories have different effects on other simultaneous audio streams. For this purpose, the Windows SDK does an odd thing by providing a second identical sample to the first, the [Playback Manager companion sample](#). This allows you run these apps at the same time (one in snapped view, the other in filled view, or one or both in the background) and play audio with different category settings to see how they combine.

How different audio streams combine is a subject that's discussed in the [Audio Playback in a Windows 8 App whitepaper](#). However, what's most important is that you assign the most appropriate category to any particular audio stream. These categories help the playback manager perform the right level of mixing between audio streams according to user expectations, both with multiple streams in the same app, and streams coming from multiple apps (with limits on how many background audio apps can be going at once). For example, users will expect that alarms, being an important form of notification, will temporarily attenuate other audio streams. Similarly, users will expect that an audio stream of a foreground app will take precedence over a stream of the same category of audio playing

in the background. As a developer, then, avoid playing games with the categories. Just assign the most appropriate category to your audio stream so that the playback manager can do its job with audio from all sources and deliver a consistent experience for the entire system.

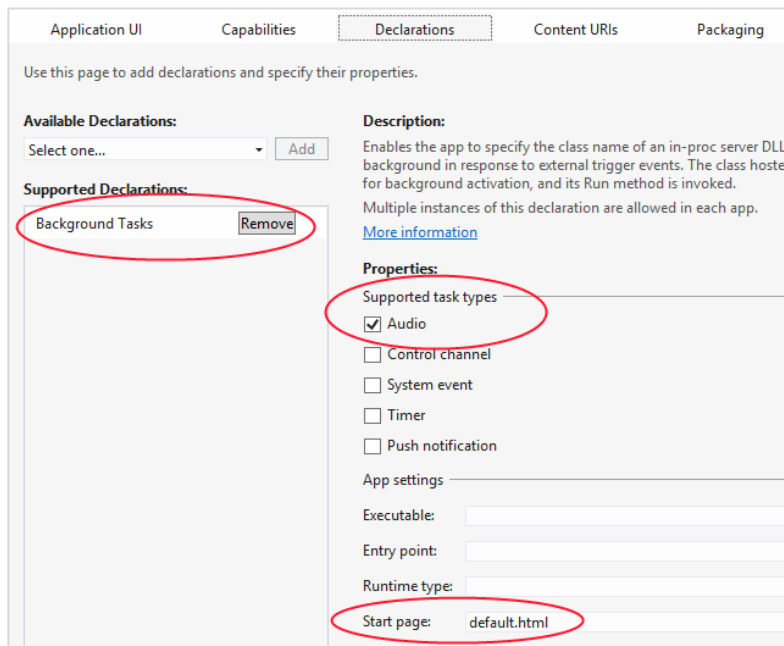
Setting an audio category for any given `audio` element is a simple matter of settings its `msAudio-Category` attribute. Every scenario in the sample does the same thing for this, making sure to set the category before setting the `src` attribute (shown here from `js/backgroundcapablemedia.js`):

```
audtag = document.createElement('audio');
audtag.setAttribute("msAudioCategory", "BackgroundCapableMedia");
audtag.setAttribute("src", fileLocation);
```

You could accomplish the same thing in markup, of course. Some examples:

```
<audio id="audio1" src="song.mp3" msAudioCategory="BackgroundCapableMedia"></audio>
<audio id="audio2" src="voip.mp3" msAudioCategory="Communications"></audio>
<audio id="audio3" src="lecture.mp3" msAudioCategory="Other"></audio>
```

With `BackgroundCapableMedia` and `Communications`, however, simply setting the category isn't sufficient: you also need to declare an audio background task extension in your manifest. This is easily accomplished by going to the Declarations tab in the manifest designer:



First, select Background Tasks from the Available Declarations drop-down list. Then check Audio under Supported Task Types, and identify a Start page under App Settings. The start page isn't really essential for background audio (because you'll never be launched for this purpose), but you need to provide something to make the manifest editor happy.

These declarations appear as follows in the manifest XML, should you care to look:

```
<Application Id="App" StartPage="default.html">
  <!-- ... -->
  <Extensions>
    <Extension Category="windows.backgroundTasks" StartPage="default.html">
      <BackgroundTasks>
        <Task Type="audio" />
      </BackgroundTasks>
    </Extension>
  </Extensions>
</Application>
```

Furthermore, background audio apps *must* also add listeners for the [Windows.Media.-MediaControl](#) events that we've already mentioned so that the user can control background audio playback through the media control UI (see the next section). They're also required because they make it possible for the playback manager to control the audio streams as the user switches between apps. If you fail to provide these listeners, your audio will always be paused and muted when the app goes into the background.

How to do this is shown in the Playback Manager sample for all its scenarios; the following is from `js/communications.js` (some code omitted):

```
mediaControl = Windows.Media.MediaControl;

mediaControl.addEventListener("soundlevelchanged", soundLevelChanged, false);
mediaControl.addEventListener("playpausetogglepressed", playpause, false);
mediaControl.addEventListener("playpressed", play, false);
mediaControl.addEventListener("stoppressed", stop, false);
mediaControl.addEventListener("pausepressed", pause, false);

// audtag variable is the global audio element for the page

function playpause() {
  if (!audtag.paused) {
    audtag.pause();
  } else {
    audtag.play();
  }
}

function play() {
  audtag.play();
}

function stop() {
  // Nothing to do here
}

function pause() {
  audtag.pause();
}
```

```

function soundLevelChanged() {
    //Catch SoundLevel notifications and determine SoundLevel state.
    //If it's muted, we'll pause the player.
    var soundLevel = Windows.Media.MediaControl.soundLevel;

    //No actions are shown here, but the options are spelled out to show the enumeration
    switch (soundLevel) {
        case Windows.Media.SoundLevel.muted:
            break;
        case Windows.Media.SoundLevel.low:
            break;
        case Windows.Media.SoundLevel.full:
            break;
    }

    appMuted();
}

function appMuted() {
    if (audtag) {
        if (!audtag.paused) {
            audtag.pause();
        }
    }
}

```

Technically speaking, a handler for `soundlevelchanged` is not required here, but the other four are. Such a minimum implementation is part of the `AudioPlayback` example with this chapter, where the code also uses the `MediaControl.isPlaying` flag to set the play/pause button in the media control UI (see next section).

A few additional notes about background audio:

- The reason for distinct `playpressed`, `pausepressed`, and `playpausepressed` events is to support a variety of hardware where some devices have separate play and pause buttons and others have a single button for both.
- If the audio is paused, a background audio app will be suspended like any other, but if the user presses a play button, the app will be resumed and audio will then continue playback.
- The use of background audio is carefully evaluated with apps submitted to the Windows Store. If you attempt to play an inaudible track as a means to avoid being suspended, the app will fail Store certification.
- A background audio app should be careful about how it uses the network for streaming media to support the low power state called connected standby. For details, refer to [Writing a power savvy background media app](#).

Now let's see the other important reason why you must implement the media control events: the UI that Windows displays in response to hardware buttons.

The Media Control UI

As mentioned in the previous section, providing handlers for the `MediaControl` object events is required for background audio so that the user can control the audio through hardware buttons (built into many devices, including keyboards and remote controls) without needing to switch to the app. This is especially important because background audio continues to play not only when the user switches to another app, but also when they switch to the Start screen switch to the desktop, or lock the device.

The default media control UI appears as shown in Figure 10-3 in the upper left of the screen, regardless of what is on the screen at the time. Tapping the app name will switch to the app.

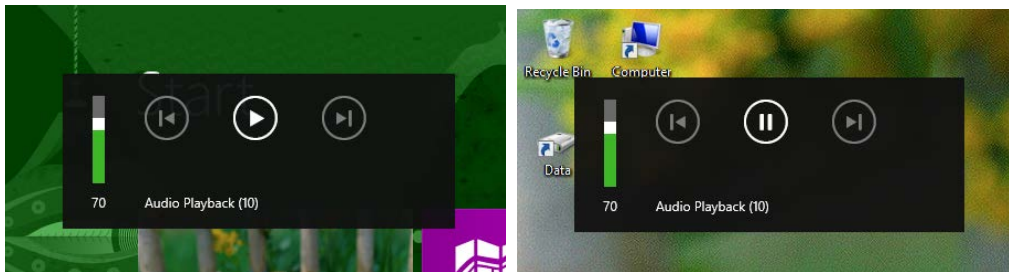
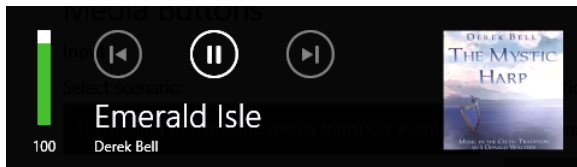


FIGURE 10-3 The media control UI appearing above the Start screen (left) and the desktop (right)

You can see in these images that the app's Display Name from the manifest is what's shown by default in the UI. Although this is an acceptable fallback, audio apps should ideally provide richer audio metadata to the `MediaControl`, specifically its `albumArt`, `trackName`, and `artistName` properties (the latter two of which must be less than 128 characters). This is done in the [Configure keys for media sample](#), which demonstrates how to obtain album art for a track, a subject we'll return to later.

With such metadata the media control UI will appear as follows; tapping the album art, track name or artist name will switch back to the audio app.



You'll notice in the images above that the forward and backward buttons are disabled. This is because the app does not have listeners for the `nexttrackpressed` and `previoustrackpressed` events of the `MediaControl` object; we'll see how to use these in the next section. There are other events as well, such as `channeldownpressed`, `channeluppressed`, `fastforwardpressed`, `rewindpressed`, `recordpressed`, and `stoppressed`, though these aren't represented in the media control UI.

Playing Sequential Audio

An app that's playing audio tracks (such as music, an audio book, or recorded lectures) will often have a list of tracks to play sequentially, especially while the app is running in the background. In this case it's important to start the next track quickly because Windows will otherwise suspend the app in 10 seconds after the current audio is finished. For this purpose, listen for the `audio` element's `ended` event and set `audio.src` to the next track. A good optimization in this case is to create a second `Audio` object and set its `src` attribute after the first track starts to play. This way that second track will be preloaded and ready to go immediately, thereby avoiding potential delays in playback between tracks. This is shown in the `AudioPlayback` example for this chapter, where I've split the one complete song into four segments for continuous playback. I've also shown here how to handle the next and previous button events, along with setting the segment number as the track name:

```
var mediaControl = Windows.Media.MediaControl;
var playlist = ["media/segment1.mp3", "media/segment2.mp3", "media/segment3.mp3",
    "media/segment4.mp3"];
var curSong = 0;
var audio1 = null;
var preload = null;

document.getElementById("btnSegments").addEventListener("click", playSegments);
audio1 = document.getElementById("audioSegments");
preload = document.createElement("audio");

function playSegments() {
    //Always reset WinRT object event listeners to prevent duplication and leaks
    mediaControl.removeEventListener("nexttrackpressed", nextHandler);
    mediaControl.removeEventListener("previoustrackpressed", prevHandler);

    curSong = 0;

    //Pause the other music
    document.getElementById("musicPlayback").pause();

    //Set up media control listeners
    setMediaControl(audio1);

    //Show the element (initially hidden) and start playback
    audio1.style.display = "";
    audio1.volume = 0.5; //50%;
    playCurrent();

    //Preload the next track in readiness for the switch
    var preload = document.createElement("audio");
    preload.setAttribute("preload", "auto");
    preload.src = playlist[1];

    //Switch to the next track as soon as one had ended or next button is pressed
    audio1.addEventListener("ended", nextHandler);
    mediaControl.addEventListener("nexttrackpressed", nextHandler);
}
```

```

function nextHandler () {
    curSong++;

    //Enable previous button if we have at least one previous track
    if (curSong > 0) {
        mediaControl.addEventListener("previoustrackpressed", prevHandler);
    }

    if (curSong < playlist.length) {
        //playlist[curSong] should already be loaded
        playCurrent();

        //Set up the next preload
        var nextTrack = curSong + 1;

        if (nextTrack < playlist.length) {
            preload.src = playlist[nextTrack];
        } else {
            preload.src = null;
            mediaControl.removeEventListener("nexttrackpressed", nextHandler);
        }
    }
}

function prevHandler() {
    //If we're already playing the last song, add the next button handler again
    if (curSong == playlist.length - 1) {
        mediaControl.addEventListener("nexttrackpressed", nextHandler);
    }

    curSong--;

    if (curSong == 0) {
        mediaControl.removeEventListener("previoustrackpressed", prevHandler);
    }

    playCurrent();
    preload.src = playlist[curSong + 1]; //This should always work
}

function playCurrent() {
    audio1.src = playlist[curSong];
    audio1.play();
    mediaControl.trackName = "Segment " + (curSong + 1);
}

```

When playing sequential tracks like this from an app written in JavaScript and HTML, you might notice brief gaps between the tracks, especially if the first track flows directly into the second. This is a present limitation of the platform given the layers that exist between the HTML `audio` element and the low-level XAudio2 APIs that are ultimately doing the real work. You can mitigate the effects to some extent—for example, you can crossfade the two tracks or crossfade a third overlay track that contains a little of the first and a little of the second track. You can also use a negative time offset to start playing the next track slightly before the previous one ends. But if want you a truly seamless transition, you'll

need to bypass the [audio](#) element and use the XAudio2 APIs from a WinRT component for direct playback. How to do this is discussed in the [Building your own Windows Runtime components to deliver great apps](#) post on the Windows 8 developer blog.

Playlists

The multisegment playback example in the previous section is clearly contrived because an app wouldn't typically have an in-memory playlist. More likely an app would load an existing playlist or create one from files that a user has selected.

WinRT supports these actions through a simple API in [Windows.Media.Playlists](#) namespace, using the WPL (Windows Media Player), ZPL (Zune), and M3U formats. The [Playlist sample](#) in the Windows SDK (which almost wins the prize for the *shortest* sample name!) shows how to perform various tasks with the API. In Scenario 1 it lets you choose multiple files by using the file picker, creates a new [Windows.Media.Playlists.Playlist](#) object, adds those files to its [files](#) list (a vector of [StorageFile](#) objects), and saves the playlist with its [saveAsAsync](#) method (this code from create.js is simplified and reformatted a bit):

```
function pickAudio() {
    var picker = new Windows.Storage.Pickers.FileOpenPicker();
    picker.suggestedStartLocation = Windows.Storage.Pickers.PickerLocationId.musicLibrary;
    picker.fileTypeFilter.replaceAll(SdkSample.audioExtensions);

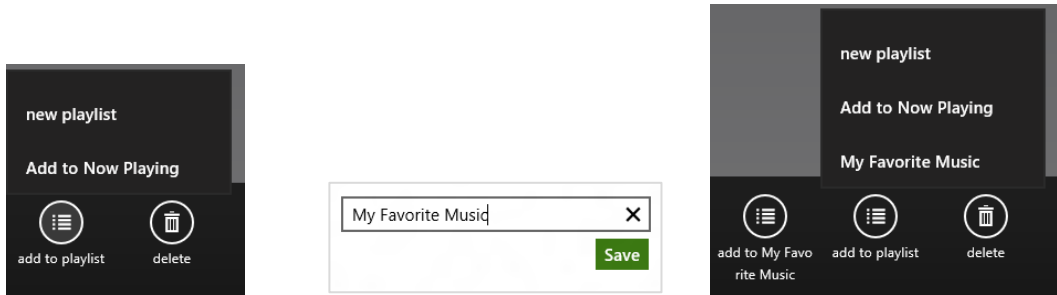
    picker.pickMultipleFilesAsync().done(function (files) {
        if (files.size > 0) {
            SdkSample.playlist = new Windows.Media.Playlists.Playlist();

            files.forEach(function (file) {
                SdkSample.playlist.files.append(file);
            });

            SdkSample.playlist.saveAsAsync(Windows.Storage.KnownFolders.musicLibrary,
                "Sample", Windows.Storage.NameCollisionOption.replaceExisting,
                Windows.Media.Playlists.PlaylistFormat.windowsMedia)
                .done();
        }
    })
}
```

Notice that [saveAsAsync](#) takes a [StorageFolder](#) and a name for the file (along with an optional format parameter). This accommodates a common use pattern for playlists where a music app has a single folder where it stores playlists and provides users with a simple means to name them and/or select them. In this way, playlists aren't typically managed like other user data files where one always goes through a file picker to do a Save As into an arbitrary folder. You could use [FileSavePicker](#), get a [StorageFile](#), and use its [path](#) property to get to the appropriate [StorageFolder](#), but more likely you'll save playlists in one place and present them as entities that appear only within the app itself.

For example, the Music app that comes with Windows 8 allows you create a new playlist when you're viewing tracks of some album. The following commands appear on the app bar (left), and when you select New Playlist, a flyout appears (middle) requesting the name, after which the flyout appears on the app bar (right):



The playlist then appears within the app as another album. In other words, though playlists might be saved in discrete files, they aren't necessarily presented that way to the user, and the API reflects that usage pattern.

Loading a playlist uses the `Playlist.loadAsync` method given a `StorageFile` for the playlist. This might be a `StorageFile` obtained from a file picker or from the enumeration of the app's private playlist folder. Scenario 2 of the Playlist sample (`display.js`) demonstrates the former, where it then goes through each file and requests their music properties:

```
function displayPlaylist() {
    var picker = new Windows.Storage.Pickers.FileOpenPicker();
    picker.suggestedStartLocation = Windows.Storage.Pickers.PickerLocationId.musicLibrary;
    picker.fileTypeFilter.replaceAll(SdkSample.playlistExtensions);

    var promiseCount = 0;

    picker.pickSingleFileAsync()
        .then(function (item) {
            if (item) {
                return Windows.Media.Playlists.Playlist.loadAsync(item);
            }
            return WinJS.Promise.wrapError("No file picked.");
        })
        .then(function (playlist) {
            SdkSample.playlist = playlist;
            var promises = {};

            // Request music properties for each file in the playlist.
            playlist.files.forEach(function (file) {
                promises[promiseCount++] = file.properties.getMusicPropertiesAsync();
            });

            // Print the music properties for each file. Due to the asynchronous
            // nature of the call to retrieve music properties, the data may appear
            // in an order different than the one specified in the original playlist.
        });
}
```

```

        // To guarantee the ordering we use Promise.join with an associative array
        // passed as a parameter, containing an index for each individual promise.
        return WinJS.Promise.join(promises);
    })
    .done(function (results) {
        var output = "Playlist content:\n\n";

        var musicProperties;
        for (var resultIndex = 0; resultIndex < promiseCount; resultIndex++) {
            musicProperties = results[resultIndex];
            output += "Title: " + musicProperties.title + "\n";
            output += "Album: " + musicProperties.album + "\n";
            output += "Artist: " + musicProperties.artist + "\n\n";
        }

        if (resultIndex === 0) {
            output += "(playlist is empty)";
        }

    }, function (error) {
        // ...
    });
}

```

We'll come back to working with these special properties in the next section, as the process also applies to other types of media.

The other method for managing a playlist is `Playlist.saveAsync`, which takes a single `StorageFile`. This is what you'd use if you've loaded and modified a playlist and simply want to save those changes (typically done automatically when the user adds or removes items from the playlist). This is demonstrated in Scenarios 3, 4, and 5 of the sample (add.js, js/remove.js, and js/clear.js), which just use methods of the `Playlist.files` vector like `append`, `removeAtEnd`, and `clear`, respectively.

Playback of a playlist depends, of course, on the type of media involved, but typically you'd load a playlist and sequentially take the next `StorageFile` object from its `files` vector, pass it to `URL.createObjectURL`, and then assign that URI to the `src` attribute of an `audio` or `video` element. You could also use playlists to manage lists of images for specific slide shows as well.

Loading and Manipulating Media

A user might store media files anywhere, but images, music, and videos are typically stored in the user's Pictures, Music, and Videos libraries specifically. Simply said, these are the folders that media apps should use by default until the user indicates otherwise through a folder picker. As we saw in Chapter 8, apps can declare programmatic access to the pictures, music, and videos libraries in their manifests and acquire the `StorageFolder` objects for these through `Windows.Storage.KnownFolders`:

```
var picLib = Windows.Storage.KnownFolders.picturesLibrary;
var musicLib = Windows.Storage.KnownFolders.musicLibrary;
var vidLib = Windows.Storage.KnownFolders.videosLibrary;
```

A photos app will typically declare the capability for the *Pictures Library* and display those contents in a *ListView*. A music and videos app will do the same for their respective libraries, as you can see in the built-in Photos, Music, and Videos apps in Windows 8. Remember too that if you forget to declare the appropriate capabilities, the lines of code above will throw access denied exceptions. You'll know right away if you forgot these important details.

I should warn you ahead of time that working with media can become very complicated and intricate. For that reason you'll probably find it helpful to refer to some of the topics in the documentation, such as [Processing image files](#), [Transcoding](#), and [Using media extensions](#).

Media File Metadata

With a *StorageFolder* in hand for some media library or subset thereof, you can use, as we also saw in Chapter 8, its *getItemsAsync* method to retrieve its contents. You can also use file queries to enumerate those files that match specific criteria. Whatever the case, you end up with a collection of *StorageFile* objects that you can work with however you want.

Now comes the interesting part. As I mentioned in Chapter 8, you can retrieve additional metadata for those files. This has a number of layers that you discover when you start opening some of the secrets doors of the *StorageFile* class, as illustrated in Figure 10-4. The following sections discuss these areas in turn.

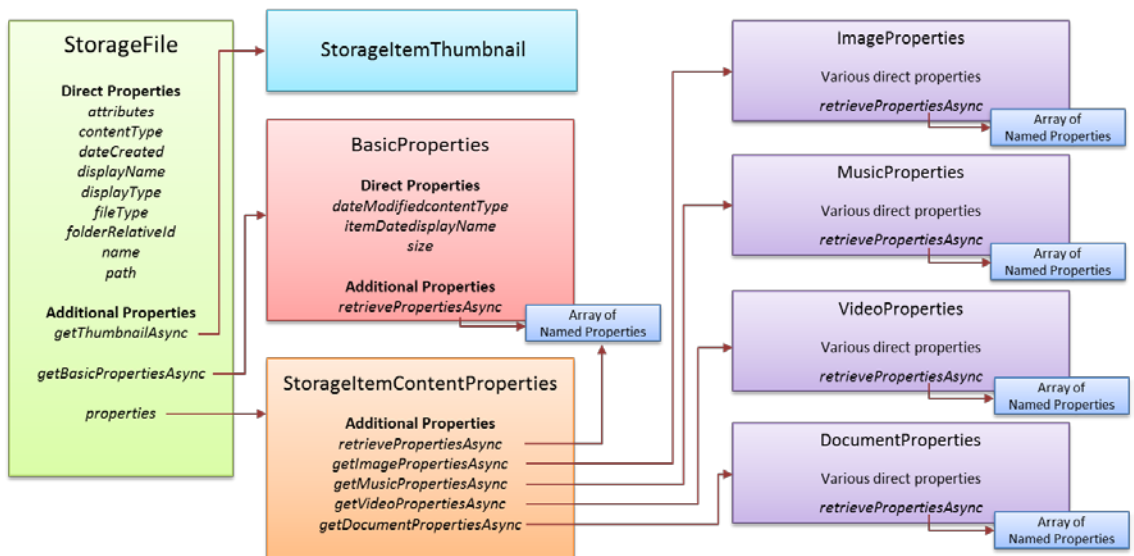


FIGURE 10-4 Relationships between the *StorageFile* object and others obtainable through it.

Thumbnails

First, `StorageFile.getThumbnailAsync` provides a thumbnail image appropriate for a particular “mode” from the [Windows.Storage.FileProperties.ThumbnailMode](#) enumeration. Options here are `picturesView`, `videosView`, `musicView`, `documentsView`, `listView`, and `singleItem`. What you receive in your completed handler is a [StorageItemThumbnail](#) object that provides thumbnail data as a stream. You can conveniently pass to our old friend `URL.createObjectURL` for display in an `img` element and whatnot.

Examples of this are found throughout the [File and folder thumbnail sample](#). Scenario 1, for instance (`js/scenario1.js`), obtains the thumbnail and displays it in an `img` element:

```
file.getThumbnailAsync(thumbnailMode, requestedSize, thumbnailOptions).done(function (thumbnail) {
    if (thumbnail) {
        outputResult(file, thumbnail, modeNames[modeSelected], requestedSize);
    }
    // ...
});

function outputResult(item, thumbnailImage, thumbnailMode, requestedSize) {
    document.getElementById("picture-thumb-imageHolder").src = URL.createObjectURL(thumbnailImage,
        { oneTimeOnly: true });
    // ...
}
```

Common File Properties

Common file properties—those that exist on all files—are found in a number of different places. Very common properties are found on the `StorageFile` object directly, like `attributes`, `contentType`, `dateCreated`, `displayName`, `displayType`, `fileType`, `name`, and `path`.

The next group is obtained through `StorageFile.getBasicPropertiesAsync`. This gives you a [Windows.Storage.FileProperties.BasicProperties](#) object that contains `dateModified`, `itemDate`, and `size` properties. “That’s a snoozer!” you’re saying to yourself. Well, this object also has an additional method called `retrievePropertiesAsync` that gives you an array of name-value pairs for all kinds of other stuff.

The trick to understand here is that you have to take an array of the property names you want and pass it to `retrievePropertiesAsync` where each name is a string that comes from a very extensive list of [Windows Properties](#), such as `System.FileOwner` and `System.FileAttributes`. An example of this is given in Scenario 5 of the [Folder enumeration sample](#) we saw in Chapter 8:

```
var dateAccessedProperty = "System.DateAccessed";
var fileOwnerProperty    = "System.FileOwner";

SdkSample.sampleFile.getBasicPropertiesAsync().then(function (basicProperties) {
    outputDiv.innerHTML += "Size: " + basicProperties.size + " bytes<br />";
    outputDiv.innerHTML += "Date modified: " + basicProperties.dateModified + "<br />";

    // Get extra properties
```

```

        return SdkSample.sampleFile.properties.retrievePropertiesAsync([fileOwnerProperty,
            dateAccessedProperty]);
    }).done(function (extraProperties) {
        var propValue = extraProperties[dateAccessedProperty];
        if (propValue !== null) {
            outputDiv.innerHTML += "Date accessed: " + propValue + "<br />";
        }
        propValue = extraProperties[fileOwnerProperty];
        if (propValue !== null) {
            outputDiv.innerHTML += "File owner: " + propValue;
        }
    });
});

```

What's very useful about this is that you can get to just about any property you want (the list of properties has hundreds of options) and then modify the array and call `BasicProperties.savePropertiesAsync`. Voila! You've just updated those properties on the file. A variation of `savePropertiesAsync` also lets you pass a specific array of name-value pairs if you only want to change specific ones.

The third set of properties is found by going through the secret door of `StorageFile.properties`. This contains a `StorageItemContentProperties` object whose `retrievePropertiesAsync` and `savePropertiesAsync` methods are like those we just saw for `BasicProperties`. What's more interesting is that it also has four other methods—`getDocumentPropertiesAsync`, `getImagePropertiesAsync`, `getMusicPropertiesAsync`, and `getVideoPropertiesAsync`—which are how you get to the really specific stuff for individual file types, as we'll see next.

Media-Specific Properties

Alongside the `BasicProperties` class in the `Windows.Storage.FileProperties` namespace we also find those returned by the `StorageFile.properties.get*PropertiesAsync` methods: `ImageProperties`, `VideoProperties`, `MusicProperties`, and `DocumentProperties`. Though we've had to dig deep to find these, they each contain deeper treasure troves of information—and I do mean deep! The tables below summarize each of these in turn. Note that each object type contains a `retrievePropertiesAsync` method, like that of `BasicProperties`, that lets you request additional properties by name that aren't already included in the main properties object. Refer to the links at the top of the table for the references that identify the most relevant Windows properties.

ImageProperties	from <code>StorageFile.properties.getImagePropertiesAsync</code>	
Additional properties	System.Image , System.Photo , System.Media	
Property	DataType	Applicable Windows Property
<code>title</code>	String	<code>System.Title</code>
<code>dateTaken</code>	Date	<code>System.Photo.DateTaken</code>
<code>latitude</code>	Double (see below)	<code>System.GPS.LatitudeDecimal</code> , or combination of <code>System.GPS.Latitude</code> , <code>System.GPS.LatitudeDenominator</code> , <code>System.GPS.LatitudeNumerator</code> , and <code>System.GPS.LatitudeRef</code>
<code>longitude</code>	Double (see below)	<code>System.GPS.LongitudeDecimal</code> ,

		or combination of System.GPS.Longitude, System.GPS.LongitudeDenominator, System.GPS.LongitudeNumerator, and System.GPS.LongitudeRef
cameraManufacturer	String	System.Photo.CameraManufacturer
cameraModel	String	System.Photo.CameraModel
width	Number in pixels	System.Image.HorizontalSize
height	Number in pixels	System.Image.VerticalSize
orientation	Windows.Storage.FileProperties.- PhotoOrientation containing unspecified, normal, flipHorizontal, flipVertical, transpose, transverse, rotate90, rotate180, rotate270	System.Photo.Orientation
peopleNames	String vector	System.Photo.PeopleNames
keywords	String vector	System.Keywords
rating	Number (1-99 with 0 meaning "no rating")	System.Rating

VideoProperties	from StorageFile.properties.getVideoPropertiesAsync	
Additional properties	System.Video , System.Media , System.Image , System.Photo	
Property	DataType	Applicable Windows Property
title	String	System.Title
subtitle	String	System.Media.SubTitle
year	Number	System.Media.Year
publisher	String	System.Media.Publisher
rating	Number	System.Rating
width	Number in pixels	System.Video.FrameWidth
height	Number in pixels	System.Video.FrameHeight
orientation	Windows.Storage.FileProperties.- VideoOrientation containing normal, rotate90, rotate180, rotate270	System.Photo.Orientation
duration	Number (in 100ns units, i.e. 1/10 th milliseconds)	System.Media.Duration
bitrate	Number (in bits/second)	System.Video.TotalBitrate, System.Video.EncodingBitrate
directors	String vector	System.Video.Director
producers	String vector	System.Media.Producer
writers	String vector	System.Media.Writer
keywords	String vector	System.Keywords
latitude	Double (see below)	System.GPS.LatitudeDecimal, or combination of System.GPS.Latitude, System.GPS.LatitudeDenominator, System.GPS.LatitudeNumerator, and System.GPS.LatitudeRef
longitude	Double (see below)	System.GPS.LongitudeDecimal, or combination of System.GPS.Longitude, System.GPS.LongitudeDenominator, System.GPS.LongitudeNumerator, and System.GPS.LongitudeRef

MusicProperties	from StorageFile.properties.getMusicPropertiesAsync	
Additional properties	System.Music , System.Media	
Property	DataType	Applicable Windows Property

title	String	System.Title, System.Music.AlbumTitle
subtitle	String	System.Media.SubTitle
trackNumber	Number	System.Music.TrackNumber
year	Number	System.Media.Year
publisher	String	System.Media.Publisher
artist	String	System.Music.Artist, System.Music.DisplayArtist
albumArtist	String	System.Music.DisplayArtist (read), System.Music.AlbumArtist (write)
genre	String vector	System.Music.Genre
composers	String vector	System.Music.Composer
conductors	String vector	System.Music.Conductor
rating	Number (1-99 with 0 meaning "no rating")	System.Rating
duration	Number (in 100ns units, i.e. 1/10 th milliseconds)	System.Media.Duration
bitrate	Number (in bits/second)	System.Video.TotalBitrate, System.Video.EncodingBitrate
producers	String vector	System.Media.Producer
writers	String vector	System.Media.Writer

DocumentProperties	from <code>StorageFile.properties.getDocumentPropertiesAsync</code>	
Additional properties	System	
Property	Data Type	Applicable Windows Property
title	String	System.Title
Author	String vector	System.Author
keywords	String vector	System.Keywords
Comments	String	System.Comment

Two notes about all this. First, the string vectors are, as we've seen before, instances of [IVector](#) that provide manipulation methods like [append](#), [insertAt](#), [removeAt](#), and so forth. In JavaScript you can access members of the vector like an array with [`i`]; just remember that the available methods are more specific.

Second, the `latitude` and `longitude` properties for images and video are `double` types but contain degrees, minutes, seconds, and a directional reference. The [Simple imaging sample](#) (in js/default.js) contains a helper function to extract the components of these values and convert them into a string:

```
"convertLatLongToString": function (latLong, isLatitude) {
    var reference;

    if (isLatitude) {
        reference = (latLong >= 0) ? "N" : "S";
    } else {
        reference = (latLong >= 0) ? "E" : "W";
    }

    latLong = Math.abs(latLong);
    var degrees = Math.floor(latLong);
    var minutes = Math.floor((latLong - degrees) * 60);
    var seconds = ((latLong - degrees - minutes / 60) * 3600).toFixed(2);
```

```

    return degrees + "°" + minutes + "'" + seconds + "\"" + reference;
}

```

To summarize, the sign of the value indicates direction. A positive value for latitude means North, negative means South; for longitude, positive means East, negative means West. The whole number portion of the value provides the degrees, and the fractional part contains the number of minutes expressed in base 60. Multiplying this value by 60 gives the whole minutes, with the remainder then containing the seconds. It's odd, but that's the kind of raw data you get from a GPS device that geolocation APIs normally convert for you directly.

Media Properties in the Samples

A few of the samples in the Windows SDK show you how to work with some of the properties described in the last section and how to work with those properties more generally. The [Simple imaging sample](#), in Scenario 1 (js/scenario1.js), provides the most complete demonstration because you can choose an image file and it will load and display various properties, as shown in Figure 10-5 (I've scrolled down to see all the properties). I can verify that the date, camera make/model, and exposure information are all accurate.

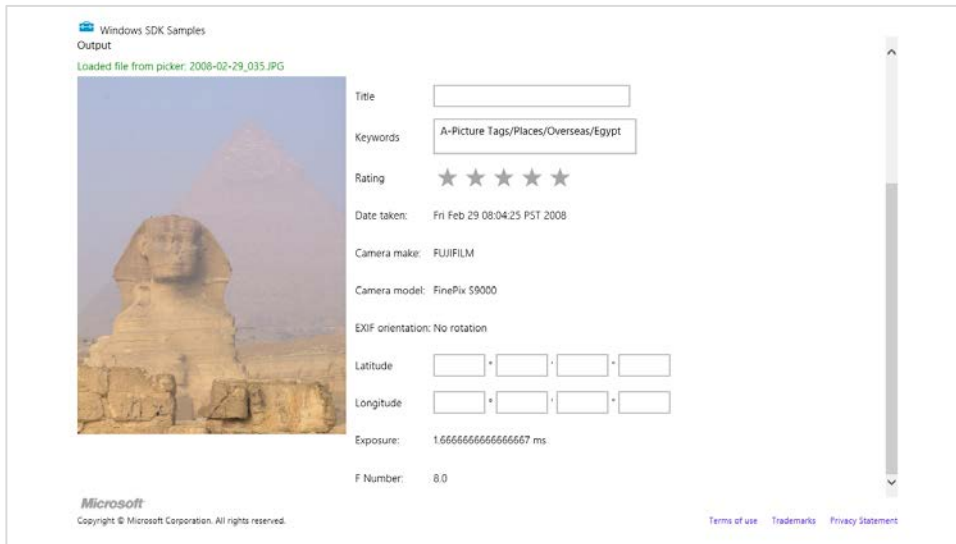


FIGURE 10-5 Image file properties in the Simple imaging sample.

The sample's `openHandler` method is what retrieves these properties from the file, specifically showing a call to `StorageFile.properties.getImagePropertiesAsync` and the use of `ImageProperties.retrievePropertiesAsync` for a couple of additional properties not already in `ImageProperties`. Then `getImagePropertiesForDisplay` coalesces these into a single object used by the sample's UI. Some lines are omitted in the code shown here:

```

var ImageProperties = {};

function openHandler() {
    // Keep data in-scope across multiple asynchronous methods.
    var file = {};

    Helpers.getFileFromOpenPickerAsync().then(function (_file) {
        file = _file;
        return file.properties.getImagePropertiesAsync();
    }).then(function (imageProps) {
        ImageProperties = imageProps;

        var requests = [
            "System.Photo.ExposureTime",    // In seconds
            "System.Photo.FNumber"          // F-stop values defined by EXIF spec
        ];

        return ImageProperties.retrievePropertiesAsync(requests);
    }).done(function (retrievedProps) {
        // Format the properties into text to display in the UI.
        displayImageUI(file, getImagePropertiesForDisplay(retrievedProps));
    });
}

function getImagePropertiesForDisplay(retrievedProps) {
    // If the specified property doesn't exist, its value will be null.
    var orientationText = Helpers.getOrientationString(ImageProperties.orientation);

    var exposureText = retrievedProps.lookup("System.Photo.ExposureTime") ?
        retrievedProps.lookup("System.Photo.ExposureTime") * 1000 + " ms" : "";

    var fNumberText = retrievedProps.lookup("System.Photo.FNumber") ?
        retrievedProps.lookup("System.Photo.FNumber").toFixed(1) : "";

    // Omitted: Code to convert ImageProperties.latitude and ImageProperties.longitude to
    // degrees, minutes, seconds, and direction

    return {
        "title": ImageProperties.title,
        "keywords": ImageProperties.keywords, // array of strings
        "rating": ImageProperties.rating, // number
        "dateTaken": ImageProperties.dateTaken,
        "make": ImageProperties.cameraManufacturer,
        "model": ImageProperties.cameraModel,
        "orientation": orientationText,
        // Omitted: lat/long properties
        "exposure": exposureText,
        "fNumber": fNumberText
    };
}

```

Most of the `displayImageUI` function to which these properties are passed just copies the data into various controls. It's good to note again, though, that displaying the picture itself is easily accomplished with our good friend, `URL.createObjectURL`:

```
function displayImageUI(file, propertyText) {
    id("outputImage").src = window.URL.createObjectURL(file, { oneTimeOnly: true });
```

For `MusicProperties` a small example can be found in the [Playlist sample](#), as we already saw earlier in "Playlists." You might go back now and look at the code listed in that section, as you should be able to understand what's going on. And while the SDK lacks samples that use `VideoProperties` and `DocumentProperties`, working with these follows the same pattern as shown above for `ImageProperties`, so it should be straightforward to write the necessary code.

Also take a look again at the [Configure keys for media sample](#), as we saw earlier in "The Media Control UI." It shows how to use the music properties to obtain album art.

As for saving properties, the Simple Imaging sample delivers there as well, also in Scenario 1. As the fields shown earlier in Figure 10-5 are editable, the sample provides an Apply button that invokes the `applyHandler` function below to write them back to the file:

```
function applyHandler() {
    ImageProperties.title = id("propertiesTitle").value;

    // Keywords are stored as an array of strings. Split the textarea text by newlines.
    ImageProperties.keywords.clear();
    if (id("propertiesKeywords").value != "") {
        var keywordsArray = id("propertiesKeywords").value.split("\n");

        keywordsArray.forEach(function (keyword) {
            ImageProperties.keywords.append(keyword);
        });
    }

    var properties = new Windows.Foundation.Collections.PropertySet();

    // When writing the rating, use the "System.Rating" property key.
    // ImageProperties.rating does not handle setting the value to 0 (no stars/unrated).
    properties.insert("System.Rating", Helpers.convertStarsToSystemRating(
        id("propertiesRatingControl").winControl.userRating
    ));

    // Code omitted: convert discrete latitude/longitude values from the UI into the
    // appropriate forms needed for the properties, and do some validation; the end result
    // is to store these in the properties list
    properties.insert("System.GPS.LatitudeRef", latitudeRef);
    properties.insert("System.GPS.LongitudeRef", longitudeRef);
    properties.insert("System.GPS.LatitudeNumerator", latNum);
    properties.insert("System.GPS.LongitudeNumerator", longNum);
    properties.insert("System.GPS.LatitudeDenominator", latDen);
    properties.insert("System.GPS.LongitudeDenominator", longDen);

    // Write the properties array to the file
```

```

ImageProperties.savePropertiesAsync(properties).done(function () {
    // ...
}, function (error) {
    // Some error handling as some properties may not be supported by all image formats.
});
}

```

A few noteworthy features of this code include the following:

- It separates keywords in the UI control and separately appends each to the `keywords` property vector.
- It creates a new collection of properties of type `Windows.Foundation.Collections.PropertySet` and uses its `insert` method to add properties to the list. This property set is what's expected by the `savePropertiesAsync` method.
- The `Helpers.convertStartsToSystemRating` method (see `js/default.js`) converts between 1–5 stars, as used in the `WinJS.UI.Rating` control, to the `System.Rating` value that uses a 1–99 range. The documentation for [System.Rating](#) specifically indicates this mapping.

In general, all the detailed information you want for any particular Windows property can be found on the reference page for that property. Again start at the [Windows Properties](#) and drill down from there.

Image Manipulation and Encoding

To do something more with an image than just loading and displaying it (where again you can apply various CSS transforms for effect), you need to get to the actual pixels by means of a *decoder*. This already happens under the covers when you assign a URI to an `img.src`, but to have direct access to pixels means decoding manually. On the flip side, saving pixels back out to an image file means using an encoder.

WinRT provides APIs for both in the [Windows.Graphics.Imaging](#) namespace, namely in the `BitmapDecoder`, `BitmapTransform`, and `BitmapEncoder` classes. Loading, manipulating, and saving an image file often involves these three classes in turn, though the `BitmapTransform` object is focused on rotation and scaling so you won't use it if you're doing other manipulations.

One demonstration of this API can be found in Scenario 2 of the [Simple imaging sample](#). I'll leave it to you to look at the code directly, however, because it gets fairly involved—up to 11 chained promises to save a file! It also does all decoding, manipulation, and encoding within a single function such as `saveHandler` (`js/scenario2.js`). Here's the process it follows:

- Open a file with `StorageFile.openAsync`, which provides a stream.
- Pass that stream to the static method `BitmapDecoder.createAsync` which provides a specific instance of `BitmapDecoder` for the stream.
- Pass that decoder to the static method `BitmapEncoder.createForTranscodingAsync`, which

provides a specific `BitmapEncoder` instance. This encoder is created with an `InMemoryRandomAccessStream`.

- Set properties in the encoder's `bitmapTransform` property (a `BitmapTransform` object) to configure scaling and rotation. This creates the transformed graphic in the in-memory stream.
- Create a property set (`Windows.Graphics.Imaging.BitmapPropertySet`) that includes `System.Photo.Orientation` and use the encoder's `bitmapProperties.setPropertiesAsync` to save it.
- Copy the in-memory stream to the output file stream by using `Windows.Storage.Stream.RandomAccessStream.copyAsync`.
- Close both streams with their respective `close` methods (this is what closes the file).

As comprehensive as this scenario is, it's helpful to look at different stages of the process separately, for which purpose we have the ImageManipulation example in this chapter's companion content. This lets you pick and load an image, convert it to grayscale, and save that converted image to a new file. Its output is shown in Figure 10-6. It also gives us an opportunity to see how we can send decoded image data to an HTML `canvas` element and save that canvas's contents to a file.

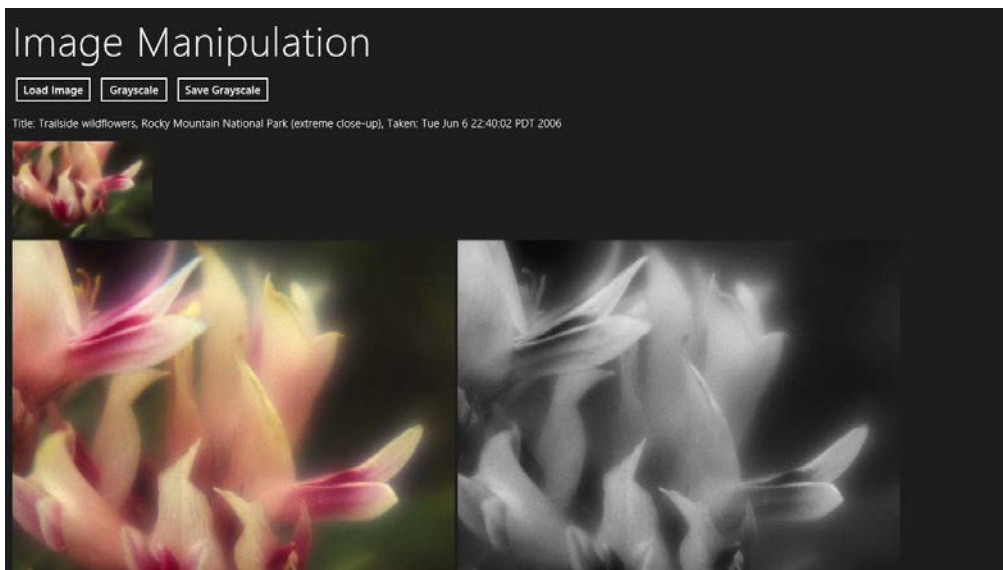


FIGURE 10-6 Output of the ImageManipulation example in the chapter's companion content.

The handler for the Load Image button (`loadImage` in `js/default.js`) provides the initial display. It lets you select an image with the file picker, displays the full-size image in an `img` element with `URL.createObjectURL`, calls `StorageFile.properties.getImagePropertiesAsync` to retrieve the `title` and `dateTaken` properties, and uses `StorageFile.getThumbnailAsync` to provide the thumbnail at the top. We've seen all of these APIs in action already.

When we click Grayscale we enter the `setGrayscale` handler where the interesting work happens. We call `StorageFile.openReadAsync` to get a stream, call `BitmapDecoder.createAsync` with that to obtain a decoder, cache some details from the decoder in a local object (`encoding`), and call `BitmapDecoder.-getPixelDataAsync` and copy those pixels to a canvas (and only three chained async operations here!):

```
var Imaging = Windows.Graphics.Imaging; //Shortcut
var imageFile; //Saved from the file picker
var decoder; //Saved from BitmapDecoder.createAsync
var encoding = {}; //To cache some details from the decoder

function setGrayscale() {
    //Decode the image file into pixel data for a canvas

    //Get an input stream for the file (StorageFile object saved from opening)
    imageFile.openReadAsync().then(function (stream) {
        //Create a decoder using static createAsync method and the file stream
        return Imaging.BitmapDecoder.createAsync(stream);
    }).then(function (decoderArg) {
        decoder = decoderArg;

        //Configure the decoder if desired. Default is BitmapPixelFormat.rgba8 and
        //BitmapAlphaMode.ignore. The parameterized version of getPixelDataAsync can also
        //control transform, ExifOrientationMode, and ColorManagementMode if needed.

        //Cache these settings for encoding later
        encoding.dpiX = decoder.dpiX;
        encoding.dpiY = decoder.dpiY;
        encoding.pixelFormat = decoder.bitmapPixelFormat;
        encoding.alphaMode = decoder.bitmapAlphaMode;
        encoding.width = decoder.pixelWidth;
        encoding.height = decoder.pixelHeight;

        return decoder.getPixelDataAsync();
    }).done(function (pixelProvider) {
        //detachPixelData gets the actual bits (array can't be returned from
        //an async operation)
        copyGrayscaleToCanvas(pixelProvider.detachPixelData(),
            decoder.pixelWidth, decoder.pixelHeight);
    });
}
```

The decoder's `getPixelDataAsync` method comes in two forms. The simple form, shown here, decodes using defaults. The full-control version lets you specify other parameters, as explained in the code comments above. A common use of this is doing a transform using a `Windows.Graphics.Imaging.BitmapTransform` object (as mentioned before), which accommodates scaling (with different interpolation modes), rotation (90-degree increments), cropping, and flipping.

Either way, what you get back from the `getPixelDataAsync` is not the actual pixel array, because of a limitation in the WinRT language projection mechanism whereby an asynchronous operation cannot return an array. Instead, the operation returns a `PixelDataProvider` object whose singular super-exciting synchronous method called `detachPixelData` gives you the array you want. (And that

method can be called only once and will fail on subsequent calls, hence the “detach” name.) In the end, though, what we have is exactly the data we need to manipulate the pixels and display the result on a canvas, as the [copyGrayscaleToCanvas](#) function demonstrates. You can, of course, replace this kind of function with any other manipulation routine:

```
function copyGrayscaleToCanvas(pixels, width, height) {
    //Set up the canvas context and get its pixel array
    var canvas = document.getElementById("canvas1");
    canvas.width = width;
    canvas.height = height;
    var ctx = canvas.getContext("2d");

    //Loop through and copy pixel values into the canvas after converting to grayscale
    var imgData = ctx.createImageData(canvas.width, canvas.height);
    var colorOffset = { red: 0, green: 1, blue: 2, alpha: 3 };
    var r, g, b, gray;
    var data = imgData.data; //Makes a huge perf difference!

    for (var i = 0; i < pixels.length; i += 4) {
        r = pixels[i + colorOffset.red];
        g = pixels[i + colorOffset.green];
        b = pixels[i + colorOffset.blue];

        //Assign each rgb value to brightness for
        gray = Math.floor(.3 * r + .55 * g + .11 * b);

        data[i + colorOffset.red] = gray;
        data[i + colorOffset.green] = gray;
        data[i + colorOffset.blue] = gray;
        data[i + colorOffset.alpha] = pixels[i + colorOffset.alpha];
    }

    //Show it on the canvas
    ctx.putImageData(imgData, 0, 0);

    //Enable save button
    document.getElementById("btnSave").disabled = false;
}
```

This is a great place to point out that JavaScript isn’t necessarily the best language for working over a pile of pixels like this, though in this case the performance of a Release build running outside the debugger is actually quite good. Such routines may be better implemented as a WinRT component in a language like C# or C++ and made callable by JavaScript. We’ll take the opportunity to do exactly this in Chapter 16, “WinRT Components,” where we’ll also see limitations of the [canvas](#) element that require us to take a slightly different approach.

Saving this canvas data to a file then happens in the [saveGrayscale](#) function, where we use the file picker to get a [StorageFile](#), open a stream, acquire the [canvas](#) pixel data, and hand it off to a [BitmapEncoder](#):

```

function saveGrayscale() {
    var picker = new Windows.Storage.Pickers.FileSavePicker();
    picker.suggestedStartLocation =
        Windows.Storage.Pickers.PickerLocationId.picturesLibrary;
    picker.suggestedFileName = imageFile.name + " - grayscale";
    picker.fileTypeChoices.insert("PNG file", [".png"]);

    var imgData, fileStream = null;

    picker.pickSaveFileAsync().then(function (file) {
        if (file) {
            return file.openAsync(Windows.Storage.FileAccessMode.readWrite);
        } else {
            return WinJS.Promise.wrapError("No file selected");
        }
    }).then(function (stream) {
        fileStream = stream;
        var canvas = document.getElementById("canvas1");
        var ctx = canvas.getContext("2d");
        imgData = ctx.getImageData(0, 0, canvas.width, canvas.height);

        return Imaging.BitmapEncoder.createAsync(
            Imaging.BitmapEncoder.pngEncoderId, stream);
    }).then(function (encoder) {
        //Set the pixel data--assume "encoding" object has options from elsewhere.
        //Conversion from canvas data to Uint8Array is necessary because the array type
        //from the canvas doesn't match what WinRT needs here.
        encoder.setPixelData(encoding.pixelFormat, encoding.alphaMode,
            encoding.width, encoding.height, encoding.dpiX, encoding.dpiY,
            new Uint8Array(imgData.data));

        //Go do the encoding
        return encoder.flushAsync();
    }).done(function () {
        fileStream.close();
    }, function () {
        //Empty error handler (do nothing if the user canceled the picker)
    });
}

```

Note how the `BitmapEncoder` takes a codec identifier in its first parameter. We're using `pngEncoderId`, which is, as you can see, defined as a static property of the `Windows.Graphics.Imaging.BitmapEncoder` class; other values are `bmpEncoderId`, `gifEncoderId`, `jpegEncoderId`, `jpegXREncoderId`, and `tiffEncoderId`. These are the formats supported by the API. You can set additional properties of the `BitmapEncoder` before setting pixel data, such as its `BitmapTransform`, which will then be applied during encoding.

One gotcha to be aware of here is that the pixel array obtained from a `canvas` element (a DOM [CanvasPixelArray](#)) is not directly compatible with the WinRT byte array required by the encoder. This is the reason for the `new Uint8Array` call down there in the last parameter.

Transcoding and Custom Image Formats

In the previous section we mostly saw the use of a [BitmapEncoder](#) created with that class's static [createAsync](#) method to write a new file. That's all well and good, but you might want to know about a few of the encoder's other capabilities.

First is the [BitmapEncoder.createForTranscodingAsync](#) method that was mentioned briefly in the context of the Simple imaging sample. This specifically creates a new encoder that is initialized from an existing [BitmapDecoder](#). This is primarily used to manipulate some aspects of the source image file while leaving the rest of the data intact. To be more specific, you can first change those aspects that are expressed through the encoder's [setPixelData](#) method: the pixel format (rgba8, rgba16, and bgra8, see [BitmapPixelFormat](#)), the alpha mode (premultiplied, straight, or ignore, see [BitmapAlphaMode](#)), the image dimensions, the image DPI, and, of course, the pixel data itself. Beyond that, you can change other properties through the encoder's [bitmapProperties.setPropertiesAsync](#) method. In fact, if all you need to do is change a few properties and you don't want to affect the pixel data, you can use [BitmapEncoder.createForInPlacePropertyEncodingAsync](#) instead (how's that for a method name!). This encoder allows calls to only [bitmapProperties.setPropertiesAsync](#), [bitmapProperties.getPropertiesAsync](#), and [flushAsync](#), and since it can assume that the underlying data in the file will remain unchanged, it executes much faster than its more flexible counterparts and has less memory overhead.

An encoder from [createForTranscodingAsync](#) does *not* accommodate a change of image file format (e.g., JPEG to PNG); for that you need to use [createAsync](#) wherein you can specify the specific kind of encoding. As we've already seen, the first argument to [createAsync](#) is a codec identifier, for which you normally pass one of the static properties on [Windows.Graphics.Imaging.BitmapEncoder](#). What I haven't mentioned is that you can also specify custom codecs in this first parameter and that the [createAsync](#) call also supports an optional third argument in which you can provide options for the particular codec in question. However, there are complications and restrictions here.

Let me address options first. The present documentation for the [BitmapEncoder](#) codec values (like [pngEncoderId](#)) lacks any details about available options. For that you need to instead refer to the docs for the Windows Imaging Component (WIC), specifically the [Native WIC Codecs](#) that are what WinRT is surfacing to Store apps. If you go into the page for a specific codec, you'll then see a section on "Encoder Options" that tells you what you can use. For example, the [JPEG codec](#) supports properties like [ImageQuality](#) (a value between 0.0 and 1.0), as well as built-in rotations. The [PNG codec](#) supports properties like [FilterOption](#) for various compression optimizations.

To provide these properties, you need to create a new [BitmapPropertySet](#) and insert an entry in that set for each desired options. If, for example, you have a variable named [quality](#) that you want to apply to a JPEG encoding, you'd create the encoder like this:

```
var options = new Windows.Graphics.Imaging.BitmapPropertySet();
options.insert("ImageQuality", quality);
var encoderPromise = Imaging.BitmapEncoder.createAsync(Imaging.BitmapEncoder.jpegEncoderId,
    stream, options);
```

You use the same `BitmapPropertySet` for any properties you might pass to an encoder's `bitmap-Properties.setPropertiesAsync` call. Here's we're just using the same mechanism for encoder options.

As for custom codecs, this simply means that the first argument to `BitmapEncoder.createAsync` (as well as `BitmapDecoder.createAsync`) is the GUID (a class identifier or CLSID) for that codec, the implementation of which must be provided by a DLL. Details on how to write one of these is provided in [How to Write a WIC-Enabled Codec](#). The catch is that including custom image codecs in your package is not presently supported. If the codec is already on the system (that is, installed via the desktop), it will work. However, the Windows Store policies do not allow apps to be dependent on other apps, so it's unlikely that you can even ship such an app unless it's preinstalled on some specific OEM device and the DLL is part of the system image. (An app written in C++ can do more here, but that's beyond the scope of this book.)

In short, for apps written in JavaScript and HTML, you're really limited, for all practical purposes, to image formats that are inherently supported in the system.

Do note that these restrictions do *not* exist for custom audio and video codecs. The [Media extensions sample](#) shows how to do this with a custom video codec, as we'll see in the next section.

Manipulating Audio and Video

As with images, if all we want to do is load the contents of a `StorageFile` into an audio or video element, we can just pass that `StorageFile` to `URL.createObjectUrl` and assign the result to a `src` attribute. Similarly, if we want to get at the raw data, we can just use the `StorageFile.openAsync` or `openReadAsync` methods to obtain a file stream.

To be honest, opening the file is probably the farthest you'd ever go in JavaScript with raw audio or video, if even that. While chewing on an image is a marginally acceptable process in the JavaScript environment, churning on audio and especially video is really best done in a highly performant C++ DLL. In fact, many third-party, platform-neutral C/C++ libraries for such manipulations are readily available that you should be able to directly incorporate into such a DLL. In this case you might as well just let the DLL open the file itself!

That said, WinRT *does* provide for transcoding (converting) between different media formats and provides an extensibility model for custom codecs, effects, and scheme handlers. In fact, we've already seen how to apply custom video effects through the [Media extensions sample](#), and the same DLLs can also be used within an encoding process, where all that the JavaScript code really does is glue the right components together (which it's very good at doing). Let's see how this works with transcoding video first and then with custom codecs.

Transcoding

Transcoding both audio and video is accomplished through the [Windows.Media.Transcoding.-MediaTranscoder](#) class, which supports output formats of mp3 and wma for audio, and mp4, wmv, and m4a for video. The transcoding process also allows you to apply effects and to trim start and end times.

Transcoding happens either from one [StorageFile](#) to another or one [RandomAccessStream](#) to another, and in each case happens according to a [MediaEncodingProfile](#). To set up a transcoding operation you call the [MediaTranscoder.prepareFileTranscodeAsync](#) or [prepareStreamTranscodeAsync](#) method, which returns back a [PrepareTranscodeResult](#) object. This represents the operation that's ready to go, but it won't happen until you call that result's [transcodeAsync](#) method. In JavaScript, each result is a promise, allowing you to provide completed and progress handlers for a single operation but also allowing you to combine operations with [WinJS.Promise.join](#). This allows them to be set up and started later, which is useful for batch processing and doing automatic uploads to a service like YouTube while you're sleeping! (And at times like these I've actually pulled ice packs from my freezer and placed them under my laptop as a poor-man's cooling system....)

The [Transcoding media sample](#) provides us with a couple of transcoding scenarios. In Scenario 1 (js/presets.js) we can pick a video file, pick a target format, select a transcoding profile, and turn the machine loose to do the job (with progress being reported), as shown in Figure 10-7.

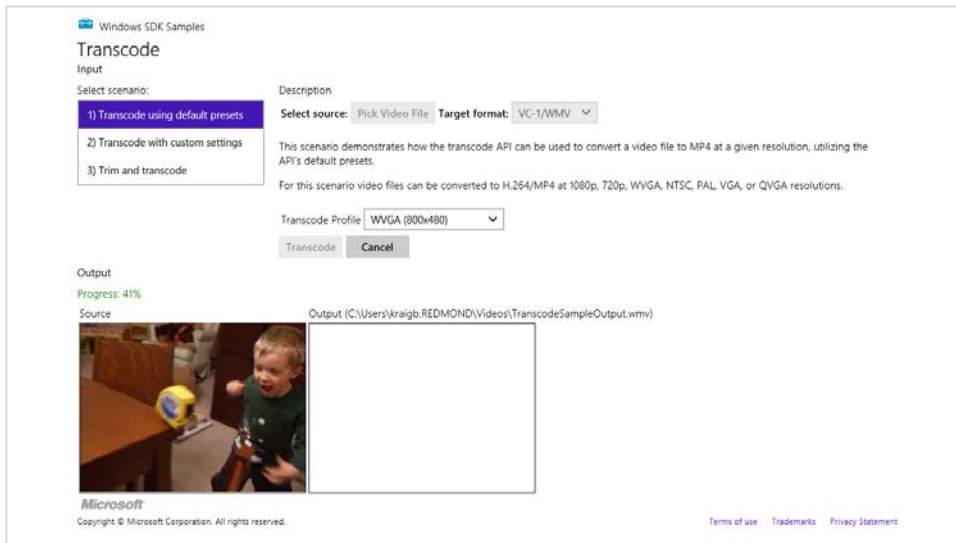


FIGURE 10-7 The Transcoding media sample cranking away on a video of my then two-year-old son discovering the joys of a tape measure.

The code that's executed when you press the Transcode button is as follows (some bits omitted; this sample happens to use nested promises, which again isn't recommended for proper error handling unless you want, as this code would show, to eat any exceptions that occur prior to the [transcode-Async](#) call):

```
function onTranscode() {
    // Create transcode object.
    var transcoder = null;
    transcoder = new Windows.Media.Transcoding.MediaTranscoder();

    // Get transcode profile.
```

```

getPresetProfile(id("profileSelect"));

// Create output file and transcode.
var videoLib = Windows.Storage.KnownFolders.videosLibrary;
var createFileOp = videoLib.createFileAsync(g_outputFileName,
    Windows.Storage.CreationCollisionOption.generateUniqueName);

createFileOp.done(function (ofile) {
    g_outputFile = ofile;
    g_transcodeOp = null;
    var prepareOp = transcoder.prepareFileTranscodeAsync(g_inputFile, g_outputFile,
        g_profile);

    prepareOp.done(function (result) {
        if (result.canTranscode) {
            g_transcodeOp = result.transcodeAsync();
            g_transcodeOp.done(transcodeComplete, transcoderErrorHandler,
                transcodeProgress);
        } else {
            transcodeFailure(result.failureReason);
        }
    }); // prepareOp.done
    id("cancel").disabled = false;
}); // createFileOp.done
}

```

The `getPresetProfile` method retrieves the appropriate profile object according to the option selected in the app. For the selections shown in Figure 10-7 (WMV and WVGA), we'd use these parts of that function:

```

function getPresetProfile(profileSelect) {
    g_profile = null;
    var mediaProperties = Windows.Media.MediaProperties;
    var videoEncodingProfile;

    switch (profileSelect.selectedIndex) {
        // other cases omitted
        case 2:
            videoEncodingProfile = mediaProperties.VideoEncodingQuality.wvga;
            break;
    }
    if (g_useMp4) {
        g_profile = mediaProperties.MediaEncodingProfile.createMp4(videoEncodingProfile);
    } else {
        g_profile = mediaProperties.MediaEncodingProfile.createWmv(videoEncodingProfile);
    }
}

```

In Scenario 2, the sample always uses the WVGA encoding but allows you to set specific values for the video dimensions, the frame rate, the audio and video bitrates, audio channels, and audio sampling. It applies these settings in `getCustomProfile` (js/custom.js) simply by configuring the profile properties after the profile is created:

```
function getCustomProfile() {
    if (g_useMp4) {
        g_profile = Windows.Media.MediaProperties.MediaEncodingProfile.createMp4(
            Windows.Media.MediaProperties.VideoEncodingQuality.wvga);
    } else {
        g_profile = Windows.Media.MediaProperties.MediaEncodingProfile.createWmv(
            Windows.Media.MediaProperties.VideoEncodingQuality.wvga);
    }

    // Pull configuration values from the UI controls
    g_profile.audio.bitsPerSample = id("AudioBPS").value;
    g_profile.audio.channelCount = id("AudioCC").value;
    g_profile.audio.bitrate = id("AudioBR").value;
    g_profile.audio.sampleRate = id("AudioSR").value;
    g_profile.video.width = id("VideoW").value;
    g_profile.video.height = id("VideoH").value;
    g_profile.video.bitrate = id("VideoBR").value;
    g_profile.video.frameRate.numerator = id("VideoFR").value;
    g_profile.video.frameRate.denominator = 1;
}
```

And to finish off, Scenario 3 is like Scenario 1, but it lets you set start and end times that are then saved in the transcoder's `trimStartTime` and `trimStopTime` properties (see js/trim.js):

```
transcoder = new Windows.Media.Transcoding.MediaTranscoder();
transcoder.trimStartTime = g_start;
transcoder.trimStopTime = g_stop;
```

Through not shown in the sample, you can apply effects to a transcoding operation by using the transcoder's `addAudioEffect` and `addVideoEffect` methods.

Custom Decoders/Encoders and Scheme Handlers

Clearly, there are many more audio and video formats in the world than Windows can support in-box, so an extensibility mechanism is provided in WinRT to allow for custom bytestream objects, custom media sources, and custom codecs and effects. It's important to note again that all such extensions are available *only* to the app itself and are not available to other apps on the system. Furthermore, Windows will always prefer in-box components over a custom one, which means don't bother wasting your time creating a new mp3 decoder or such since it will never actually be used.

As suggested earlier with custom image formats, this subject will certainly take you into some pretty vast territory around the entire [Windows Media Foundation \(WMF\) SDK](#). What's in WinRT is really just a wrapper, so knowledge of WMF is essential and not for the faint of heart!

Audio and video extensions are declared in the app manifest where you'll need to edit the XML

directly. As seen in the [Media extensions sample](#) for all the DLLs in its overall solution, each declaration looks like this:

```
<Extension Category="windows.activatableClass.inProcessServer">
  <InProcessServer>
    <Path>MPEG1Decoder.dll</Path>
    <ActivatableClass ActivatableClassId="MPEG1Decoder.MPEG1Decoder"
      ThreadingModel="both" />
  </InProcessServer>
</Extension>
```

The `ActivatableClassId` is how an extension is identified when calling the WinRT APIs, which is clearly mapped in the manifest to the specific DLL that needs to be loaded.

Depending, then, on the use of the extension, you might need to register it with WinRT through the methods of [Windows.Media.MediaExtensionManager](#): `registerAudio[Decoder | Encoder]`, `registerByteStreamHandler` (media sinks), `registerSchemeHandler` (media sources/file containers), and `registerVideo[Decoder | Encoder]`. In Scenario 1 of the Media extensions sample (`js/LocalDecoder.js`), we can see how to set up a custom decoder for video playback:

```
var page = WinJS.UI.Pages.define("/html/LocalDecoder.html", {
  extensions: null,
  MFVideoFormat_MPG1: { value: "{3147504d-0000-0010-8000-00aa00389b71}" },
  NULL_GUID: { value: "{00000000-0000-0000-0000-000000000000}" },

  ready: function (element, options) {
    if (!this.extensions) {
      // Add any initialization code here
      this.extensions = new Windows.Media.MediaExtensionManager();
      // Register custom ByteStreamHandler and custom decoder.
      this.extensions.registerByteStreamHandler("MPEG1Source.MPEG1ByteStreamHandler",
        ".mpg", null);
      this.extensions.registerVideoDecoder("MPEG1Decoder.MPEG1Decoder",
        this.MFVideoFormat_MPG1, this.NULL_GUID);
    }

    // ...
  }
});
```

where the `MPEG1Source.MPEG1ByteStreamHandler` CLSID is implemented in one DLL (see the `MPEG1Source C++` project in the sample's solution) and the `MPEG1Decoder.MPEG1.Decoder` CLSID is implemented in another (the `MPEG1Decoder C++` project).

Scenario 2, for its part, shows the use of a custom scheme handler, where the handler (in the `GeometricSource C++` project) generates video frames on the fly. Fascinating stuff, but again beyond the scope of this book.

Effects, as we've seen, are quite simple to use once you have one implemented: just pass their CLSID to methods like `msInsertVideoEffect` and `msInsertAudioEffect` on `video` and `audio` elements. You can also apply effects during the transcoding process in the `MediaTranscoder` class's `addAudio-Effect` and `addVideoEffect` methods. The same is also true for media capture, as we'll see shortly.

Media Capture

There are times when we can really appreciate the work that people have done to protect individual privacy, such as making sure I know when my computer's camera is being used since I am often using it in the late evening, sitting in bed, or in the early pre-shower mornings when I have, in the words of my father-in-law, "pineapple head."

And there are times when we want to turn on a camera or a microphone and record something: a picture, a video, or audio. Of course, an app cannot know ahead of time what exact camera and microphones might be on a system. A key step in capturing media, then, is determining which device to use—something that the [Windows.Media.Capture](#) APIs provide for nicely, along with the process of doing the capture itself into a file, a stream, or some other custom "sink" depending on how an app wants to manipulate or process the capture.

Back in Chapter 2, "Quickstart," we learned how to use WinRT to easily capture a photograph in the Here My Am! app. To quickly review, we only needed to declare the *Webcam* capability in the manifest and add a few lines of code:

```
function capturePhoto() {
    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.");
        });
}
```

The UI that Windows brings up through this API provides for cropping, retakes, and adjusting camera settings. Another example of taking a photo can also be found in Scenario 1 of the [CameraCaptureUI Sample](#), along with an example of capturing video in Scenario 2. In this latter case ([js/capturevideo.js](#)) we configure the capture UI object for a video format and indicate a video mode in the call to `captureFileAsync`:

```
function captureVideo() {
    var dialog = new Windows.Media.Capture.CameraCaptureUI();
    dialog.videoSettings.format = Windows.Media.Capture.CameraCaptureUIVideoFormat.mp4;

    dialog.captureFileAsync(Windows.Media.Capture.CameraCaptureUIMode.video)
        .done(function (file) {
            if (file) {
                var videoBlobUrl = URL.createObjectURL(file, {oneTimeOnly: true});
            } else {
                //...
            }
        }, function (err) {
            //...
        });
}
```

It should be noted that the *Webcam* capability in the manifest applies only to the image or video side of camera capture. If you want to capture audio, be sure to also select the *Microphone* capability on the Capabilities tab of the manifest editor.

If you look in the `Windows.Media.Capture.CameraCaptureUI` object, you'll also see many other options you can configure. Its `photoSettings` property, a `CameraCaptureUIPhotoCapture-Settings` object, lets you indicate cropping size and aspect ratio, format, and maximum resolution. Its `videoSettings` property, a `CameraCaptureUIVideoCaptureSettings` object, lets you set the format, set the maximum duration and resolution, and indicate whether the UI should allow for trimming. All useful stuff! You can find discussions of some of these in the docs on [Capturing or rendering audio, video, and images](#), including coverage of managing calls on a Bluetooth device.

Flexible Capture with the MediaCapture Object

Of course, the default capture UI won't necessarily suffice in every use case. For one, it always sends output to a file, but if you're writing a communications app, for example, you'd rather send captured video to a stream or send it over a network without any files involved at all. You might also want to preview a video before any capture actually happens. Furthermore, you may want to add effects during the capture, apply rotation, and perhaps apply a custom encoding.

All of these capabilities are available through the [Windows.Media.Capture.MediaCapture](#) class:

Properties	Description (classes are in the <code>Windows.Media.Capture</code> namespace unless note)
<code>audioController</code>	An <code>AudioDeviceController</code> that controls volume and provides the ability to manage other arbitrary properties that affect the audio stream.
<code>mediaCaptureSettings</code>	A <code>MediaCaptureSettings</code> that contains device IDs and mode settings, and lets you set the source (audio, videoPreview, photo).
<code>videoController</code>	A <code>VideoDeviceController</code> that controls picture properties (brightness, hue, pan/tilt, zoom, etc.). provides adjustments for backlight and AC power frequency, and provides the ability to manage other arbitrary properties that affect the video stream.
Events	Description
<code>failed</code>	Fired when an error occurs during capture.

<code>recordLimitationExceeded</code>	Fired when the user tried to record video or audio past the allowable duration.
Methods	Description
<code>initializeAsync</code>	Initialize the <code>MediaCapture</code> object (with defaults or with a <code>MediaCaptureInitialization-Settings</code> object that contains the same stuff as <code>MediaCaptureSettings</code>).
<code>addEffectAsync</code>	Applies an effect.
<code>clearEffectsAsync</code>	Clears all current effects.
<code>capturePhotoToStorageFileAsync</code> <code>capturePhotoToStreamAsync</code>	Captures an image to a storage file or a random access stream. Both take an instance of <code>ImageEncodingProperties</code> to control format (JPEG or PNG), type, dimensions, and other arbitrary Windows Properties as described earlier in the section “Common File Properties.”
<code>getEncoderProperty</code> <code>setEncoderProperty</code>	Manages specific encoder properties.
<code>startRecordToStorageFileAsync</code> <code>startRecordToStreamAsync</code> <code>stopRecordAsync</code>	Starts and stops recording to a storage file or random access stream, a <code>MediaEncodingProfile</code> that determines the audio/video format, along with bitrate, quality, video dimensions, etc.
<code>getRecordRotation</code> <code>setRecordRotation</code>	For videos, these manage a <code>VideoRotation</code> value (90-degree increments) to apply to the recording. These do not affect audio.
<code>startRecordToCustomSinkAsync</code>	Starts recording into a custom sink that's described either by an implementation of <code>Windows.Media.IMediaExtension</code> or by an ID plus a property set of settings.
<code>startPreviewAsync</code> <code>startPreviewToCustomSinkAsync</code> <code>stopPreviewAsync</code> <code>getPreviewRotation</code> <code>setPreviewRotation</code>	Same as recording but works for previews. In this case, if you call <code>URL.createObjectURL</code> and pass the <code>MediaCapture</code> object as the first parameter, the result can be assigned to the <code>src</code> attribute of a <code>video</code> element and the preview shows in that element when you call the <code>video.play</code> method.
<code>getPreviewMirroring</code> <code>setPreviewMirroring</code>	Controls preview mirroring, which means to flip the preview horizontally; this accounts for differences in camera direction which can be in the same direction as the user (rear-mounted camera as on a tablet computer), or the opposite direction (camera mounted on a monitor or built into a laptop display). See the next section, “Selecting a Media Capture Device.”

For a very simple demonstration of previewing video in a `video` element we can look at the [CameraOptionsUI sample](#) in `js/showoptionsui.js`. When you tap the Start Preview button, it creates an initializes a `MediaCapture` object as follows:

```
function initializeMediaCapture() {
    mediaCaptureMgr = new Windows.Media.Capture.MediaCapture();
    mediaCaptureMgr.initializeAsync().done(initializeComplete, initializeError);
}
```

where the `initializeComplete` handler calls into `startPreview`:

```
function startPreview() {
    document.getElementById("previewTag").src = URL.createObjectURL(mediaCaptureMgr);
    document.getElementById("previewTag").play();
    startPreviewButton.disabled = true;
    showSettingsButton.style.visibility = "visible";
}
```

```

    previewStarted = true;
}

```

The other little bit shown in this sample is invoking the `Windows.Media.Capture.CameraOptionsUI`, which happens when you tap its Show Settings button; see Figure 10-8. This is just a system-provided flyout with options that are relevant to the current media stream being captured:

```

function showSettings() {
    if (mediaCaptureMgr) {
        Windows.Media.Capture.CameraOptionsUI.show(mediaCaptureMgr);
    }
}

```

By the way, if you have trouble running a sample like this in the Visual Studio simulator—specifically, you see exceptions when trying to turn on the camera—try running on the local machine or a remote machine instead.

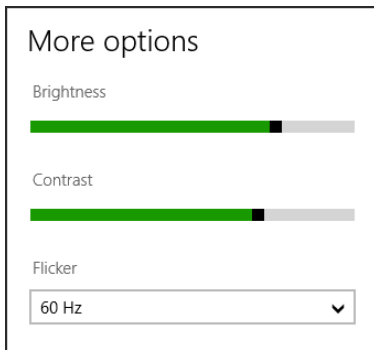


FIGURE 10-8 The Camera Options UI, as shown in the CameraOptionsUI sample (empty bottom is cropped).

More complex scenarios involving the `MediaCapture` class (and a few others) can be found now in the [Media capture using capture device sample](#), such as previewing and capturing video, changing properties dynamically (Scenario 1), selecting a specific media device (Scenario 2), and recording just audio (Scenario 3).

Starting with Scenario 3 (`js/AudioCapture.js`, the simplest), here's the core code to create and initialize the `MediaCapture` object for an audio stream (see the `streamingCaptureMode` property in the initialization settings), where that stream is directed to a file in the music library via `startRecordToStorageFileAsync` (some code omitted for brevity):

```

var mediaCaptureMgr = null;
var captureInitSettings = null;
var encodingProfile = null;
var storageFile = null;

// This is called when the page is loaded
function initCaptureSettings() {
    captureInitSettings = new Windows.Media.Capture.MediaCaptureInitializationSettings();
    captureInitSettings.audioDeviceId = "";
}

```

```

    captureInitSettings.videoDeviceId = "";
    captureInitSettings.streamingCaptureMode =
        Windows.Media.Capture.StreamingCaptureMode.audio;
}

function startDevice() {
    mediaCaptureMgr = new Windows.Media.Capture.MediaCapture();

    mediaCaptureMgr.initializeAsync(captureInitSettings).done(function (result) {
        // ...
    });
}

function startRecord() {
    // ...
    // Start recording.
    Windows.Storage.KnownFolders.videosLibrary.createFileAsync("cameraCapture.m4a",
        Windows.Storage.CreationCollisionOption.generateUniqueName)
        .done(function (newFile) {
            storageFile = newFile;
            encodingProfile = Windows.Media.MediaProperties
                .MediaEncodingProfile.createM4a(Windows.Media.MediaProperties
                    .AudioEncodingQuality.auto);
            mediaCaptureMgr.startRecordToStorageFileAsync(encodingProfile,
                storageFile).done(function (result) {
                    // ...
                });
        });
}

function stopRecord() {
    mediaCaptureMgr.stopRecordAsync().done(function (result) {
        displayStatus("Record Stopped. File " + storageFile.path + " ");

        // Playback the recorded audio
        var audio = id("capturePlayback" + scenarioId);
        audio.src = URL.createObjectURL(storageFile, { oneTimeOnly: true });
        audio.play();
    });
}

```

Scenario 1 is essentially the same code but captures a video stream as well as photos, with results shown in Figure 10-9. This variation is enabled through these properties in the initialization settings (see `js/BasicCapture.js` within `initCaptureSettings`):

```

captureInitSettings.photoCaptureSource =
    Windows.Media.Capture.PhotoCaptureSource.videoPreview;
captureInitSettings.streamingCaptureMode =
    Windows.Media.Capture.StreamingCaptureMode.audioAndVideo;

```

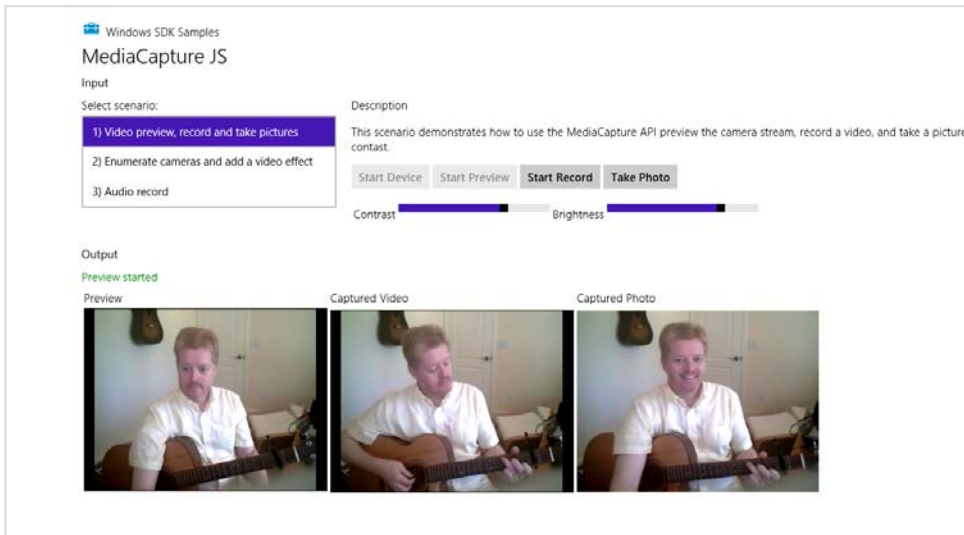


FIGURE 10-9 Previewing and recording video with the default device in the Media capture sample, Scenario 1. (The output is cropped because I needed to run the app using the Local Machine option in Visual Studio, and I didn't think you needed to see a 1920x1200 screenshot with lots of whitespace!).

Notice the Contrast and Brightness controls in Figure 10-9. Changing these will change the preview video, along with the recorded video. The sample does this through the `MediaCapture.video-DeviceController` object's `contrast` and `brightness` properties, showing that these (and any others in the controller) can be adjusted dynamically. Refer to the `getCameraSettings` function in `js/BasicCapture.js` that basically wires the slider `change` events into a generic anonymous function to update the desired property.

Selecting a Media Capture Device

Looking now at Scenario 2 (`js/AdvancedCapture.js`), it's more or less like Scenario 1 but it allows you to select the specific input device. Until now, everything we've done has simply used the default device, but you're not limited to that, of course. You can use the [Windows.Devices.Enumeration](#) API to retrieve a list of devices within a particular device interface class; the sample uses the predefined `videoCapture` class:

```
function enumerateCameras() {
    var cameraSelect = id("cameraSelect");
    deviceList = null;
    deviceList = new Array();
    while (cameraSelect.length > 0) {
        cameraSelect.remove(0);
    }
    //Enumerate webcams and add them to the list
    var deviceInfo = Windows.Devices.Enumeration.DeviceInformation;
    deviceInfo.findAllAsync(Windows.Devices.Enumeration.DeviceClass.videoCapture)
        .done(function (devices) {
```

```

        // Add the devices to deviceList
        if (devices.length > 0) {
            for (var i = 0; i < devices.length; i++) {
                deviceList.push(devices[i]);
                cameraSelect.add(new Option(deviceList[i].name), i);
            }
            //Select the first webcam
            cameraSelect.selectedIndex = 0;
            initCaptureSettings();
        } else {
            // disable buttons.
        }
    }, errorHandler);
}

```

The selected device's ID is then copied within `initCaptureSettings` to the `MediaCapture-InitializationSetting.videoDeviceId` property:

```

var selectedIndex = id("cameraSelect").selectedIndex;
var deviceInfo = deviceList[selectedIndex];
captureInitSettings.videoDeviceId = deviceInfo.id;

```

By the way, you can retrieve the default device ID at any time through the methods of the [Windows.Media.Devices.MediaDevice](#) object and listen to its events for changes in the default devices. It's also important to note that [DeviceInformation](#) (in the `deviceInfo` variable above) includes a property called `enclosureLocation`. This tells you whether a camera is forward or back-ward facing, which you can use to rotate the video or photo as appropriate for the user's perspective:

```

var cameraLocation = null;

if (deviceInfo.enclosureLocation) {
    cameraLocation = deviceInfo.enclosureLocation.panel;
}

if (cameraLocation === Windows.Devices.Enumeration.Panel.back) {
    rotateVideoOnOrientationChange = true;
    reverseVideoRotation = false;
} else if (cameraLocation === Windows.Devices.Enumeration.Panel.front) {
    rotateVideoOnOrientationChange = true;
    reverseVideoRotation = true;
} else {
    rotateVideoOnOrientationChange = false;
}

```

The other bit that Scenario 2 demonstrates is using the `MediaCapture.addEffectAsync` with a grayscale effect, shown in Figure 10-10, that's implemented in a DLL (the GrayscaleTransform project in the sample's solution). This works exactly as it did with transcoding, and you can refer to the `addRemoveEffect` and `addEffectToImageStream` functions in `js/AdvancedCapture.js` for the details. You'll notice there that these functions do a number of checks using the `MediaCaptureSettings.videoDeviceCharacteristic` value to make sure that the effect is added in the right place.

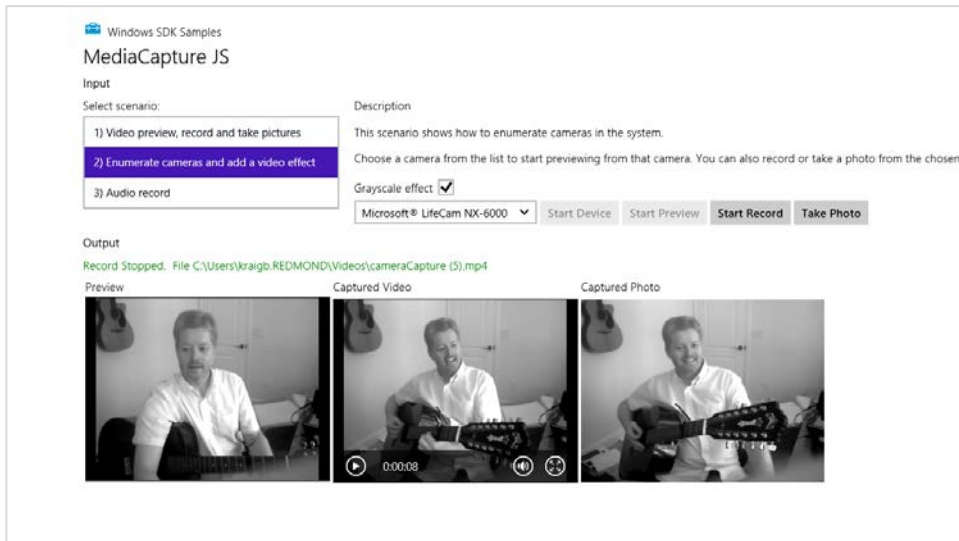


FIGURE 10-10 Scenario 2 of the Media capture sample in which one can select a specific device and apply an effect. (The output here is again cropped from a larger screen shot.) Were you also paying attention enough to notice that I switched guitars?

Streaming Media and PlayTo

To say that streaming media is popular is certainly a gross understatement. As mentioned in this chapter's introduction, Netflix alone consumes for a large percentage of today's Internet bandwidth (including that of my own home). YouTube certainly does its part as well—so your app might as well contribute to the cause!

Streaming media from a server to your app is easily the most common case, and it happens automatically when you set an audio or video `src` attribute to a remote URI. To improve on this, Microsoft also has a [Smooth Streaming SDK for Windows 8 Apps](#) (in beta at the time of writing) that helps you build media apps with a number of rich features including live playback and PlayReady content protection. I won't be covering that SDK in this book, so I wanted to make sure you were aware of it.

What we'll focus on here, in the few pages we have left before my editors at Microsoft Press pull the plug on this chapter, are considerations for digital rights management (DRM) and streaming not from a network but *to* a network, for example, audio/video capture in a communications app, as well as streaming media from an app to a PlayTo device.

Streaming from a Server and Digital Rights Management (DRM)

Again, streaming media from a server is what you already do whenever you're using an `audio` or `video` element with a remote URI. The details just happen for you. Indeed, much of what a great media client app does is talking to web services, retrieving metadata and the catalog, helping the user navigate all of that information, and ultimately getting to a URI that can be dropped in the `src` attribute of a `video` or `audio` element. Then, once the app receives the `canplay` event, you can call the element's `play` method to get everything going.

Of course, media is often protected with DRM, otherwise the content on paid services wouldn't be generating much income for the owners of those rights! So there needs to be a mechanism to acquire and verify rights somewhere between setting the element's `src` and receiving `canplay`. Fortunately, there's a simple means to do exactly that:

- Before setting the `src` attribute, create an instance of [Windows.Media.Protection.MediaProtectionManager](#) and configure its `properties`.
- Listen to this object's `serviceRequested` event, the handler for which performs the appropriate rights checks and sets a completed flag when all is well. (Two other events, just to mention them, are `componentloadfailed` and `rebootneeded`.)
- Assign the protection manager to the audio/video element with the [msSetMediaProtectionManager](#) extension method.
- Set the `src` attribute. This will trigger the `serviceRequested` event to start the DRM process which will prevent `canplay` until DRM checks are completed successfully.
- In the event of an error, the media element's `error` event will be fired. The element's `error` property will then contain an `msExtendedCode` with more details.

You can refer to [How to use pluggable DRM](#) and [How to handle DRM errors](#) for additional details, but here's a minimal and hypothetical example of all this in code:

```
var video1 = document.getElementById("video1");

video1.addEventListener('error', function () {
    var error = video1.error.msExtendedCode;
    //...
}, false);

video1.addEventListener('canplay', function () {
    video1.play();
}, false);

var cpm = new Windows.Media.Protection.MediaProtectionManager();
cpm.addEventListener('servicerequested', enableContent, false); //Remove this later
video1.msSetContentProtectionManager(cpm);
video1.src = "http://some.content.server.url/protected.wmv";

function enableContent(e) {
```

```

if (typeof (e.request) != 'undefined') {
    var req = e.request;
    var system = req.protectionSystem;
    var type = req.type;

    //Take necessary actions based on the system and type
}

if (typeof (e.completion) != 'undefined') {
    //Requested action completed
    var comp = e.completion;
    comp.complete(true);
}
}

```

How you specifically check for rights, of course, is particular to the service you're drawing from—and not something you'd want to publish in any case!

For a more complete demonstration of handling DRM, check out the [Simple PlayReady sample](#), which will require that you download and install the [Microsoft PlayReady Client SDK](#). PlayReady, if you aren't familiar with it yet, is a license service that Microsoft provides so that you don't have to create one from scratch. The PlayReady client SDK provides additional tools and framework support for apps wanting to implement both online and offline media scenarios, such as progressive download, download to own, rentals, and subscriptions. Plus, with the SDK you don't need to submit your app for

DRM Conformance testing. In any case, here's how the Simple PlayReady sample sets up its content protection manager, just to give an idea of how the WinRT APIs are used with specific DRM service identifiers:

```

mediaProtectionManager = new Windows.Media.Protection.MediaProtectionManager();
mediaProtectionManager.properties["Windows.Media.Protection.MediaProtectionSystemId"] =
    '{F4637010-03C3-42CD-B932-B48ADF3A6A54}'

var cpsystems = new Windows.Foundation.Collections.PropertySet();
cpsystems["{F4637010-03C3-42CD-B932-B48ADF3A6A54}"] =
    "Microsoft.Media.PlayReadyClient.PlayReadyWinRTTrustedInput";
mediaProtectionManager.properties[
    "Windows.Media.Protection.MediaProtectionSystemIdMapping"] = cpsystems;

```

Streaming from App to Network

The next case to consider is when an app is the source of streaming media rather than the consumer, which means that client apps elsewhere are acting in that capacity. In reality, in this scenario—audio or video communications and conferencing—it's usually the case that the app plays both roles, streaming media to other clients and consuming media from them. This is the case with Windows Live Messenger, Skype, and other such utilities, along with apps like games that include chat capabilities.

Here's how such apps generally work:

- Set up the necessary communication channels over the network, which could be a peer-to-peer system or could involve a central service of some kind.
- Capture audio or video to a stream using the WinRT APIs we've seen (specifically [MediaCapture.startRecordToStreamAsync](#)) or capturing to a custom sink.
- Do any additional processing to the stream data. Note, however, that effects are plugged into the capture mechanism ([MediaCapture.addEffectAsync](#)) rather than something you do in post-processing.
- Encode the stream for transmission however you need.
- Transmit the stream over the network channel.
- Receive transmissions from other connected apps.
- Decode transmitted streams and convert to a blob by using [MSApp.createBlobFromRandomAccessStream](#).
- Use [URL.createObjectURL](#) to hook an [audio](#) or [video](#) element to the stream.

To see such features in action, check out the [Real-time communications sample](#) that implements video chat in Scenario 2 and demonstrates working with different latency modes in Scenario 1. The latter two steps in the list above are also shown in the [PlayToReceiver sample](#) that is set up to receive a media stream from another source.

PlayTo

The final case of streaming is centered on the PlayTo capabilities that were introduced in Windows 7. Simply said, PlayTo is a means through which an app can connect local playback/display for [audio](#), [video](#), and [img](#) elements to a remote device.

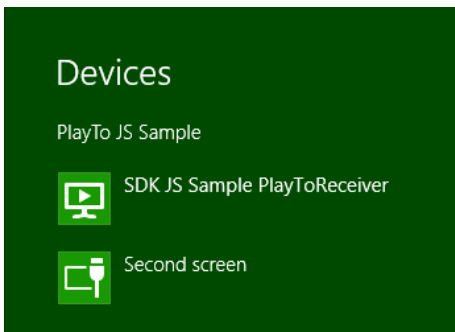
The details happen through the [Windows.Media.PlayTo](#) APIs along with the extension methods added to media elements. If, for example, you want to specifically start a process of streaming to a PlayTo device, invoking the selection UI directly, you'd do the following:

- [Windows.Media.PlayTo.PlayToManager](#):
 - [getForCurrentView](#) returns the object.
 - [showPlayToUI](#) invokes the flyout UI where the user selects a receiver.
 - [sourceRequested](#) event is fired when user selects a receiver.
- In [sourceRequested](#)
 - Get [PlayToSource](#) object from [audio](#), [video](#), or [img](#) element ([msPlayToSource](#) property) and pass to [e.setSource](#).

- Set `PlayToSource.next` property to the `msPlayToSource` of another element for continual playing.
- Pick up the media element's `ended` event to stage additional media

Another approach, as demonstrated in the [Media Play To sample](#), is to go ahead and play media locally and then let the user choose a PlayTo device on the fly from the Devices charm. In this case you don't need to do anything because Windows will pick up the current playback element and direct it accordingly. But the app can listen to the `statechanged` event of the element's `msPlayToSource.connection` object (a [PlayToConnection](#)) that will fire when the user selects a PlayTo device and when other changes happen.

Generally speaking, PlayTo is primarily intended for streaming to a media receiver device that's probably connected to a TV or other large screen. This way you can select local content on a Windows 8 device and send it straight to that receiver. But it's also possible to make a software receiver—that is, an app that can receive streamed content from a PlayTo source. The [PlayToReceiver sample](#) does exactly this, and when you run it on another device on your local network, it will show up in the Devices charms as follows:



You can even run the app from your primary machine using the remote debugging tools of Visual Studio, allowing you to step through the code of both source and receiver apps at the same time! (Another option is to run Windows Media Player on one machine and check its Stream > Allow Remote Control of My Player menu option. This should make that machine appear in the PlayTo target list.)

To be a receiver, an app will generally want to declare some additional networking capabilities in the manifest—namely, *Internet (Client & Server)* and *Private Networks (Client & Server)*—otherwise it won't see much action! It then creates an instance of `Windows.Media.PlayTo.PlayToReceiver`, as shown in the PlayTo Receiver sample's `startPlayToReceiver` function (`js/audiovideoptr.js`):

```
function startPlayToReceiver() {
    if (!g_receiver) {
        g_receiver = new Windows.Media.PlayTo.PlayToReceiver();
    }
}
```

Next you'll want to wire up handlers for the element that will play the media stream:

```
var dmrVideo = id("dmrVideo");
dmrVideo.addEventListener("volumechange", g_elementHandler.volumechange, false);
dmrVideo.addEventListener("ratechange", g_elementHandler.ratechange, false);
dmrVideo.addEventListener("loadedmetadata", g_elementHandler.loadedmetadata, false);
dmrVideo.addEventListener("durationchange", g_elementHandler.durationchange, false);
dmrVideo.addEventListener("seeking", g_elementHandler.seeking, false);
dmrVideo.addEventListener("seeked", g_elementHandler.seeked, false);
dmrVideo.addEventListener("playing", g_elementHandler.playing, false);
dmrVideo.addEventListener("pause", g_elementHandler.pause, false);
dmrVideo.addEventListener("ended", g_elementHandler.ended, false);
dmrVideo.addEventListener("error", g_elementHandler.error, false);
```

along with handlers for events that the receiver object will fire:

```
g_receiver.addEventListener("playrequested", g_receiverHandler.playrequested, false);
g_receiver.addEventListener("pauserequested", g_receiverHandler.pauserequested, false);
g_receiver.addEventListener("sourcechangerequested", g_receiverHandler.sourcechangerequested, false);
g_receiver.addEventListener("playbackratechangerequested", g_receiverHandler.playbackratechangerequested, false);
g_receiver.addEventListener("currenttimechangerequested", g_receiverHandler.currenttimechangerequested, false);
g_receiver.addEventListener("mutechangerequested",
    g_receiverHandler.mutedchangerequested, false);
g_receiver.addEventListener("volumechangerequested", g_receiverHandler.volumechangerequested, false);
g_receiver.addEventListener("timeupdaterequested",
    g_receiverHandler.timeupdaterequested, false);
g_receiver.addEventListener("stoprequested", g_receiverHandler.stoprequested, false);
g_receiver.supportsVideo = true;
g_receiver.supportsAudio = true;
g_receiver.supportsImage = false;
g_receiver.friendlyName = 'SDK JS Sample PlayToReceiver';
```

The last line above, as you can tell from the earlier image, is the string that will show in the Devices charm for this receiver once it's made available on the network. This is done by calling `startAsync`:

```
// Advertise the receiver on the local network and start receiving commands
g_receiver.startAsync().then(function () {
    g_receiverStarted = true;

    // Prevent the screen from locking
    if (!g_displayRequest) {
        g_displayRequest = new Windows.System.Display.DisplayRequest();
    }
    g_displayRequest.requestActive();
});
```

Of all the receiver object's events, the critical one is `sourcechangerequested` where `eventArgs.stream` contains the media we want to play in whatever element we choose. This is easily accomplished by creating a blob from the stream and then a URI from the blob that we can assign to an element's `src` attribute:

```

sourcechangerequested: function (eventIn) {
    if (!eventIn.stream) {
        id("dmrVideo").src = "";
    } else {
        var blob = MSApp.createBlobFromRandomAccessStream(eventIn.stream.contentType,
            eventIn.stream);
        id("dmrVideo").src = URL.createObjectURL(blob, {oneTimeOnly: true});
    }
}

```

All the other events, as you can imagine, are primarily for wiring together the source's media controls to the receiver such that pressing a pause button, switching tracks, or acting on the media in some other way at the source will be reflected in the receiver. There may be a lot of events, but handling them is quite simple as you can see in the sample.

What We Have Learned

- Creating media elements can be done in markup or code by using the standard [img](#), [svg](#), [canvas](#), [audio](#), and [video](#) elements.
- The three graphics elements—[img](#), [svg](#), and [canvas](#)—can all produce essentially the same output, only with different characteristics as to how they are generated and how they scale. All of them can be styled with CSS, however.
- The [Windows.System.Display.DisplayRequest](#) object allows for disabling screen savers and the lock screen during video playback (or any other appropriate scenario).
- Both the audio and video elements provide a number of extension APIs (properties, methods, and events) for working with various platform-specific capabilities in Windows 8, such as horizontal mirroring, zooming, playback optimization, 3D video, low-latency rendering, PlayTo, playback management of different audio types or categories, effects (generally provided as DLLs in the app package), and digital rights management.
- Background audio is supported for several categories given the necessary declarations in the manifest and handlers for media control events (so the audio can be appropriately paused and played). Media control events are important to also support the media control UI.
- Through the WinRT APIs, apps can manage very rich metadata and properties for media files, including thumbnails, album art, and properties specific to the media type, including access to a very extensive list of [Windows Properties](#).
- The WinRT APIs provide for decoding and encoding of media files and streams, through which the media can be converted or properties changed. This includes support for custom codecs.

- WinRT provides a rich API for media capture (photo, video, and audio), including a built-in capture UI, along with the ability to provide your own and yet still easily enumerate and access available devices.
- Streaming media is supported from a server (with and without digital rights management, including PlayReady), between apps (inbound and outbound), and from apps to PlayTo devices. An app can also be configured as a PlayTo receiver.

Chapter 11

Purposeful Animations

In the early 1990s, the wonderful world of multimedia first became prevalent on Windows PCs. Before that time it was difficult for such machines to play audio and video, access compact discs (remember those?), and otherwise provide the rich experience we take for granted today. The multimedia experience was new and exciting, and many people jumped in wholeheartedly, including the group of developer support engineers at Microsoft specializing in this area. Though my team (specializing in UI) sat more than 100 feet away from their area, we could clearly hear—for most of the day!—the various chirps and bleeps emitting from their speakers, against the background of a soft Amazon basin rainfall.

At that time too, many consumers of Windows were having fun attaching all kinds of crazy sounds to every mouse click, window transition, email arrival, and every other system event they could think of. Yet after a month or two of this sensual overload—not unlike being at a busy carnival—most people started to remove quite a few of those sounds, if not disable them altogether. I, for one, eventually turned off all my sounds. Simply said, I got tired of the extra noise.

Along these same lines, you may remember that when DVDs first appeared in their full glory, just about every title had fancy menus with clever transitions. No more: most consumers, I think, got tired of waiting for all this to happen and just want to get on with the business of watching the movie as quickly as possible.

Today we're reliving this same experience with fluid animations. Now that most systems have highly responsive touch screens and GPUs capable of providing very smooth graphical transitions, it's tempting to use animations superfluously. However, unless the animations actually add meaning and function to an app, consumers will likely tire of them like they did with DVD menus, especially if they end up interfering with a workflow by making one constantly wait for the animations to finish. I'll bet that every reader of this book has, at least once, repeatedly hit the Menu button on a DVD remote to no avail....

This is why Windows Store app design speaks of *purposeful* animations: if there's no real purpose behind an animation in your app, ask yourself, "Why am I wanting to use this?" Take a moment, in fact, to use Windows 8 and some of the built-in apps to explore how animations are both used and *not* used. Notice how many animations are specifically to track or otherwise give immediate feedback for touch interactions, which purposefully help users know that their input is being registered by the system (and is, in fact, a Store certification requirement). Other animations, such as when items are added or removed from a list, are intended to draw attention to the change, soften its visual impact, and give it a sense of fluidity. In other cases, you may find apps that perhaps overuse animations, simply using animations because they're available or trying too hard to emulate physical motion where it's simply not

necessary. In this way, excessive animations constitute a kind of “chrome” with the same effect as other chrome: distracting the user from the content they really care about. (If you can’t resist the temptation to add little effects that are like this, consider at least providing a setting to turn them off.)

Let me put it another way. When thinking about animations, ask yourself, “What do they communicate?” Animations are a form of communication, a kind of visual language. I would even venture to say (as I am venturing now) that animations really only say one or a combination of three things:

- “Thanks, I heard you,” as when something on the screen moves naturally in response to a user gesture. Without this communication, the user might think that their gesture didn’t register and will almost certainly poke at the app again.
- “Hello” and “Goodbye,” as when items appear or disappear from view, or transition one to another. Without this communication, changes that happen to on-screen elements can be as jarring as Bilbo Baggins in *Lord of the Rings* slipping on the Ring of Power and instantly vanishing. This is not to say that most consumers are incredulous hobbits, of course, but you get the idea.
- “Hey, look at me!” as when something moves to only gain attention or look cute.

If I were to assign percentages to these categories to represent how often they would or should be used, I’d probably put them at 80%, 15%, and 5%, respectively (although some animations will serve multiple purposes). Put another way, the first bit of communication is really about listening and responding, which is what an app should be doing most of the time. The second bit is about courtesy, which is another good quality to express within reason—courtesy can, like handshakes, hugs, bows, and salutes, be overused to the point of annoyance. The third bit, for its part, can be helpful when there’s a real and sincere reason to raise your hand or offer a friendly wave, but otherwise can easily become just another means of showing off.

There’s another good reason to be judicious about the use of animations and really make them count: power consumption. No matter how it’s accomplished, via GPU or CPU, animation is an expensive process. Every watt of juice in a consumer’s batteries should be directed toward fulfilling their goals with their device rather than scattered to the wind. Again, this is why this chapter is called “*Purposeful Animations*” and not just “*Animations*”!

In any case, you and your designers are the ultimate arbiters of how and when you’ll use animations. Let me emphasize here that animations should be part of an app’s design, not just an implementation detail. Animations are very much part of the overall user experience of an app. Oftentimes app designs focus on static wireframes and static mockups, neither of which indicate dynamic elements like animations and transitions. Animations are also tightly coupled to the app’s layout and should be designed alongside that layout from the earliest stages of design.

In this uncommonly short chapter, then, we'll first look at what's provided for you in the WinJS Animations Library, a collection of animations built on CSS that already embody the Windows 8 look and feel for many common operations. After that we'll review the underlying CSS capabilities that you can, of course, use directly. In fact, aside from games and other apps whose primary content consists of animated objects, you can probably use CSS for most other animation needs. This is a good idea because the CSS engine is very much optimized to take advantage of hardware acceleration, something that isn't true when doing frame-by-frame animations in JavaScript yourself. Nevertheless, we'll end this chapter on that latter subject, as there are some tips and tricks for doing it well within Windows Store apps.

Systemwide Enabling and Disabling of Animations

Before we go any further, there's a setting buried deep inside the desktop Control Panel's Ease of Access Center that you need to know about because it affects how the WinJS Animations Library behaves and should affect whether you do animations of your own. From the desktop, invoke the Settings Charms and select Control Panel. Then navigate to Ease of Access > Ease of Access Center > Make The Computer Easier To See. Scroll down close to the bottom and you'll see the item "Turn off all unnecessary animations (when possible)," as shown in Figure 11-1.

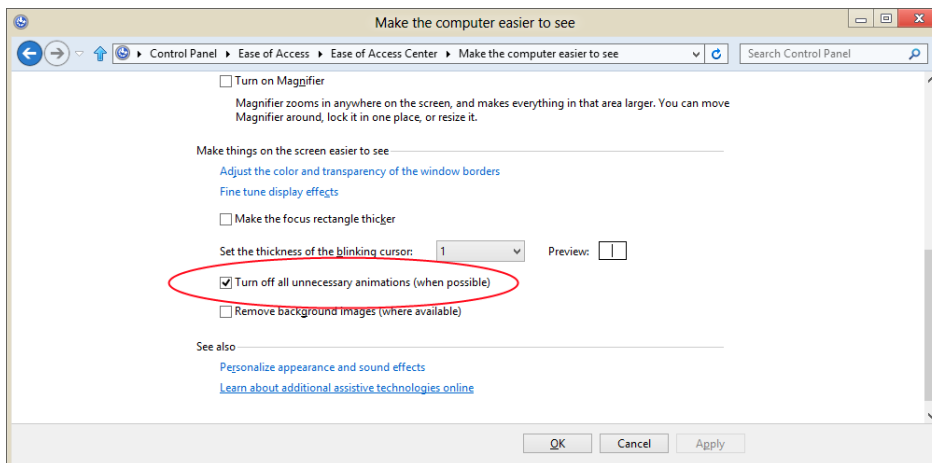


FIGURE 11-1 A very important setting for animation in the desktop control panel.

The idea behind this check box is that for some users, animations are a real distraction that can make the entire machine more difficult to use. For medical reasons too, some users might elect to minimize on-screen movement just to keep the whole experience more calm. So when this option is checked, the WinJS animations don't actually do anything, and it's recommended that apps also disable many if not all of their own custom animations as well.

The Control Panel setting can be obtained through the [Windows.UI.ViewManagement.UISettings](#) class in its `animationsEnabled` property:

```
var settings = new Windows.UI.ViewManagement.UISettings();  
var enabled = settings.animationsEnabled;
```

You can also just call the `WinJS.UI.isAnimationEnabled` method that will return `true` or `false` depending on this property. WinJS obviously uses this internally to manage its own animation behavior.

WinJS also adds an enablement count that you can use to temporarily enable or disable animations in conjunction with the `animationsEnabled` value. You change this count by calling `WinJS.UI.enableAnimations` and `WinJS.UI.disableAnimations`, the effects of which are cumulative, and the `animationsEnabled` property counts as 0 if the Control Panel option is checked and 1 if it's unchecked.

When implementing your own animations either with CSS or with mechanisms like `setInterval` or `requestAnimationFrame`, it's a good idea to be sensitive to the `animationsEnabled` setting where appropriate. I add this condition because if an animation is essential to the actual content of an app, like a game, then it's not appropriate to apply this setting. The same goes for animating something like a clock within a clock app. It's really about animations that add a fast-and-fluid effect to the content, but it can be turned off without ill effect.

The WinJS Animations Library

When considering animations for your app, the first place you should turn is the Animations Library in WinJS, found in the [WinJS.UI.Animation](#) namespace. Each animation is basically a function within this namespace that you call when you want a certain kind of animation or transition to happen. The benefit of using these is that they directly embody the Windows 8 look and feel and, in fact, are what WinJS itself uses to animate its own controls, flyouts, and so forth to match the user interface design guidelines. What's more, because they are built with CSS transitions and animations, they aren't dependent on WinRT and are fully functional within web context pages that have loaded WinJS (but they do again pay attention to whether animations are enabled as described in the previous section).

All of the animations, as listed in the table below, have guidance as to when and how they should be applied. These are again really design questions more than implementation questions, as stated earlier. By being aware of what's in the animations library, designers can more readily see where animations are appropriately applied and include them early on in their app design, which makes your life as a developer all the more predictable.

You can find full guidance in the [Animating Your UI](#) and [Animating UI Surfaces](#) topics in the documentation, which will also contain specific guidelines for the individual animations below. I will only summarize here.

Key Point Built-in controls and other UI elements like those we've worked with in previous chapters already make use of the appropriate animations. For example, you don't need to animate a button tap in the `button` element nor animate the appearance or disappearance of controls like `WinJS.UI.Appbar`. You'll primarily use them when implementing UI directly with HTML layout or when building custom controls.

Animation Name	WinJS.UI.Animation methods	Description and Usage
Page Transition	<code>enterPage</code> , <code>exitPage</code>	Animates a whole page into or out of view, such as when bringing in the first page of an app after the splash screen or when switching between app pages. Avoid using <code>enterPage</code> when content is already on screen—that is, use it only when changing the entirety of the content.
Content Transition	<code>enterContent</code> , <code>exitContent</code>	Animates one or more elements into or out of view, specifically used for content that wasn't ready when a page was loaded or when a section of a page is changing within a container. If other content needs to move in response to the container change, such as if it is resizing, you can move those other elements by using <code>expand/collapse</code> or <code>reposition</code> animations.
Fade In/Out	<code>fadeIn</code> , <code>fadeOut</code>	Used to show or hide transient UI or controls, as is done with scrollbars or when a placeholder is replaced with a loaded item. These are also good default animations for situations where other specific animations don't apply.
Crossfade	<code>crossFade</code>	Softens the transition between different states of an item. This is also used in refresh scenarios, such as when a news app updates all of its content at once.
Pointer Up/Down	<code>pointerUp</code> , <code>pointerDown</code>	Provides immediate feedback for a successful tap or click on an item or tile-like elements. Note that built-in controls like the button and ListView already incorporate these animations.
Expand/Collapse	<code>createExpandAnimation</code> , <code>createCollapseAnimation</code>	Adds or removes extra space within content, such as making room for error messages or hiding an option that isn't needed.
Reposition	<code>createRepositionAnimation</code>	Used when moving an element to a new position.
Show/Hide Popup	<code>showPopup</code> , <code>hidePopup</code>	Used to show and hide popup UI like menus, flyouts, tooltips, and other contextual UI that appears above an app canvas (dialogs, however, use Fade In). Avoid using for elements that are part of that canvas directly—use Content Transition and Fade In/Out animations instead. You also don't need to use these directly when using built-in controls, as those controls already apply the animations.
Show/Hide Edge UI Show/Hide Panel UI	<code>showEdgeUI</code> , <code>hideEdgeUI</code> , <code>showPanel</code> , <code>hidePanel</code>	Used to show and hide edge-oriented UI like app bars and the soft keyboard. The Edge UI animations are for elements that only move a short distance onto the

		<p>screen; the Panel animations are for those that move longer distances.</p> <p>These should not be used for UI that's not moving from or toward an edge; use the Reposition animation instead. Crossfade is also typically applied after showing and simultaneous with hiding. The built-in edge controls like the app bar and settings pane already apply these animations.</p>
Peek (for tiles)	<code>createPeekAnimation</code>	<p>Animates a tile update when alternating between image and text areas; see Chapter 13, "Tiles, Notifications, the Lock Screen, and Background Tasks." Can also be used to cycle through tile updates. This is the animation used for live tiles on the Windows Start screen.</p>
Badge Update	<code>updateBadge</code>	Used to update the number on a tile badge.
Swipe Hint	<code>swipeReveal</code>	Used in response to a tap-and-hold event to indicate that an item can be selected with a swipe.
Swipe Select/Deselect	<code>swipeSelect</code> , <code>swipeDeselect</code>	Animates an item when swiped to select or deselect it.
Add/Delete from List	<code>createAddToListAnimation</code> , <code>createDeleteFromListAnimation</code>	<p>Animates the insertion or deletion of items from a list, as used by the ListView control. The add animation repositions existing items to make space for the new items and then brings them; the delete animation pulls items out and repositions those that remain. Avoid using these to display or remove a container or to add or remove the entire contents of the collection; use Content Transitions instead.</p>
Add/Delete from Search List	<code>createAddToSearchListAnimation</code> , <code>createDeleteFromSearchListAnimation</code>	<p>These animations are similar to those for adding and removing from a list, but they are designed for much more rapid changes as happens when populating a list of search results. Simply said, they have shorter durations.</p>
Start/End Drag-Drop	<code>dragSourceStart</code> , <code>dragSourceEnd</code> , <code>dragBetweenEnter</code> , <code>dragBetweenLeave</code>	<p>Provides visual feedback during drag-and-drop operations as seen on the Start screen when you move tiles around. The start and end animations are for the item being moved and should always be used together; the enter and leave animations are for rearranging the area around a potential drop point, which helps to show how the content will appear if the drop happens. For this purpose you'll need to define the size of potential target areas (rectangles) so that you can track pointer movement in and out of those areas.</p>

If you want to see what these animations are actually doing, you can find all of that in the WinJS source code's `ui.js` file. Just search for the method, and you'll see how they're set up. The Crossfade animation, for example, animates the incoming element's opacity property from 0 to 1 over 167ms with a linear timing function, while animating the outgoing element's opacity from 1 to 0 in the same way.

The Pointer Down animation changes the element's scale from 100% to 97.5% over 167ms according to a cubic-bezier curve, while Pointer Up does the opposite.

Knowing these characteristics or *animation metrics* can be important when creating web-based content to integrate with your app. As noted before the Windows 8 app certification requirements, specifically section 4.5, indicates that touch targets must provide visual feedback. Because you cannot use WinJS on a remote web page, knowing how these animations work will help you emulate that behavior on such pages.

Within your app, though, you should acquire these animation metrics programmatically through the API in [Windows.Core.UI.AnimationMetrics](#), rather than hardcoding any values. You can find a demonstration of using this API in the [Animation metrics sample](#).

As interesting as such details might be, of course, they are always subject to change. And in the end, what's important is that you choose animations not for their visual effects but for their semantic meaning, using the right animations at the right times in the right places. So let's see how we do that.

Tip #1 All of the WinJS animations are implemented using the [WinJS.UI.executeAnimation](#) and [WinJS.UI.executeTransition](#) functions, which you can use for custom animations as well.

Tip #2 While an animation is running always avoid changing an element's contents and its CSS styles that affect the same properties. The results are unpredictable and unreliable and can cause performance problems.

Animations in Action

To see all of the WinJS animations in action, run the [HTML animation library sample](#). There are many different animations to illustrate, and this sample most certainly earns the award for the largest number of scenarios: twenty-two! In fact, the first thing you should do is go to Scenario 22 and see whether animations are enabled, as that will most certainly affect your experience with the rest of the same. The output of that scenario will show you whether the [UISettings.animationsEnabled](#) flag is set and allow you to increment or decrement the WinJS enablement count. So go check that now, because if you're like me (I dislike waiting for my task bar to animate up and down), you might have turned off system animations a long time ago for a snappier desktop experience. I didn't realize at first that it affected WinJS in this way!

Clearly, with 22 scenarios in the sample I won't be showing code for all of them here; indeed, doing so isn't necessary because many operate in the same way. The only real distinction is between those whose methods start with [create](#) and those that don't, as we'll see in a bit.

All the animation methods return a promise that you can use to take additional action when the animation is complete (at which point your completed handler will be called). If you already know something about CSS transitions and animations, you'll rightly guess that these promises encapsulate events like [transitionend](#) and [animationend](#), so you won't need to listen for those events directly if you

want to chain or synchronize animations. For chaining, you can just chain the promises; for synchronization, you can obtain the promises for multiple animations and wait for their completion using methods like `WinJS.Promise.join` or `WinJS.Promise.any`.

Animation promises also support the `cancel` method, which removes the animation from the element. This immediately sets the affected property values to their final states, causing an immediate visual jump to that end state. And whether you cancel an animation or it ends on its own, the promise is considered to have completed successfully; canceling an animation, in other words, will call the promise's completed handler and not its error handler.

Do be aware that because all of the WinJS animations are implemented with CSS, they won't actually start until you give control back to the UI thread. This means that you can set up multiple animations knowing that they'll more or less start together once you return from the function. So even though the animation methods return promises, they are not like other asynchronous operations in WinRT that start running on another thread altogether.

Anyway, let's look at some code! In the simplest case, all you need to do is call one of the animation methods and the animation will execute when you yield. Scenario 6 of the sample, for instance, just adds these handlers to the `MSPointerDown` and `MSPointerUp` events of three different elements (`js/pointerFeedback.js`):

```
function onPointerDown(evt) {
    WinJS.UI.Animation.pointerDown(evt.srcElement);
}

function onPointerUp(evt) {
    WinJS.UI.Animation.pointerUp(evt.srcElement);
}
```

We typically don't need to do anything when the animations complete, so there's no need for us to call `done` or provide a completed function. Truly, using many of these animations is just this simple.

The `crossFade` animation, for its part (Scenario 10), takes two elements: the incoming element and the outgoing element (all of which must be visible and part of the DOM throughout the animation, mind you!). Calling it then looks like this (`js/crossfade.js`):

```
WinJS.UI.Animation.crossFade(incoming, outgoing);
```

Yet this isn't the whole story. A common feature among the animations is that you can provide an *array* of elements on which to execute the same animation or, in the case of `crossFade`, two arrays of elements. While this isn't useful for animations like `pointerDown` and `pointerUp` (each pointer event should be handled independently), it's certainly handy for most others.

Consider the `enterPage` animation. In its singular form it accepts an element to animate and an optional initial offset where the element begins relative to its final position. (Generally speaking, you should omit this offset if you don't need it, because it will give result in better performance—the sample passes `null` here, which I've omitted in the code below.) `enterPage` can also accept a collection of elements, such as the result of a `querySelectorAll`. Scenario 1 (`html/enterPage.html` and

js/enterPage.js) provides a choice of how many elements are animated separately:

```
switch (pageSections) {
  case "1":
    // Animate the whole page together
    enterPage = WinJS.UI.Animation.enterPage(rootGrid);
    break;
  case "2":
    // Stagger the header and body
    enterPage = WinJS.UI.Animation.enterPage([[header, featureLabel], [contentHost,
      footer]]);
    break;
  case "3":
    // Stagger the header, input, and output areas
    enterPage = WinJS.UI.Animation.enterPage([[header, featureLabel],
      [inputLabel, input], [outputLabel, output, footer]]);
    break;
}
```

When the element argument is an array, the offset argument, if provided, can be either a single offset that is applied to all elements, or an array to indicate individual offsets for each element. Each offset is an object whose properties that define the offset. See js/dragBetween.js for Scenario 13 where this is used with the [dragBetweenEnter](#) animation:

```
WinJS.UI.Animation.dragBetweenEnter([box1, box2],
  [{ top: "-40px", left: "0px" }, { top: "40px", left: "0px" }]);
```

Here's a modification showing a single offset that's applied to both elements:

```
WinJS.UI.Animation.dragBetweenEnter([box1, box2], { top: "0px", left: "40px" });
```

Scenario 4 (js/transitioncontent.js) shows how you can chain a couple of promises together to transition between two different blocks of content:⁵⁵

```
WinJS.UI.Animation.exitContent(outgoing, null).done( function () {
  outgoing.style.display = "none";
  incoming.style.display = "block";
  return WinJS.UI.Animation.enterContent(incoming, null);
});
```

Things get a little more interesting when we look at the [create*](#) animation methods, together referred to as the *layout animations*, which are for adding and removing items from lists, expanding and collapsing content, and so forth. Each of these has a three-step process where you create the animation, manipulate the DOM, and then execute the animation, as shown in Scenario 7 (js/addAndDeleteFromList.js):

⁵⁵ Note that the actual sample passes a variable [output](#) as the first parameter to [exitContent](#) and [enterContent](#); the code should appear as shown here, with [outgoing](#) being passed to [exitContent](#) and [incoming](#) passed to [enterContent](#).


```
// Create addToList animation.
var addToList = WinJS.UI.Animation.createAddToListAnimation(newItem, affectedItems);

// Insert new item into DOM tree. This causes the affected items to change position.
list.insertBefore(newItem, list.firstChild);

// Execute the animation.
addToList.execute();
```

The reason for the three-step process is that in order to carry out the animation on newly added items, or items that are being removed, they all need to be in the DOM when the animation executes. The process here lets you create the animation with the initial state of everything, manipulate the DOM (or just set styles and so forth) to create the ending state, and then execute the animation to “let ‘er rip.” You can then use the `done` method on the promise returned from `execute` to perform any final cleanup. Scenario 5 (js/expandAndCollapse.js) makes this point clear:

```
// Create collapse animation.
var collapseAnimation = WinJS.UI.Animation.createCollapseAnimation(element, affected);

// Remove collapsing item from document flow so that affected items reflow to their new
// position. Do not remove collapsing item from DOM or display at this point, otherwise the
// animation on the collapsing item will not display.
element.style.position = "absolute";
element.style.opacity = "0";

// Execute collapse animation.
collapseAnimation.execute().done(
    // After animation is complete (or on error), remove from display.
    function () { element.style.display = "none"; },
    function () { element.style.display = "none"; }
);
```

As a final example—because I know you’re smart enough to look at most of the other cases on your own—Scenario 21 (js/customAnimation.js) shows how to use the `WinJS.UI.executeAnimation` and `WinJS.UI.executeTransition` methods.

```
function runCustomShowAnimation() {
    var showAnimation = WinJS.UI.executeAnimation(
        target,
        {
            // Note: this keyframe refers to a keyframe defined in customAnimation.css.
            // If it's not defined in CSS, the animation won't work.
            keyframe: "custom-opacity-in",
            property: "opacity",
            delay: 0,
            duration: 500,
            timing: "linear",
            from: 0,
            to: 1
        }
    );
}
```

```
function runCustomShowTransition() {
    var showTransition = WinJS.UI.executeTransition(
        target,
        {
            property: "opacity",
            delay: 0,
            duration: 500,
            timing: "linear",
            to: 1
        }
    );
}
```

If you want to combine multiple animations (as many of the WinJS animations do), note that both of these functions return promises so that you can combine multiple results with [WinJS.Promise.join](#) and have a single completed handler in which to do post-processing. This is exactly what WinJS does internally.

And if you know anything about CSS animations and transitions already, you can probably tell that the objects you pass to [executeAnimation](#) and [executeTransition](#) are simply shorthand expressions of the CSS styles you would use otherwise. In short, these methods give you an easy way to set up your own custom animations and transitions through the capabilities of CSS. Let's now look at those capabilities directly.

Sidebar: Parallax/Panorama Animations

Developers have often asked how to create a *parallax* or *panorama* background animation as seen on the Windows Start screen. If you're not familiar with this concept, go to the start screen and pan around a little, noticing how the background pans as well but slower than the tiles. This creates a sense of the tiles floating above the background.

While it is possible to implement this effect in JavaScript (see the [KidsBook example](#) on the Internet Explorer TestDrive site), we don't recommend it or at least recommend providing a setting to turn the effect off. At issue is the fact that the threading and rendering model of JavaScript results in choppy movement except on high-power devices; the effect will be very pronounced on low-power and especially ARM devices. In addition, such animations can be costly in terms of CPU and battery utilization.

This is one case in which using C++ and DirectX (or even C#/VB and XAML) has a clear advantage over JavaScript, and would be a consideration if you absolutely must have this effect in your app.

CSS Animations and Transitions

As noted before, many animation needs can be achieved through CSS rather than with JavaScript code running on intervals or animation frames. The WinJS Animations Library, as we've just seen, is entirely built on CSS. Using CSS relieves us from writing a bunch of code that worries about how much to move every element in every frame based on elapsed time and synchronized to the refresh rate. Instead, we can simply declare what we want to happen (perhaps using the [WinJS.UI.executeAnimation](#) and [WinJS.UI.executeTransition](#) helpers) and let the app host take care of the details. Delegation at its best! In this section, then, let's take a closer look at the capabilities of CSS for Windows Store apps.

Another huge benefit of performing animations and transitions through CSS—specifically those that affect only transform and opacity properties—is that they can be used to create what are called *independent animations* that run on a GPU thread rather than the UI thread. This makes them smoother and more power-efficient than *dependent animations* that are using the UI thread. Dependent animations happen when you create animations in JavaScript using intervals, use CSS animations and transitions with properties other than transform and opacity, or run animations on elements that are partly or wholly obscured by other elements.

We'll come back to this subject in a bit when we look at sample code. As I assume that you're already at least a little familiar with the subject, let's first review how CSS animations and transitions work. I say animations and transitions both because there are, in fact, two separate CSS specifications: [CSS animations](#) and [CSS transitions](#). So what's the difference?

Normally when a CSS property changes, its value jumps immediately from the old value to the new value, resulting in a sudden visual change. *Transitions* instruct the app host how to change one or more property values gradually, according to specific delay, duration, and timing curve parameters. All of this is declared within a specific style rule for an element (as well as `:before` and `:after` pseudo-elements) using four individual styles:

- `transition-property` (`transitionProperty` in JavaScript) Identifies the CSS properties affected by the transition (the transitionable properties are listed in section 7 of the transitions spec).
- `transition-duration` (`transitionDuration` in JavaScript) Defines the duration of the transition in seconds (fractional seconds are supported, as in `.125s`; negative values are normalized to `0s`).
- `transition-delay` (`transitionDelay` in JavaScript) Defines the delayed start of the transitions relative to the moment the property is changed, in seconds. If a negative value is given, the transition will appear to have started earlier but the effect will not have been visible.
- `transition-timing-function` (`transitionTimingFunction` in JavaScript) Defines how the property values change over time. The functions are `ease`, `linear`, `ease-in`, `ease-out`, `ease-in-out`, `cubic-bezier`, `step-start`, and `step-end`. The W3C spec has some helpful diagrams that explain these, but the best way to see the difference is to try them out in running code.

For example, a transition for a single property appears as so:

```
#div1 {  
  transition-property: left;  
  transition-duration: 2s;  
  transition-delay: .5s;  
  transition-timing-function: linear;  
}
```

When defining transitions for multiple properties, each value in each style is separated by a comma:

```
.class2 {  
  transition-property: opacity, left;  
  transition-duration: 1s, 0.25s;  
}
```

Again, transitions don't specify any actual beginning or ending property values—they define how the change actually happens *whenever* a new property is set through another CSS rule or through JavaScript. So, in the first case above, if `left` is initially 100px and it's set to 300px through a `:hover` rule, it will transition after 0.5 seconds from 100px to 300px over a period of 2 seconds. Doing the math, the visual movement with a `linear` timing function will run at 100px/second. Other timing functions will show different rates of movement at different points along the 2-second duration.

If a bit of JavaScript then sets the value to -200px—ideally after the first transition completes and fires its `transitionend` event—the value will again transition over the same amount of time but now from 300px to -200px (a total of 500px). As a result, the element will move at a higher speed (250px/second, again with the `linear` timing function) because it has more ground to cover for the same transition duration.

What's also true for transitions is that if you assign a style (e.g., `class2` above) to an element, nothing will happen until an affected property changes value. Changing a style like this also has no effect if a transition is already in progress. The exception is if you change the `transition-property` value, in which case that transition will stop. With this, it's important to note that the default value of this property is `all`, so clearing it (setting it to `""`) doesn't stop all transitions...it enables them! You instead need to set the property to `none`.

Note Elements with `display: none` do not run CSS animations and transitions at all, for obvious reasons. The same cannot be said about elements with `display: hidden`, `visibility: hidden`, `visibility: collapsed`, or `opacity: 0`, which means that hiding elements with some means other than `display: none` might end up running animations on nonvisible elements, which is a complete waste of resources. In short, use `display: none`.

Animations work in an opposite manner to transitions. Animations are defined separately from any CSS style rules but are then attached to rules. Assigning that style to an element then triggers the animation immediately. Furthermore, groups of affected properties are defined together in *keyframes* and are thus animated together.

A CSS animation, in other words, is an instruction to progressively update one or more CSS property values over a period of time. The values change from an initial state to a final state through various intermediate states defined by a set of keyframes. Here's an example (from Scenario 1 of the [HTML independent animations sample](#) we'll be referring to):

```
@keyframes move {  
  from { transform: translateX(0px); }  
  50% { transform: translateX(400px); }  
  to { transform: translateX(800px); }  
}
```

More generally:

- Start with `@keyframes <identifier>` where `<identifier>` is whatever name you want to assign to the keyframe (like *move* above). You'll refer to this identifier elsewhere in style rules.
- Within this keyframe, you create any number of *rule sets*, each of which represents a different snapshot of the animated element at different stages in the overall animation, demarked by percentages. The `from` and `to` keywords, as shown above, are simply aliases for 0% and 100%, respectively.
- Within each rule set you then define the desired value of *any number of style properties* (just `transform` in the example above), with each separated by a semicolon as with CSS styles. If a value for a property is the same as in the previous rule set, no animation will occur for that property. If the value is different, the rendering engine will animate the change between the two values of that property across the amount of time equivalent to `<overall animation time> * (<toPercentage> - <fromPercentage>)/100`. A timing function can also be specified for each rule set using the `animation-timing-function` style. For example:

```
50% { transform: translateX(400px); animation-timing-function: ease-in;}
```

One thing you'll notice here is that while the keyframe can indicate a timing function, it doesn't say anything about actual *timings*. This is left for the specific style rules that refer to the keyframe. In Scenario 1 of the sample, for instance:

```
.ball {  
  animation-name: move;  
  animation-duration: 2s;  
  animation-timing-function: linear;  
  animation-delay: 0s;  
  animation-iteration-count: infinite;  
  animation-play-state: running;  
}
```

Here, the `animation-name` style (`animationName` in JavaScript) identifies the keyframe to apply. The other `animation-*` styles then describe how the keyframe should execute:

- `animation-duration` (`animationDuration` in JavaScript) The duration of the animation in seconds (fractions allowed, of course). Negatives are the same as `0s`.
- `animation-timing-function` (`animationTimingFunction` in JavaScript) Defines, as with transitions, how the property values are interpolated over time—`ease` (the default), `linear`, `ease-in`, `ease-out`, `ease-in-out`, `cubic-bezier`, `step-start`, and `step-end`.
- `animation-delay` (`animationDelay` in JavaScript) Defines the number of seconds after which the animation will start when the style is applied. This can be negative, as with transitions, which will start the animation partway through its cycle.
- `animation-iteration-count` (`animationIterationCount` in JavaScript) Indicates how many times the animation will repeat (default is 1). This can be a number or `infinite`, as shown above.
- `animation-direction` (`animationDirection` in JavaScript) Indicates whether the animation should play `normal` (forward), `reverse`, `alternate` (back and forth), or `alternate-reverse` (back and forth starting with `reverse`). The default is `normal`.
- `animation-play-state` (`animationPlayState` in JavaScript) Allows you to play or pause an animation. The default state of `running` plays the animation; setting this to `paused` will pause it until you set the style back to `running`.
- `animation-fill-mode` (`animationFillMode` in JavaScript) Defines which property values of the named keyframe will be applied when the animation is not executing, such as during the initial delay or after it is completed. The default value of `none` applies the values of the `0%` or `from` rule set if the direction is `forward` and `alternate` directions; it applies those of the `100%` or `to` rule set if the direction is `reverse` or `alternate-reverse`. A fill mode of `backwards` flips this around. A fill mode of `forwards` always applies the `100%` or `to` values (unless the iteration count is zero, in which case it acts like `backwards`). The other option, both, is the same as indicating both `forwards` and `backwards`.
- `animation` (`animation` in JavaScript) The shorthand style for all of the above (except for play-state) in the order of name, duration, timing function, delay, iteration count, direction, and fill mode.

Applying a style that contains `animation-name` will trigger the animation for that element. This can happen automatically if the animation is named in a style that's applied by default. It can also happen on demand if the style is assigned to an element in JavaScript or if you set the `animation` property for an element.

Keyframes, while typically defined in CSS, can also be defined in JavaScript. The first step is to build up a string that matches what you'd write in CSS, and then you insert that string to the stylesheet. This is shown in Scenario 7 of the HTML independent animations sample ([js/scenario7.js](#)):

```
var styleSheet = document.styleSheets[1];
var element1 = document.getElementById("ballcontainer");
var animationString = '@keyframes bounce1 {'
    // ...
    + '}'

styleSheet.insertRule(animationString, 0);

window.setImmediate(function () {
    element1.style.animationName = 'bounce1';
});
```

Note how this code uses `setImmediate` to yield to the UI thread before setting the `animationName` property that will trigger the animation. This makes sure that other code that follows (not shown here) will execute first, as it does some other work the sample wants to complete before the animation begins.

More generally, it's good to again remember that CSS animations and transitions start only when you return from whatever function is setting them up. That is, nothing happens visually until you yield back to the UI thread and the rendering engine kicks in again, just like when you change nonanimated properties. This means you can set up however many animations and transitions as desired, and they'll all execute simultaneously. Using a callback with `setImmediate`, as shown above, is a simple way to say, "Run this code as soon as there is no pending work on the UI thread."⁵⁶ Such a pattern is typically for triggering one or more animations once everything else is set up.

As a final note for this section, you might be interested in [The Guide to CSS Animation: Principles and Examples](#) from Smashing Magazine. This will tell you a lot about animation design beyond just how CSS animations are set up in your code.

The Independent Animations Sample

Turning now to the [HTML independent animations sample](#), Scenario 1 gives a demonstration of an independent versus a dependent animation by eating some time on the UI thread (that is, blocking that thread) according to a slider. As a result, the top red ball (see image below) moves choppyly, especially as you increase the work on the UI thread by moving the slider. The green ball on the bottom, on the other hand, continues to move smoothly the whole time.

⁵⁶ For more on this topic, see <http://dvcs.w3.org/hg/webperf/raw-file/tip/specs/setImmediate/Overview.html>.



What's tricky to understand about this sample is that both balls use the same CSS style rule named *ball* that we saw in the previous section. In fact, just about everything about the two elements is exactly the same. So why does the movement of the red ball get choppy when additional work is happening on the UI thread?

The secret is in the `z-index: -1;` style on the red ball in `css/scenario1.css` (and a corresponding lack of `position: static` which negates `z-index`). For animations to run independently, they must be free of obstruction. This really gets into the subject of how layout is being composed within the HTML/CSS rendering engine of the app host, as an animating element that's somewhere in the middle of the z-order might end up being independent or dependent. The short of it is that the `z-index` style is the only lever that's available for you to pull here.

As I noted before, independent animations are limited to those that affect only the `transform` and `opacity` properties for an element. If you animate any property that affects layout, like `width` or `left`, the animation will run as dependent (and similar results can be achieved with a scaling and translation transform anyway). Other factors also affect independent animations, as described on the [Animating](#) topic in the documentation. For example, if the system lacks a GPU, if you overload the GPU with too many independent animations, or if the elements are too large, some of the animations will revert to dependent. This is another good reason to be purposeful in your use of animations—overusing them will produce a terrible user experience on lower-end devices, thereby defeating the whole point of using animations to enhance the user experience!

The other scenarios of the HTML Independent Animations sample lets you play with CSS transitions and animations by setting values within various controls and then running the animation. Scenarios 2 and 3 work with CSS transitions for 2D and 3D transforms, respectively, with an effect of the latter shown in Figure 11-2. As you can see, the element that the sample animates is the container for all the input controls! Scenarios 5 and 6 then let you do similar things with CSS animations. In all these cases, the necessary styles are set directly in JavaScript rather than using declarative CSS, so look in the `.js` files and not the `.css` files for the details.

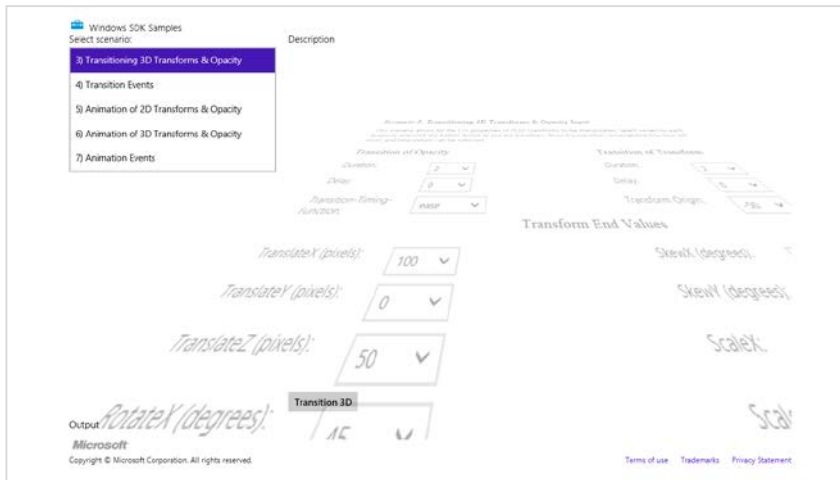


FIGURE 11-2 Output of Scenario 3 of the HTML independent animations sample.

Scenarios 4 and 7 then show something we haven't talked about yet, which are the few simple events that are raised for transitions and animations (and actually have nothing to do with independent versus dependent animations). In the former case, any element on which you execute a CSS transition will fire `transitionstart` and `transitionend` events. You can use these to chain transitions together.

With animations, there are three events: `animationstart` (which comes after any delay has passed), `animationend` (when the animation finished), and `animationiteration` (at the end of each iteration, unless `animationend` also fires are the same time). As with transitions, all of these can be used to chain animations or otherwise synchronize them. The `animationiteration` event is also helpful if you need to run a little code every time an animation finishes a cycle. In such a handler you might check conditions that would cause you to stop an animation, in which case you can set the `animationPlayState` to paused when needed.

Rolling Your Own: Tips and Tricks

If you're anything like me, I imagine that one of the first things you did when you started playing with JavaScript is to do some kind of animation: set up some initial conditions, create an timer with `setInterval`, do some calculations in the handler and update elements (or draw on a `canvas`), and keep looping until you're done. After all, this sort of thing is at the heart of many of our favorite games! (For an introductory discussion on this, just in case you haven't done this on your own yet, see [How to animate canvas graphics](#).)

As such, there is considerable wisdom available in the community on this subject if you decide to go this route. I put it this way because by now, having looked at the WinJS animations library and the capabilities of CSS, you should be in a good position to decide whether you actually *need* to go this route at all. Some people have estimated that a vast majority of animations needed by most apps can

be handled entirely through CSS: just set a style and let the app host do the rest. But if you decide that you still need to do low-level animation, the first thing you should do is ask yourself this question:

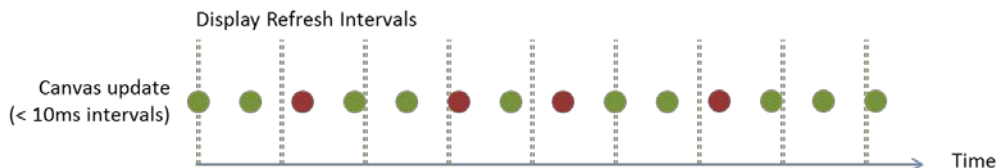
What is the appropriate animation interval?

This is a very important question because oftentimes developers have no idea what kind of interval to use for animation. It's not so much of an issue for long intervals, like 500ms or 1s, but developers often just use 10ms because it seems "fast enough."

To be honest, 10ms is overkill for a number of reasons. 60 frames per second (fps)—an animation interval of 16.7ms—is about the best that human beings can even discern and is also the best that most displays can even handle in the first place. In fact, the best results are obtained when your animation frames are synchronized with the screen refresh rate.

Let's explore this a little more. Have you ever looked at a screen while eating something really crunchy, and noticed how the pixels seem to dance all over the place? This is because display devices aren't typically just passive viewports onto the graphics memory. Instead, displays (even LCD and LED displays) typically cycle through graphics memory at a set refresh rate, which is most commonly 60Hz or 60fps (but can also be as low as 50Hz or as high as 100Hz).

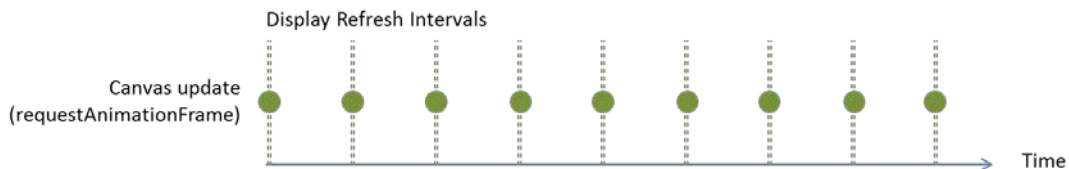
This means that trying to draw animations at an interval faster than the refresh rate is a waste of time, is a waste of power (it has been shown to reduce battery life by as much as 25%!), and results in dropped frames. The latter point is illustrated below, where the red dots are frames that get drawn on something like a `canvas` but never make it to the screen because another frame is drawn before the screen refreshes:



This is why it's common to animate on multiples of 16.7ms using `setInterval`. However, using 16.7 assumes a 60Hz display refresh, which isn't always the case. The right solution, then, for Windows Store apps in JavaScript and web apps both, is to use `requestAnimationFrame`. This API simply takes a function to call for each frame:

```
requestAnimationFrame(renderLoop);
```

You'll notice that there's not an interval parameter; the function rather gives you a way to align your frame updates with display refreshes so that you draw only when the system is ready to display something:



What's more, `requestAnimationFrame` also takes page visibility into account, meaning that if you're not visible (and animations are thus wasteful), you won't be asked to render the frame at all. This means you don't need to handle page visibility events yourself to turn animations on and off: you can just rely on the behavior of `requestAnimationFrame` directly.

Tip If you really want an optimized display, consider doing all drawing work of your app (not just animations) within a `requestAnimationFrame` callback. That is, when processing a change, as in response to an input event, update your data and call `requestAnimationFrame` with your rendering function rather than doing the rendering immediately. And always be mindful to redraw only when you need to redraw, as we'll see in a moment, to make the best use of CPU and battery power.

It's also good to know that attempting to animate a `canvas` that's partly obscured by an element with `display: inline-block` has been found to result in very poor performance and large gaps between frames because of excessive region invalidation. Using a different display model such as `table-cell` avoids this issue.

Calling this method once will invoke your callback for a single frame. To keep up a continuous animation, your handler should call `requestAnimationFrame` again. This is shown in the [Using requestAnimationFrame for power efficient animations sample](#) (this wins second place for long sample names!), which draws and animates a clock with a second hand:



The first call to `requestAnimationFrame` happens in the page's `ready` method, and then the callback refreshes the request (`js/scenario1.js`):

```
window.requestAnimationFrame(renderLoopRAF);

function renderLoopRAF() {
  drawClock();
  window.requestAnimationFrame(renderLoopRAF);
}
```

where the `drawClock` function gets the current time and calculates the angle at which to draw the clock hands:

```
function drawClock() {  
  // ...  
  
  // Note: this is modified from the sample to only create a Date once, not each time  
  var date = new Date();  
  var hour = date.getHours();  
  var minute = date.getMinutes();  
  var second = date.getSeconds();  
  
  // ...  
  
  var sDegree = second / 60 * 360 - 180;  
  var mDegree = minute / 60 * 360 - 180;  
  var hDegree = ((hour + (minute / 60)) / 12) * 360 - 180;  
  
  // Code to use the context's translate, rotate, and drawImage methods  
  // to render each clock hand  
}
```

Here's a challenge for you: *What's wrong with this code?* Run the sample and look at the second hand. Then think about how `requestAnimationFrame` aligns to screen refresh cycles with an interval like 16.7ms. What's wrong with this picture?

What's wrong is that even though the second hand is moving visibly only *once per second*, the `drawClock` code is actually executing nearly *50, 60, or 100 times more frequently* than that! Thus the "Efficient and Smooth Animations" title that the sample shows on screen is anything but! Indeed, if you run Task Manager, you can see that this simple "efficient" clock is ironically consuming 15–20% of the CPU. Yikes!

 Efficient Animations JS sample	20.2%	39.6 MB	0 MB/s	0 Mbps
--	-------	---------	--------	--------

Remember that an interval aligned with ~16.7ms screen refreshes (on a 60Hz display) implies 60fps rendering; if you don't need that much, you should skip frames yourself according to elapsed time, thereby saving power, and not blindly redraw as this sample is doing. In fact, if all we need is a once-per-second movement in a clock like this, repeated calls to `requestAnimationFrame` is sheer overkill. We could instead use `setInterval(function () { requestAnimationFrame(drawClock) }, 1000)` to coordinate 1s intervals with screen refreshes. If you make this change in the `ready` method, for example, the CPU usage will drop precipitously:

 Efficient Animations JS sample	0.5%	39.6 MB	0 MB/s	0 Mbps
--	------	---------	--------	--------

But let's say we actually want to put 60fps animation and 20% of the CPU to good use—in that case, we should at least make the clock's second hand move smoothly, which can be done by simply adding milliseconds into the angle calculation:

```
var second = date.getSeconds() + date.getMilliseconds() / 1000;
```

Still, 20% is a lot of CPU power to spend on something so simple and 60fps is still serious overkill. ~10fps is probably sufficient for good effect. In this case we can calculate elapsed time within `renderLoopRAF` to call `drawClock` only when 0.1 seconds have passed:

```
var lastTime = 0;

function renderLoopRAF() {
  var fps = 10; // Target frames per second
  var interval = 1000 / fps;
  var curTime = Math.floor(Date.now() / interval);

  if (lastTime !== curTime) {
    lastTime = curTime;
    drawClock();
  }

  requestAnimationFrame(renderLoopRAF);
}
```

That's not quite as smooth—10fps creates the sense of a slight mechanical movement—but it certainly has much less impact on the CPU:

 Efficient Animations JS sample	5.2%	39.8 MB	0 MB/s	0 Mbps
--	------	---------	--------	--------

I encourage you to play around with variations on this theme to see what kind of interval you can actually discern with your eyes. 10fps and 15fps give a sense of mechanical movement; at 20fps I don't see much difference from 60fps at all, and the CPU is running at about 7–10%. You might also try something like 4fps (quarter-second intervals) to see the effect. In this chapter's companion content I've included a variation of the original sample where you can select from various target rendering rates.

The other thing you can do in the modified sample is draw the hour and minute hand at fractional angles. In the original code, the minute hand will move suddenly when the second hand comes around to the top. Many analog clocks don't actually work this way: their complex gearing moves both the hour and the minute hand ever so slightly with every tick. To simulate that same behavior, we just need to include the seconds in the minutes calculation, and the resulting minutes in the hours, like so:

```
var second = date.getSeconds() + date.getMilliseconds() / 1000;
var minute = date.getMinutes() + second / 60;
var hour = date.getHours() + minute / 60;
```

In real practice, like a game, you'd generally want to avoid just running a continuous animation loop like this: if there's nothing moving on the screen that needs animating (for which you might be using `setInterval` as a timer) and there are no input events to respond to, there's no reason to call `requestAnimationFrame`. Also, be sure when the app is paused that you stop calling `requestAnimationFrame` until the animation starts up again. (You can also use `cancelAnimationFrame` to stop one you've already requested.) The same is true for `setTimeout` and `setInterval`: don't generate unnecessary calls to your callback functions unless you really need to do the animation. For this, use the `visibilitychange` event to know if your app is visible on screen. While `requestAnimationFrame` takes visibility into account (the sample's CPU use will drop to 0% before it is suspended), you need to do this

on your own with `setTimeout` and `setInterval`.

In the end, the whole point here is that really understanding the animation interval you need (that is, your frame rate) will help you make the best use of `requestAnimationFrame`, if that's needed, or `setInterval`/`setTimeout`. They all have their valid uses to deliver the right user experience with the right level of consumption of system resources.

Did you know? One change for Windows 8 and Internet Explorer 10 is that `setTimeout` and `setInterval`, along with `setImmediate`, all support including parameters you can pass to the callback functions?

What We've Just Learned

- In the desktop control panel, users can elect to disable most (that is, nonessential) animations. Apps should honor this, as does WinJS, by checking the `Windows.UI.ViewManagement.UISettings.animationsEnabled` property.
- The WinJS animations library has many built-in animations that embody the Windows 8 personality. These are highly recommended for apps to use for the scenarios they support, such as content and page transitions, selections, list manipulation, and others.
- All WinJS animations are built using CSS and thus benefit from hardware acceleration. When the right conditions are met, such animations run in the GPU and are thus not affected by activity on the UI thread.
- Apps can also use CSS animations and transitions directly, according to the W3C specifications.
- Apart from WinJS and CSS, apps can also use functions like `setInterval` and `requestAnimationFrame` to implement direct frame-by-frame animation. The `requestAnimationFrame` method aligns frames with the display refresh rate, leading to the best overall performance.

Chapter 12

Contracts

Recently I discovered a delightfully quirky comedy called *Interstate 60* that is full of delightfully quirky characters. One of them, played by Chris Cooper, is a former advertising executive who, having discovered he was terminally ill with lung cancer, decided to make up for a career built on lies by encouraging others to be more truthful. As such, he was very particular about agreements and contracts, especially those in writing.

We really get to see the character's quirkiness in a scene at a gas station. He's approached by a beggar with a sign, "Will work for food." Seeing this, he offers the man an apple in exchange for cleaning his car's windshield. But when the man refuses to honor the written contract on his sign, Cooper's character gets increasingly upset over the breach...to the point where he announces his terminal illness, rips open his shirt, and reveals the dynamite wrapped around his body and the 10-second timer that's already counting down!

In the end, he drives away with a clean windshield and the satisfaction of having helped someone—in his delightfully quirky way—to fulfill their part of a written contract. And he reappears later in the movie in a town that's 100% populated with lawyers; I'll leave it to you to imagine the result, or at least enjoy the film.

Setting the dynamite aside, agreements between two parties are exceptionally important in a civil society as they are in a well-running computer system. Agreements are especially important where apps provide extensions to the system and where apps written by different people at different points in time cooperate to fulfill certain tasks.

Such is the nature of various contracts within Windows 8, which as a whole is perhaps one of the most powerful features of the entire system. The overarching purpose of contracts has been described as "launching apps for a purpose and with context." That is, instead of just starting apps in isolation, contracts make it possible to start them in relationship to other apps and in the context of those other apps. Information can then be shared directly between those apps for a real purpose, rather than through the generic intermediary of the file system where such context is lost.

With any given contract, one party is the consumer or receiver of information that's managed through the contract. The other party is then the source or provider of that information. The contract itself is then generic: neither party needs any specific knowledge of the other, just knowledge of their side of the contract. It might not sound like much, but what this allows is a degree of extensibility that gets richer and richer as more apps that support those contracts are added to the system. I figure that when users really start to experience what these contracts provide, they'll more and more look for and choose apps from the Windows Store that use contracts to enrich their system and create increasingly powerful user experiences.

Within the apps themselves, consuming contracts typically happens through an API call, such as the file pickers, or is already built into the system through UI like the Charms bar. *Providing* information for a contract is often the more interesting part, because an app needs to respond to specific events (when running), or announce the capability through its manifest and then handle different contract activations.

The table below summarizes all the contracts and other extensions in Windows 8 (in alphabetical order), some of which serve to allow apps to work together while others serve to allow apps to extend system functionality. Full descriptions can be found on [App contracts and extensions](#). Those that are covered in this chapter are colored in green: share, search, file type and URI scheme associations, file pickers, cached file updater, and contacts (people). Others contracts have been or will be covered in the chapters indicated, and a few I've simply left for you to explore through the linked documentation and samples.

Tip For a comparison of the different options for exchanging data—the share contract, the clipboard, and the file save picker contract—see [Sharing and exchanging data](#) on the Windows Developer Center. It outlines different scenarios for each option and when you might implement more than one in the same app.

Also note that there are many WinRT events involved in these different contracts, so be mindful of the need to call `removeEventListener` as described in Chapter 3, “App Anatomy and Page Navigation,” in the section “WinRT Events and `removeEventListener`.”

Contract/Extension	Provider	Consumer	Description, Documentation, and Samples
Account picture provider (Chapter 14)	Apps that can take a picture	Windows (account picture)	When user changes an account picture, they can either select an existing one or acquire a new one from a provider; see Account picture name sample .
AutoPlay (Chapter 15)	Apps that want to be listed as an AutoPlay option	Windows	See Auto-launching with AutoPlay and the Removable storage sample .
Background tasks (Chapters 13 and 14)	Apps that have background tasks	Windows	Allows apps to run small tasks in the background (that is, when otherwise suspended or not running) without user interaction. See Introduction to background tasks as well as Chapter 13. Background file transfers are a special case supported by specific APIs; see Transferring data in the background and Chapter 14.
Cached file updater	Apps that provide access to their data through file pickers and want to synchronize updates	Apps using the file picker API and the file APIs to manage them	Provider apps can allow the consumer to maintain a cached copy of a file. Through this contract, the provider can manage updates between the local copy and the source copy. See Integrating with file picker contracts .
Camera settings	Apps with custom camera UI	Windows Camera Capture UI	See Developing Windows 8 device apps for cameras .
Contact picker	Apps that manage contact data (like an address book)	Apps that use the contact picker API (like email)	Launches an app to provide a list of possible contacts to select. See Managing user contacts .

File activation (file type association)	Apps that can open files of a particular type	Windows Explorer and apps that use the launcher API	Launches an app to open/service a file when needed. See How to handle file activation and Auto-launching with file and URI associations .
File open picker/file save picker	App with data that can appear as files to other apps for opening and/or saving (two separate contracts).	Apps using the file picker API (also certain Windows features)	Makes data that is otherwise hidden inside and managed by apps appear as if they were part of the file system. See Integrating with file picker contracts .
Game explorer	Game apps with a Game Definition File	Windows (parental controls)	Manages age ratings for games. See Creating a GDF file .
Play To (Chapter 10)	Apps that can play media to a DLNA device	Windows (Devices charm > Connect)	See Streaming media to devices using Play To .
Print task settings	Printer device apps	Windows (Device charm > Print)	See Developing Windows 8 device apps for printers .
Protocol activation (URI scheme association)	Apps that can open URIs that begin with a particular URI scheme	Windows Explorer and apps that use the launcher API	Launches an app to open/service a URI when needed. See How to handle protocol activation and Auto-launching with file and URI associations .
Search	Apps with searchable data	Windows (Search charm)	Provides the ubiquitous ability to search any app from anywhere. See Adding search to an app .
Settings (Chapter 8)	Apps with settings	Windows (Settings charm)	Provides a standard place for app settings. See Adding app settings .
Share	Apps with sharable data	Apps that can receive data to incorporate into itself or share to a service	Provides a linkage of data transfer between apps so that source apps don't need to be particularly aware of individual targets like Facebook, Twitter, etc. See Adding share .
SSL/certificates	Apps that need to install a certificate	Apps needing to supply a certificate to another service	See Encrypting data and working with certificates .

Share

Though Search appears at the top of the Charms bar, the first contract I want to look at in depth is Share—after all, it's one of the first things you learn as a child! In truth, I'm starting with Share because we've already seen the source side of the story starting back in Chapter 2, "Quickstart," with the Here My Am! app, and our coverage here will also include a brief look at the age-old clipboard at the end of this section.

Here's what we've already learned about Share, with the more complete process illustrated in Figure 12-1:

- An app with sharable content listens for the `datarequested` event from the object returned by `Windows.ApplicationModel.DataTransfer.DataTransferManager.GetForCurrentView()`. This WinRT event (for which you should be mindful of using `removeEventListener`) is fired whenever the user invokes the Share charm. Note that if an app doesn't listen for this event at all, the Share charm will show a default "unable to share" message (one that is certain to be disappointing to users!).

- In its event handler, the app determines whether it has anything to share in its current state. If it does, it populates the `Windows.ApplicationModel.DataTransfer.DataPackage` provided in the event. (This can vary with the selection or lack thereof; if the user needs to make a selection for share to work, the app can display a message to that effect.)
- Based on the data formats in the package, Windows—that is, the *share broker* who manages the contract—determines the share target apps to display to the user. The user can also control which apps are shown through Change PC Settings > Share.
- When the user picks a target, the associated app is activated and receives the data package to process however it wants.

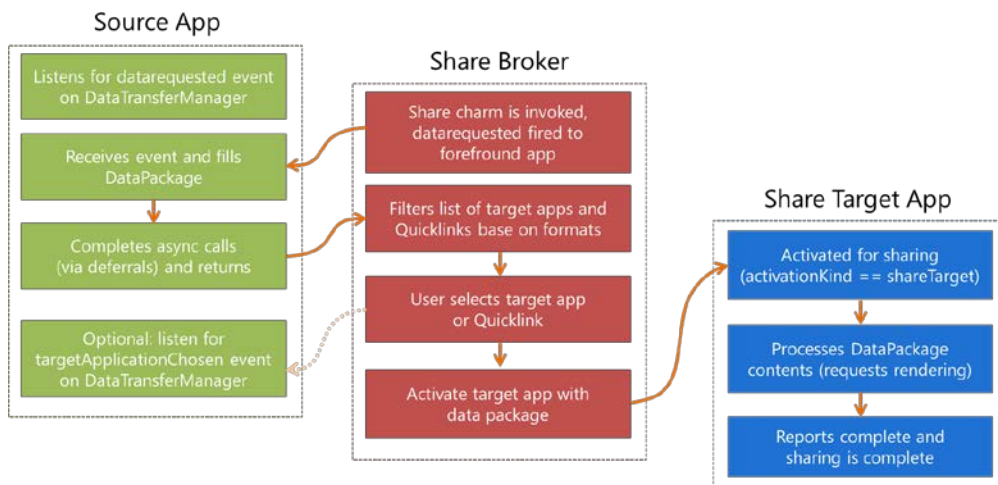


FIGURE 12-1 Processing the Share contract as initiated by the user's selection of the Share charm.

This whole process provides a very convenient shortcut for users to take something they love in one app and get it into another app with a simple edge gesture and target app selection. It's like a semantically rich clipboard in which you don't have to figure out how to get connected to other apps. What's very cool about the Share contract, in other words, is that the source doesn't have to care what happens to the data—its only role is to provide whatever data is appropriate for sharing at the moment the user invokes the Share charm (if, in fact, there is appropriate data—sometimes there isn't). This liberates source apps from the burden of having to predict, anticipate, or second-guess what users might want to do with the data. Perhaps they want to email it, share it via social networking, drop it into a content management app...who knows?

Well, only the user knows, so what the share broker does with that data is let the user decide! Given the data package from the source, the broker matches the formats in that package to target apps that have indicated support for those formats in their manifests. The broker then displays that list to the user. That list can contain apps, for one, but also something called a quicklink (a `Windows.ApplicationModel.DataTransfer.ShareTarget.Quicklink` object, to be precise), which is serviced by some app but is much

more specific. For instance, when an email app is shown as an option for sharing, the best it can do is create a new message with no particular recipients. A quicklink, however, can identify specific email addresses, say, for a person or persons you email frequently. The quicklink, then, is essentially an app plus specific configuration information.

Whatever the case, some app is launched when the user selects a target. With the Share contract, the app is launched with an activation kind of `shareTarget`. This tells it to *not* bring up its default UI but to rather show a specific share pane (with light-dismiss behavior) in which the user can refine exactly what is being shared and how. A share target for a social network, for instance, will often provide a place to add a comment on the shared data before posting it. An email app would provide a means to edit the message before sending it. A front-end app for a photo service could allow for adding a caption, specifying a location, identifying people, and so on. You get the idea. All of this combines together to provide a smooth flow from having something to share to an app that facilitates the sharing and allows the user to add customizations.

Overall, then, the Share contract gets apps connected to one another for this common purpose without either one having to know anything about the other. This creates a very extensible and scalable experience: since all the potential target choices appear only in the Share charm pane, they never need to clutter a source app as we see happening on many web pages. This is the “content before chrome” design principle in action.

Source apps also don’t need to update themselves when a new target becomes popular (e.g., a new kind of social network): all that’s needed is a single target app. As for those target apps, they don’t have to evangelize themselves to the world: through the contract, source apps are automatically able to use any target apps that come along in the future. And from the end user’s point of view, their experience of the Share charm gets better and better as they acquire more Share-capable apps.

At the same time, it is possible for the source app to know something about how its shared data is being used. Alongside the `dataRequested` event, the `DataTransferManager` also fires a `targetApplicationChosen` event to those sources who are listening. The `eventArgs` in this case contain only a single property: `applicationName`. This isn’t really useful for any other WinRT APIs, mind you, but is something you can tally within your own analytics. Such data can help you understand whether you’d provide a better user experience by sharing richer data formats, for example, or, if common target apps also support custom formats that you can support in future updates.

Source Apps

Let’s complete our understanding of source apps now by looking at a number of details we haven’t fully explored yet, primarily around how the source populates the data package and the options it has for handling the request. For this purpose, I suggest you obtain and run both the [Sharing content source app sample](#) and the [Sharing content target app sample](#). We’ll be looking at both of these, and the latter provides a helpful way to see how a target app will consume the data package created in the source.

The source app sample provides a number of scenarios that demonstrate how to share different types of data. They also show how to programmatically invoke the Share charm. This isn't typically recommended, but it is possible. If it really fits in your app scenario, here's how:

```
Windows.ApplicationModel.DataTransfer.DataTransferManager.ShowShareUI();
```

Calling this will, as when the user invokes the charm, trigger the `datarequested` event where `eventArgs.request` object is a [Windows.ApplicationModel.DataTransfer.DataRequest](#) object. This request object contains two properties and two methods:

- `data` is the [DataPackage](#) to populate. It contains methods to make various data formats available, though it's important to note that not all formats will be immediately rendered. Instead, they're rendered only when a share target asks for them.
- `deadline` is a [Date](#) property indicating the time in the future when the data you're making available will no longer be valid (that is, will not render). This recognizes that there might be an indeterminate amount of time between when the source app is asked for data and when the target actually tries to use it. With delayed rendering, as noted above for the `data` property, it's possible that some transient source data might disappear after some time. By indicating that time in `deadline`, rendering requests that occur past the deadline will be ignored.
- `failWithDisplayText` is a method to tell the share broker that sharing isn't possible right now, along with a string that will tell the user why (perhaps the lack of a usable selection). You call this when you don't have appropriate data formats or an appropriate selection to share, or if there's an error in populating the data package for whatever reason. The text you provide will then be displayed in the Share charm (and thus should be localized). Scenario 8 of the source app sample shows the use of this in the simple case when don't provide data in response to the `datarequested` event..
- `getDeferral` provides for async operations you might need to perform while populating the data package (just like other deferrals elsewhere in the WinRT API). Do note that `datarequested` has a 200ms timeout period, after which time the Share charm will display "can't share right now". Requesting a deferral does not change that timeout; it only prevents `datarequested` from assuming that the data package is ready once you return from your handler.

The basic structure of a `datarequested` handler, then, will attempt to populate the minimal properties of `eventArgs.request.data` and call `eventArgs.request.failWithDisplayText` when errors occur. We see this structure in most of the scenarios in the sample:

```
var dataTransferManager =  
    Windows.ApplicationModel.DataTransfer.DataTransferManager.GetForCurrentView();  
// Remove this listener as required  
dataTransferManager.AddEventListener("datarequested", dataRequested);  
  
function dataRequested(e) {  
    var request = e.request;  
  
    // Title is required
```

```

var dataPackageTitle = document.getElementById("titleInputBox").value;

if ( /* Check if there is appropriate data to share */ ) {
    request.data.properties.title = dataPackageTitle;

    // The description is optional.
    var dataPackageDescription = document.getElementById("descriptionInputBox").value;
    request.data.properties.description = dataPackageDescription;

    // Call request.data.setText, setUri, setBitmap, setData, etc.
} else {
    request.failWithDisplayText(/* Error message */ );
}
}

```

As we see here, the `request.data.properties` object (of type [DataPackagePropertySet](#)) is where you set things like a title and description for the data package. Other properties are as follows:

- `applicationListingUri` The URI of your app's page in the Windows Store, which should be set to the return value of [Windows.ApplicationModel.Store.CurrentApp.linkUri](#)⁵⁷)
- `applicationName` A string, which helps share targets gather the same kind of information that the source can obtain from the `targetApplicationChosen` event
- `fileTypes` A string vector, where strings should come from the [StandardDataFormats](#) enumeration but can also be custom formats
- `size` The number of items when the data in the package is a collection, e.g., files
- `thumbnail` A stream containing a thumbnail image; obtaining this image is typically why you'd use the `DataRequest.getDeferral` method).

Beyond this the `data.properties` object also supports custom properties through its `insert`, `remove`, and other methods. This makes it possible for the source app to pass custom properties along with custom formats, making all of this extensible if new data formats are widely adopted in the future.

Within this code structure above, the primary job of the source app is to populate the data package by calling the package's various `set*` methods. For standard formats, which are again those described in the `StandardDataFormats` enumeration, there are discrete methods: `setText`, `setUri`, `setHtmlFormat`, `setRtf` (rich text format, a comparably ancient precursor to HTML), `setBitmap`, and `setStorageItems` (for files and folders). All of these except for `setRtf` are represented in the source app sample as follows.

⁵⁷ This URI along the `applicationName` would allow a target to indicate where the data originally came from, especially for scenarios where the data goes to a social network, in an email message, or elsewhere off the device with the source app. This way, recipients can be invited to acquire the source app themselves, so source apps will probably want to include it. You always want to use the `Windows.ApplicationModel.Store.CurrentApp.linkUri` property to populate this field because you won't know your URI until your completed app is uploaded to the Store the first time.

Sharing text—Scenario 1 (js/text.js):

```
var dataPackageText = document.getElementById("textInputBox").value;
request.data.setText(dataPackageText);
```

Sharing a link—Scenario 2 (js/link.js), which can be used for local and remote content alike:

```
request.data.setUri(new Windows.Foundation.Uri(document.getElementById("linkInputBox").value));
```

Sharing an image and a storage item—Scenario 3 (js/image.js):

```
var imageFile; // A StorageFile obtained through the file picker

// In the data requested event
var streamReference =
    Windows.Storage.Streams.RandomAccessStreamReference.createFromFile(imageFile);
request.data.properties.thumbnail = streamReference;

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

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

// The setBitmap method requires a RandomAccessStreamReference
request.data.setBitmap(streamReference);
```

Sharing files—Scenario 4 (js/file.js):

```
var selectedFiles; // A collection of StorageFile objects obtained through the file picker

// In the data requested event
request.data.setStorageItems(selectedFiles);
```

As for sharing HTML, this can be quite simple if you just have HTML in a string:

```
request.data.setHtmlFormat(someHtml);
```

For this purpose you might find the [Windows.ApplicationModel.DataTransfer.-HtmlFormatHelper](#) object, well, helpful, as it provides methods to build properly formatted markup. What's also true with HTML is that it often refers to other content like images that aren't directly contained in the markup. So how do you handle that? Fortunately, the designers of this API thought through this need: you employ the data package's `resourceMap` property to associate relative URIs in the HTML with an image stream. We see this in Scenario 6 of the sample (js/html.js):

```
var path = document.getElementById("htmlFragmentImage").getAttribute("src");
var imageUri = new Windows.Foundation.Uri(path);
var streamReference =
    Windows.Storage.Streams.RandomAccessStreamReference.createFromUri(imageUri);
request.data.resourceMap[path] = streamReference;
```

The other interesting part of Scenario 6 is that it actually replaces the data package in the `eventArgs` with a new one that it creates as follows:

```
var range = document.createRange();
range.selectNode(document.getElementById("htmlFragment"));
request.data = MSApp.createDataPackage(range);
```

As you can see, the `MSApp.createDataPackage` method takes a DOM range (in this case a portion of the current page) and creates a data package from it, where the package's `setHtmlFormat` method is called in the process (which is why you don't see that method called explicitly in Scenario 6). For what it's worth, there is also `MSApp.createDataPackageFromSelection` that does the same job with whatever is currently selected in the DOM. You would obviously use this if you have editable elements on your page from which you'd like to share.

Sharing Multiple Data Formats

As shown in Scenario 3, it is certainly allowable—and *encouraged!*—to share data in as many formats as makes sense, thereby enabling more potential targets. All this means is that you call all the `set*` methods that make sense within your `datarequested` handler. This includes calling `setData` for custom formats and `setDataProvider` for deferred rendering, as described in the next two sections.

Custom Data Formats: schema.org

Long ago, I imagine, API designers decided it was an exercise in futility to try to predict every data format that apps might want to exchange in the future. The WinRT API is no different, so alongside the format-specific `set*` methods of the `DataPackage` we find the generic `setData` method. This takes a format identifier (a string) and the data to share. This is illustrated in Scenario 7 of the sample using the format `"http://schema.org/Book"` and data in a JSON string (`js/custom.js`):

```
request.data.setData(dataFormat, JSON.stringify(book));
```

Since the custom format identifier is just a string, you can literally use anything you want here; a very specific format string might be useful, for example, in a sharing scenario where you want to target a very specific app, perhaps one that you authored yourself. However, unless you're very good at evangelizing your custom formats to the rest of the developer community (and have a budget for such!), chances are that other share targets won't have any clue what you're talking about.

Fortunately, there is a growing body of conventions for custom data formats maintained by <http://schema.org>. This site is the point of agreement where custom formats are concerned, so we highly recommend that you draw formats from it. See <http://schema.org/docs/schemas.html> for a complete list.

Here's the JSON book data used in the sample:

```
var book = {
  type: "http://schema.org/Book",
  properties: {
```

```

        image: "http://sourceuri.com/catcher-in-the-rye-book-cover.jpg",
        name: "The Catcher in the Rye",
        bookFormat: "http://schema.org/Paperback",
        author: "http://sourceuri.com/author/jd_salinger.html",
        numberOfPages: 224,
        publisher: "Little, Brown, and Company",
        datePublished: "1991-05-01",
        inLanguage: "English",
        isbn: "0316769487"
    }
};

```

You can easily express this same data as plain text, as HTML (or RTF), as a link (perhaps to a page with this information), and an image (of the book cover). This way you can populate the data package with all the standard formats alongside specific custom formats.

Deferrals and Delayed Rendering

Deferrals, as mentioned before, are a simple mechanism to delay completion of the `datarequested` event until the deferral's `complete` method is called within an async operation's completed handler. The documentation for [DataRequest.getDeferral](#) shows an example of using this when loading an image file:

```

var deferral = request.getDeferral();

Windows.ApplicationModel.Package.current.installedLocation.getFileAsync(
    "images\\smalllogo.png")
    .then(function (thumbnailFile) {
        request.data.properties.thumbnail = Windows.Storage.Streams.
            RandomAccessStreamReference.createFromFile(thumbnailFile);
        return Windows.ApplicationModel.Package.current.installedLocation.getFileAsync(
            "images\\logo.png");
    })
    .done(function (imageFile) {
        request.data.setBitmap(
            Windows.Storage.Streams.RandomAccessStreamReference.createFromFile(imageFile));
        deferral.complete();
    });

```

Delayed rendering is a different matter, though the process typically employs the deferral. The purpose here is to avoid rendering the shared data until a target actually requires it, sometimes referred to as a *pull operation*. The `set*` methods that we've seen so far all copy the full data into the package. Delayed rendering means calling the data package's [setDataProvider](#) method with a data format identifier and a function to call when the data is needed. Here's how it's done in Scenario 5 of the source app sample where `imageFile` is selected with a file picker (`js/image.js`):


```
// When sharing an image, don't forget to set the thumbnail for the DataPackage
var streamReference =
    Windows.Storage.Streams.RandomAccessStreamReference.createFromFile(imageFile);
request.data.properties.thumbnail = streamReference;
request.data.setDataProvider(
    Windows.ApplicationModel.DataTransfer.StandardDataFormats.bitmap,
    onDeferredImageRequested);
```

As indicated in the comments, it's a really good idea to provide a thumbnail with delayed rendering so the target app has something to show the user. Then, when the target needs the full data, the data provider function gets called—in this case, `onDeferredImageRequested`:

```
function onDeferredImageRequested(request) {
    if (imageFile) {
        // Here we provide updated Bitmap data using delayed rendering
        var deferral = request.getDeferral();

        var imageDecoder, inMemoryStream;

        imageFile.openAsync(Windows.Storage.FileAccessMode.read).then(function (stream) {
            // Decode the image
            return Windows.Graphics.Imaging.BitmapDecoder.createAsync(stream);
        }).then(function (decoder) {
            // Re-encode the image at 50% width and height
            inMemoryStream = new Windows.Storage.Streams.InMemoryRandomAccessStream();
            imageDecoder = decoder;
            return Windows.Graphics.Imaging.BitmapEncoder.createForTranscodingAsync(
                inMemoryStream, decoder);
        }).then(function (encoder) {
            encoder.bitmapTransform.scaledWidth = imageDecoder.orientedPixelWidth * 0.5;
            encoder.bitmapTransform.scaledHeight = imageDecoder.orientedPixelHeight * 0.5;
            return encoder.flushAsync();
        }).done(function () {
            var streamReference = Windows.Storage.Streams.RandomAccessStreamReference
                .createFromStream(inMemoryStream);
            request.setData(streamReference);
            deferral.complete();
        }, function (e) {
            // didn't succeed, but we still need to release the deferral to avoid
            // a hang in the target app
            deferral.complete();
        });
    }
}
```

Note that this function receives a simplified hybrid of the `DataRequest` and `DataPackage` objects: a `DataProviderRequest` that contains `deadline` and `formatId` properties, a `getDeferral` method, and a `setData` method through which you provide the data that matched `formatId`. The `deadline` property, as you can guess, is the same as what the `daterequested` handler might have stored in the `DataRequest` object.

Target Apps

Looking back to Figure 12-1, we can see that while the interaction between a source app and the share broker is driven by the single `datarequested` event, the interaction between the broker and a target app is a little more involved. For one, the broker needs to determine which apps can potentially handle a particular data package, for which purpose each target app includes appropriate details in its manifest. When an app is selected, it gets launched with an activation kind of `shareTarget`, in response to which it should show a specific share UI rather than the full app experience.

Let's see how all this works with the [Sharing content target app sample](#) whose appearance is shown in Figure 12-2 (borrowing from Figure 2-22 we saw ages ago). Be sure to run this app once in Visual Studio so that it's effectively installed and it will appear on the list of apps when we invoke the Share charm.

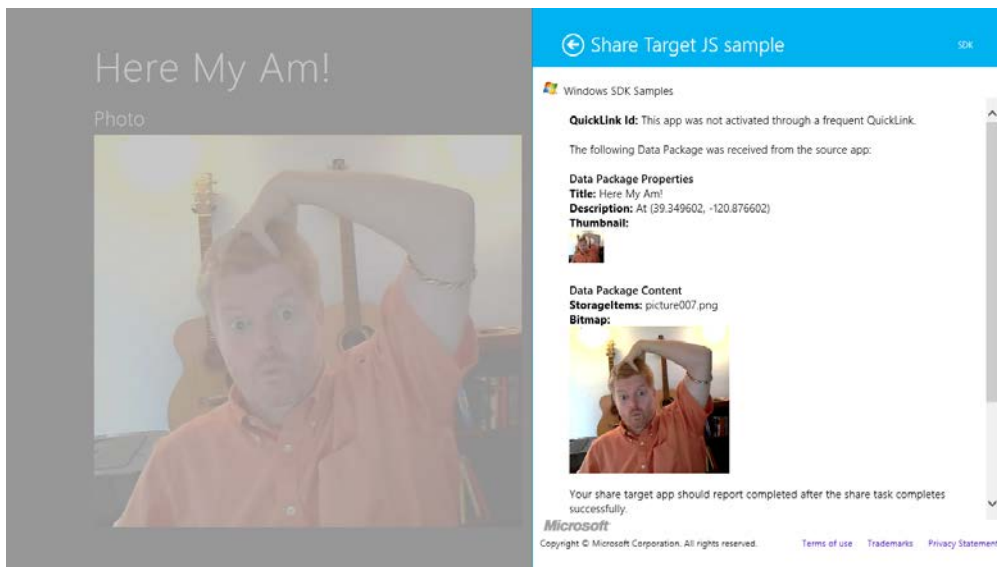


FIGURE 12-2 The appearance of the Sharing content target app sample (the right-hand nonfaded part).

The first step for a share target is to declare the data formats it can accept in the Declarations section of its manifest, along with the page that will be invoked when the app is selected as a target. As shown in Figure 12-3, the target app sample declares it can handle text, URI, bitmap, html, and the <http://schema.org/Book> formats, and it also declares it can handle whatever files might be in a data package (you can indicate specific file types here). Way down at the bottom it then points to `target.html` as its Share target page.

Application UI
Capabilities
Declarations
Content URIs
Packaging

Use this page to add declarations and specify their properties.

Available Declarations:
Select one...
Add

Supported Declarations:

Share Target
Remove

Description:
Registers the app as a share target, which allows the app to receive shareable content.
Only one instance of this declaration is allowed per app.
[More information](#)

Properties:

Data formats

Specifies the data formats supported by the app; for example: "Text", "URI", "Bitmap", "HTML", "StorageItems", or "RTF". The app will be displayed in the Share charm whenever one of the supported data formats is shared from another app.

Data format
Remove

Data format: text

Data format
Remove

Data format: uri

Data format
Remove

Data format: bitmap

Data format
Remove

Data format: html

Data format
Remove

Data format: http://schema.org/Book

Add New

Supported file types

Specifies the file types supported by the app; for example, ".jpg". The Share target declaration requires the app support at least one data format or file type. The app will be displayed in the Share charm whenever a file with a supported type is shared from another app. If no file types are declared, make sure to add one or more data formats.

☒ Supports any file type

Add New

App settings

Executable:

Entry point:

Runtime type:

Start page: target.html

FIGURE 12-3 The Share content target app sample's manifest declarations.

The Share start page, `target.html`, is just a typical HTML page with whatever layout you require for performing the share task. This page typically operates independently of your main app: when your app is chosen through Share, this page is loaded and activated by itself and thus has an entirely separate script context. This page should not provide navigation to other parts of the app and should thus load only whatever code is necessary for the sharing task. (The Executable and Entry Point options are not used for apps written in HTML and JavaScript; those exist for apps written in other languages.)

Much of this structure is built for you automatically through the Share Target Contract *item* template provided by Visual Studio and Blend, as shown in Figure 12-4; the dialog appears when you right-click your project and select Add > New Item or select the Project > Add New Item menu command.

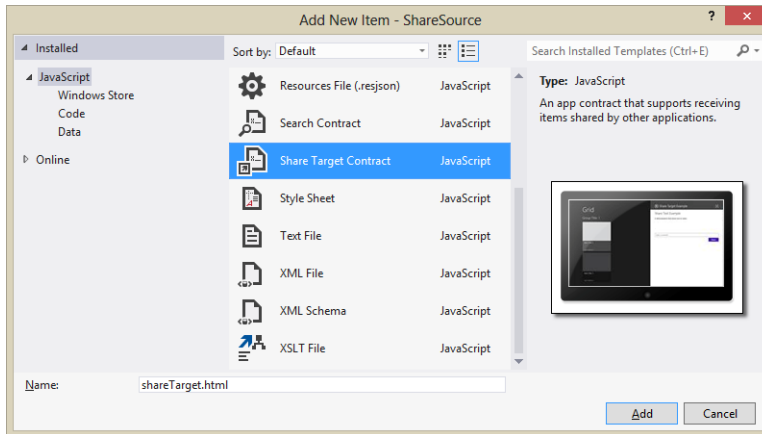


FIGURE 12-4 The Share Target Contract item template in Visual Studio and Blend.

This item template will give you HTML, JS, and CSS files for the share target page and will add that page to your manifest declarations along with text and URI formats. So you'll want to update those as appropriate.

Before we jump into the code, a few notes about the design of a share target page, summarized from [Guidelines for sharing content](#):

- Maintain the app's identity and its look and feel, consistent with the primary app experience.
- Keep interactions simple to quickly complete the share flow: avoid text formatting, tagging, and setup tasks, but do consider providing editing capabilities especially if posting to social networks or sending a message. (See Figure 12-5 from the Mail app for an example.) A social networking target app would generally want to include the ability to add comment; a photo-sharing target would probably include the ability to add captions.
- Avoid navigation: sharing is a specific task flow, so use inline controls and inline errors instead of switching to other pages. Another reason to avoid this is that the share page of the target app runs in its own script context, so being able to navigate elsewhere in the app within a separate context could be very confusing to users.
- Avoid links that would distract from or take the user away from the sharing experience. Remember that sharing is a way to shortcut the oft-tedious process of getting data from one app to another, so keep the target app focused on that purpose.
- Avoid light-dismiss flyouts since the Share charm already works that way.

- Acknowledge user actions when you start sending the data off (to an online service, for example) so that users know something is actually happening.
- Put important buttons within reach of the thumbs on a touch device; refer to [Windows 8 Touch Posture](#) topic in the documentation for placement guidance.
- Make previews match the actual content—in other words, don't play tricks on the user!

With this page design, it's good to know that you do *not* need to worry about different view states—this page really just has one state (and as a flyout, it cannot be snapped). It does need to adapt itself well to varying dimensions, mind you, but not different view states. Basing the layout on a CSS grid with fractional rows and columns is a good approach here.

Caution Because a target app can receive data from any source app, it should treat all such content as untrusted and potentially malicious, especially with HTML, URIs, and files. The target app should avoid adding such HTML or file contents to the DOM, executing code from URIs, navigating to the URI or some other page based on the URI, modifying database records, using `eval` with the data, and so on.

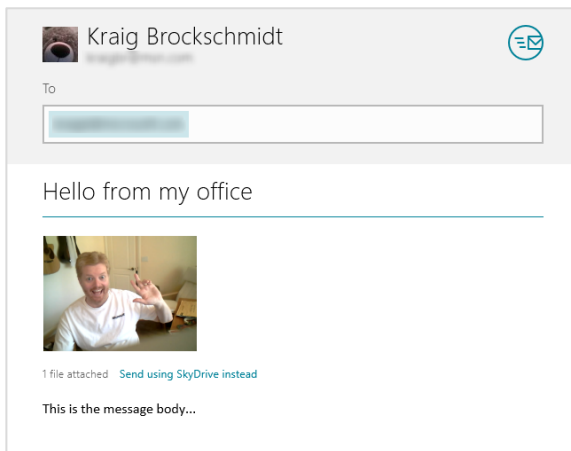


FIGURE 12-5 The sharing UI of the Windows Mail app (the bottom blank portion has been cropped); this UI allows editing of the recipient, subject, and message body, and sending an image as an attachment or as a link to SkyDrive.

Let's now look at the contents of the template's JavaScript file as a whole, because it shows us the basics of being a target. First, as you can see, we have the same structure as a typical `default.js` for the app, using the `WinJS.Application` object's methods and events.

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

    var app = WinJS.Application;
    var share;

    function onShareSubmit() {
        document.querySelector(".progressindicators").style.visibility = "visible";
    }
```

```

document.querySelector(".commentbox").disabled = true;
document.querySelector(".submitbutton").disabled = true;

// TODO: Do something with the shared data stored in the 'share' var.

share.reportCompleted();
}

// This function responds to all application activations.
app.onactivated = function (args) {
    var thumbnail;

    if (args.detail.kind ===
        Windows.ApplicationModel.Activation.ActivationKind.shareTarget) {
        document.querySelector(".submitbutton").onclick = onShareSubmit;
        share = args.detail.shareOperation;

        document.querySelector(".shared-title").textContent =
            share.data.properties.title;
        document.querySelector(".shared-description").textContent =
            share.data.properties.description;

        thumbnail = share.data.properties.thumbnail;
        if (thumbnail) {
            // If the share data includes a thumbnail, display it.
            args.setPromise(thumbnail.openReadAsync().then(
                function displayThumbnail(stream) {
                    document.querySelector(".shared-thumbnail").src =
                        window.URL.createObjectURL(stream, { oneTimeOnly: true });
                }
            ));
        } else {
            // If no thumbnail is present, expand the description and
            // title elements to fill the unused space.
            document.querySelector("section[role=main] header").style
                .setProperty("-ms-grid-columns", "0px 0px 1fr");
            document.querySelector(".shared-thumbnail").style.visibility = "hidden";
        }
    }
};

app.start();
})();

```

When this page is loaded and activated, during which time the app's splash screen will appear, its `WinJS.Application.onactivated` event will fire—again independently of your app's main `activated` handler that's typically in `default.js`. As a share target you just want to make sure that the activation kind is `shareTarget`, after which your primary responsibility is to provide a preview of the data you'll be sharing along with whatever UI you have to edit it, comment on it, and so forth. Typically, you'll also have a button to complete or submit the sharing, on which you tell the share broker that you've completed the process.

The key here is the `args.detail.shareOperation` object provided to the `activated` handler. This is a [Windows.ApplicationModel.DataTransfer.ShareTarget.ShareOperation](#) object, whose `data` property contains a read-only package called a [DataPackageView](#) from which you obtain all the goods:

- To check whether the package has formats you can consume, use the `contains` method or the `availableFormats` collection.
- To obtain data from the package, use its `get*` methods such as `getTextAsync`, `getBitmapAsync`, and `getDataAsync` (for custom formats). When pasting HTML you can also use the `getResourceMapAsync` method to get relative resource URLs. The view's `properties` like the `thumbnail` are also useful to provide a preview of the data.

As you can see, the Share target item template code above doesn't do anything with shared data other than display the title, description, and thumbnail; clearly your app will do something more by requesting data from the package, like the examples we see in the share target sample. Its `js/target.js` file contains an `activated` handler for the `target.html` page (in the project root), and it also displays the thumbnail in the data package by default. It then looks for different data formats and displays those contents if they exist:

```
if (shareOperation.data.contains(Windows.ApplicationModel.DataTransfer.StandardDataFormats.  
    text)) {  
    shareOperation.data.getTextAsync().done(function (text) {  
        displayContent("Text: ", text, false);  
    });  
}
```

The same kind of code appears for the simpler formats. Consuming a bitmap is a little more work but straightforward:

```
if (shareOperation.data.contains(Windows.ApplicationModel.DataTransfer.StandardDataFormats.  
    bitmap)) {  
    shareOperation.data.getBitmapAsync().done(function (bitmapStreamReference) {  
        bitmapStreamReference.openReadAsync().done(function (bitmapStream) {  
            if (bitmapStream) {  
                var blob = MSApp.createBlobFromRandomAccessStream(bitmapStream.contentType,  
                    bitmapStream);  
                document.getElementById("imageHolder").src = URL.createObjectURL(blob,  
                    { oneTimeOnly: true });  
                document.getElementById("imageArea").className = "unhidden";  
            }  
        });  
    });  
}
```

For HTML, it looks through the markup for `img` elements and then sets up their `src` attributes from the resource map (the `iframe` already has the HTML content from the package by this time):

```

var images = iFrame.contentDocument.documentElement.getElementsByTagName("img");
if (images.length > 0) {
    shareOperation.data.getResourceMapAsync().done(function (resourceMap) {
        if (resourceMap.size > 0) {
            for (var i = 0, len = images.length; i < len; i++) {
                var streamReference = resourceMap[images[i].getAttribute("src")];
                if (streamReference) {
                    // Call a helper function to map the image element's src
                    // to a corresponding blob URL generated from the streamReference
                    setResourceMapURL(streamReference, images[i]);
                }
            }
        }
    });
}

```

The `setResourceMapURL` helper function does pretty much what the bitmap-specific code did, which is call `openReadAsync` on the stream, call `MSApp.createBlobFromRandomAccessStream`, pass that blob to `URL.createObjectURL`, set the `img.src` with the result, and close the stream.

After the target app has completed a sharing operation, it should call the `ShareOperation.reportCompleted` method, as shown earlier with the template code. This lets the system know that the data package has been consumed, the share flow is complete, and all related resources can be released. The share target sample does this when you explicitly click a button for this purpose, but normally you automatically call the method whenever you've completed the share. Do be aware that calling `reportCompleted` will close the target app's sharing UI, so avoid calling it as soon as the target activates: you want the user to feel confident that the operation was carried out.

Long-Running Operations

When you run the share target sample and invoke the Share charm from a suitable source app, there's a little expansion control near the bottom labeled "Long-running Share support." If you expand that, you'll see some additional controls and a bunch of descriptive text, as shown in Figure 12-6. The buttons shown here tie into a number of other methods on the `ShareOperation` object alongside `reportCompleted`. These help Windows understand exactly how the share operation is happening within the target: `reportStarted`, `reportDataRetrieved`, `reportSubmittedBackgroundTask`, and `reportError`. As you can see from the descriptions in Figure 12-6, these generally relate to telling Windows when the target app has finished cooking its meal, so to speak, and the system can clean the dishes and put away the utensils:

- `reportStarted` informs Windows that your sharing operation might take a while, as if you're uploading the data from the package to another place, or just sending an email attachment with what ends up being large images and such. This specific method indicates that you've obtained all necessary user input and that the share pane can be dismissed.
- `reportDataRetrieved` informs Windows that you've extracted what you need from the data package such that it can be released. If you've called `MSApp.createBlobFromRandomAccessStream` for an image stream, for example, the blob now contains a copy of the image that's local to the

target app. If you're using images from the package's [resourceMap](#), on the other hand, you'll not want to call [reportDataRetrieved](#) unless you explicitly make a copy of those references whose URIs refer to bits inside the data package. In any case, if you need to hold onto the package throughout the operation, you don't need to call this method as you'll later call [reportCompleted](#) to release the package.

- [reportSubmittedBackgroundTask](#) tells Windows that you've started a background transfer using the [Windows.Networking.BackgroundTransfer.BackgroundUploader](#) class. As the sample description in Figure 12-6 indicates, this lets Windows know that it can suspend the target app and not disturb the sharing operation. If you call this method with a local copy of the data being uploaded, go ahead and call [reportCompleted](#) method so that Windows can clean up the package; otherwise wait until the transfer is complete.
- [reportError](#) lets Windows know if there's been an error during the sharing operation.

Report Completed

Long-running Share Support ^

This API is required if your app supports uploading a format that may take some time, such as images or videos. A user should be able to dismiss your app and have the upload continue in the background while they do other things. In order for the dismiss behavior to work correctly, you need to report to the share platform that you finished getting user input. After you call this, a user can go back to the share pane and see your application in the share progress list.

Report Started

This API is optional and helps Windows to optimize resource usage of the system. You should report this if you have finished extracting data from the Data Package so that Windows can suspend or terminate the source app as necessary to reclaim system resources.

Report Data Retrieved

This API is optional and helps Windows to optimize resource usage of the system. You should report this if you have called the Windows Runtime Background Transfer class to upload your content. Then Windows can suspend your app as necessary to reclaim system resources. If you use this API, call it after Report Started.

Report Submitted To BackgroundTask

If for any reason the long-running share was unsuccessful and failed in the background, you should report failure and include a message for the user about how they can recover from the error. When the user goes back to the share pane they can see your message in the progress list. You must never call Report Error if your app is visible in the foreground.

Error message:

Report Error

FIGURE 12-6 Expanded controls in the Sharing content target app sample for Long-Running Share Support. The Report Completed button is always shown and isn't specific to long-running tasks despite its placement in the sample's UI. Don't let that confuse you!

Quicklinks

The last aspect of the Share contract for us to explore is something we mentioned early on in this section: quicklinks. These serve to streamline the Share process such that users don't need to re-enter information in a target app. For example, if a user commonly shares data with particular people through email, each contact can be a quicklink for the email app. If a user commonly shares with different people or groups through a social networking app, those people and/or groups can be represented with quicklinks. And as these targets are much more user-specific than target apps in general, the Share charm UI shows these at the top of its list (see Figure 12-7 below).

Each quicklink is associated with and serviced by a particular target app and simply provides an identifier to that target. When the target is invoked through a quicklink, it then uses that identifier to retrieve whatever data is associated with that quicklink and prepopulates or otherwise configures its UI. It's important to understand that quicklinks contain *only* an identifier, so the target app must store and retrieve the associated data from some other source, typically local app data where the identifier is a filename, the name of a settings container, etc. The target app could also use roaming app data or the cloud for this purpose, but quicklinks themselves do not roam to another device—they are strictly local. Thus, it makes the most sense to store the associated data locally.

A quicklink itself is just an instance of the [Windows.ApplicationModel.DataTransfer.-Quicklink](#) class. You create one with the `new` operator and then populate its `title`, `thumbnail`, `supportedDataFormats`, `supportedFileTypes`, and `id` properties. The data formats and file types are what Windows uses to determine if this quicklink should be shown in the list of targets for whatever data is being shared from a source app (independent of the app's manifest declarations). The `title` and `thumbnail` are used to display that choice in the Share charm, and the `id` is what gets passed to the target app when the quicklink is chosen.

Tip For the thumbnail, use an image that's more specifically representative of the quicklink (such as a contact photo) rather than just the target app. This helps distinguish the quicklink from the general use of the target app.

An app then registers a quicklink with the system by passing it to the [ShareOperation.report-Completed](#) method. As this is the *only* way in which a quicklink is registered, it tells us that creating a quicklink *always* happens as part of another sharing operation. It's a way to create a specific target that might save the user some time and encourage her to choose your target app again in the future.


Let's follow the process within the Sharing content target app sample to see how this all works. First, when you invoke the Share charm and choose the sample, you'll see that it provides a check box for creating a quicklink (Figure 12-7). When you check this, it provides fields in which you can enter an id and a title (the thumbnail just uses a default image). When you press the Report Completed button, it calls `reportCompleted` and the quicklink is registered. On subsequent invocations of the Share charm with the appropriate data formats from the source app, this quicklink will then appear in the list, as shown in Figure 12-8 where the app servicing the quicklink is always indicated under the provided title.

When reporting completed, you can optionally add a QuickLink to make it easier for users to repeat the way they share most often. This saves them from having to select that person or group in your app every time they share to them.

☒ Add a QuickLink (optional)

QuickLink Id:

Title: ✕

Icon: 

Long-running Share Support ⌵

Report Completed

FIGURE 12-7 Controls to create a quicklink in the Sharing content target app sample.

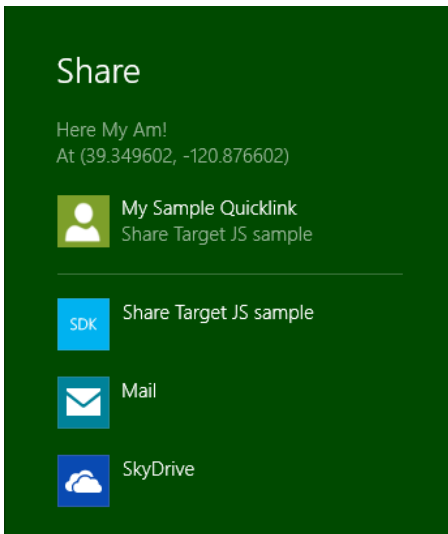


FIGURE 12-8 A quicklink from the Sharing content target app sample as it appears in the Share charm target list.

Here's how the share target sample creates the quicklink within the function `reportCompleted` (`js/target.js`) that's attached to the Report Completed button (some error checking omitted):

```
if (addQuickLink) {
    var quickLink = new Windows.ApplicationModel.DataTransfer.ShareTarget.QuickLink();

    var quickLinkId = document.getElementById("quickLinkId").value;
    quickLink.id = quickLinkId;

    var quickLinkTitle = document.getElementById("quickLinkTitle").value;
    quickLink.title = quickLinkTitle;

    // For quicklinks, the supported FileTypes and DataFormats are set independently
```

```

// from the manifest
var dataFormats = Windows.ApplicationModel.DataTransfer.StandardDataFormats;
quickLink.supportedFileTypes.replaceAll(["*"]);
quickLink.supportedDataFormats.replaceAll([dataFormats.text, dataFormats.uri,
    dataFormats.bitmap,
    dataFormats.storageItems, dataFormats.html, customFormatName]);

// Prepare the icon for a QuickLink
Windows.ApplicationModel.Package.current.installedLocation.getFileAsync(
    "images\\user.png").done(function (iconFile) {
    quickLink.thumbnail = Windows.Storage.Streams.RandomAccessStreamReference
        .createFromFile(iconFile);
    shareOperation.reportCompleted(quickLink);
});

```

Again, the process just creates the [Quicklink](#) object, sets its properties (perhaps settings a more specific thumbnail such as a contact person's picture), and passes it to [reportCompleted](#). In the share target sample, you can see that it doesn't actually store any other local app data; for its purposes the properties in the quicklink are sufficient. Most target apps, however, will likely save some app data for the quicklink that's associated with the [quicklink.id](#) property and reload that data when activated later on through the quicklink.

When the app is activated in this way, the [eventArgs.detail.shareOperation](#) object within the [activated](#) event handler will contain the [quickLinkId](#). The Source target app simply displays this id, but your would certainly use it to load app data and prepopulate your share UI:

```

// If this app was activated via a QuickLink, display the QuickLinkId
if (shareOperation.quickLinkId !== "") {
    document.getElementById("selectedQuickLinkId").innerText = shareOperation.quickLinkId;
    document.getElementById("quickLinkArea").className = "hidden";
}

```

Note that when the target app is invoked *through* a quicklink, it doesn't display the same UI to *create* a quicklink, because doing so would be redundant. However, if the user edited the information related to the quicklink, you might provide the ability to update the quicklink, which means to update the data you save related to the id, or to create a new quicklink with a new id.

The Clipboard

Before the Share contract was ever conceived, the mechanism we know as the Clipboard was once the poster child of app-to-app cooperation. And while it may not garner any media attention nowadays, it's still a tried-and-true means for apps to share and consume data.

For Windows Store apps, clipboard interactions build on the same [DataPackage](#) mechanisms we've already seen for sharing, so everything we've learned about populating that package, using custom formats, and using delayed rendering still apply. Indeed, if you make data available on the clipboard, you should make sure the same data is available for the Share contract!

The question is how to wire up commands like copy, cut, and paste—from the app bar, a context menu, or keystrokes—should an app provide them for its own content (many controls handle the clipboard automatically). For this we turn to the [Windows.ApplicationModel.DataTransfer.Clipboard](#) class.

As shown in the [Clipboard app sample](#), the processes here are straightforward. For copy and cut:

- Create a new `Windows.ApplicationModel.DataTransfer.DataPackage` (or use `MSApp.createDataPackage` or `MSApp.createDataPackageFromSelection`), and populate it with the desired data.

```
var dataPackage = new Windows.ApplicationModel.DataTransfer.DataPackage();
dataPackage.setText(textValue);
//...
```

- (Optional) Set the package's `requestedOperation` property to values from [DataPackageOperation](#): `copy`, `move`, `link`, or `none` (the latter is used with delayed rendering). Note that these values can be combined using the bitwise OR operator, as in:

```
var dpo = Windows.ApplicationModel.DataTransfer.DataPackageOperation;
dataPackage.requestedOperation = dpo.copy | dpo.move | dpo.link;
```

- Pass the data package to `Windows.ApplicationModel.DataTransfer.Clipboard.setContent`:

```
Windows.ApplicationModel.DataTransfer.Clipboard.setContent(dataPackage);
```

To perform a paste:

- Call `Windows.ApplicationModel.DataTransfer.Clipboard.getContent` to obtain a read-only data package called a [DataPackageView](#):

```
var dataView = Windows.ApplicationModel.DataTransfer.Clipboard.getContent();
```

- Check whether it contains formats you can consume with the `contains` method (alternately, you can check the contents of the `availableFormats` vector):

```
if (dataView.contains(Windows.ApplicationModel.DataTransfer.StandardDataFormats.
    text)) {
    //...
}
```

- Obtain data using the view's `get*` methods such as `getTextAsync`, `getBitmapAsync`, and `getDataAsync` (for custom formats). When pasting HTML, you can also use the `getResourceMapAsync` method to get relative resource URLs. The view's `properties` like the `thumbnail` are also useful, along with the `requestedOperation` value or values.

```
dataView.getTextAsync().done(function (text) {
    // Consume the data
})
```

If at any time you want to clear the clipboard contents, call the `Clipboard` class's `clear` method. You can also make sure data is available to other apps even if yours is shut down by calling the `flush` method (which will trigger any deferred rendering you might have set up).

Apps that use the clipboard also need to know when to enable or disable a paste command depending on available formats. At any time you can get the data package view from the clipboard and use its `contains` method or `availableFormats` property and decide accordingly. You should also then listen to the Clipboard object's `contentChanged` event (a WinRT event), which will be fired when you or some other app calls the clipboard's `setContent` method. At time time you'd again enable or disable the commands. Of course, you won't receive this event when your app is suspended, so you should refresh the state of those commands within your `resuming` handler.

Again, the Clipboard app sample provides examples of these various scenarios, including copy/paste of text and HTML (Scenario 1); copy and paste of an image (Scenario 2); copy and paste of files (Scenario 3); and clearing the clipboard, enumerating formats, and handling `contentChanged` (Scenario 4).

Note, finally, that pasted data can come from anywhere. Apps that consume data from the clipboard should, like a share target, treat the content they receive as potentially malicious and take appropriate precautions.

Search

Search has become such a ubiquitous feature for apps that the designers of Windows 8 decided to provide a system-level keyword search UI (with built-in Input Method Editor support) directly alongside Share, Devices, and Settings in the Charms bar, as shown in Figure 12-9. This means that apps don't need to (and generally shouldn't) provide their own UI controls for search, and, by participating in this contract, the user can not only easily search the app that's in the foreground but also quickly and easily search within other apps without having to go off and start those apps separately. And because those other apps can be searching content that doesn't necessarily exist on your machine, the Search charm fills the middle ground between material on your file system and the rest of the world. It's quite powerful in this way!

The Search charm also means that users never need to explicitly start your app to search within it. Simply by changing the search target within the search pane, that target app is launched and asked to perform a search with the current keywords. This is also what makes Search work even if the current foreground app doesn't support the contract at all—the search target just defaults to the first app in the list.

Tip If you need to know which side of the screen the Search pane appears on, so you can place controls on your results page so they won't be obscured, check the [`Windows.UI.Application-Settings.SettingsPane.edge`](#) property.

The Search contract that makes this happen is composed of a set of interactions between the Windows-provided Search UI and the search target app. (In this section, when I refer to a *target* app, I'm referring now to search, not share.) This interaction communicates the keywords (even if empty) to the app when the user presses Enter, clicks the icon to the right of the entry field, or changes apps. The interaction also allows the target app to provide suggested search terms, as well as suggested results (with result-specific graphics) that appear in the search pane directly, as shown in Figure 12-10.

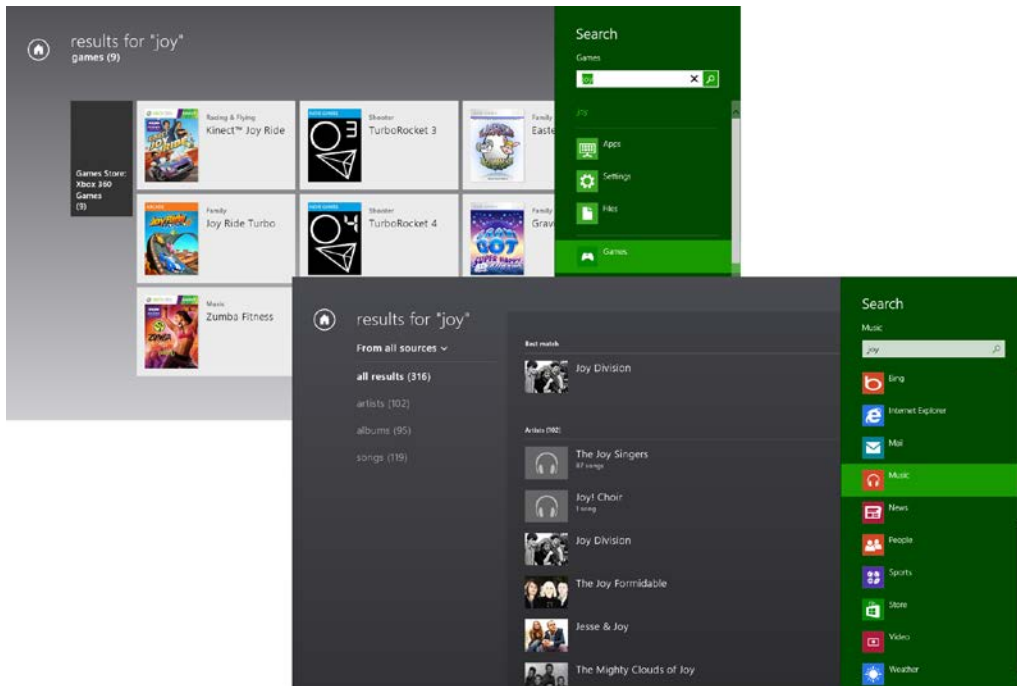


FIGURE 12-9 The Search pane invoked through the Search charm, with results shown in the Games app and the Photos app. As with Share, the user can control which apps are shown through Change PC Settings > Search. That same settings panel also allows the user to clear search history and control a few other aspects of the UI.

Designwise, Search should work with whatever data the app manages, whether local or online (or both); it's really the primary means to search within everything that the app can access. For this reason, Microsoft highly recommends that apps *don't* provide their own search UI (which otherwise distracts from the app's content) unless it's really all the app does and where it would need additional search criteria up-front. Otherwise, it's best to let the user first search through the charm and then filter, sort, and otherwise organize the results within the app through on-canvas or app bar commands. On the

flip side, the Search charm is *not* intended for finding data *within* a page; that is, it is expected that apps provide their own controls for essentially scanning and highlighting results that are already in view (like the find function in browsers). Many details on such design questions can be found on [Guidelines for search](#).

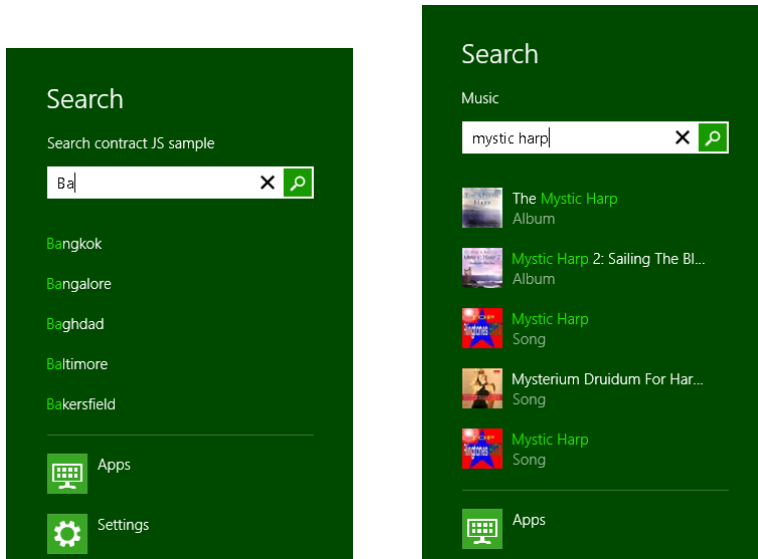


FIGURE 12-10 Suggested searches (left) and search results (right) from a target app appear directly in the search pane.

Searching within an app effectively navigates the app to its search results page, as we see in Figure 12-9, and thus activates the app in the same script context as when it's run normally. Again, if the app needs to be launched to service the search contract, it will be launched directly into that page (we'll see this mechanism shortly). Tapping on a result then navigates the app directly to the details for that result. Of course, if the app was already running when invoked via Search, the result page's back button should navigate to whatever page the user was on before. Even if the app is launched to service the Search charm, it's helpful to provide the user with a means to navigate to its home page, especially when there are no results through which to navigate elsewhere.

Let's now look at the basic search contract interaction, after which we'll explore the richer aspects of search suggestions, suggested results, and type to search.

Search in the App Manifest and the Search Item Template

An app's life as a search target begins, as with other contracts, in the app manifest on the Declarations tab, as shown in Figure 12-11.

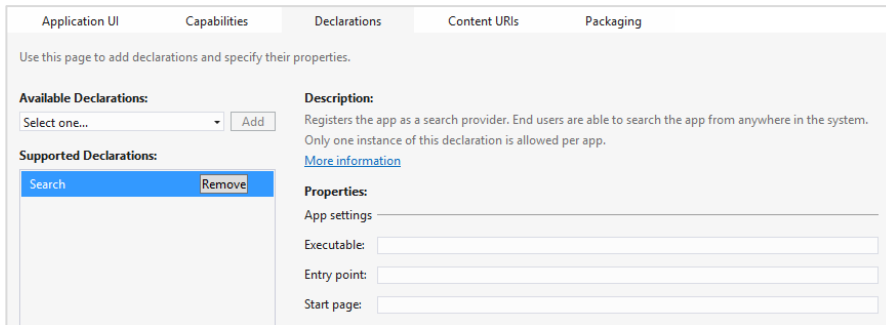


FIGURE 12-11 The Search declarations page within Visual Studio; typically, the App Settings properties are left blanks in an HTML/JavaScript app.

Since search is not tied to any particular data format (like share is), all you really need to specify here is a Start page, if in fact you want it to be separate from the rest of your app at all. Unlike the share contract, search is much more integrated with in-app navigation: when the user taps a result on your results page, you want to navigate to that page directly as if they'd tapped on the same item in some other list. Similarly, if the user taps the back button in your results page, they should navigate to whatever page they were on when the charm was first invoked. For this reason, then, activation via search typically gets handled by through the app's main `activated` event. We'll get to that in the next section.

An easy way to add the Search contract is through the Search contract item template in Visual Studio and Blend. (You can see this listed back in Figure 12-4 just above the Share target contract.) If you right-click your project and select Add > New Item, or use the Project > Add New Item... menu command, you can choose the Search Contract item in from the list of templates. This will add the Search declaration in your app manifest and add three page control files (.html, .js, and .css) for a search results page. There's not much exciting to show here visually because the template code very much relies on there being some real data to work with. Nevertheless, the template gives you a great structure to work from, including the recommended UI for providing filters and so forth. Some further details can be found on [Adding a Search Contract item template](#).

Basic Search and Search Activation

The most basic interaction with the Search contract is receiving a query when the app is already running. This is a great example of how search really just triggers navigation in the app. To receive such a query, you need only listen to the `querysubmitted` event of the `Windows.ApplicationModel.Search.SearchPane` object. The exact code looks something like this where `searchPageURI` identifies the results page:

```
var searchPane = Windows.ApplicationModel.Search.SearchPane.getForCurrentView();
searchPane.onquerysubmitted = function (eventArgs) {
    WinJS.Navigation.navigate(searchPageURI, eventArgs);
};
```

The `eventArgs` object here will be a [SearchPaneQuerySubmittedEventArgs](#) that contains just two properties: `queryText` (the contents of the text box in the search pane) and `language` (the BCP 47 language tag currently in used). In the code above, these are just passed to the `WinJS.Navigation.navigate` method that passes them onto to the results page (whatever `searchPageURI` contains). From there, that page will just process `queryText` appropriate to `language` and populate the page contents with appropriate items. For this purpose an app typically uses a `ListView` control, as you might expect for a variable-length results collection.

Through the same `SearchPane` object you can also set the `placeholderText` property with whatever should appear in the initial search box. Its `show` method allows you to show the pane programmatically, its `visible` property and `visibilitychanged` event will tell you its status, and its `queryText` property will give you the current contents of the input control.

You can also listen for its `querychanged` event. This is a precursor to `querySubmitted` and is appropriate if you have logic you need to run outside of providing suggestions, such as previewing results (the behavior you see on the start screen when searching for apps, also known as word wheeling). Its `eventArgs` will contain `queryText` and `language` properties, as with `query-submitted`, along with a `linguisticDetails` property that provides details about text entered through an Input Method Editor (IME), specifically linguistic alternatives. If you expect to have Japanese or Chinese users, it's highly recommended to also search for these alternatives in response to `querychanged` and `suggestionsrequested` (see the next section)

Let's see how search affects app activation, which again typically comesthrough your default `activated` handler in the same script context as when the app is run normally.

In this case the activation kind value will be `search`, a case that you want to handle separately from `launch`. To see this in action, let's turn to the [Search contract sample](#). Its activation code is found in `js/default.js`—code that's applicable to the entire app:

```
function activated(eventObject) {
    if (eventObject.detail.kind ===
        Windows.ApplicationModel.Activation.ActivationKind.launch) {
        eventObject.setPromise(WinJS.UI.processAll().then(function () {
            var url = WinJS.Application.sessionState.lastUrl || scenarios[0].url;
            return WinJS.Navigation.navigate(url);
        }));
    } else if (eventObject.detail.kind ===
        Windows.ApplicationModel.Activation.ActivationKind.search) {
        eventObject.setPromise(WinJS.UI.processAll().then(function () {
            if (eventObject.detail.queryText === "") {
                // Navigate to your landing page since the user is pre-scoping to your app.
            } else {
                // Display results in UI for eventObject.detail.queryText and
                // eventObject.detail.language (that represents user's locale).
            }

            // Navigate to the first scenario since it handles search activation.
            var url = scenarios[0].url;
            return WinJS.Navigation.navigate(url, { searchDetails: eventObject.detail });
        }));
    }
}
```

```

    }));
  }
}

```

In the search activation path, it's clearly good to avoid any processing that isn't needed by the search page itself, but you still need to be prepared to navigate to other parts of the app when a result is chosen. Also, if the app is being launched in response to a search, be sure to reload both general settings as you would with a normal launch as well as session state when `previousExecutionState` is `terminated`. This means, in fact, that the state of a results page is part of the app's session state; you'll normally want to save the last search term as part of that state so that you can rehydrate the results page when needed.

The sample doesn't actually search any real data—it just outputs messages when certain events happen. But you can test this activation path in a couple of ways. First, if the app isn't running, invoke the search charm, enter some query text, and then select the search sample. You'll find that it ends up on the page for Scenario 1 and shows the search term right away. This tells you that it processed the activation and picked up the search term from `eventObject.detail.queryText`, as you can see in the code above. (Look also at `js/scenario1.js` where it outputs the term in the page's `processed` method.)

To step through the same code, set a breakpoint within the `searchTarget` case of the `activated` handler and run the app in the Visual Studio debugger. Invoke the search charm, enter a query, select some other app (which will do a search), and then switch back to the sample. You should hit your breakpoint as the activated handler will be called with the activation kind of `search`.

When activated through search, be sure that the page gets fully processed with calls like `WinJS.UI.processAll`. (You don't need to worry if the app is already running; `processAll` won't do redundant work.)

It is important when your app is activated—as with handling `querysubmitted` and/or `querychanged` events—to note that the `queryText` might be empty. In this case you can show default results or navigate to your home page if that's more appropriate. See “Sidebar: Testing Search.”

Sidebar: Testing Search

A number of variations with the Search charm can affect how a search target app is launched and with what parameters. To be sure that you've exercised all applicable code paths, be sure to test these conditions:

- App is running and search is invoked with no query text, query text with known results, and query text that returns no results.
- App is not running and is invoked from the search charm, with all the variations on text listed above.

- App is in the snapped state and is invoked as above, in which case Search will go to the Start screen.
- App is suspended and is invoked as above.

You should also be mindful of how you present results, taking care that the primary results are not hidden by the Search pane, which will remain visible until the user dismisses it.

Sidebar: Synchronizing In-App Search with the Search Pane

Some types of apps will still maintain their own in-app search UI in addition to using the search pane, or in other ways they might have some kind of search term that would be good to keep in sync with the term shown in the search pane. To do this, the app can ask the search pane for its `queryText` value and can attempt to set that value through the [SearchPane.trySet-QueryText](#) method. This call will fail, mind you, if the app isn't itself visible or if the search pane is already visible or becoming visible.

Providing Query Suggestions

Using `querysubmitted` and the activation sequence in the previous section gives you the basic level of search interaction, and Windows will automatically provide a history of the user's recent searches. Still, with just a little more work you can make the experience much richer. Because writing the code to actually perform the search, process the results, and display them beautifully is the bulk of the work with the Search contract anyway, adding support for query suggestions (this section) and result suggestions (next section) is a relatively small investment with a huge impact on the overall user experience.

To go beyond the default search history and provide as-the-user-is-typing query suggestions, which appear to the user as shown on the left side of Figure 12-10, you have two options. Which one you use depends on what you want to suggest and the data that you're searching.

First, to provide suggestions from folders on the file system, such as the music, pictures, and videos libraries, the search pane provides a built-in implementation through its [setLocalContent-SuggestionsSettings](#) method with results like those in Figure 12-12. As shown in Scenario 4 of the sample, you first create a [Windows.ApplicationModel.Search.LocalContentSuggestion-Settings](#) object, populate its properties, and then pass that object to [setLocalContent-SuggestionsSettings](#) (js/scenario4.js):

```
var page = WinJS.UI.Pages.define("/html/scenario4.html", {
    ready: function (element, options) {
        var localSuggestionSettings = new
            Windows.ApplicationModel.Search.LocalContentSuggestionSettings();
        localSuggestionSettings.enabled = true;
        localSuggestionSettings.locations.append(Windows.Storage.KnownFolders.musicLibrary);
        localSuggestionSettings.aqsFilter = "kind:=music";
    }
});
```

```

Windows.ApplicationModel.Search.SearchPane.getForCurrentView()
    .setLocalContentSuggestionSettings(localSuggestionSettings);
}
});

```

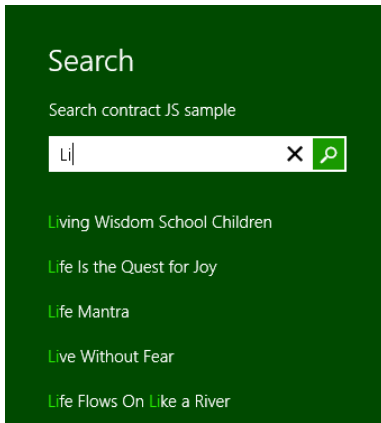


FIGURE 12-12 Suggestions from local folders as automatically provided by the search pane.

In populating the [LocalContentSuggestionSettings](#) properties, be sure first to set [enabled](#) to [true](#). The [locations](#) collection (a vector) contains one or more [StorageFolder](#) objects to indicate where the search should take place. Because enumerating files to provide suggestions requires programmatic access to those folders, you need to make sure your app has the appropriate capabilities set in its manifest, retrieves the folder from the AccessCache, or has obtained programmatic access through the folder picker. In the latter case, the app would provide UI elsewhere to configure the search locations (perhaps through the Settings pane, for instance).

You can also specify an [Advanced Query Syntax](#) (AQS) string in the [aqFilter](#) property and/or some number of [Windows Properties](#) (like *System.Title*) within [propertiesToMatch](#) (a string vector). This is typically used to filter file types, as when searching a folder, but it can be as specific as you need. For more on AQS, refer to “Rich Enumeration with File Queries” in Chapter 8, “State, Settings, Files, and Documents”; for more on Windows properties, refer to “Media File Metadata” in Chapter 10, “Media.”

As for the second option, [LocalContentSuggestionSettings](#) can do a lot for you, but clearly many apps will be searching on some other data source (whether local or online) and will thus need to supply suggestions from those sources. In these cases, listen for and handle the search pane’s [suggestionsrequested](#) event. Its [eventArgs](#) will contain the [queryText](#), [language](#), and [linguistic-Details](#) as always, and in response you populate a collection of up to five suggestions in the [eventArgs.request.searchSuggestionCollection](#) (again including the alternatives in the [linguistic-Details](#) object if needed). Ideally this takes half a second or less, and it’s important to know that all the results need to be in the collection once you return from your event handler.

Here's how it's done in Scenario 2 of the Search contract sample (where `suggestionList` is just a hard-coded list of city names):

```
Windows.ApplicationModel.Search.SearchPane.getForCurrentView().onsuggestionsrequested =
    function (eventObject) {
        var queryText = eventObject.queryText;
        var suggestionRequest = eventObject.request;
        var query = queryText.toLowerCase();
        var maxNumberOfSuggestions = 5;
        for (var i = 0, len = suggestionList.length; i < len; i++) {
            if (suggestionList[i].substr(0, query.length).toLowerCase() === query) {
                suggestionRequest.searchSuggestionCollection.appendQuerySuggestion(
                    suggestionList[i]);
                if (suggestionRequest.searchSuggestionCollection.size ===
                    maxNumberOfSuggestions) {
                    break;
                }
            }
        }
    };
```

So if `query` contains "ba" as it would in Figure 12-10, the first 5 names in `suggestionList` will be Bangkok, Bangalore, Baghdad, Baltimore, and Bakersfield. Of course, a real app will be drawing suggestions from its own database or from a service (simulated in Scenarios 5 and 6, by the way), but you get the idea. With a service, though, you should also check the `suggestionResult.isCanceled` property before starting a new request: this flag indicates that the search query hasn't actually changed from a previous query and it's not necessary to create new suggestions.

Note When the `SearchPane.searchHistoryEnabled` property is `true` (the default), a user's search history will be automatically tracked with prior searches appearing as suggestions when the search charm is first invoked (before the user types any other characters). Setting this property to `false` will disable the behavior, in which case an app can maintain its own history of previous `queryText` values. If an app does this, we recommend providing a means to clear the history through the app's Settings.

Apps can also use the `SearchPane.searchHistoryContext` property to create different histories depending on different contexts. When this value is set prior to the search charm being invoked, automatically managed search terms (`searchHistoryEnabled` is `true`) will be saved for that context. This has no effect when an app manages its own history, in which case it can manage different histories directly.

Now the `eventArgs.request` property, a `SearchPaneSuggestionsRequest` object, has a few features you want to know about. Its `searchSuggestedCollection` property is unique—it's not an array or other generic vector but a `SearchSuggestionCollection` object with a `size` property and four methods: `appendQuerySuggestion` (to add a single item to the list, as shown above), `appendQuerySuggestions` (to add an array of items at once, as you might receive from a query to a service), `appendResultSuggestion` (see next section) and `appendSearchSeparator` (which is used to group suggestions). In the latter case, a separator is given a label and appears as follows:



The request object also has a `getDeferral` method if you need to perform an asynchronous operation to retrieve your suggestions. It works like all other deferral's we've seen: before starting the async operation (like `WinJS.xhr`), call `getDeferral` to retrieve the deferral object, start the operation, return from the `suggestionsrequested` method, and call the deferral's `complete` method inside the async completed handler. This is demonstrated again in Scenarios 5 and 6 of the sample since this would clearly be needed when querying a service for this purpose (code here derived from `js/scenario5.js`):

```
Windows.ApplicationModel.Search.SearchPane.getForCurrentView().onsuggestionsrequested =
function (eventObject) {
    var queryText = eventObject.queryText;
    var suggestionRequest = eventObject.request;

    var deferral = suggestionRequest.getDeferral();

    // Create request to obtain suggestions from service and supply them to the Search Pane.
    // Depending on design of the service, you might vary URL based on eventObject.language.
    // You might also compose queryText in the URL to let the service do the filtering.
    xhrRequest = WinJS.xhr({ url: /* URL to suggestion service */ });
    xhrRequest.done(
        function (request) {
            if (request.responseText /* or responseXML */) {
                // Populate suggestionRequest.searchSuggestionCollection based on response
            }

            deferral.complete(); // Indicate we're done supplying suggestions.
        },
        function (error) {
            // Call complete on the deferral when there is an error.
            deferral.complete();
        }
    );
};
```

You can use any JSON or XML response format you want, but since your app is doing the parsing, there are existing standards for returning search suggestions. For JSON, refer to the [OpenSearch Suggestions specification](#) and Scenario 5 in the sample where a JSON response can be directly parsed into an array and passed in one call to `appendQuerySuggestions`. For XML, refer to the [XML Search Suggestions Format Specification](#) and Scenario 6. In the latter case, a function named `generate-Suggestions` provides a generic parser routine for such a response, and although the sample doesn't demonstrate using separators, URIs, and images in those suggestions, the `generateSuggestions` function shows how to parse them and send them onto `appendQuerySuggestion[s]` as well as `appendResultSuggestion`, which we'll see next.

Providing Result Suggestions

As shown in Figure 12-10 (on the right side), a search target app can provide suggested *results* and not just suggested queries. This is also accomplished by handling the search pane's `suggestions-requested` event as described in the previous section, only make sure you use `suggestion-Request.searchSuggestionCollection.appendResultSuggestion` to populate the results and not `appendQuerySuggestion[s]` (`appendSearchSeparator` can still be used). You also then need to handle the search pane's `resultSuggestionChosen` event to handle the user's selection as a result and not as a query.

In other words, handling the `querysubmitted` event means that you're taking the query text and populating a list of results in your own page. Because of this, you'll be handling click or tap events for those items directly, navigating to the appropriate details page. The `resultSuggestionChosen` event tells you that the same thing has happened in the system-owned search pane with results that are shown there from your suggestions. You thus process the `resultSuggestionChosen` event in the same way that you would handle an item invocation in your own page. The `eventArgs.tag` property in this case will contain the tag you provide for the suggested result in the `appendResultSuggestion` call.

This method takes five arguments in this order, and be mindful of any necessary localization here:

- `text` The first line text to show in the search pane (as in Figure 12-10).
- `detailText` The second line of text for a search result (as in Figure 12-10) that is also used for tooltips.
- `tag` The string you want to receive in the `resultSuggestionChosen` event.
- `image` An `IRandomAccessStreamReference` for the image to display. The base size of this image is 40x40 for 100% scale, 56x56 for 140%, and 72x72 for 180%. Take these sizes into account if you dynamically generate images for the result suggestions.
- `imageAlternateText` The `alt` attribute for the image.

As noted in the previous section, the `generateSuggestions` function found in `js/scenario6.js` of the sample provides a generic parser that turns XML search suggestions into the appropriate `appendResultSuggestion` calls, including the use of `Windows.Storage.Streams.RandomAccess-StreamReference.createFromUri` to convert an image URI to the appropriate stream reference. Typically, such URIs point to a remote source where ideally you'd be able to ask your service for different sized images based on the resolution scaling.

Local `ms-appx://` and `ms-appdata://` URIs are also allowable using the appropriate `.scale-1x0` naming convention. You should always, in fact, have a default image for suggested results in your package (using an `ms-appx://` URI to refer to it when necessary); the system will not provide one for you.

Type to Search

The final feature of Search is the ability to emulate the “type to search” behavior of the Windows Start screen, where the user doesn’t explicitly invoke the Search charm. If you haven’t done it before and you have a computer with a physical keyboard, press the Windows key to return to the Start screen, and start typing some app name *without* invoking the search charm first. Voila! The search charm appears automatically with results immediately filtered and displayed. This is essentially the same behavior that the Start button provided in previous versions of Windows, but it’s now much more visually engaging!

To enable this in your own app, simply set the [SearchPane.showOnKeyboardInput](#) property to `true`. You can enable or disable the behavior at any time through this property. Generally speaking, we recommend providing this behavior on your app’s main page(s) and on search results pages, but not on other subsidiary pages where there can be other input controls, nor on details pages showing content for a single item, nor on pages that support an in-page find capability. For details, see [Guidelines for Enabling Type to Search](#).

Launching Apps: File Type and URI Scheme Associations

Developers of Windows 8 apps have often asked whether it’s possible for one app to launch another. The answer is yes, with some restrictions (aren’t you surprised!). First, apps can be launched only through a file type or URI scheme association, not directly by name or path. To be specific, the only way for a Windows 8 app to launch another app—including desktop applications—is through the [Windows.System.Launcher](#) API that provides you with two choices:

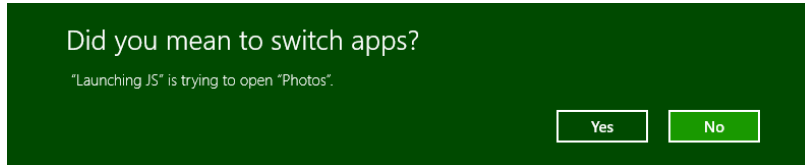
- [launchFileAsync](#) Launches another app associated with a given [StorageFile](#). An optional [LauncherOptions](#) object lets you specify a number of details (see below).
- [launchUriAsync](#) Launches another app associated with a given URI scheme, again with or without [LauncherOptions](#).

Note With both [launchFileAsync](#) and [launchUriAsync](#), Windows 8 specifically blocks apps from launching any file or URI scheme that is handled by a system component and for which there is no legitimate scenario for a Windows 8 app to insert itself into that process. The [How to handle file activation](#) and [How to handle protocol activation](#) topics lists the specific file types and URI schemes in question. The `file://` URI scheme is allowed in [launchUriAsync](#), but only for intranet URIs when you have declared the *Private Networks (Client & Server)* capability in the manifest.

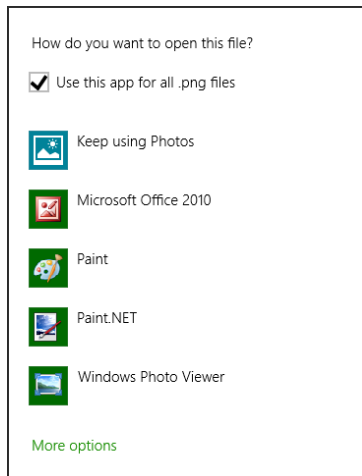
The result of both these async methods, as passed to your completed handler, is a Boolean: `true` if the launch succeeded, `false` if not. That is, barring a catastrophic failure such as a low memory condition where the async operation will outright fail, these operations normally report success to your completed handler with a Boolean indicating the outcome. You’ll get a `false` result, for example, if you try to launch a file that itself contains executable code or other files that are blocked for security reasons.

However, you cannot know ahead of time what the result will be. This is the reason for the [LauncherOptions](#) parameter, through which you can provide fallback mechanisms:

- The [treatAsUntrusted](#) option (a Boolean, default is `false`) will display a warning to the user that they'll be switching apps if they proceed (see image below). This is good to use when you're unsure about the source of the association, such as launching a URI found inside a PDF or other document, and want to prevent the user from experiencing a classic bait-and-switch!



- [displayApplicationPicker](#) (a Boolean, default is `false`) will let the user choose which app to launch as part of the process (see image below). Note that the UI allows the user to change the default app for subsequent invocations. Also, the [LauncherOptions.ui](#) property can be used to control the placement of the app picker.



- [preferredApplicationDisplayName](#) and [preferredApplicationPackageFamilyName](#) provide a suggestion to the user to acquire a specific app from the Windows Store if no other app is available to service the request. This is very useful with a particular URI scheme or file type for which you provide an app yourself.
- Similarly, [fallbackUri](#) specifies a URI to which the user will be taken if no app can be found to handle the request and you don't have a specific suggestion in the Windows Store.
- Finally, for [launchUriAsync](#), the [contentType](#) option identifies the content type associated with a URI that controls which app is launched. This is primarily useful when the URI doesn't contain a specific scheme but simply refers to a file on a network using a scheme such as `http` or `file` that would normally launch a browser for file download. With [contentType](#), the default app that's

registered for that type, rather than the scheme, will be launched. That app, of course, must be able to then use the URI to access the file. In other words, this option is a way to pass a URI, rather than a whole file, to a handler app that you know can work with that URI.

Scenarios 1 and 2 of the [Association launching sample](#) provide a demonstration of using these methods with some of the options so you can see their effects.

On the flip side, as demonstrated in Scenarios 3 and 4 of the same sample, is the question of how an app associates itself with a file type or URI scheme so that it can be launched in these ways. These associations constitute the File Activation contract and the Protocol Activation contract. In both cases the target app must declare the file types and/or URI schemes it wishes to service in its manifest and must then provide for those activation kinds, as we'll see in the following sections.

Again, file or URI scheme association is the *only* means through which a Windows 8 app can launch another, so there's no guarantee that you'll actually launch a specific app. Of course, the more unique and specific the file type or URI scheme, the less likely it is that a consumer would have multiple apps to handle the association or even that there would be many such apps in the Windows Store. Indeed, designing a unique URI scheme interface, where the scheme is fairly app-specific, is really the best way to have one Windows 8 app launch and delegate a task to another, since all kinds of data can be passed in the URI string itself. The Maps app in Windows 8, for example, supports a *bingmaps* scheme for accomplishing mapping tasks from other apps. You can imagine the same for a stocks app, a calendar app, an email app (beyond *mailto*), and so forth. If you create such a scheme and want other apps to use it, you'll certainly need to provide documentation for its usage details, which means that another app can implement the same scheme and thus offer itself as another choice in the Windows Store. So, there's no guarantee even with a very specific scheme that you can know for certain that you'll be launching another known app, but this is about as close as you can get to that capability.⁵⁸

File Activation

To declare file activation capability, first go to the Declarations section of the manifest and add a "File Type Associations" declaration, the Visual Studio UI for which is shown in Figure 12-13. Each file type can have multiple specific types (notice the Add New button under Supported File Types), such as a JPEG having .jpg and .jpeg file extensions. Note that some file types are disallowed for apps; see [How to handle file activation](#) for the complete list.

Under Properties, the Display Name is the overall name for a group of file types (this is optional; not needed if you have only one type). The Name, on the other hand, is required—it's the internal identity for the file group and one that should remain consistent for the entire lifetime of your app across all updates. In a way, the Name/Display Name properties for the whole group of file types is like your real name, and all the individual file types are nicknames—any of them ultimately refer to the core file type and your app.

⁵⁸ In any case, it's a good idea to register your URI scheme with the Internet Assigned Numbers Authority ([IANA](#)). [RFC 4395](#) is the particular specification for defining new URI schemes.

Info Tip is tooltip text for when the user hovers over a file of this type and the app is the primary association. The Logo is a little tricky; in Visual Studio here, you simply refer to a base name for an image file, like you do with other images in the manifest. In your actual project, however, you should have multiple files for the same image in different target sizes (not resolution scales): 16x16, 32x32, 48x48, and 256x256. The [Association launching sample](#) uses such images with *targetsize-** suffixes in the filenames.⁵⁹ These various sizes help Windows provide the best user experience across many different types of devices.

FIGURE 12-13 The Declarations > File Type Associations UI in the Visual Studio manifest designer.

Under Edit Flags, these options control whether an “Open” verb is available for a downloaded file of this type: checking Open Is Safe will enable the verb in various parts of the Windows UI; checking Always Unsafe disables the verb. Leaving both blank might enable the verb, depending on where the file is coming from and other settings within the system.

⁵⁹ Ignore, however, the sample’s use of *targetsize-** naming conventions for the app’s tile images; target sizes apply only to file and URI scheme associations.

At the very bottom of this UI you can also set a discrete start page for handling activations, but typically you'll use your main activation handler, as shown in `js/default.js` of the Association launching sample (leading into `js/scenario3.js`).

There you'll receive the activation kind of `file`, in which case `eventArgs.detail` is a [WebUIFileActivatedEventArgs](#): its `files` property contains the array of `StorageFile` objects from `Windows.System.Launcher.launchFileAsync`, and its `verb` property will be `"open"`. You respond, of course, by opening and presenting the file contents in whatever way is appropriate to the app.

Of course, since the file might have come from anywhere, treat it as untrusted content, as we mentioned earlier for share targets. Avoid taking permanent actions based on those the file contents.

As with the Search contract, too, be sure to test file activation when the app is already running and when it must be started anew. In all cases be sure to load app settings and restore session state if `eventArgs.detail.previousExecutionState` is `terminated`.

Protocol Activation

Creating a URI scheme association for an app is much like a file type association. In the Declarations section of the manifest, add a Protocol declaration, as shown in Figure 12-14.

The screenshot shows the Visual Studio manifest designer with the 'Declarations' tab selected. The interface is divided into several sections:

- Available Declarations:** A dropdown menu labeled 'Select one...' and an 'Add' button.
- Supported Declarations:** A list box titled 'File Type Associations' containing a single entry 'Protocol' with a 'Remove' button next to it.
- Description:** Text stating 'Register for URL Protocols, such as "mailto", on behalf of a Windows Store app. Multiple instances of this declaration are allowed in each app.' with a link to 'More information'.
- Properties:** A series of input fields:
 - Logo:** A text field containing 'images\Icon.png' with a clear button (X) and a browse button (...).
 - Display name:** An empty text field.
 - Name:** A text field containing 'alsdkjs'.
 - App settings:** An empty text field.
 - Executable:** An empty text field.
 - Entry point:** An empty text field.
 - Start page:** An empty text field.

FIGURE 12-14 The Declarations > Protocol UI in the Visual Studio manifest designer.

Under Properties, the Logo, Display Name, and Name all have the same meaning as with file type associations (see the previous section). Similarly, while you can specify a discrete start page, you'll typically handle activation in your main activation handler, as demonstrated in again in `js/default.js` of the Association launching sample (leading into `js/scenario4.js`).

There you'll receive the activation kind of `protocol`, in which case `eventArgs.detail` is a [WebUIProtocolActivatedEventArgs](#): its `uri` property contains the URI from `Windows.System.Launcher.launchUriAsync`.

Once again be warned that URIs with some unique scheme can come from anywhere, including potentially malicious sources. Be wary of any data or queries in the URI, and avoid taking permanent actions with it. For instance, you can perhaps navigate to a new page but don't modify database records to try to `eval` anything in the URI.

Nevertheless, protocol associations are a primary way that an app can provide valuable services to others when appropriate. The built-in Maps app, for example, supports a `bingmaps://` URI scheme and association, so you can just launch a URI with the appropriate format to show the user a fully interactive map instead of trying to implement such capabilities yourself. This is similar to how you rely on an email client with the `mailto:` scheme; other kinds of apps can easily create a URI scheme interface for other services and workflows.

Tip To debug protocol activation you need to be able to have the app start directly within the debugger when it's activated. To do this, open the project's properties (Project > Properties menu command in Visual Studio), and then under Configuration Properties > Debugging set Launch Application to No.

File Picker Providers

Back in Chapter 8 we looked at how the file/folder picker can be used to reference not only locations on the file system but also content that's managed by other apps or even created on-the-fly within other apps. Let's be clear on this point: the app that's *using* the file picker is doing so to obtain a `StorageFile` or `StorageFolder` for some purpose. But this does not mean that *provider* apps that can be invoked through the file picker necessarily manage their data *as* files or folders. Their role is to take whatever kind of data they manage and package it up so that it *looks* like a file/folder to the picker.

In the "Using the File Picker" section of Chapter 8, for instance, we saw how the Windows 8 Camera app can be used to take a photo and return it through the file picker. Such a photo did not exist at the time the target app was invoked; instead, it displayed its UI through which the user could essentially create a file that was then passed back through the file picker. In this way, the Camera app shortcuts the whole process of creating a new picture, providing that function exactly when the user is trying to select a picture file. Otherwise the user would have to start the Camera app separately, take a photo, store it locally, and switch to the original app to invoke the file picker and relocate that new picture.

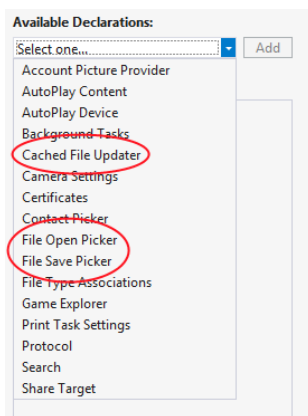
The file picker is not limited to pictures, of course: it works with any file type, depending on what the caller indicates it wants. One app might let the user go into a music library, purchase and download a track, and then return that file to the file picker. Another app might perform some kind of database query and return the results as a file, and still others might allow the user to browse online databases of file-type entities, again hiding the details of downloading and packaging that data as a file such that the user's experience of the file picker is seamless across the local file system, online resources, and apps that just create data dynamically. It's also possible to create an app that generates or acquires file-like data on the fly, such as the Camera app that allows you to take a picture or an audio app that could

record a new sound. In such cases, however, note that the file picker contracts are designed for relatively quick in-and-out experiences. For this reason an app should provide only basic editing capabilities (like cropping a photo or trimming the audio) in this context.

As with the Search and Share target contracts, Visual Studio and Blend provide an item template for file picker providers, specifically the File Open Picker contract item in the Add > New Item dialog as we've seen before (it's hiding off the top of the list in Figure 12-4). This gives you a basic selection structure built around a ListView control, but not much else. For our purposes here we won't be using this template; we'll draw on samples instead. Generally speaking, when servicing the file picker contracts, an app should use the same views and UI as it does when launched normally, thereby keeping the app experience consistent in both scenarios.

Manifest Declarations

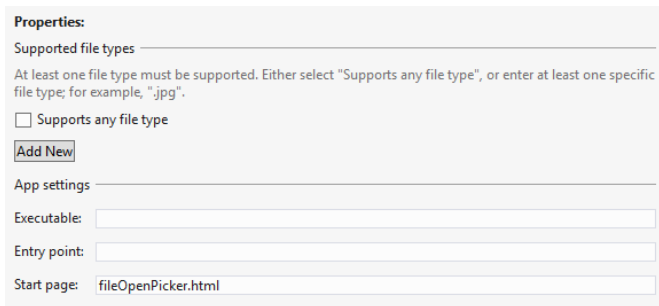
To be a provider for the file picker, an app starts by—what else!—adding the appropriate declaration to its manifest. In this case there are actually three declarations: File Open Picker, File Save Picker, and Cached File Updater, as shown below in Visual Studio's manifest designer. Each of these declarations can be made once within any given app.



The File Open Picker and File Save Picker declarations are what make a provider app available in the dialogs invoked through the `Windows.Storage.Pickers.FileOpenPicker` and `FileSavePicker` API. The calling app in both cases is completely unaware that another app might be invoked—all the interaction is between the picker and the provider app through the contract, with the contract broker being responsible for first displaying a UI through which to select an object and second for returning a `StorageFile` object for that item.

With both the File Open Picker and File Save Picker contracts, the provider app indicates in its manifest those file types that it can service. This is done through the Add New button in the image below; the file picker will then make that app available as a choice only when the calling app indicates a matching file type. The Supports Any File Type option that you see here will make the app always appear in the list, but this is appropriate only for apps like SkyDrive that provide a general storage

location. Apps that work only with specific file types should indicate only those types.



The screenshot shows the 'Properties' dialog box for a provider app. It has two main sections: 'Supported file types' and 'App settings'. The 'Supported file types' section includes a text area for supported file types, a note stating 'At least one file type must be supported. Either select "Supports any file type", or enter at least one specific file type; for example, ".jpg".', and a checkbox for 'Supports any file type'. Below this is an 'Add New' button. The 'App settings' section contains three text fields: 'Executable:', 'Entry point:', and 'Start page:'. The 'Start page:' field is populated with the value 'fileOpenPicker.html'.

The provider app indicates a Start Page for the open and save provider contracts separately—the operations are distinct and independent. In both cases, as we’ve seen for other contracts, these are the pages that the file picker will load when the user selects this particular provider app. As with Share targets, these pages are typically independent of the main app and will have their own script contexts and activation handlers, as we’ll see in the next section. (Again, the Executable and Entry Point options are there for other languages.)

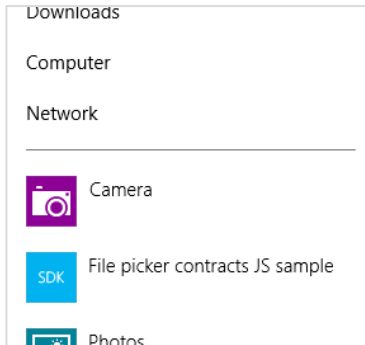
You might be asking: why are the open and save contracts separate? Won’t most apps generally provide both? Well, not necessarily. If you’re creating a provider app for a web service that is effectively read-only (like the image results from a search engine), you can serve only the file open case. If the service supports the creation of new files and updating existing files, such as a photo or document management service would, then you can also serve the file save case. There might also be scenarios where the provider would serve only the save case, such as writing to a sharing service. In short, Windows cannot presume the nature of the data sources that provider apps will work with, so the two contracts are kept separate.

While the next main section in this chapter covers the Cached File Updater contract, it’s good to know how it relates to the others here. This contract allows a provider app to synchronize local and remote copies of a file, essentially to subscribe to and manage change/access notifications for provided files. This is primarily of use to apps that represent a file repository where the user will frequently open and save files, like SkyDrive or a database app. It’s essentially a two-way binding service for files when either local or remote copies can be updated independently. As such, it’s always implemented in conjunction with the file picker provider contracts.

Tip As noted earlier in this chapter, the [Sharing and exchanging data](#) topic on the Windows Developer Center has some helpful guidance as to when you might choose to be a provider for the file save picker contract and when being a share target is more appropriate.

Activation of a File Picker Provider

Demonstrations of the file picker provider contracts—for open and save—are found in the [File picker contracts sample](#), which I'll refer to as the *provider sample* for clarity. Declarations for both are included in the manifest with Supports Any File Type, so the sample will be listed with other apps in all file pickers, as shown here:



When invoked, the Start page listed in the manifest for the appropriate contract (open or save) is loaded. These are fileOpenPicker.html and fileSavePicker.html, found in the root of the project. Both of these pages are again loaded independently of the main app and appear as shown in Figures 12-15 and 12-16. Note that the title of the app and the color scheme is determined by the Application UI settings in the provider app's manifest. In particular, the text comes from the Display Name field and the colors come from the Foreground Text and Background Color settings under Tile, as shown in Figure 12-17. Note that the system automatically adds the down chevron (v) next to the title in Figures 12-15 and 12-16 through which the user can select a different picker location or provider app.

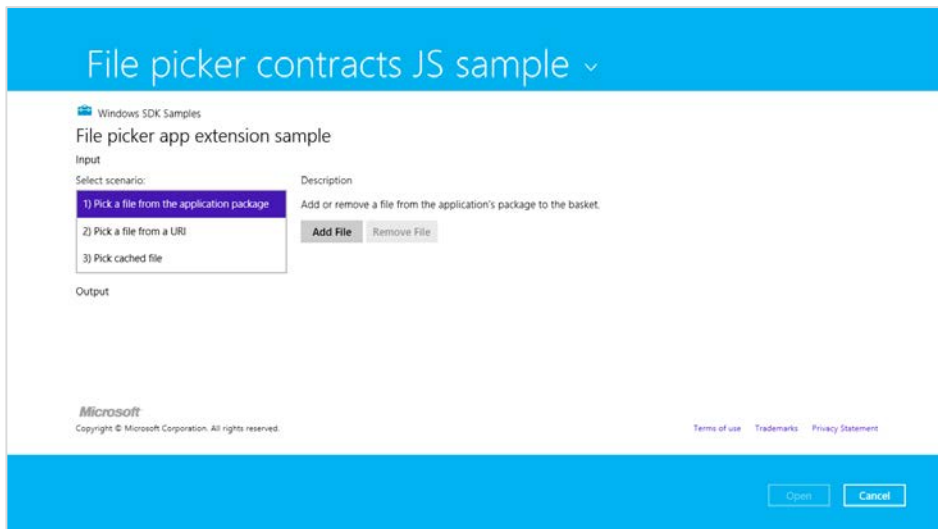


FIGURE 12-15 The Open UI as displayed by the sample.

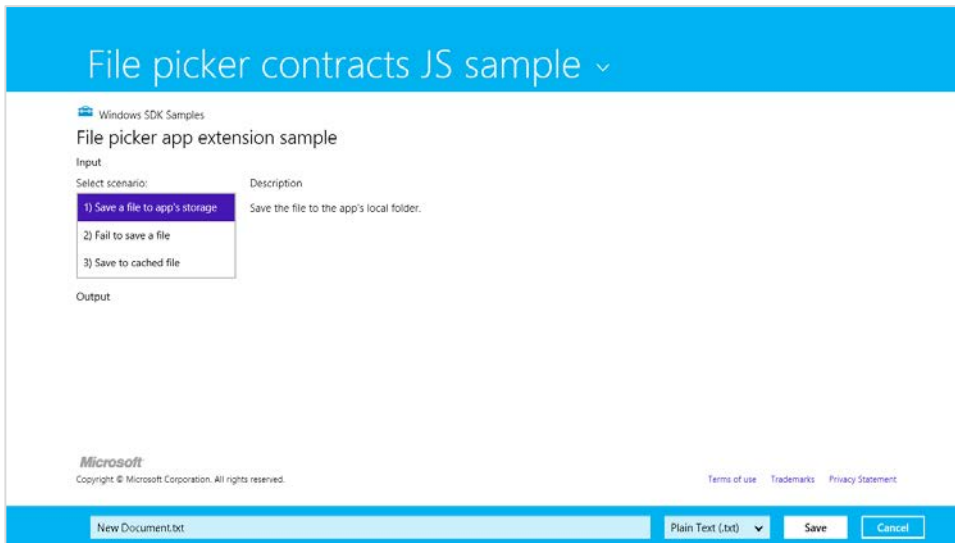


FIGURE 12-16 The Save UI as displayed by the sample.

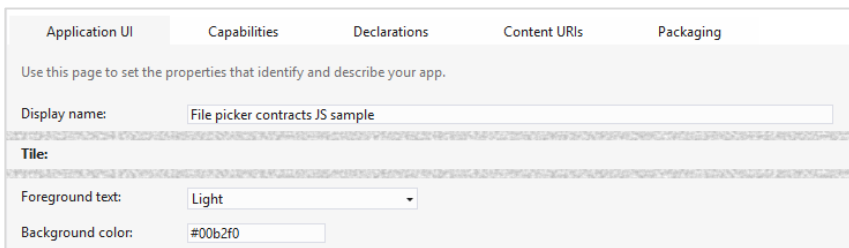


FIGURE 12-17 Application UI settings in the manifest that affect the appearance of the open and save picker UI for a provider app. The gray bars in this image represent other fields that I've omitted for brevity.

When you first run this sample, you won't see either of these pages. Instead you'll see a page through which you can invoke the file open or save pickers and then choose this app as a provider. You can do this if you like, but I recommend using a different app to invoke the pickers, just so we're clear on which app is playing which role. For this purpose you can use the sample we used in Chapter 8, the [File picker sample](#) (this is the consumer side). You can even use something like the Windows 8 Music app where the Open File command on its app bar will invoke a picker wherein the provider sample will be listed.

Whatever your choice, the important parts of the provider sample are its separate pages for servicing its contracts, which are again `fileOpenPicker.html` and `fileSavePicker.html`. In the first case, the code is contained in `js/fileOpenPicker.js` where we can see the `activated` event handler with the activation kind of `fileOpenPicker`:

```
function activated(eventObject) {
    if (eventObject.detail.kind ===
```

```

Windows.ApplicationModel.Activation.ActivationKind.fileOpenPicker) {
    fileOpenPickerUI = eventObject.detail.fileOpenPickerUI;

    eventObject.setPromise(WinJS.UI.processAll().then(function () {
        // Navigate to a scenario page...
    }));
}
}

```

Here `eventObject.detail` is a [WebUIFileOpenPickerActivatedEventArgs](#) object, whose `fileOpenPickerUI` property (a [Windows.Storage.Pickers.Providers.FileOpenPickerUI](#) object) provides the means to fulfill the provider's responsibilities with the contract.

In the second case, the code is in `js/fileSavePicker.js` where the activation kind is `fileSavePicker`:

```

function activated(eventObject) {
    if (eventObject.detail.kind ===
        Windows.ApplicationModel.Activation.ActivationKind.fileSavePicker) {
        fileSavePickerUI = eventObject.detail.fileSavePickerUI;

        eventObject.setPromise(WinJS.UI.processAll().then(function () {
            // Navigate to a scenario page
        }));
    }
}

```

where `eventObject.detail` is a [WebUIFileSavePickerActivatedEventArgs](#) object. As with the open contract, the `fileSavePickerUI` property of this (a [Windows.Storage.Pickers.Providers.-FileSavePickerUI](#) object) provides the means to fulfill the provider's side of the contract.

In both open and save cases, the contents of the contract's Start page is displayed within the letterboxed area between the system-provided top and bottom bands. If that content overflows the provided space, scrollbars would be provided only within that area—the top and bottom bands always remain in place. In both cases, WinRT also provides the usual features for activation, such as the `splashScreen` and `previousExecutionState` properties, just as we saw in Chapter 3, "App Anatomy and Page Navigation," meaning that you should reload necessary session state and use extended splash screens as needed.

What's most interesting, though, are the contract-specific interactions that are represented in the different scenarios for these pages (as you can see in Figures 12-15 and 12-16). Let's look at each.

Note For specific details on designing a file picker experience, see [Guidelines for file pickers](#).

File Open Provider: Local File

The provider for file open works through the `FileOpenPickerUI` object supplied with the `fileOpen-Picker` activation kind. Simply said, whatever kind of UI the provider offers to select some file or data will be wired to the various methods, properties, and events of this object.

First, the UI will use the `allowedFileTypes` property to filter what it displays for selection—clearly, the provider should not display items that don't match what the file picker is being asked to pick! Next, the UI can use the `selectionMode` property (a `FileSelectionMode` value) to determine if the file picker was invoked for `single` or `multiple` selection.

When the user selects an item within the UI, the provider calls the `addFile` method with the `StorageFile` object as appropriate for that item. Clearly, the provider has to somehow create that `StorageFile` object. In the sample's open picker > Scenario 1, this is accomplished with a `StorageFolder.getFileAsync` (where the `StorageFolder` is the package location).

```
Windows.ApplicationModel.Package.current.installedLocation
    .getFileAsync("images\\squareTile-sdk.png").then(function (fileToAdd) {
        addFileToBasket(localFileId, fileToAdd);
    })
```

where `addFileToBasket` just calls `FileOpenPickerUI.addFile` and displays messages for the result. That result is a value from `Windows.Storage.Pickers.Provider.AddFileResult`: `added` (success), `alreadyAdded` (redundant operations, so the file is already there), `notAllowed` (adding is denied due to a mismatched file type), and `unavailable` (app is not visible). These really just help you report the result to users in your UI. Note also that the `canAddFile` method might be helpful for enabling or disabling add commands in your UI as well, which will prevent some of these error cases from ever arising in the first place.

The provider app must also respond to requests to remove a previously added item, as when the user removes a selection from the “basket” in the multi-select file picker UI. To do this, listen for the `FileOpenPickerUI` object's `fileRemoved` event, which provides a file ID as an argument. You pass this ID to `containsFile` followed by `removeFile` as in the sample (`js/fileOpenPickerScenario1.js`):

```
// Wire up the event in the page's initialization code
fileOpenPickerUI.addEventListener("fileremoved", onFileRemovedFromBasket, false);

function removeFileFromBasket(fileId) {
    if (fileOpenPickerUI.containsFile(fileId)) {
        fileOpenPickerUI.removeFile(fileId);
    }
}
```

If you need to know when the file picker UI is closing your page (such as the user pressing the Open or Cancel buttons as shown in Figure 12-15), listen for the `closing` event. This gives you a chance to close any sessions you might have opened with an online service and otherwise perform any necessary cleanup tasks. In the `eventArgs` you'll find an `isCanceled` property that indicates whether the file picker is being canceled (`true`) or if it's being closed due to the Open button (`false`). The `eventArgs.closingOperation` object also contains a `getDeferral` method and a `deadline` property that allows you to carry out async operations as well, similar to what we saw in Chapter 3 for the `suspending` event.

A final note is that a file picker provider should respect the `FileOpenPickerUI.settings-Identifier` to relaunch the provider to a previous state (that is, a previous picker session). If you remember from the other side of this story, an app that's using the file picker can use the `settings-Identifier` to distinguish different use cases within itself—perhaps to differentiate certain file types or feature contexts. The identifier can also differ between different apps that invoke the file picker. By honoring this property, then, a provider app can maintain a case-specific context each time it's invoked (basically using `settingsIdentifier` in its appdata filenames and the names of settings containers), which is how the built-in file pickers for the file system works.

It's also possible for the provider app to be suspended while displaying its UI and could possibly be shut down if the calling app is closed. However, if you manage picker state based on `settings-Identifier` values, you don't need to save or manage any other session state where your picker functionality is concerned.

File Open Provider: URI

For the most part, Scenario 2 of the open file picker case in the provider sample is just like we've seen in the previous section. The only difference is that it shows how to create a `StorageFile` from a nonfile source, such as an image that's obtained from a remote URI. In this situation we need to obtain a data stream for the remote URI and convert that stream into a `StorageFile`. Fortunately, a few WinRT APIs make this very simple, as shown in `js/fileOpenPickerScenario2.js` within its `onAddFileUri` method:

```
function onAddUriFile() {
    // Respond to the "Add" button being clicked
    var imageSrcInput = document.getElementById("imageSrcInput");

    if (imageSrcInput.value !== "") {
        var uri = new Windows.Foundation.Uri(imageSrcInput.value);
        var thumbnail =
            Windows.Storage.Streams.RandomAccessStreamReference.createFromUri(uri);

        // Retrieve a file from a URI to be added to the picker basket
        Windows.Storage.StorageFile.createStreamedFileFromUriAsync("URI.png", uri,
            thumbnail).then(function (fileToAdd) {
                addFileToBasket(uriFileId, fileToAdd);
            },
            function (error) {
                // ...
            });
    } else {
        // ...
    }
}
```

Here `Windows.Storage.StorageFile.createStreamedFileFromUriAsync` does the honors to give us a `StorageFile` for a URI, and `addFileToBasket` is again an internal method that just calls the `addFile` method of the `FileOpenPickerUI` object.

Note that if you need to perform authentication or take any other special steps to obtain content from a web service, you'll generally want to use the [Windows.Networking.BackgroundTransfer](#) API to acquire the content (where you can provide credentials), followed by [StorageFile.createStreamedFile](#) to then serve that file up through the contract. [StorageFile.createStreamedFileFromUriAsync](#) does exactly this but doesn't provide for authentication.

File Save Provider: Save a File

Similar to how the file open provider interacts with a [FileOpenPickerUI](#) object, a provider app for saving files works with the specific methods, properties, and events [FileSavePickerUI](#) class. Again, the open and save contracts are separate concerns because the data source for which you might create a provider app might or might not support save operations independently of open. If you do support both, you will likely reuse the same UI and would thus use the same Start page and activation path.

Within the [FileSavePickerUI](#) class, we first have the [allowedFileTypes](#) as provided by the app that invoked the file save picker UI in the first place. As with open, you'll use this to filter what you show in your own UI so that users can clearly see what items for these types already exist. You'll also typically want to populate a file type drop-down list with these types as well.

For restoring the provider's save UI for the specific calling app from a previous session, there is again the [settingsIdentifier](#) property.

Referring back to Figure 12-16, notice the controls along the bottom of the screen, the ones that are automatically provided by the file picker UI when the provider app is invoked. When the user changes the filename field, the provider app can listen for and handle the [FileSavePickerUI](#) object's [filenameChanged](#) event; in your handler you can get the new value from the [fileName](#) property. If the provider app has UI for setting the filename, it cannot write to this property, however. It must instead call [trySetFileName](#), whose return value from the [SetFileNameResult](#) enumeration is either [succeeded](#), [notAllowed](#) (typically a mismatched file type), or [unavailable](#). This is typically used when the user taps an item in your list, where the expected behavior is to set the filename to the name of that item.

The most important event, of course, happens when the user finally taps the Save button. This will fire the [FileSavePickerUI](#) object's [targetFileRequested](#) event. You must provide a handler for this event, in which you must create an empty [StorageFile](#) object in which the app that invoked the file picker UI can save its data. The name of this [StorageFile](#) must match the [fileName](#) property.

The [eventArgs](#) for this event is a [Windows.Storage.Pickers.Providers.TargetFile-Requested-EventArgs](#) object. This contains a single property named [request](#), which is a [TargetFileRequest](#). Its [targetFile](#) property is where you place the [StorageFile](#) you create (or [null](#) if there's an error). You must set this property before returning from the event handler, but of course you might need to perform asynchronous operations to do this at all. For this purpose, as we've seen many times, the request also contains a [getDeferral](#) method. This is used in Scenario 1 of the provider sample's save case ([js/fileSavePickerScenario1.js](#)):

```
function onTargetFileRequested(e) {  
    var deferral = e.request.getDeferral();
```

```

// Create a file to provide back to the Picker
Windows.Storage.ApplicationData.current.localFolder.createFileAsync(
    fileSavePickerUI.fileName).done(function (file) {
    // Assign the resulting file to the targetFile property and complete the deferral
    e.request.targetFile = file;
    deferral.complete();
}, function () {
    // Set the targetFile property to null and complete the deferral to indicate failure
    e.request.targetFile = null;
    deferral.complete();
});
};

```

In your own app you will, of course, replace the `createFileAsync` call in the local folder with whatever steps are necessary to create a file or data object. Where remote files are concerned, on the other hand, you'll need to employ the Cached File Updater contract (see "Cached File Updater" below).

File Save Provider: Failure Case

Scenario 2 of the provider sample's save UI just shows one other aspect of the process: displaying errors in case there is a real failure to create the necessary `StorageFile`. Generally speaking, you can use whatever UI you feel is best and consistent with the app in general, to let the user know what they need to do. The sample uses a `MessageDialog` like so:

```

function onTargetFileRequestedFail(e) {
    var deferral = e.request.getDeferral();

    var messageDialog = new Windows.UI.Popups.MessageDialog("If the app needs the user to
correct a problem before the app can save the file, the app can use a message like this to
tell the user about the problem and how to correct it.");

    messageDialog.showAsync().done(function () {
        // Set the targetFile property to null and complete the deferral to indicate failure
        // once the user has closed the dialog. This will allow the user to take any
        // necessary corrective action and click the Save button once again.
        e.request.targetFile = null;
        deferral.complete();
    });
};

```

Cached File Updater

Using the cached file updater contract provides for keeping a local copy of a file in sync with one managed by a provider app on some remote resources. This contract is specifically meant for apps that provide access to a storage location where users regularly save, access, and update files. The SkyDrive app in Windows is a good example of this. In other cases where the user is generally going to pick a file and use it some scenario but not otherwise come back to it, using the file picker contracts is entirely sufficient.

Back in Chapter 8, we saw some of the method calls that are made by an app that uses the file picker: `Windows.Storage.CachedFileManager.deferUpdates` and `Windows.Storage.CachedFileManager.completeUpdatesAsync`. This usage is shown in Scenarios 4 and 6 of the [File picker sample](#) we worked with in that chapter. Simply said, these are the calls that a file-*consuming* app makes if and when it writes to a file that it obtained from a file picker. It does this because it won't know (and shouldn't care) whether the file provider has another copy in database, web service, etc., that needs to be kept in sync. If the provider needs to handle synchronization, the consuming app's calls to these methods will trigger the necessary cached file updater UI of the provider app, which might or might not be shown, depending on the need. Even if the consuming app doesn't call these methods, the provider app will still be notified of changes but won't be able to show any UI.

There are two directions with which this contract works, depending on whether it's needed to update a *local* (cached) copy of a file or the *remote* (source) copy. In the first case, the provider is asked to update the local copy, typically when the consuming app attempts to access that file (pulling it from the `FutureAccessList` or `MostRecentlyUsed` list of `Windows.Storage.AccessCache`; it does not explicitly ask for an update). In the second case, the consuming app has modified the file such that the provider needs to propagate those changes to its source copy.

From a provider app's point of view, the need for such updates comes into play whenever it supplies a file to another app. This can happen through the file picker contracts, as we've seen in the previous section, but also through file type associations as well as the share contract. In the latter case a share source app is, in a sense, a file provider and might make use of the cached file updater contract as well. In short, if you want your file-providing app to be able to track and synchronize updates between local and remote copies of a file, this is the contract to use.

Supporting the contract begins with a manifest declaration as shown below, where the Start page indicates the page implementing the cached file updater UI. That page will handle the necessary events to update files and might or might not actually be displayed to the user, as we'll see later.

Available Declarations: Select one... <input type="button" value="Add"/>	Description: Registers the app as a cached file updater, allowing the app to provide updates to files that are accessed by other Windows 8 apps. Only one instance of this declaration is allowed per app. More information
Supported Declarations: Cached File Updater <input type="button" value="Remove"/> File Open Picker File Save Picker	Properties: App settings: _____ Executable: <input type="text"/> Entry point: <input type="text"/> Start page: <input type="text" value="cachedFileUpdater.html"/>

The next step for the provider is to indicate when a given [StorageFile](#) should be hooked up with this contract. It does so by calling [Windows.Storage.Provider.CachedFileUpdater.setUpdateInformation](#) on a provided file as shown in Scenario 3 of the [File picker contracts sample](#), which I'll again refer to as the *provider sample* for simplicity (js/fileOpenPickerScenario3.js):

```
function onAddFile() {
    // Respond to the "Add" button being clicked

    Windows.Storage.ApplicationData.current.localFolder.createFileAsync("CachedFile.txt",
        Windows.Storage.CreationCollisionOption.replaceExisting).then(function (file) {
        Windows.Storage.FileIO.writeTextAsync(file, "Cached file created...").then(
            function () {
                Windows.Storage.Provider.CachedFileUpdater.setUpdateInformation(
                    file, "CachedFile",
                    Windows.Storage.Provider.ReadActivationMode.beforeAccess,
                    Windows.Storage.Provider.WriteActivationMode.notNeeded,
                    Windows.Storage.Provider.CachedFileOptions.requireUpdateOnAccess);
                addFileToBasket(localFileId, file);
            }, onError);
        }, onError);
};
```

Note [setUpdateInformation](#) is within the [Windows.Storage.Provider](#) namespace and is different from the [Windows.Storage.CachedFileManager](#) object that's used on the other side of the contract; be careful to not confuse the two.

The [setUpdateInformation](#) method takes the following arguments:

- A [StorageFile](#) for the file in question.
- A content identifier string that identifies the remote resource to keep in sync.
- A [ReadActivationMode](#) indicating whether the calling app can read its local file without updating it; values are [notNeeded](#) and [beforeAccess](#).
- A [WriteActivationMode](#) indicating whether the calling app can write to the local file and whether writing triggers an update; values are [notNeeded](#), [readOnly](#), and [afterWrite](#).
- One or more values from [CachedFileOptions](#) (that can be combined with bitwise-OR) that describes the ways in which the local file can be accessed without triggering an update; values are [none](#) (no update), [requireUpdateAccess](#) (update on accessing the local file), [useCachedFileWhenOffline](#) (will update on access if the calling app desires, and access is allowed if there's no network connection), and [denyAccessWhenOnline](#) (triggers an update on access and requires a network connection).

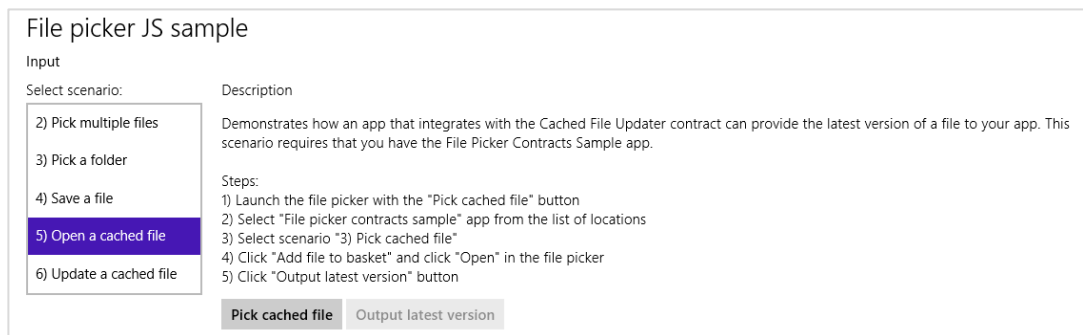
It's through this call, in other words, that the provider specifically controls how and when it should be activated to handle updates when a local file is accessed.

So, together we have two cases where the provider app will be invoked and might be asked to show its UI: one where the calling app updates the file, and another when the calling app attempts to access the file but needs an update before reading its contents.

Before going into the technical details, let's see how these interactions appear to the user. To see the cached file updater in action using the sample, invoke it by using the file picker from another app. First, then, run the provider sample to make sure its contracts are registered. Then run the aforementioned [File picker sample](#). In the latter, Scenarios 4, 5, and 6 cause interactions with the cached file updater contract. Scenarios 4 and 6 write to a file to trigger an update to the remote copy; Scenario 5 accesses a local file that will trigger a local update as part of the process.

Updating a Local File: UI

In Scenario 5 (updating a local file), start by tapping the Pick Cached File button in the UI shown here:



This will launch the provider sample. In that view, select Scenario 3 so that you see the UI shown in Figure 12-18. This is the mode of the provider sample that is just a file picker *provider*, (`js/fileOpenPickerScenario3.js`) where it calls `setUpdateInformation`. This is *not* the UI for the cached file updater yet. Click the Add File to Basket button, and tap the Open button. This will return you to the first app (the picker sample in the above graphic) where the Output Latest Version button will now be enabled. Tapping *that* button will then invoke the provider sample through the cached file updater contract, as shown in Figure 12-19. This is what appears when there's a need to update the local copy of the cached file.

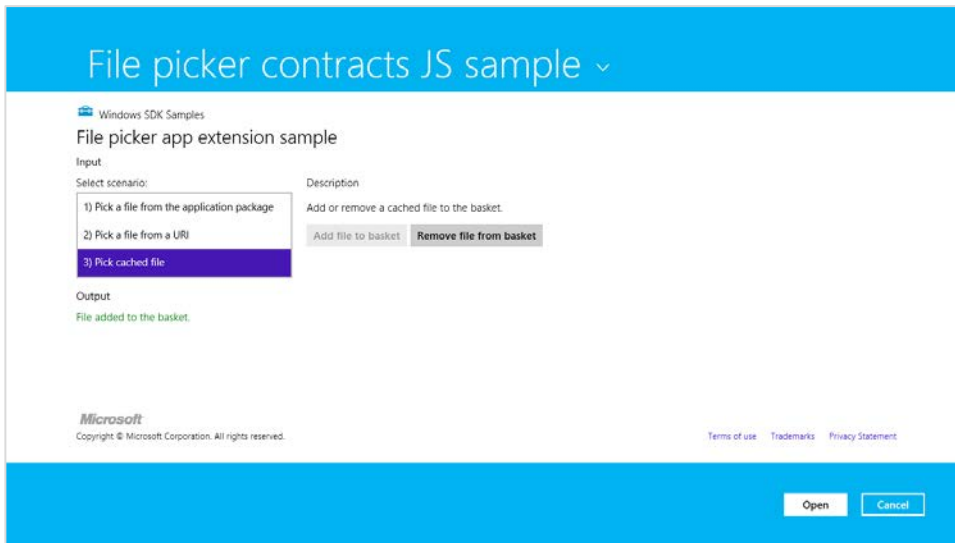


FIGURE 12-18 The provider sample's UI for picking a file; the `setUpdateInformation` method is called on the provided file to set up the cached file updater relationship.

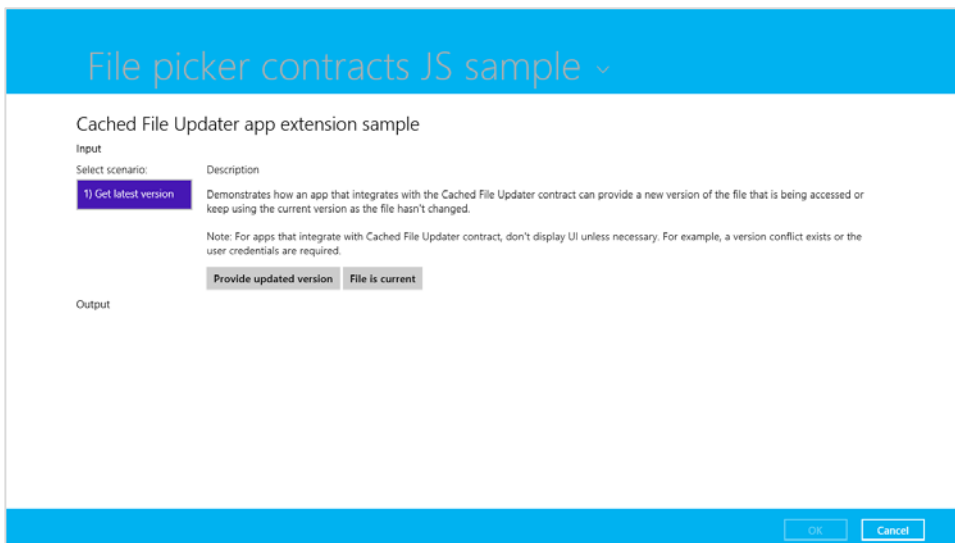
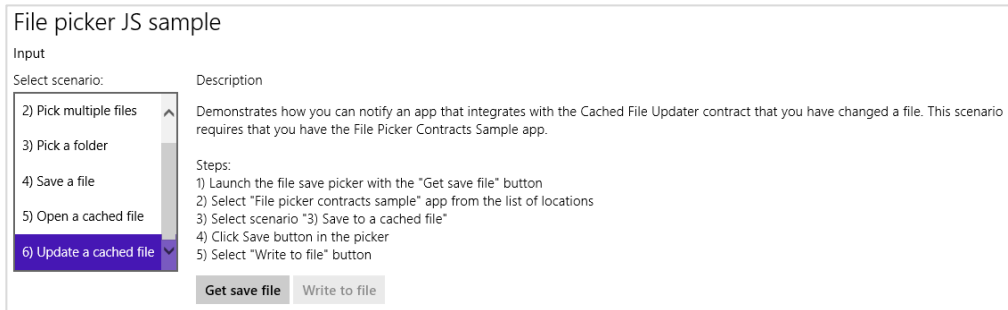


FIGURE 12-19 The provider sample's UI for the cached file updater contract on a local file.

Take careful note of the description in the sample. While the sample shows this UI by default, a cached file updater app will not show it unless it's necessary to resolve conflicts or collect credentials. Oftentimes no such interaction is necessary and the provider silently provides an update to the local file or indicated that the file is current. The sample's UI here is simply providing both those options as explicit choices (and be sure to choose one of them because selecting Cancel will throw an exception).

Updating a Remote File: UI

In Scenario 6 (updating a remote file) of the file picker sample, we can see the interactions that take place when the consuming app writes changes to its local copy, thereby triggering an update to the remote copy. Start by tapping the Get Save File button in the UI shown next:



In the picker, select the provider sample as the picker source, which invokes the UI of Figure 12-20 through the file save picker contract, implemented through `html/fileSavePickerScenario3.html` and `js/fileSavePickerScenario3.js`. If you look in the JavaScript file, you'll again see a call to `setUpdateInformation` that's called when you enter a file name and tap Save. Doing so also returns you to the picker sample above where Write To File should now be enabled. Tapping Write To File then reinvokes the provider sample through the cached file updater contract with the UI shown in Figure 12-21. This UI is intended to demonstrate how such a provider app would accommodate overwriting or renaming the remote file.

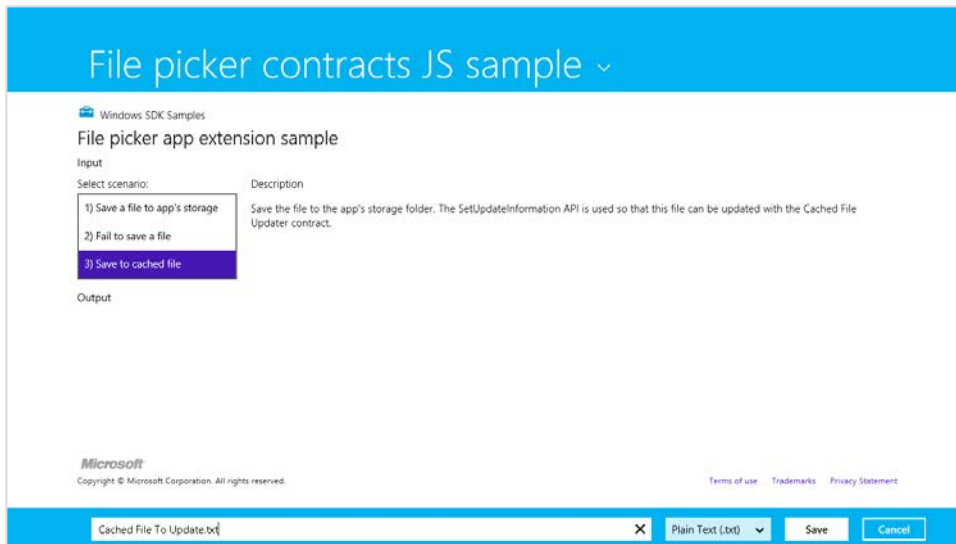


FIGURE 12-20 The provider sample's UI for saving a file; the `setUpdateInformation` method is again called on the provided file to set up the cached file updater relationship.

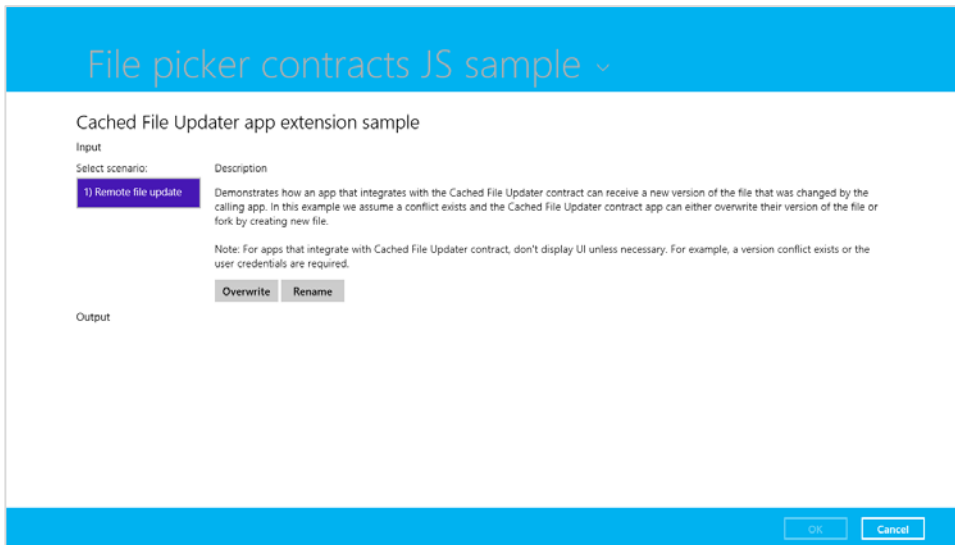


FIGURE 12-21 The provider sample's UI for the cached file updater contract on a remote file.

Update Events

Let's see how the cached file updater contract looks in code. As you will by now expect, the provider app is launched, the Start page (`cachedFileUpdater.html` in the project root) is loaded, and the activated handler is called with the activation kind of `cachedFileUpdater`. This will happen for both local and remote cases, and as we'll see here, you use the same activation code for both. Here `eventObject.detail` is a `WebUICachedFileUpdaterActivatedEventArgs` that contains a `cachedFileUpdaterUI` property (a `CachedFileUpdaterUI`) along with the usual roster of `kind`, `previousExecutionState`, and `splashScreen`. Here's how it looks in `js/cachedFileUpdater.js` of the provider sample:

```
function activated(eventObject) {
    if (eventObject.detail.kind ===
        Windows.ApplicationModel.Activation.ActivationKind.cachedFileUpdater) {
        cachedFileUpdaterUI = eventObject.detail.cachedFileUpdaterUI;

        cachedFileUpdaterUI.addEventListener("fileupdaterequested", onFileUpdateRequest);
        cachedFileUpdaterUI.addEventListener("uirequested", onUIRequested);

        switch (cachedFileUpdaterUI.updateTarget) {
            case Windows.Storage.Provider.CachedFileTarget.local:
                // Code omitted: configures sample to show cachedFileUpdaterScenario1
                // if needed.
                break;

            case Windows.Storage.Provider.CachedFileTarget.remote:
                // Code omitted: configures sample to show cachedFileUpdaterScenario2
                // if needed.
                break;
        }
    }
}
```

```

    }
}
}

```

When the provider app is invoked to update a local file from the remote source, the `cachedFileUpdaterUI.updateTarget` property will be `local`, as you can see above. When the app is being asked to update a remote file with local changes, the target is `remote`. All the sample does in these cases is point to either `html/cachedFileUpdaterScenario1.html` (Figure 12-19) or `html/cachedFileUpdaterScenario2.html` (Figure 12-21) as the update UI.

The UI is not actually shown initially. What happens first is that the `CachedFileUpdaterUI` object fires its `fileUpdateRequested` event to attempt a silent update. Here the `eventArgs` is a `File-UpdateRequestedEventArgs` object with a single `request` property (`FileUpdateRequest`), an object that you'll want to save in a variable that's accessible from your update UI.

If it's possible to silently update a local file, follow these steps:

- Because you'll likely be doing async operations to perform the update, obtain a deferral from `request.getDeferral`.
- To update the contents of the local file, use one of these options:
 - If you already have a `StorageFile` with the new contents, just call `request.update-LocalFile`. This is a synchronous call, in which case you do not need to obtain a deferral.
 - The local file's `StorageFile` object will be in `request.file`. You can open this file and write whatever contents you need within it. This will typically start an async operation, after which you return from the event handler.
- To update the contents of a remote file, copy the contents from `request.file` to the remote source.
- Depending on the outcome of the update, set `request.status` to a value from `FileUpdate-Status`: `complete` (the copies are sync'd), `incomplete` (sync didn't work but the local copy is still available), `userInputNeeded` (the update failed for need of credentials or conflict resolution), `currentlyUnavailable` (source can't be reached, and the local file is inaccessible), `failed` (sync cannot happen now or ever, as when the source file has been deleted), and `completeAndRenamed` (the source version has been renamed, generally to resolve conflicts).
- If you asked for a deferral and processed the outcome within completed and error handlers, call the deferral's `complete` method to finalize the update.

Now the provider might know ahead of time that it can't do a silent update at all—a user might not be logged into the back-end service (or credentials are needed each time), there might be a conflict to resolve, and so forth. In these cases the event handler here should check the value of `cachedFile-UpdaterUI.uiStatus` (a `UIStatus`) and set the `request.status` property accordingly:

- If the UI status is `visible`, switch to that UI and return from the event handler. Complete the deferral when the user has responded through the UI.
- If UI status is `hidden`, set `request.status` to `userInputNeeded` and return. This will trigger the `CachedFileUpdaterUI.onuiRequested` event followed by another `fileUpdate-Requested` event where `uiStatus` will be `visible`, in which case you'll switch to your UI.
- If the UI status is `unavailable`, set `request.status` to `currentlyUnavailable`.

You can see some of this in the sample's `onFileUpdateRequest` handler; it really handles only the `uiStatus` check because it doesn't attempt silent updates at all (as described in the comments below):

```
function onFileUpdateRequest(e) {
    fileUpdateRequest = e.request;
    fileUpdateRequestDeferral = fileUpdateRequest.getDeferral();

    // Attempt a silent update using fileUpdateRequest.file silently, or call
    // fileUpdateRequest.updateLocalFile in the local case, setting fileUpdateRequest.status
    // accordingly, then calling fileUpdateRequestDeferral.complete(). Otherwise, if you
    // know that user action will be required, execute the following code.

    switch (cachedFileUpdaterUI.uiStatus) {
        case Windows.Storage.Provider.UIStatus.hidden:
            fileUpdateRequest.status =
                Windows.Storage.Provider.FileUpdateStatus.userInputNeeded;
            fileUpdateRequestDeferral.complete();
            break;

        case Windows.Storage.Provider.UIStatus.visible:
            // Switch to the update UI (configured in the activated event)
            var url = scenarios[0].url;
            WinJS.Navigation.navigate(url, cachedFileUpdaterUI);
            break;

        case Windows.Storage.Provider.UIStatus.unavailable:
            fileUpdateRequest.status = Windows.Storage.Provider.FileUpdateStatus.failed;
            fileUpdateRequestDeferral.complete();
            break;
    }
}
```

Again, if a silent update succeeds, the provider app's UI never appears to the user. In the case of the provider sample, since it never attempts to do a silent update, it always does the check on `uiStatus`. When the app was just launched to service the contract, we'll end up in the `hidden` case and return `userInputNeeded`, as would happen if you attempted a silent update but returned the same status. Either way, the `CachedFileUpdateUI` object will fire its `uiRequested` event, telling the provider app that the system is making the UI visible. The app, in fact, can defer initializing its UI until this event occurs because there's no need to do so for a silent update.

After this, the `fileUpdateRequested` event will fire again with `uiStatus` now set to `visible`. Notice how the code above will have called `request.getDeferral` in this case but has not called its `complete`.

We save that step for when the UI has done what it needs to do (and, in fact, we save both the request and the deferral for use from the UI code).

The update UI is responsible for gathering whatever user input is necessary to accomplish the task: collecting credentials, choosing which copy of a file to keep (the local or remote version), allowing for renaming a conflicting file (when updating a remote file), and so forth. When updating a local file, it writes to the `StorageFile` within `request.file` or calls `request.updateLocalFile`; in the remote case it copies data from the local copy in `request.file`.

To complete the update, the UI code then sets `request.status` to `complete` (or any other appropriate code if there's a failure) and calls the deferral's `complete` method. This will change the status of the system-provided buttons along the bottom of the screen, as you can see in Figure 12-19 and Figure 12-21—enabling the OK button and disabling Cancel. In the provider sample, both buttons just execute these two lines for this purpose:

```
fileUpdateRequest.status = Windows.Storage.Provider.FileUpdateStatus.complete;  
fileUpdateRequestDeferral.complete();
```

All in all, the interactions between the system and the app for the cached file updater contract are simple and straightforward in themselves: handle the events, copy data around as needed, and update the request status. The real work with this contract is first deciding when to call `setUpdateInformation` and then providing the UI to support updates of local and remote files under the necessary circumstances. This will, of course, involve more interactions with your backend storage system.

Contacts

The last contract we'll explore in this chapter (whew!) is that of the contact picker. We haven't seen this feature of Windows 8 in action yet. Let's take a look at it first and then explore how the picker is used from one side of the contract and how an provider app fulfills the other side.

A contact, as you probably expect, is information about a person that includes details like name, phone numbers, email addresses, and so forth. An obvious place you'd need a contact is to compose an email, as shown in Figure 12-22. Here, tapping the + controls to the right of the To and Cc fields will open the contact picker, which defaults to the Windows 8 People app, as shown in Figure 12-23 (its splash screen) and Figure 12-24 (its multiselect picker view, where I have blurred my friends' identities so that they don't start blaming me for unwanted attention!). As we saw with the File Picker UI, the provider app supplies the UI for the middle portion of the screen while Windows supplies the top and bottom bars, the header, and the down-arrow menu control using information from the provider app's manifest. (Refer back to Figure 12-17.) Figure 12-25 shows the appearance of the [Contact Picker app sample](#) in its provider mode, as well as the menu that allows you to select a different provider (those who have declared themselves as a contact provider).

When I select one or more contacts in any provider app and press the Select button along the bottom of the screen, those contacts are then brought directly back to the first app—Mail in this case.

Just as the file picker contract allowed the user to navigate into data surfaced as files by any other app, the contact contract (say that ten times fast!) lets the user easily navigate to people you might select from any other source.

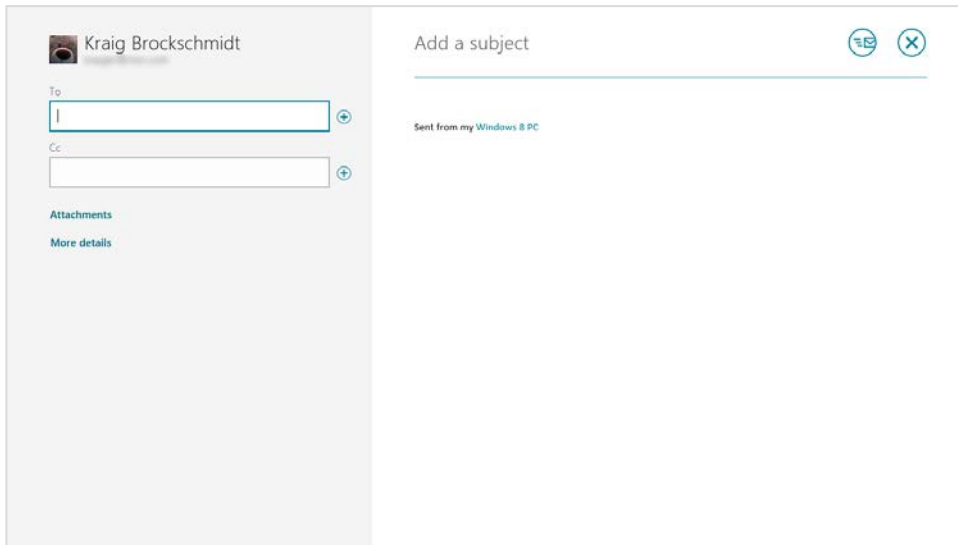


FIGURE 12-22 The Mail app uses the contact picker to choose a recipient.

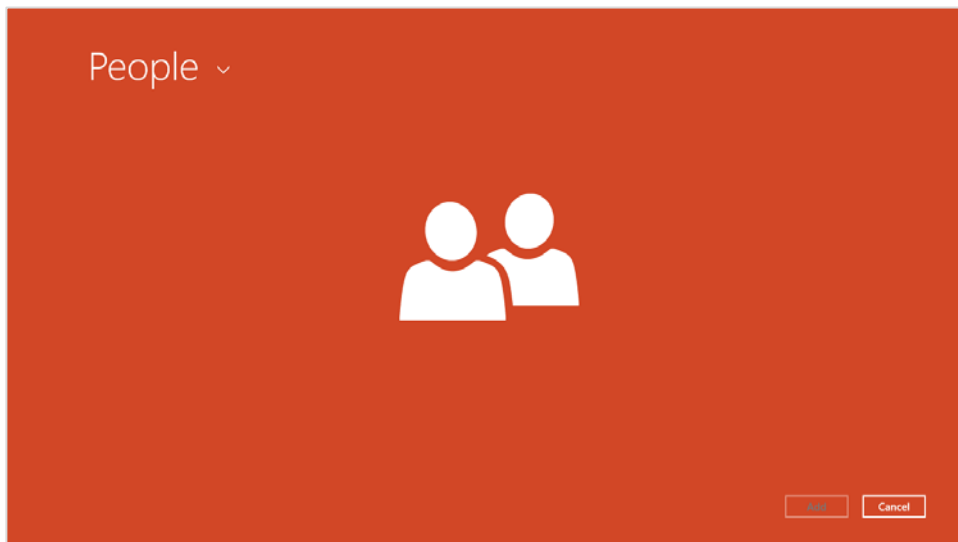


FIGURE 12-23 The People app on startup when launched as a contact provider.

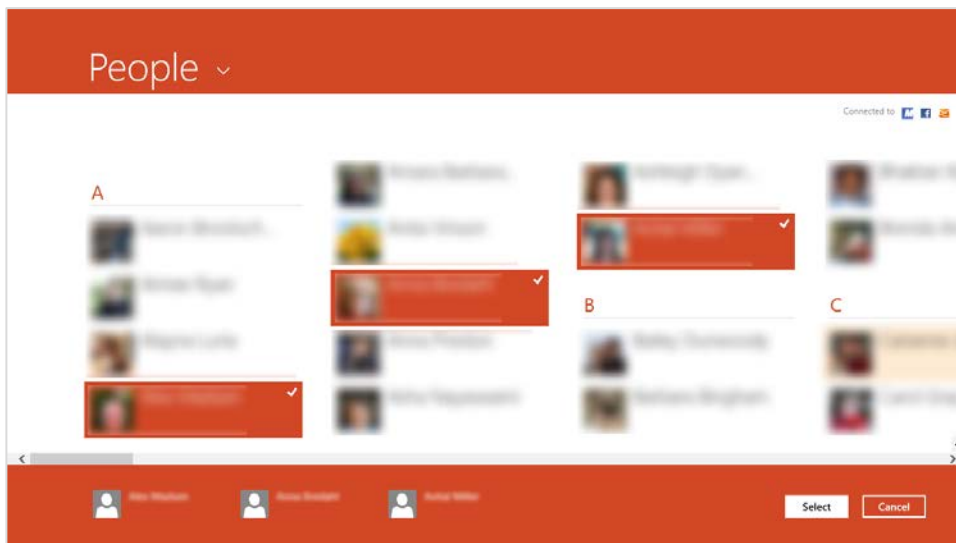


FIGURE 12-24 The picker UI within the People app, shown for multiple selection (with my friends blurred because they're generally not looking for fame amongst developers). The selections are gathered along the bottom in the basket.

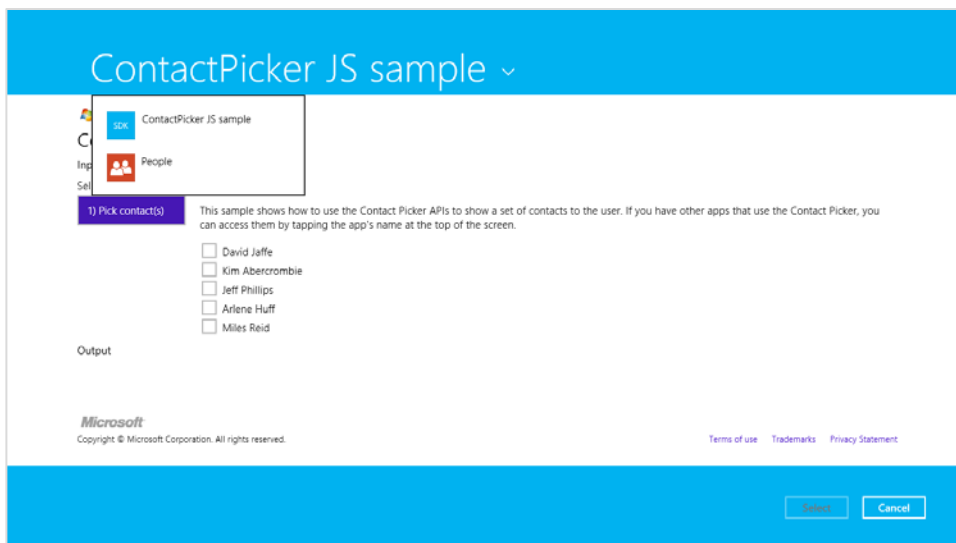


FIGURE 12-25 The Contact Picker sample's UI when used as a provider, along with the header flyout menu allowing selection of a picker provider.

Using the Contact Picker

Contacts as a whole work with the API in the `Windows.ApplicationModel.Contacts` namespace. An app that consumes contacts sees each one represented by an instance of the [Contact-Information](#) class, whose properties like `name`, `phoneNumbers`, `locations`, `emails`, `instant-Messages`, and `customFields` give you the contact information, along with the `getThumbnailAsync` and `queryCustomFields` methods.

Choosing a contract happens through a picker UI much like the file picker, invoked through [Windows.ApplicationModel.Contacts.ContactPicker](#). After creating an instance of this object, you can set the `commitButtonText` for the first (left) button in the picker UI (as with “Select” in the earlier figures). You can also set the `selectionMode` property to a value from the [Contact-SelectionMode](#) enumeration: either `contact` (the default) or `fields`. In the former case, the whole contact information is returned; in the latter, the picker works against the contents of the picker’s [desiredFields](#). Refer to the documentation on that property for details.

When you’re ready to show the UI, call the picker’s `pickSingleContactAsync` or `pickMultipleContactsAsync` methods. These provide your completed handler with a single [ContactInformation](#) object or a vector of them, respectively. As with the file picker, note that these APIs will throw an exception if called when the app is in snapped view, so you’ll want to avoid this case.

Picking a single contact and displaying its information is demonstrated in Scenario 1 of the [Contact Picker app sample](#) (`js/scenarioSingle.js`):

```
var picker = new Windows.ApplicationModel.Contacts.ContactPicker();
picker.commitButtonText = "Select";

// Open the picker for the user to select a contact
picker.pickSingleContactAsync().done(function (contact) {
    if (contact !== null) {
        // Consume the contact information...
    }
});
```

Choosing multiple contacts (Scenario 2, `js/scenarioMultiple.js`) works the same way, just using `pickMultipleContactsAsync`. In either case, the calling app then applies the [Contact-Information](#) data however it sees fit, such as populating a To or Cc field like the Mail app. However, other than the `name` property in that object, which is just a string, its properties have a little more structure, as shown in the following table.

Property	Type	Field Properties and Types
emails phoneNumbers customFields	Vector of ContactField	category (ContactFieldCategory), name (string), type (a ContactFieldType), value (string)
instantMessages	Vector of ContactInstant-MessageField	Same as ContactField above plus <code>displayText</code> , <code>launchUri</code> , <code>service</code> , and <code>userName</code> (all strings)
locations	Vector of ContactLocationField	Same as ContactField above plus <code>city</code> , <code>country</code> , <code>postalCode</code> , <code>region</code> , <code>street</code> , and <code>unstructuredAddress</code> (all strings)

Accordingly, the sample consumes a [ContactInformation](#) object as follows, first extracting the individual vector properties:

```
appendFields("Emails:", contact.emails, contactElement);
appendFields("Phone Numbers:", contact.phoneNumbers, contactElement);
appendFields("Addresses:", contact.locations, contactElement);
```

and then enumerating the contents of those vectors and in this case creating elements with their contents. Other apps will, of course, transfer the values to appropriate fields or other parts of the app UI—what’s shown here demonstrates processing of the different categories:

```
function appendFields(title, fields, container) {
    // Creates UI for a list of contact fields of the same type, e.g. emails or phones
    fields.forEach(function (field) {
        if (field.value) {
            // Append the title once we have a non-empty contact field
            if (title) {
                container.appendChild(createTextElement("h4", title));
                title = "";
            }

            // Display the category next to the field value
            switch (field.category) {
                case Windows.ApplicationModel.Contacts.ContactFieldCategory.home:
                    container.appendChild(createTextElement("div",
                        field.value + " (home)"));
                    break;
                case Windows.ApplicationModel.Contacts.ContactFieldCategory.work:
                    container.appendChild(createTextElement("div",
                        field.value + " (work)"));
                    break;
                case Windows.ApplicationModel.Contacts.ContactFieldCategory.mobile:
                    container.appendChild(createTextElement("div",
                        field.value + " (mobile)"));
                    break;
                case Windows.ApplicationModel.Contacts.ContactFieldCategory.other:
                    container.appendChild(createTextElement("div",
                        field.value + " (other)"));
                    break;
                case Windows.ApplicationModel.Contacts.ContactFieldCategory.none:
                default:
                    container.appendChild(createTextElement("div", field.value));
                    break;
            }
        }
    })
}
```

```

    }
  });
}

```

Contact Picker Providers

On the provider side, which is also demonstrated in the [Contact Picker app sample](#), we see the same pattern as for file picker providers. First, a provider app needs to declare the Contact Picker contract in its manifest, where it indicates the Start page to load within the context of the picker. In the sample, the Start page is `contactPicker.html` that in turn loads `html/contactPickerScenario.html` (with their associated JavaScript files):

The screenshot shows the 'Available Declarations' section with a dropdown menu set to 'Select one...' and an 'Add' button. Below it, the 'Supported Declarations' section lists 'Contact Picker' with a 'Remove' button. To the right, the 'Description' section states: 'Registers the app as a people picker, making contact details in the app available to other Windows 8 apps. Only one instance of this declaration is allowed per app.' with a 'More information' link. The 'Properties' section includes 'App settings', 'Executable:', 'Entry point:', and 'Start page:' (set to 'contactPicker.html').

As with the file picker, having a separate Start page means having a separate activated handler, and in this case it looks for the activation kind of `contactPicker` (`js/contactPicker.js`):

```

function activated(eventObject) {
    if (eventObject.detail.kind ===
        Windows.ApplicationModel.Activation.ActivationKind.contactPicker) {
        contactPickerUI = eventObject.detail.contactPickerUI;
        eventObject.setPromise(WinJS.UI.processAll().then(function () {
            // ...
        }));
    }
}

```

The `eventObject.detail` here is a [ContactPickerActivatedEventArgs](#) (these names are long, but at least they're predictable!). As with all activations, it contains `kind`, `previous-ExecutionState`, and `splashScreen` properties for the usual purposes. Its `contactPickerUI` property, a [ContactPickerUI](#), then contains the information specific for the contact picker contract:

- The `selectionMode` and `desiredFields` properties as supplied by the calling app.
- Three methods—`addContact`, `removeContact`, and `containsContact`—for managing what's returned to the calling app. These methods correspond to the actions of a typical selection UI.
- One event, `contactsRemoved`, which informs the provider when the user removes an item from the basket along the bottom of the screen. (Refer back to Figure 12-24.)

Within a provider, each contact is represented by a `Windows.ApplicationModel.Contacts.Contact` object. A provider will create an object for each contact it supplies. In the sample (`js/contactPickerScenario.js`), there's an array called `sampleContacts` that simulates what would more typically come from a database. That array just contains JSON records like this:

```
{
  name: "David Jaffe",
  homeEmail: "david@contoso.com",
  workEmail: "david@cpandl.com",
  workPhone: "",
  homePhone: "248-555-0150",
  mobilePhone: "",
  address: {
    full: "3456 Broadway Ln, Los Angeles, CA",
    street: "",
    city: "",
    state: "",
    zipCode: ""
  },
  id: "761cb6fb-0270-451e-8725-bb575eeb24d5"
},
```

Each record is shown as a check box in the sample's UI (generated in the `createContactUI` function), which is a quick and easy way to show a selectable list of items! Of course, your own provider app will likely use a `ListView` for this purpose; the sample is just trying to keep things simple so that you can see what's happening with the contract itself.

When a contact is selected, the sample's `addContactToBasket` function is called. This is the point at which we create the actual `Contact` object and call `ContactPickerUI.addContact`. The process here for each field follows a chain of other function calls, so let's see how it works for the single `homeEmail` field in the source record, starting with `addContactToBasket` (again in `js/contactPicker-Scenario.js`). The rest of the field values are handled pretty much the same way:

```
function addContactToBasket(sampleContact) {
  var contact = new Windows.ApplicationModel.Contacts.Contact();
  contact.name = sampleContact.name;

  appendEmail(contact.fields, sampleContact.homeEmail,
    Windows.ApplicationModel.Contacts.ContactFieldCategory.home);

  // Add other fields...

  // Add the contact to the basket
  switch (contactPickerUI.addContact(sampleContact.id, contact)) {
    // Show various messages based on the result, which is of type
    // Windows.ApplicationModel.Contacts.Provider.AddContactResult
  }
}
```

As you can see, the *homeEmail* field is passed to a function called `appendEmail`, where the first argument is the vector (`Contact.fields`) in which to add the field and the third parameter is the category (*home*). These are then passed through to another generic function, `appendField`, where the *type* of the field has been thrown into the mix, all of which is used to create a `ContactField` object and add it to the contact:

```
function appendEmail(fields, email, category) {
    // Adds a new email to the contact fields vector
    appendField(fields, email,
        Windows.ApplicationModel.Contacts.ContactFieldType.email, category);
}

function appendField(fields, value, type, category) {
    // Adds a new field of the desired type, either email or phone number
    if (value) {
        fields.append(new Windows.ApplicationModel.Contacts.ContactField(value,
            type, category));
    }
}
```

In short, this is essentially how all the fields in a contact are assembled, one bit at a time.

Now, when an item is unselected in the list, it needs to be removed from the basket:

```
function removeContactFromBasket(sampleContact) {
    // Programmatically remove the contact from the basket
    if (contactPickerUI.containsContact(sampleContact.id)) {
        contactPickerUI.removeContact(sampleContact.id);
    }
}
```

Similarly, when the user removes an item from the basket, the contact provider needs to update its selection UI by handling the `contactremoved` event:

```
contactPickerUI.addEventListener("contactremoved", onContactRemoved, false);

function onContactRemoved(e) {
    // Add any code to be called when a contact is removed from the basket by the user
    var contactElement = document.getElementById(e.id);
    var sampleContact = sampleContacts[contactElement.value];
    contactElement.checked = false;
}
```

You'll notice that we haven't said anything about closing the UI, and in fact the `ContactPickerUI` object does not have an event for this. Simply said, when the user selects the commit button (with whatever text the caller provided), it gets back whatever the provider has added to the basket. If the user taps the cancel button, the operation returns a `null` contact. In both cases, the provider app will be suspended and, if it wasn't running prior to being activated for the contact, close automatically.

Do note that as with file picker providers, a contact provider needs to be ready to save its session state when suspended such that it can restore that state when relaunched with `previousExecution-State` set to `terminated`. Although not demonstrated in the sample, a real provider app should save its current selections and viewing position within its list, along with whatever else, to session state and restore that in its `activated` handler when necessary.

What We've Just Learned

- Contracts in Windows 8 provide the ability for any number of apps to extend system functionality as well as extend the functionality of other apps. Through contracts, installing more apps that support them creates a richer overall environment for users.
- The Share contract provides a shortcut means through which data from one app can be sent to another, eliminating many intermediate steps and keeping the user in the context of the same app. A source app packages data it can share when the Share charm is invoked; target apps consume that data, often copying it elsewhere as in an email message, text message, social networking service, and so forth.
- The Share target provides for delayed rendering of items (such as graphics), for long-running operations (such as when it's necessary to upload large data files to a service), and for providing quicklinks to specific targets within the same app (such as frequent email recipients).
- The Search contract provides integration between an app and the Search charm. From the charm users can search the current app as well as any others that support the contract, easily viewing results from other apps without having to manually launch them or switch to them. The search contract allows apps to also provide query suggestions and result suggestions.
- File type and URI scheme associations are how apps can launch other apps. an app's associations are declared in its manifest allowing it to be launched to service those associations. URI scheme associations are an excellent means for an app to provide workflow services to others.
- Apps that implement the provider side of the file picker contract appear as choices within the file picker UI. This is how apps can present data sources they manage as if they were part of the local file system, even though they might exist in databases, online services, or other such locations. To the user, the necessary transport considerations are transparent, and through the cached file updater contract a provider app can also handle synchronization of local and remote copies of the file.
- The contract for Contacts works similarly to the file picker but with information about people. A consuming app can easily invoke the contact picker UI and any number of provider apps can implement the other side of the contract to surface an address book, database, or other source through that UI.

Chapter 13

Tiles, Notifications, the Lock Screen, and Background Tasks

At the risk of seriously dating myself once again, I can still remember how a friend and I marveled when we first acquired modems that allowed us to do an online chat. At that time the modems ran at a whopping 300 baud (not Kb, not Mb—just b) and we connected by one of us calling the other’s phone number directly. It would have been far more efficient for us to just talk over the phone lines we were tying up with our bitstreams! Such was the early days of the kind of connectivity we enjoy today, where millions of services are ready to provide us with just about any kind of information we seek with transfer speeds that once challenged the limits of believability.

Even so, almost from the genesis of online services it’s been necessary to enter some kind of app, be it a client app or a web app, to view that information and get updates. When computers could run *only* a single app at a time (like the one I was using with that 300 baud modem⁶⁰), this could become quite cumbersome, and it made it difficult, if not impossible, to take data from one program and transfer it to another. With the advent of multitasking operating systems like Windows, you could run such apps side-by-side and transfer information between them, a model that has stuck with us for several decades now. Even many web apps, for the most part, still operate this way. There have been innovations in this space, certainly, such as mashups that bring disparate information together into a more convenient place, but such an experience is still hidden within an app.

That changes with Windows 8. As one columnist recently put it, “Using Windows 8 is like living in a house made out of Internet...The Start screen is a brilliant innovation, [a] huge improvement on the folder-littered desktops on every other OS, which serve exactly no purpose except to show a background photo. The Start screen makes it possible to check a dozen things”—if not more, I might add!—“in five seconds—from any app, just tap the Windows key and you can check to see if you have a new email, an upcoming appointment, inclement weather, or any breaking news. Tap the Windows key again and you’re back to your original app.”⁶¹ He goes on to suggest how long this would take if you had to go into individual apps to check the same information, even with high-speed broadband!

What makes the Start screen come alive in this way are what we call *live tiles*, Microsoft’s answer to the need to bring information from many sources together at the core of the user experience, an experience that “is constantly changing and updating,” as the same writer puts it, “because its every fiber is connected to the Internet.” With live tiles, each one is a small window onto whatever wealth of

⁶⁰ If you want the actual make and model, you’ll have to look for it in the footnotes of Chapter 1 of my book *Mystic Microsoft*, found on mysticmicrosoft.com or through my website, kraigbrockschmidt.com.

⁶¹ From [This is my next: Windows 8](#) by David Pierce.

information an app is built around; the app is essentially extracting the most important pieces of that information according to each user's particular interests. And as the user adds more apps to the system—which adds more tiles to the Start screen that the user can rearrange and group however he or she likes—the whole information experience becomes richer.

Even so, live tiles and the Start screen are just the beginning of the story. It's ironic that this chapter has one of the longest titles in the entire book, listing off four things that do not, at first glance, appear to be related: tiles, notifications, the lock screen, and background tasks. Maybe you're just thinking that I couldn't figure out where else to put them all! In truth, however, they together form what is essentially a single topic: how apps work with Windows 8 to create an environment that is *alive with activity* while those apps are often not actually running or are allowed to run just a little bit.

Let's begin, then, by exploring those relationships and the general means through which apps wire their tiles and other notifications to their information sources.

Before going further Refer back to the section named "Systemwide Enabling and Disabling of Animations" early in Chapter 11, "Purposeful Animations," and check your Control Panel setting. If "unnecessary" animations are turned off, live tiles won't be animated and you won't see the complete experience they can provide.

Second, because all the topics of this chapter are related to one another, various aspects that I'll discuss in one section of this chapter—in the context of tiles, for example—also apply to other sections—such as toast notifications. For this reason it is best to read this chapter from start to finish.

Third, many aspects of what we cover in this chapter are not enabled within the Windows Simulator, such as live tiles, toast notifications, and the lock screen. When running some of the samples within Visual Studio, be sure to use the Local Machine or Remote Machine debugging options.

Finally, the tile and notifications API is generally found within `Windows.UI.Notifications`, which is a lot to spell out every time. Unless noted, assume that the WinRT APIs we're talking about come from that namespace.

Alive with Activity: A Visual Tour

When an app is first acquired from the Windows Store and installed on a device, its primary *app tile* is, as we know well already, added to the Start screen. These tiles can be *square* or *wide*, depending on what graphics the app provides in its package. If an app provides both a square and wide graphic, the user can, as shown in Figure 13-1, use an app bar command to change the width.



FIGURE 13-1 The typical default Start screen with the built-in apps and the app bar showing the command (third from left) to make a wide tile smaller, into a square tile. The same command on a square tile might appear as Larger (see overlay) if the app supports wide tiles.

When you first installed Windows 8 on a device, you might not have noticed that the Start screen was somewhat quiet, tiles for a few built-in apps (like Weather, News, and Bing) were updating, but most of them were static. But as soon as you ran some of those apps—which I imagine you did within a couple of seconds!—the Start screen suddenly lit up much more, with many tiles changing every few seconds as I attempt to show in Figures 13-2 and 13-3. This is because apps need to be run once to make their initial connections to their associated web services and enable their *live tiles*⁶².

⁶² This is assuming two things. First is that you have Internet connectivity, which I mention with great irony because at this very moment there's a fiber optic breakdown between Sacramento and Oakland, California, that has myself and many thousands of others completely offline! Second, I'm assuming that you've acquired and installed a copy of Windows 8 on a development machine where the only preinstalled apps are those built into Windows. If you have a machine that came with Windows 8 already on it, chances are you have some additional preinstalled third-party apps. These and the built-in apps are effectively allowed to have live tiles from the get-go because the apps can be initially run prior to the system image being placed on the machine.

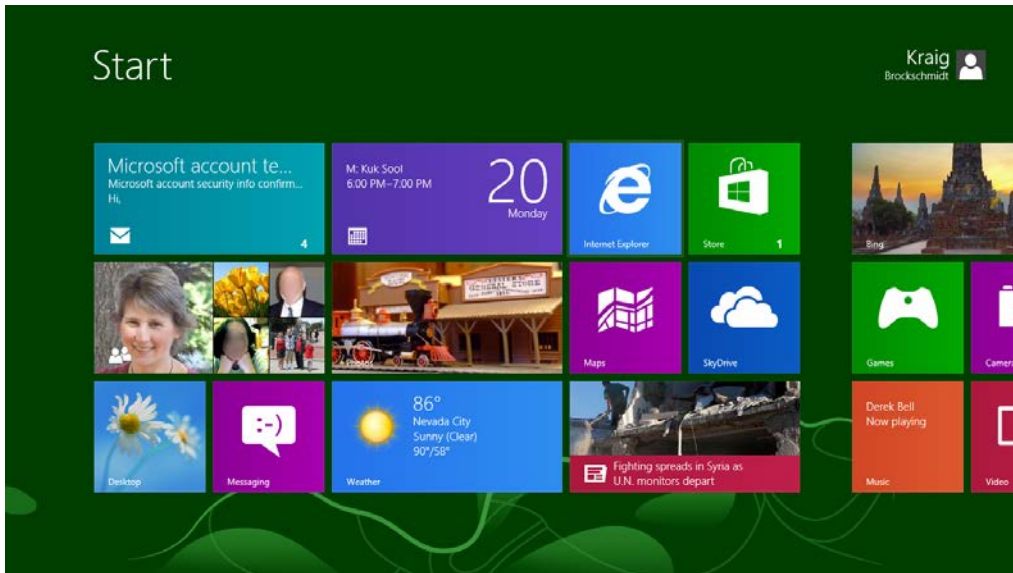


FIGURE 13-2 A Start screen after running most of the built-in apps and taking some initial configuration steps, such as connecting the People app to my Facebook account.



FIGURE 13-3 The same Start screen a few seconds later after a number of apps have updated their tiles.

What can appear on any given tile is quite extensive and varied. As you can see in the previous figures, square and wide tiles can display text, images, an app name or logo (at the lower left), and other small glyphs or numbers called *badges* at the lower right (on the Mail and Store tiles in Figure 13-3, for example).

Selecting an item on the Start screen also invokes the app bar, shown in Figure 13-4, which offers commands to unpin a tile from the Start screen, uninstall the app, change the tile size (as we've seen), and turn off updates for a particular live tile.

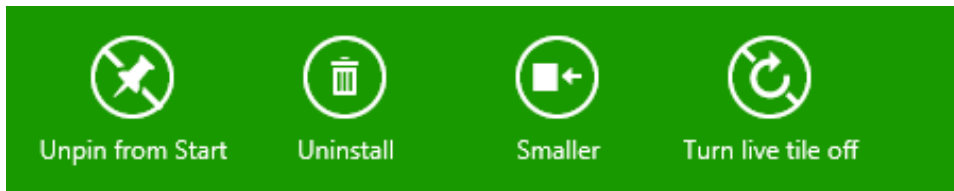


FIGURE 13-4 The app bar for the Start screen when a tile is selected. The Turn Live Tile Off command will disable updates for a given tile, so be careful not to annoy your customers with too much noise!

Tiles can receive updates even when the app isn't running, as we'll see in the next section, "The Four Sources of Updates and Notifications." Tiles can also cycle through up to five updates, an important feature that reduces the overall number of updates that actually need to be retrieved from the Internet (thus using less power). That is, by cycling through different updates a tile will continue to appear alive even though it is only receiving new updates in a timeframe of 5–15 minutes instead of 5–15 seconds.

Tip Even though live tiles can be updated frequently through push notifications, be careful not to abuse that right. Think of live tiles as views into app content rather than gadgets: avoid trying to make a live tile an app experience unto itself (like a clock) because you cannot rely on high-frequency updates. Furthermore, a tile update consists only of XML that defines the tile content—updates cannot trigger the execution of any code. In the end, think about the real experience you want to deliver through your live tile and use the longest update period you can that will still achieve that goal.

In the introduction I mentioned how acquiring more apps from the Windows Store is a way that the Start screen becomes increasingly richer. But new apps are not the only way that more tiles might appear. Apps can also create *secondary tiles* with all the capabilities of the app tile. Secondary tiles are essentially ways to create bookmarks into views of an app. A secondary tile is typically created through a Pin command on the app bar. Upon the app's request to create the tile, Windows automatically prompts the user for confirmation as shown for the Weather app in Figure 13-5, thus always keeping the user in control of their Start screen (that is, apps cannot become litterbugs on that real estate!). In this case the Weather app lets you pin secondary tiles for each location you've configured; the secondary tile always includes specific information that is given back to the app when it's launched, allowing it to navigate to the appropriate page.

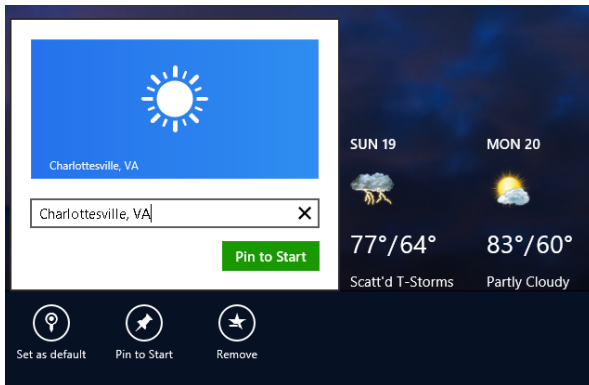
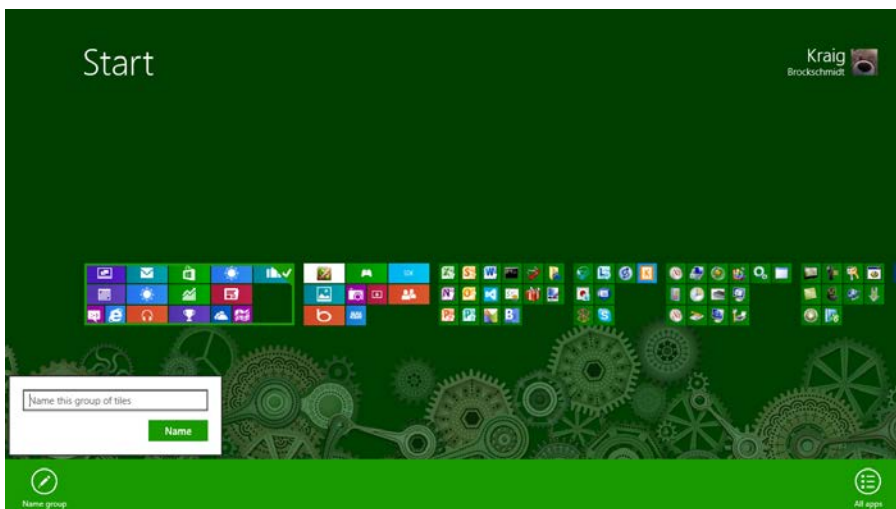


FIGURE 13-5 Pinning a secondary tile in the Weather app by using a Pin To Start app bar command, shown here with the automatic confirmation prompt.

In the People app, similarly, you can pin—that is, create secondary tiles for—specific individuals. In the Mail app you can pin different accounts and folders. In Internet Explorer you can pin your favorite websites. You get the idea: secondary tiles let you populate the Start screen with very personalized views into different apps. The user can also unpin any app tile at any time (including the primary app tile, as can happen when one has created a number of secondary tiles for more specific views). An app can ask to unpin a tile as well, in response to which Windows will again prompt the user for confirmation.

User tip You probably know that you can drag and drop tiles around on the Start screen into different sections. But did you also know that you can create group headers for those sections? To do this, do a semantic zoom out on the Start screen (a pinch gesture, Ctrl+mouse wheel down, or the Ctrl+minus key), select a group, and then invoke the Name Group command on the app bar.



In many ways, live tiles might reduce the need for a user to ever launch the app that's associated with a tile. Yet this isn't really the case. Because tiles are limited in size and must adhere to predefined configurations (templates), they simultaneously provide essential details while serving as teasers. They give you enough useful information for an at-a-glance view but not so much that your appetite for details is fully satisfied. Instead of being a deterrent to starting apps, they're actually an invitation: they both inform and engage. For this reason, I suspect that live tiles will be considered an essential app feature where they are appropriate and that apps that should provide them but don't will see lower ratings in the Windows Store.

I encourage you to be creative in thinking about what kinds of interesting information you might surface on a tile, even if your app doesn't have anything to do with the Internet. Games, for example, can cycle through tile updates that show progress on various levels, high scores, new challenges, and so forth—all of which invite the user to re-engage with that app. Do remember, though, that the user can always disable updates for any given tile, so don't give them a reason to defeat your purpose altogether!

As additional background on live tiles, check out the [Updating live tiles without draining your battery post](#) on the Building Windows 8 blog. It's good background on the system's view of efficiently managing tiles.

Now, for all the excellence of live tiles, the Start screen isn't actually where users will be spending the majority of their time—we expect them, of course, to mostly be engaged in apps themselves. Even so, users may want to be notified when important events occur, such as the arrival of an email, the triggering of an alarm, or perhaps a change in traffic conditions that indicates a good time to head home for the day (or a change in weather conditions that indicates a great time to go out skiing!).

For this purpose—surfacing typically time-sensitive information from apps that aren't in the foreground—Windows 8 provides *toast notifications*. These transient messages pop up (like real toast but without the bread crumbs) in the upper right corner of the screen (upper left in right-to-left languages). They appear on top of the foreground app as shown in Figure 13-6, as well as the Start screen and the desktop. Up to three toasts can appear at any one time, and each can be accompanied by a predefined sound, if desired.

Toasts are, like tile updates, created using predefined templates and can be composed of images, text, and logos; they always use the originating app's color scheme, as defined in that app's manifest (the foreground text and Background color settings in the Application UI section).

The purpose of toasts is, again, to give the user alerts and other time-sensitive information, but by default they appear only for a short time before disappearing. The default toast duration is five seconds, but this can be set to as long as five minutes in PC Settings > Ease of Access, as shown in Figure 13-7. Apps can create long-duration toasts that remain visible for 25 seconds or the Ease of Access setting, whichever is longer. Furthermore, apps can create a looping toast for events like a phone call or other situation where another human being might be waiting on the other end and it's appropriate to keep the notification active for some time.



FIGURE 13-6 Up to three toast notifications can appear on top of the foreground app (including the desktop and the Start screen). Each notification can also play one of a small number of predefined sounds.

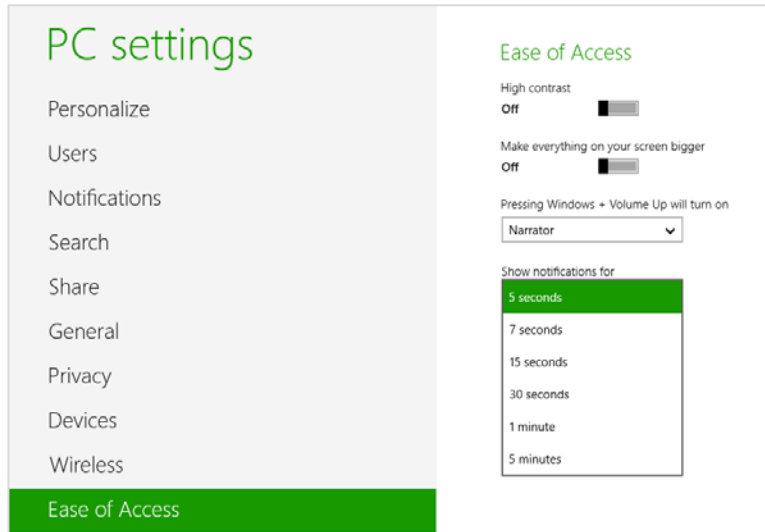


FIGURE 13-7 Toast duration settings (a drop-down list) in PC Settings > Ease of Access.

As with tile updates, the user has complete control over toast notifications: for the entire system, for the lock screen, and for individual apps. Users do this through PC Settings > Notifications, as shown in Figure 13-8. This ultimately means that you want to make your notifications valuable to the user; if you

toss up lots of superfluous toast, chances are that the user will turn them off for your app or for the whole system (and give you bad reviews in the Windows Store).

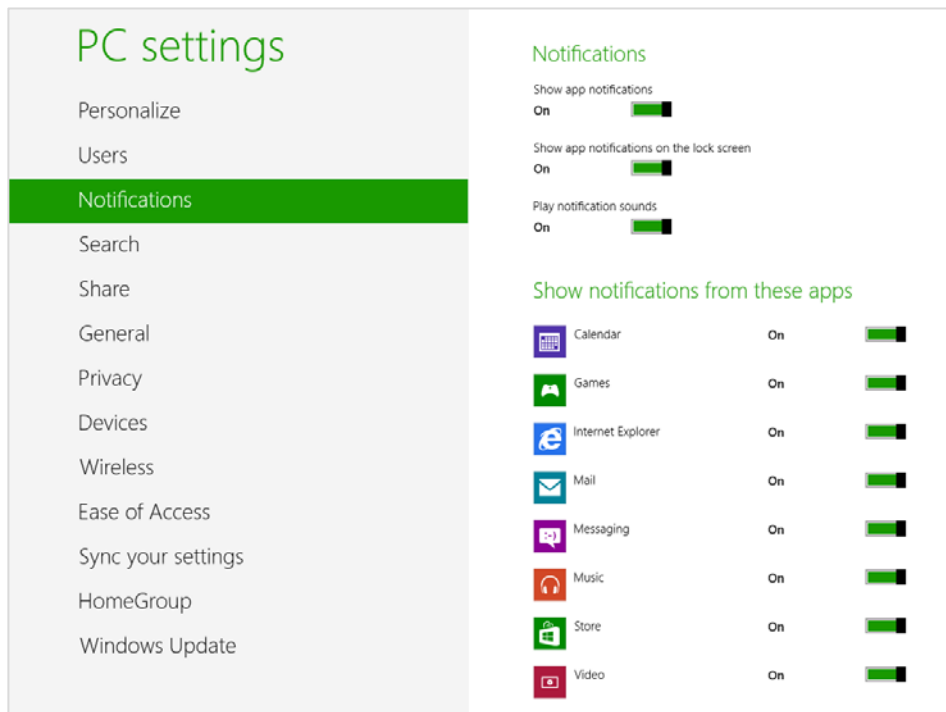


FIGURE 13-8 The user can exercise fine control over notifications in PC Settings > Notifications.

As with secondary tiles, each toast notification contains specific data that is given to its associated app when it's activated. If the app is suspended, of course, Windows switches to that app and fires its `activated` event with the notification data. If the app isn't running, Windows will launch it. (The Win+V key, by the way, will cycle the keyboard focus through active toasts, such that pressing Enter will activate it.)

This brings up the point that toast notifications, like tile updates, can originate from sources other than a running app—which should be obvious because nonforeground apps will typically be suspended! Again, we'll talk about those sources in the next section. At the same time, you might be wondering if the last item in this chapter's title—background tasks—comes into play here.

Indeed it does! As we've already seen with background audio apps in Chapter 10, "Media," it's not a hard-and-fast rule that apps are always suspended in the background.⁶³ It's just that Windows, on its quest to optimize battery life, doesn't allow arbitrary apps to keep themselves running for arbitrary reasons. Instead, Windows allows apps to run focused background tasks for specific purposes—called

⁶³ There are also APIs to configure background data transfers while an app isn't running, as we'll see in Chapter 14, "Networking."

triggers—subject to specific quotas on CPU time and network I/O. As you might expect, an app declares such background tasks in its manifest.

Triggers include a change in network connectivity, a time zone change, an update of an app, the expiration of a timer (with a 15-minute resolution), or the arrival of a *push notification* from an online source (that is, a notification sent in response to a condition that’s completely external to the device itself). Each trigger can also be configured with conditions such as whether there is Internet connectivity or not. Whatever the case, the whole purpose of background tasks is not to launch an app—in fact, background tasks cannot display arbitrary UI. It is rather to allow them to update their internal state and, when needed, issue tile updates or toast notifications through which the user can make the choice to activate the app for further action.

One additional aspect of background tasks is that Windows also places a limit on the total number of apps that can handle certain kinds of triggers: timers, receipt of push notifications, and receipt of network traffic on a *control channel* as used by real-time communications apps. The limit is imposed by the fact that such apps must be added to the *lock screen* for their tasks to run at all.

The lock screen, as you certainly know by now and as shown in Figure 13-9, is what’s displayed anytime the user must log into the device. A device will be locked directly by the user or after a period of inactivity. An exception is made when an app has disabled auto-locking through the [Windows.System.Display.DisplayRequest](#) API, as discussed in Chapter 10 in the “Disabling Screen Savers and the Lock Screen During Playback” section.



FIGURE 13-9 A typical lock screen. Up to seven apps can display badges along the bottom of the screen; one app can display text next to the clock.

Yet Windows doesn't want to force the user to log in just to see the most important information from their most important apps. Through PC Settings, as shown in Figure 13-10, the user can add up to seven apps to the lock screen (provided those apps have requested access, which is subject to user consent). These apps must be registered for lock screen–related background tasks during which they can issue badge updates to the lock screen—these are what you see above along the bottom of Figure 13-9, where each badge glyph (the numbers) is also accompanied by a monochrome graphic, referred to as the Badge Logo in the app manifest. This graphic is 24x24 at 100%, 33x33 at 140%, and 43x43 at 180%, and it must contain only white or transparent pixels.

In addition, the user can indicate a single app that can display a piece of text (but not an image) next to the clock. Note that toast notifications raised by these apps will surface on the lock screen; if tapped, the lock screen will bounce and the app will be activated once the user signs in.

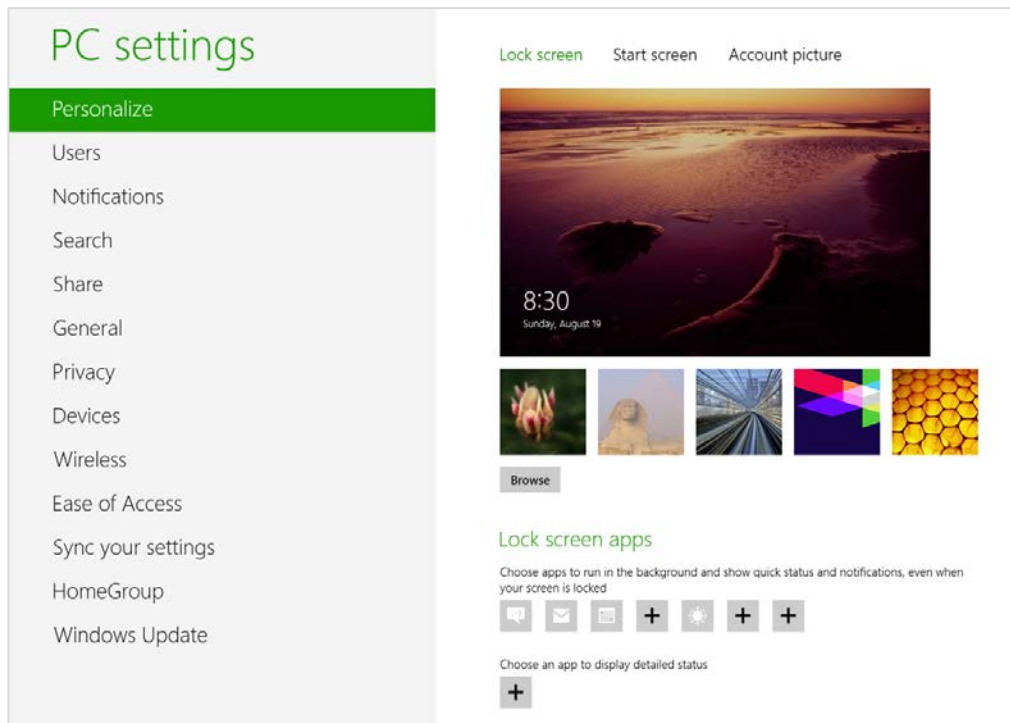


FIGURE 13-10 Configuring the lock screen and lock screen apps in PC Settings.

Thus we complete the story of how Windows 8 works with apps to be alive with activity—on the Start screen, on the lock screen, and while the user is engaged in other apps—while yet conserving battery power by intelligently managing how and when apps can issue their various updates. Let's now see exactly how that's accomplished, ideally without needing apps to run at all.

The Four Sources of Updates and Notifications

When an app is active in the foreground, it can clearly issue whatever notifications it wants: updates to any of its tiles, badge updates, and toast notifications. Together these are simply referred to as *local* updates because they originate from the running app and are applied immediately, as shown in Figure 13-11.⁶⁴ A running news app, as an example, might issue up to five updates to its tiles so that recent headlines continue to cycle when the user switches to another app. Such updates can also be set to expire at some date and time in the future so that they'll disappear automatically (perhaps fulfilling the adage, “No news is good news!”). With toasts, note that a foreground app should use inline messages, flyouts, and message dialogs for errors that pertain to the currently visible content; toasts are only appropriate for alerts about content in some other part of the app.

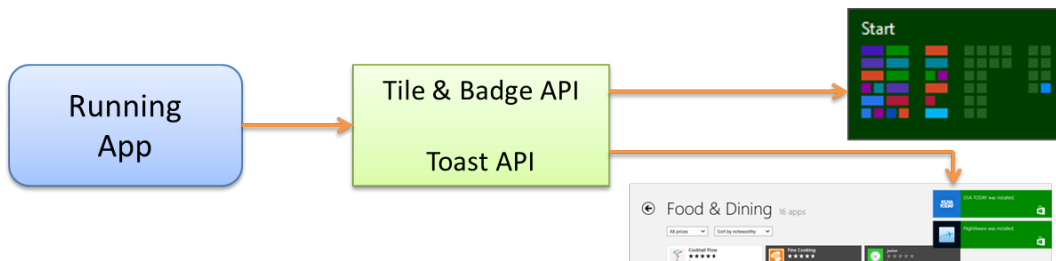


FIGURE 13-11 Local updates from a running app are applied immediately.

The second source of updates are *scheduled notifications* that apply to tile updates and toasts. A running app issues these to the system with the date and time when the update or notification should appear, regardless of whether the app will be running, suspended, or not running at that future time. This is illustrated in Figure 13-12. A calendar app, for example, would typically use scheduled notifications for appointment reminders.

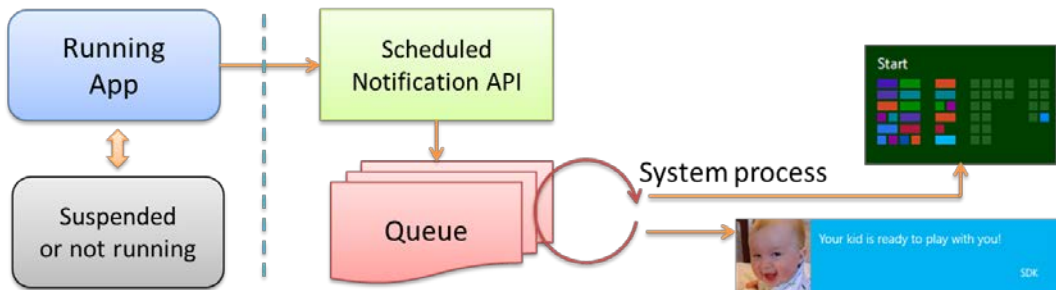


FIGURE 13-12 Scheduled notifications are managed by the system and will appear at the requested time irrespective of the state of the originating app.

⁶⁴ Background tasks, as discussed in this chapter, are not a concern for the foreground app as it can run whatever background processes it wants using web workers or WinRT components (see Chapter 16).

The third way an app can issue updates—in this case for tiles and badges only—is through a *periodic update*. As illustrated in Figure 13-13, a running app configures the system’s tile and badge updaters to request an update from a specific web service URI at certain low-frequency intervals (the minimum is 30 minutes) beginning at a specified time, if desired. The web service responds to this HTTP request with an XML payload that’s equivalent to what a running app would provide in a local update, and updates can be set with an expiration date/time so that they’re automatically removed from the update cycle when appropriate. With all these capabilities, periodic updates are wholly sufficient for many apps to create very dynamic live tiles with relatively little effort.

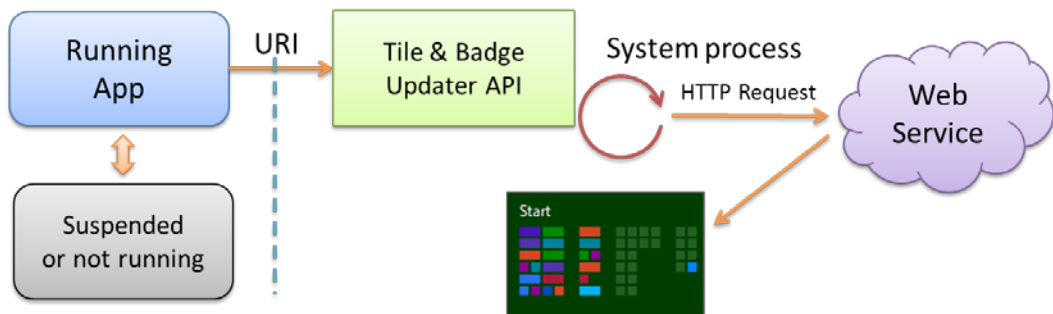


FIGURE 13-13 Periodic updates for tiles and badges are registered with the system’s tile updater, which will request an update from a web service at regular intervals.

Of course, a 30-minute minimum interval is simply not fast enough when an app wants to notify a user as soon as possible. Thus we have the fourth means for updates—*push notifications*—and this method applies across tiles and toasts, as well as non-UI (raw) notifications.

Push notifications are, as the name implies, sent directly to a device not at the request of an app but at the request of some associated web service that is typically monitoring information or other conditions around the clock. As illustrated in Figure 13-14, that web service employs the free Windows Push Notification Service (WNS for short) to send notifications to those apps that have created a channel for this purpose. Each channel is specific to a user and the device. As with other updates, this requires the app to be run at least once, because it’s during that first launch that the app establishes a WNS channel for the given device.

A push notification can contain an XML payload as with other tile updates and toast notifications, but it can also be used to send a non-UI *raw notification* that contains arbitrary data. A raw notification must be received by a running app or a by lock screen app with a background task for with the push notification trigger—otherwise the system clearly won’t know what to do with it!

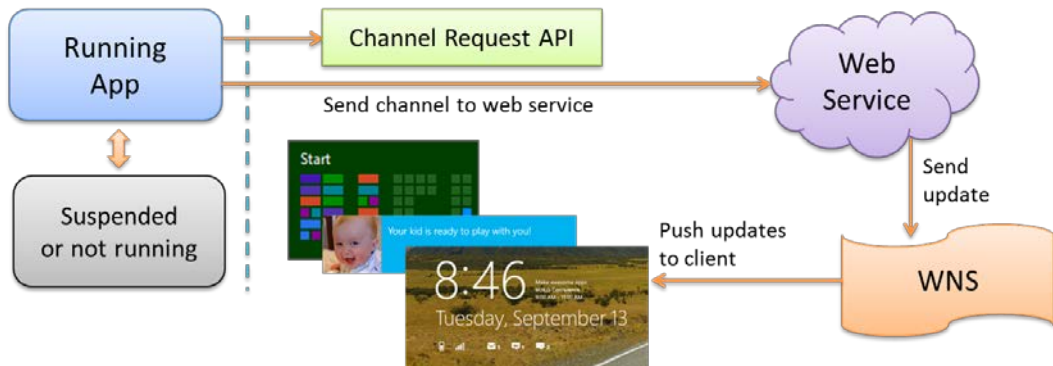


FIGURE 13-14 Push notifications originate with an always-running web service and are then sent to the Windows Push Notification Service for delivery to specific clients (a specific app on a specific user device) through their registered WNS channels.

A helpful summary of these different update mechanisms can be found on [Choosing a notification delivery method](#) in the documentation, a topic that includes various examples of when you might use each method. Whatever the case, we’re now ready to see the details of how we employ all of them in an app to help keep a system alive with activity.

Tiles, Secondary Tiles, and Badges

The very first thing you should know about your app tile is that if you want to enable live *wide* tiles (including secondary ones), you must include a wide logo image in the Application UI section of your manifest as shown below. Without it, you can still have live square tiles, but wide tile updates will be ignored.

Tile:	
Logo:	<input type="text" value="images\squareTile.png"/> ✕ ...
Required size: 150 x 150 pixels	
Wide logo:	<input type="text" value="images\wideTile.png"/> ✕ ...
Required size: 310 x 150 pixels	

At this point I encourage you to go back to Chapter 3, “App Anatomy and Page Navigation,” and review the “Branding Your App 101” section where we discuss how different bits in the manifest affect your tiles, such as the Short Name and Show Name settings. As also covered in that section, remember to provide different scaled versions of your logo and wide logo images. Even though you might issue tile updates as soon as your app is run, your static tiles will be essential to the user’s first impression of your app after it’s acquired from the Windows Store. The static tiles are also what the user will see if he or she turns your live tiles off or if all your updates expire. So, even if you plan for live tiles, be sure to still invest in great static tile designs as well.

Providing both square and wide static tiles enables you to issue live tile updates to both, including square and wide secondary tiles. In both cases, try to think through what the user would most want to see. When users select a wide tile, which is to say they're electing to have your tile occupy more prime real estate on the Start screen, it's likely that they're looking for details that add value to the Start screen. If users choose a square tile, on the other hand, they're probably more interested in only the most essential information: the number of new email messages (as expressed through a badge), for example, rather than the first line of those messages, or the current temperature in a location rather than a more extended forecast.

The [Guidelines and checklist for tiles and badges](#) provides rather extensive guidance on this particular topic along with appropriate use of logos, names, badges, and updates. There is also a helpful post on the Windows 8 Developer Blog called [Creating a great tile experience](#). Here we'll concern ourselves with how such updates and badges are sent to a tile, a process that involves what are called *tile XML templates*, predefined XML configurations that you populate with text, images, and other properties. These templates apply to all forms of tiles and update methods, which we'll examine in a moment. First, however, let's see how secondary tiles are managed because everything we talk about thereafter applies equally to all tiles for the app.

Note The tile and notifications API is generally found within `Windows.UI.Notifications`, except those for creating secondary tiles that come from `Windows.UI.StartScreen`. Unless otherwise noted, assume that the APIs we're talking about come from that namespace. That way we don't have to spell it out every time!

Secondary Tiles

A secondary tile is a kind of bookmark into an app, to achieve what's also called *deep linking*: a way to launch an app into a particular state or to a particular page. Secondary tiles allow the user to personalize the Start screen with more specific views of an app. As suggested on [Guidelines and checklist for secondary tiles](#) (a topic I highly recommend you read), offering the ability to create a secondary tile is a good idea whenever you have app state that could be a useful target or destination unto itself. Don't create secondary tiles, however, for static content or use them as virtual command buttons—that would only educate your customers that they shouldn't bother to pin tiles from your app!

An app creates a secondary tile in response to a Pin command that it typically includes on its app bar (using the `WinJS.UI.AppBarIcon.pin` icon). Offer this command when the app is displaying pinnable content or the user has made an appropriately pinnable selection; hide or disable the command if the content or selection is not pinnable. In addition, change it to an Unpin command if the content is already pinned. For details on managing commands in the app bar, refer to Chapter 7, "Commanding UI."

When the Pin command is invoked, the app makes the request to create the tile. Windows then prompts the user for their consent, as shown earlier in Figure 13-5.

Once created, a secondary tile has all the same capabilities as your app tile, including the ability to receive updates from any source. The key difference between the app tile and secondary tiles is that the former launches the app into its default (or current) state, whereas the latter launches the app with specific arguments that your activation handler uses to launch (or activate) the app into a specific state. Let's see how it all works.

Creating Secondary Tiles

The process for creating a secondary tile in response to a pin command is quite simple: first create an instance of [Windows.UI.StartScreen.SecondaryTile](#) with the desired options, and then call either its [requestCreateAsync](#) or [requestCreateForSelectionAsync](#) method. If the user confirms the creation of the tile, it will be added to the Start screen and your completed handler will receive a result argument of [true](#). If the user dismisses the flyout (by tapping outside it), the completed handler will be called with a result argument of [false](#). The error handler for these methods will be called if there is an exception, as if you fail to provide required properties in the [SecondaryTile](#).

When creating a [SecondaryTile](#) object, you can use four different constructors:

- [SecondaryTile\(\)](#) Creates a [SecondaryTile](#) with default properties.
- [SecondaryTile\(tileId\)](#) Initializes the [SecondaryTile](#) with a specific ID, typically used when creating an object before an update or when unpinning the file.
- [SecondaryTile\(tileId, shortName, displayName, arguments, tileOptions, logo\)](#) Creates a [SecondaryTile](#) with all the required properties for a square tile.
- [SecondaryTile\(tileId, shortName, displayName, arguments, tileOptions, logo, wideLogo\)](#) Creates a [SecondaryTile](#) with all the required properties for a wide tile.

These options clearly correspond to the following [SecondaryTile](#) properties, all of which are required when you call a [requestCreate*](#) method (except [wideLogo](#) that is only required for a wide tile):

- [tileId](#) A unique string (a maximum of 64 alphanumeric characters including . and _) that identifies the tile within the package. You need this when you want to update or delete a tile, and it should always be set. This value is typically derived from the content related to the file. If you create secondary tiles with a [tileId](#) that already exists, the new one will take its place.
- [shortName](#) The text string (40 characters max) that initializes the contents of the tile name control, as shown earlier in Figure 13-5. This is displayed directly on the tile but can be modified by the user before the tile is actually created. Once the tile is created, this value will contain the string as it appears on the tile.
- [displayName](#) The tile's display name that will be shown in the tile's tooltip, next to the app in the Start screen's All Tiles list, and a few other areas within Windows. This can be whatever length you want and can contain any characters.

- `arguments` A string that's passed to the app's activation handler when the secondary tile is invoked.
- `tileOptions` One or more values from the `TileOptions` enumeration, combined with the `|` (bitwise OR) operator. Options include `none` (the default), `showNameOnLogo` (displays `shortName` on the square tile), `showNameOnWideLogo` (displays `shortName` on the wide tile), and `copyOnDeployment` (indicates that the secondary tile is roamed to the cloud and replicated on other devices where the same user installs the same app).
- `logo` A URI for the square tile image. This can use either the `ms-appx:///` or `ms-appdata:///local` schema. Be sure to avoid storing a dynamically generated image in temporary storage, and avoid deleting it unless all secondary tiles that reference it are deleted.
- `wideLogo` A URI for the wide tile image, again with either the `ms-appx:///` or `ms-appdata:///local` schema.

You can, of course, modify any of these properties after creating the `SecondaryTile` object along with the remaining properties that let you override the defaults defined in the app manifest: `backgroundColor` (a `Windows.UI.Color` value), `foregroundText` (a `ForegroundText` value, either `dark` or `light`), and `smallLogo` (a URI again with `ms-appx:///` or `ms-appdata:///local`). Two other properties, `lockScreenBadgeLogo` and `lockScreenDisplayBadgeAndTileText`, relate to secondary tiles on the lock screen. We'll come back to these later in "Background Tasks and Lock Screen Apps," specifically the subsection "Lock Screen Dependent Tasks and Triggers."

At runtime, you can also retrieve any of these properties to check the state of the secondary tile if needed. If you modify any properties for a `SecondaryTile` that has already been pinned, be sure to call its `updateAsync` method to propagate those changes.

The `requestCreate*` methods also have a couple of variations that allow you to control the placement of the user consent flyout (again see Figure 13-5). Calling `requestCreateAsync` by itself results in a default placement in a lower corner of the display. It's usually better, however, for that flyout to appear close to the command that invoked it. For this purpose `requestCreateAsync` accepts an optional `Windows.Foundation.Point`, specifying where to place the lower right corner of the flyout.

With `requestCreateForSelectionAsync` there are also two variations. The first takes a `Windows.Foundation.Rect` describing the selection. The flyout will appear above that rectangle if possible. If you expect that this default placement will obscure the secondary tile's content, you can also pass an optional value from `Windows.Popup.Placement` to indicate where the flyout should appear relative to that rectangle: `above`, `below`, `left`, and `right`.

You can play around with all of these options in the [Secondary tiles sample](#). Scenarios 1 and 2 pin and unpin a secondary tile using on-canvas buttons, respectively, with Scenario 7 doing the same through the app bar. We'll see some of the other scenarios in the sections that follow. For the moment, the pinning function in Scenario 1 (`js/pintile.js`) shows the creation process using `requestCreateForSelectionAsync`:

```

function pinSecondaryTile() {
    var Scenario1TileId = "SecondaryTile.Logo";
    var uriLogo = new Windows.Foundation.Uri(
        "ms-appx:///images/SecondaryTileDefault-sdk.png");
    var uriSmallLogo = new Windows.Foundation.Uri(
        "ms-appx:///images/smallLogoSecondaryTile-sdk.png");

    // Create activation arguments...
    var currentTime = new Date();
    var newTileActivationArguments = Scenario1TileId + " WasPinnedAt=" + currentTime;

    var tile = new Windows.UI.StartScreen.SecondaryTile(Scenario1TileId,
        "Title text shown on the tile",
        "Name of the tile the user sees when searching for the tile",
        newTileActivationArguments,
        Windows.UI.StartScreen.TileOptions.showNameOnLogo, uriLogo);

    // Setting other options
    tile.foregroundText = Windows.UI.StartScreen.ForegroundText.dark;
    tile.smallLogo = uriSmallLogo;

    var selectionRect = document.getElementById("pinButton").getBoundingClientRect();

    tile.requestCreateForSelectionAsync(
        { x: selectionRect.left, y: selectionRect.top, width: selectionRect.width,
          height: selectionRect.height },
        Windows.UI.Popups.Placement.below)
        .done(function (isCreated) {
            if (isCreated) {
                // The tile was successfully created
            } else {
                // The tile was not created
            }
        });
}

```

Note As mentioned in Chapter 7, the system flyout displayed when creating a secondary tile (and when removing it, see “Managing Secondary Tiles” below), will cause the app to lose focus and will dismiss a nonsticky app bar as a result. For this reason, Scenario 7 of the Secondary tiles sample keeps the app bar visible by setting its *sticky* property to *true* before calling the secondary tile API.

App Activation From a Secondary Tile

Secondary tiles provide a way to activate an app to something other than its default state, similar to how command-line arguments work with desktop or console apps. This process depends entirely on the contents of the secondary tile’s *arguments* property. When a secondary tile is tapped or clicked, the app’s *activated* event is fired with an activation kind of launch and the tile’s *arguments* value in *eventArgs.detail.arguments*. The app then takes whatever action is appropriate for that data, such as navigating to a particular page of content, retrieving a piece of content from an online source, and so on. In the Secondary tiles sample, the activation code in *js/default.js* navigates to its Scenario 5 page, where we pass *arguments* as the options parameter of *WinJS.Navigation.navigate*:

```

function activated(eventObject) {
    if (eventObject.detail.kind ===
        Windows.ApplicationModel.Activation.ActivationKind.launch) {
        if (eventObject.detail.arguments !== "") {
            // Activation arguments are present (declared when the
            // secondary tile was pinned)
            eventObject.setPromise(WinJS.UI.processAll().done(function () {
                // Navigate to Scenario 5, where the user will be shown
                // the activation arguments
                return WinJS.Navigation.navigate(scenarios[4].url,
                    eventObject.detail.arguments);
            }));
        } else {
            // Activate in default state
        }
    }
}

```

The page control (`js/LaunchedFromSecondaryTile.js`) receives the arguments string in the `options` parameter of both the `processed` and `ready` methods. In the case of the sample it just copies that string to the display:

```

var page = WinJS.UI.Pages.define("/html/LaunchedFromSecondaryTile.html", {
    processed: function (element, options) {
        if (options) {
            document.getElementById("launchedFromSecondaryTileOutput").innerHTML += "<p>" +
                "App was activated from a secondary tile with the following activation" +
                "arguments : " + options + "</p>";
        }
    },

    ready: function (element, options) {
    }
});

```

Your own app, of course, will do something much more interesting with `arguments`!

Managing Secondary Tiles

In addition to the methods and properties to create secondary tiles, the `SecondaryTile` class has two static methods to generally manage your app's secondary tiles:

- `exists` Returns a `Boolean` indicating whether a secondary tile, identified with its `tileId`, is present on the Start screen. This tells you whether calling a `requestCreate*` method for a tile with that same `tileId` will replace an existing one. This is demonstrated in Scenario 4 of the Secondary tiles sample.
- `findAllAsync` Retrieves a vector of `SecondaryTile` objects that have been created by the app. This will include any tiles roamed from another device (those created with the `copyOnDeployment`

option).⁶⁵ This is demonstrated in Scenario 3 of the sample.

In addition, there are a few other methods to work with a specific `SecondaryTile` instance:

- `requestDeleteAsync` and `requestDeleteForSelectionAsync` Direct analogs, with the same placement variations, to the `requestCreate*` methods, as deletion of a secondary tile (unpinning) is also subject to user consent. This is demonstrated again in Scenario 2 and 7 of the sample.
- `updateAsync` Propagates any changes made to the `SecondaryTile` properties since it was added to the Start screen. This is demonstrated in Scenario 8 of the sample.

If you've been keep score throughout this section, you might have noticed that I've yet to mention Scenario 6 of the sample. That's because it shows how to make a secondary tile into a live tile with updates. To understand that, we need to look at updates more generally because the mechanisms involved apply to all tiles alike. This just so happens to be the next topic in this chapter—yes, I planned it that way!

Basic Tile Updates

A local update for a tile, as described earlier in this chapter, is one that an app issues while it's running. Clearly, this is one of the best times to issue updates because it's highly likely that the app already has the information it needs for those updates to any of its tiles. In a number of cases—especially when an app is *not* related to a web service—the information needed for the app's live tiles is available only while it's running. A game, for example, can send updates showing best scores, new challenges, progress toward achievements, and other kinds of compelling invitations to re-engage with the app. (I must personally admit that this works quite well with the Fruit Ninja game.)

The process of sending a local tile update is very straightforward using the APIs in the `Windows.UI.Notifications` namespace:

- Create the XML *payload*, as it's called, that describes the update within an `XmlDocument` object. The XML must always match one of the predefined tile templates. You can start with a system-provided `XmlDocument`, create it from scratch, or use the Notifications Extensions Library that provides an object model and IntelliSense for this.
- Create a `TileNotification` object with that XML. The XML becomes the `TileNotification` object's `content` property and can be set separately.
- Optionally set the `expirationTime` and `tag` properties of the `TileNotification`. By default, a locally issued update does not expire and is removed only if it's evicted by a newer update or explicitly cleared. Setting `expirationDate` will automatically remove it at that particular time. (Cloud-issued notifications automatically expire after three days.) The `tag` property is a string of

⁶⁵ The `SecondaryTile` class also has a variant of `findAllAsync` that takes a different app name along with `findAllForPackageAsync` that's described as enumerating secondary tiles for all apps in the same package. These were meant for packages that contain multiple apps, a feature that is not currently supported through the Windows Store.

16 or fewer characters that is used to manage the stack of updates that are cycled on the tile. More on this a little later.

- Call [TileUpdateManager.createTileUpdaterForApplication](#) to obtain a [TileUpdater](#) object that's linked to your app tile; call [TileUpdateManager.createTileUpdater-ForSecondaryTile](#) (chew on that name!) to obtain a [TileUpdater](#) object for a secondary tile with a given [tileId](#).
- Call [TileUpdater.update](#) with your [TileNotification](#) object. (The animation used to bring the update into view is similar to [WinJS.UI.Animation.createPeekAnimation](#), as described in Chapter 11.)

Tip If you issue tile updates or other notifications when your app is running, think about whether it's also appropriate to issue updates within a [resuming](#) event handler if you aren't going to use other means like periodic updates or push notifications to refresh the tile. It may have been a while since you were suspended, so being resumed is a good opportunity to send updates.

Let's turn now to the [App tiles and badges sample](#) for how updates appear in code. Because the Visual Studio simulator doesn't enable live tiles and toast notifications, remember to run the sample with the Local Machine or Remote Machine options.

Assuming that we have our update [XMLDocument](#) in a variable named [tileXml](#), sending the update just takes two lines of code (see [js/sendTextTile.js](#)):

```
var tileNotification = new Windows.UI.Notifications.TileNotification(tileXml);
Windows.UI.Notifications.TileUpdateManager.createTileUpdaterForApplication()
    .update(tileNotification);
```

and similarly for secondary tiles in Scenario 6 of the Secondary tiles sample ([js/SecondaryTileNotification.js](#)):

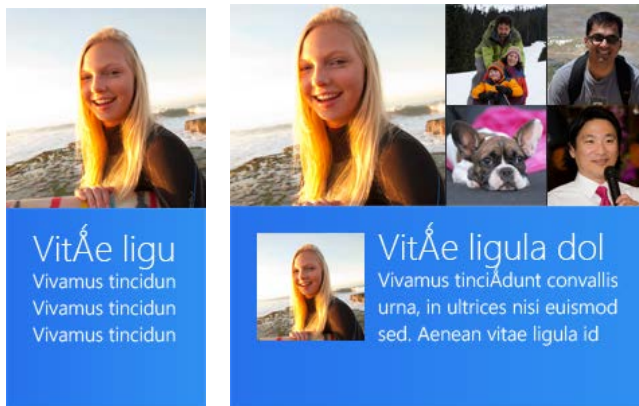
```
var tileNotification = new Windows.UI.Notifications.TileNotification(tileXml);
Windows.UI.Notifications.TileUpdateManager.createTileUpdaterForSecondaryTile(
    "SecondaryTile.LiveTile").update(tileNotification);
```

The more interesting question is how we create that [tileXml](#) payload in the first place. This involves choosing one of the predefined visual tile templates and then choosing a method to create the [XMLDocument](#). Then we'll see how to use images with the updates along with considerations for branding. Localization and accessibility are additional concerns for tile updates, but we'll return to that subject later in Chapter 17, "Apps for Everyone."

Choosing a Tile Template

The first step in creating a tile update is to select an appropriate template from the [Tile template catalog](#). Here you will find descriptions, images, and the exact XML for the 10 available square templates and the 36 available wide templates—yes, 46 different templates in all (so I hope you understand why I'm not showing them all here!). Some are text only, some are image only, some are text and image (wide tiles only), and then there are a number referred to as *peek* templates. These, if you look at them

in the topic linked to above,⁶⁶ are really composed of two sections that are each the size of the whole tile, as shown below for square tiles (left) and wide tiles (right):



With peek templates you effectively get to show twice the content as the other templates. When a peek update is shown on a live tile, the upper portion will appear first and then the tile will flip or give you a “peek” at the lower portion, and then it will switch back to the upper portion, after which the live tile will switch to the next update in the cycle, if one exists. (The Travel app uses peek templates if you want an example; and the animation that’s employed here is again similar to [WinJS.UI.Animation.createPeekAnimation](#).) Of course, both sections should contain related content because they are part of the same singular update.

There are several important notes with the template layouts. First, in many of the templates at present, the last line of text will not display if you’re also showing a logo or a short name on the tile (to avoid overlaps). This will likely be changed in the future, but it’s the reality for Windows 8.

Second, images are limited to 1024x1024 and 200KB maximum; if any image exceeds these limits, the entire update will not appear at all. Clearly, it’s better to avoid large images if you can help it because such images just increase memory consumption and possibly network usage (if the image is being downloaded). It’s also good to take the 80%, 100%, 140%, and 180% scale factors into account for tile images. However, if you don’t want to deal with individual scaling factors, size your tile images for 180% and let the system scale them down (which uses a high-quality algorithm so that images will look as good as if you scaled them down with photo-editing software). Also, for photograph, consider using JPEG instead of PNG as the former has better compression for such images.

⁶⁶ A more succinct list of templates is also found on the reference page for [TileTemplateType](#). This includes the name of the template and a representative image, but doesn’t include the XML.

Third, if you supply an image that doesn't match the final aspect ratio, the image will be scaled for width and cropped on the top and bottom. Note too that a wide tile is not exactly a 2:1 aspect ratio; at 100% the wide tile is 310x150 pixels, meaning that an image occupying half of it will be 155x150 pixels (not quite square) and those in a collection view (the upper right portion of the rightmost image above) will be 77.5x75.

Fourth, if you want a tile with images and text that doesn't fit any of the templates, you can always use an image-only template (TileSquareImage and TileWideImage) with a graphic you generate at run time. However, don't make the tile appear to have separate buttons or other clickable areas: the whole tile always acts as a single unit to invoke the app, so such a design would be misleading.

Hint If you see any apps using tile updates that don't seem to match any of the templates, they are likely just using the TileWideImage template and drawing all the text and graphics directly.

Scenario 5 of the App tiles and badges sample also provides a very helpful design and experimentation tool for tiles, as shown in Figure 13-15. This part of the sample is intended as a tool rather than being code you duplicate in an app. It's meant to let you easily play around with all the templates and their contents, including images referenced from local and remote sources, without having to write specific code every time. It also lets you exercise the various options for branding the app and sending the result as an update to the sample's tile on the Start screen.

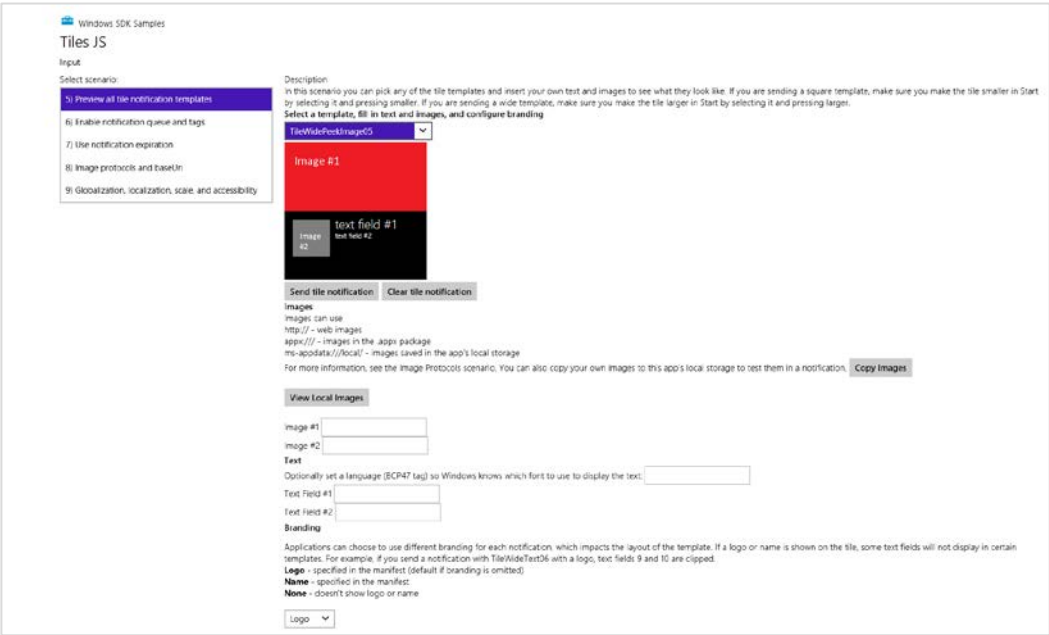


FIGURE 13-15 Scenario 5 of the App tiles and badges sample is a tool for testing out all the different tile templates.

Creating the Payload, Method 1: Populating Template Content

The first way to create the XML payload for a given template is to use the [TileUpdateManager.-getTemplateContent](#) method, to which you pass the name of a template (a value from [TileTemplateType](#)). This is shown in Scenario 1 of the sample (js/sendTextTile.js):

```
function sendTileTextNotificationWithXmlManipulation() {  
    var tileXml = Windows.UI.Notifications.TileUpdateManager.getTemplateContent(  
        Windows.UI.Notifications.TileTemplateType.tileWideText03);
```

This method returns an [XmlDocument](#) object that contains the structure of the XML for the template but not any specific content. If you run the sample and examine `tileXml` just after the call above, it will contain only the following—elements but no real data values:

```
<tile>  
  <visual>  
    <binding template="TileWideText03">  
      <text id="1"></text>  
    </binding>  
  </visual>  
</tile>
```

The next step, then, is to fill in the blanks (primarily attributes) by using the [XmlDocument](#) methods you probably already know (and may or may not love):

```
var tileAttributes = tileXml.getElementsByTagName("text");  
tileAttributes[0].appendChild(tileXml.createTextNode(  
    "Hello World! My very own tile notification"));
```

In general, if your tile supports a wide format, include XML for both square and wide formats in the payload, because the user can change the size of the tile at any time. The sample does it this way:

```
var squareTileXml = Windows.UI.Notifications.TileUpdateManager.getTemplateContent(  
    Windows.UI.Notifications.TileTemplateType.tileSquareText04);  
var squareTileTextAttributes = squareTileXml.getElementsByTagName("text");  
squareTileTextAttributes[0].appendChild(squareTileXml.createTextNode(  
    "Hello World! My very own tile notification"));  
  
var node = tileXml.importNode(squareTileXml.getElementsByTagName("binding").item(0), true);  
tileXml.getElementsByTagName("visual").item(0).appendChild(node);
```

We're then ready to send the update to the tile:

```
// send the notification to the app's application tile  
Windows.UI.Notifications.TileUpdateManager.createTileUpdaterForApplication()  
    .update(tileNotification);  
}
```

Note that the `visual` element in the XML supports a `version` attribute whose default value is 1. This will help accommodate future changes where elements added in newer versions of the XML that

arrive on a Windows 8 machine will simply be ignored. The exact tile schema, should be you interested, can be found on the [Tile schema](#) reference page.

Creating the Payload, Method 2: XML Strings

Instead of calling `TileUpdateManager.getTemplateContent` to obtain an `XmlDocument` with the tile template contents, you can just create that `XmlDocument` directly from a string. This is just like creating elements in the DOM by using `innerHTML` instead of the DOM API—it takes fewer overall function calls to create the payload you need and lends itself well to predefining a bunch of mostly populated tile updates ahead of time.

This method is simple: define an XML string with the update contents, create a new `XmlDocument`, and use its `loadXml` method to turn the string into the payload. In Scenario 1 we see how this is done to create the exact same payload as in the previous section:

```
function sendTileTextNotificationWithStringManipulation() {
    // create a string with the tile template xml
    var tileXmlString = "<tile>"
        + "<visual>"
        + "<binding template='TileWideText03'>"
        + "<text id='1'>Hello World! My very own tile notification</text>"
        + "</binding>"
        + "<binding template='TileSquareText04'>"
        + "<text id='1'>Hello World! My very own tile notification</text>"
        + "</binding>"
        + "</visual>"
        + "</tile>";

    var tileDOM = new Windows.Data.Xml.Dom.XmlDocument();
    tileDOM.loadXml(tileXmlString); // Good idea to put this in a try/catch block

    var tile = new Windows.UI.Notifications.TileNotification(tileDOM);
    Windows.UI.Notifications.TileUpdateManager.createTileUpdaterForApplication()
        .update(tile);
}
```

Clearly, this method is very simple but has the drawback of requiring you to do manual escaping. It is also more difficult to debug. (Looking for tiny errors in strings is not my favorite pastime!) Fortunately, there is a third available method: the Notifications Extensions Library, which offers the simplicity of using strings with a high degree of reliability.

Creating the Payload, Method 3: The Notifications Extensions Library

The third means of creating the necessary `XmlDocument` for a tile update is to use what's called the Notifications Extensions Library. (Yes, it's a double plural.) This is a WinRT component written in C# that's included with a number of the SDK samples, including the App tiles and badges sample we're looking at here. (Notice that it's included in the project's References.) We'll be looking at the structure of such components in Chapter 16, "WinRT Components." It's likely that this library will become part of the Windows API in the future, so we do encourage developers to leverage it.

The library makes it easier to populate a template through object properties rather than `XmlDocument` methods, and because it's been very well-tested within Microsoft it's a more robust approach than creating an `XmlDocument` directly from strings. Here's how it's used in Scenario 1 to create, once again, the same payload we've already seen:

```
function sendTileTextNotification() {
    var tileContent =
        NotificationsExtensions.TileContent.TileContentFactory.createTileWideText03();
    tileContent.textHeadingWrap.text = "Hello World! My very own tile notification";

    var squareTileContent = NotificationsExtensions.TileContent.TileContentFactory
        .createTileSquareText04();
    squareTileContent.textBodyWrap.text = "Hello World! My very own tile notification";
    tileContent.squareContent = squareTileContent;

    Windows.UI.Notifications.TileUpdateManager.createTileUpdaterForApplication()
        .update(tileContent.createNotification());
}
```

Simply said, the library's `TileContentFactory` object provides methods to create objects equivalent to the XML documents provided by `TileUpdateManager.getTemplateContent`. As shown in the code above, those objects have properties equivalent to each field in the template, and when you're ready to pass it to `TileUpdater.update`, you just call its `createNotification` method.

The other reason this library exists is to simplify the process of creating an ASP.NET web service for periodic updates and push notifications (where the latter can send tile updates, badge updates, and toast notifications). Instead of creating the XML payloads manually—a fragile and highly error-prone practice at best—the service can use the Notifications Extensions Library to easily and consistently create the XML for all these notifications.

Because the object model in the library clearly describes the XML, it's fairly easy to use. There is also a topic in the documentation for it called [Quickstart: Using the NotificationsExtensions library in your code](#). The samples we look at in this chapter also show most use cases.

Using Local and Web Images

Scenario 1 of the sample, as we've seen, shows tile updates using text, but the more interesting ones include graphics as well. These can come either from the app package, local app data, or the web, using `ms-appx:///`, `ms-appdata:///local`, and `http://` URIs, respectively. These URIs are simply assigned to the `src` attributes of `image` elements within the tile templates. (These are `image`, not `img` as in HTML.) Note again that the first two URIs typically have three slashes at the beginning to denote "the current app"; `http://` URIs also require that the *Internet (Client)* capability be declared in the app's manifest.

Scenario 2 of the sample (`js/sendLocalImage.js`) shows the use of `ms-appx:///` for images within the app package, with variants for all three methods we've just seen to create the payload. When using `XmlDocument` methods, setting an image source looks like this:

```
var tileImageAttributes = tileXml.getElementsByTagName("image");
tileImageAttributes[0].setAttribute("src", "ms-appx:///images/redWide.png");
```

The Notifications Extensions Library gives us properties to which we can assign a URI:

```
var tileContent = NotificationsExtensions.TileContent.TileContentFactory
    .createTileWideImageAndText01();
tileContent.textCaptionWrap.text = "This tile notification uses ms-appx images";
tileContent.image.src = "ms-appx:///images/redwide.png";
```

And when using XML strings, you can just include the URI directly in the `image` element.

Scenario 3 (`js/sendWebImage.js`) shows the same things except you can enter an <http://> URI of your choice. This is a good way to see the effects of pointing to images that have varying aspect ratios as well as those that exceed the allowable 1024px dimensions and 200KB file size. As you'll see, the updates simply aren't shown in those cases.

As for `ms-appdata:///local` URIs (roaming and temp are not allowed), their use is demonstrated in Scenario 8 where you choose an image with the file picker and the sample copies it to the local app data folder. It then references that file with an `ms-appdata:///local` URI in the update payload (`js/imageprotocols.js`):

```
tileContent = NotificationsExtensions.TileContent.TileContentFactory.createTileWideImage();
tileContent.image.src = "ms-appdata:///local/" + imageRelativePath;
```

The same scenario lets you play with in-package and remote URIs as well, as does the tile update designer in Scenario 5. I also updated the Here My Am! app for this chapter (in the companion content) with a peek tile update containing the most recent image and the location; see "Sidebar: PNG vs. JPEG Image Sizes" below.

Speaking of tools and images, also check out Scenario 10 in the SDK sample. This gives you another helpful tool for tile updates where you can crop and adjust images according to varying pixel densities so that they'll work well with a selected tile template. You can save the images you adjust for inclusion in your app package, and the code for the scenario can also be used to adjust images at run time. Like I said, very helpful!

A final capability with images is an option to have Windows automatically append a query string to <http://> URIs. This query string will describe the current scaling factor, contrast setting (for accessibility), and language. This enables web services to adjust images accordingly, avoiding the need to handle such concerns in the app itself. As described in the [Tile schema](#) reference, specifically for the `image` element, you indicate this option by setting the `addImageQuery` attribute of `image` to `true` (also supported on the `visual` and `binding` elements):

```
// XmlDocument form
var tileImageAttributes = tileXml.getElementsByTagName("image");
tileImageAttributes[0].setAttribute("addImageQuery", "true");

// XML string form (other lines omitted)
var tileXmlString = /* ... */ "<image id='1' addImageQuery='true'
    src='ms-appx:///images/redwide.png'/>" /* ... */

// If using Notifications Extensions Library (see Scenario 9 in the sample)
```

```
var tileContent = NotificationsExtensions.TileContent.TileContentFactory
    .createTileWideImageAndText01();
tileContent.image.src = "ms-appx:///images/redWide.png";
tileContent.image.addImageQuery = true;
```

In all of these cases, the appended string will be of the form

```
?ms-scale=<scale>&ms-contrast=<contrast>&ms-lang=<language>
```

where `<scale>` is 80, 100, 140, or 180, `<contrast>` is `standard`, `black`, or `white`, and `<language>` is a BCP-47 language tag such as `en-US`, `jp-JP`, `de-DE`, and so forth. All of these are described on the [Globalization and accessibility for tile and toast notifications](#) in the documentation, including how to localize update text.

Sidebar: PNG vs. JPEG Image Sizes

When considering tile images for the larger 140% and 180% scales, the encoding you use for your images can make a big difference and keep them below the 200K size limit. As we saw in “Branding Your App 101” in Chapter 3, a wide tile at 180% is 558x270 pixels and a square is 270x270 pixels. With the wide tile, a typical photographic PNG at this size will easily exceed 200K.

I encountered this when adding tile support to Here My Am! in this chapter, where it makes a smaller version of the current photo in the local appdata folder and uses `ms-appdata:///local` URIs in the tile XML payload. At first, I borrowed code from Scenario 10 of the App tiles and badges sample, as we’ve been working with here, to create a PNG from the `img` element using a temporary `canvas` and the blob APIs. This worked fine for a 270x270 tile image (a 180% scale that can be downsized), but for a 558x270 the file was too large. So I borrowed code from Scenario 3 of the [Simple Imaging sample](#) to directly transcode the `StorageFile` for the current image into a JPEG, where the compression is much better and we don’t need to use the canvas. This code is in the `transcodeImageFile` function in `pages/home/home.js`, a routine that we’ll also rewrite in Chapter 17 using C# in a WinRT component.

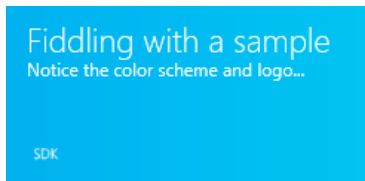
Such considerations are certainly important for services that handle the `addImageQuery` parameters for scale. For larger image sizes, it’s probably wise to stick with the JPEG format to avoid going over the 200K limit.

Branding

If you’re the kind of person who likes to read XML schema specs (like the [Tile schema](#) reference I pointed to a few moments ago), you might have noticed another attribute of the `visual` and `binding` elements called `branding`. This can be set to `none`, `logo` (the default), or `name` to indicate whether to include the app’s small logo or short name on the tile, both of which are provided in the app’s manifest. Scenario 5 of the App tiles and badges sample lets you play with these variations.

The other bits of the manifest that affect a tile update are the Foreground Text and Background Color settings in the Application UI section. These define how tile text appears for all tile updates (and toasts for that matter), and they cannot be altered in the tile payload. This keeps the branding of the app consistent between updates; users would certainly find it confusing if multiple tiles from the same app showed up in different colors.

As a quick example, the SDK sample we've been working with here uses Light foreground text and a background color of #00b2f0. If I go to Scenario 5, choose the TileWideText09 template, add some text, and select Logo for the branding (where the small logo contains a block with "SDK" in it), the result is as follows:



Cycling, Scheduled, and Expiring Updates

Although you might read the heading for this section and think it's just going to be a grab bag of randomness, all it really means is that we're looking at additional methods of the [TileUpdater](#) object and revisiting the two properties of the [TileNotification](#) object that we already mentioned. Simply said, now that we've seen how to do all the basic tile updates we're ready to start exploring the additional capabilities. Again, everything here applies to all tiles in the app.

First is the ability to programmatically clear all updates and reset the tile to its default state as defined in the manifest. This happens with a simple call to [TileUpdater.clear](#) (shown in Scenario 1):

```
Windows.UI.Notifications.TileUpdateManager.createTileUpdaterForApplication().clear();
```

The next capability, as already mentioned, is to set the [TileNotification.expirationTime](#) property before sending that notification to [TileUpdater.update](#). This ensures that a locally issued notification will be automatically removed, and it lets you override the default three-day expiration period for cloud-issued updates. The update will appear immediately (at the next tile refresh, that is) and will then be removed from the tile after it expires. This is demonstrated in Scenario 7 of the sample—sending an update with an expiration date will display an update as on the left below. When it expires, it's removed, which in this case causes the tile to revert to its default state, as shown on the right (and yeah, I'm working on this book on a Sunday night!):



To *delay* the appearance of an update until a specified time, with or without an expiration, you do something a little different at the beginning: instead of creating a [TileNotification](#) object to send to the updater, create a [ScheduledTileNotification](#) and send it to [TileUpdater.-addToSchedule](#).

A [ScheduledTileNotification](#) is exactly the same as a [TileNotification](#) (including the expiration time) except that it contains an extra property called `deliveryTime` that indicates when—in UTC time, not local time!—the tile should first appear. For an example of this we have to take a brief detour to Scenario 1 of the [Scheduled notifications sample](#). But all that's really different is that we take whatever [XmlDocument](#) we've created with the payload—through any of the methods covered earlier—and create the notification with the delivery time. Here's a condensation of the code in the sample's `js/scenario1.js` file:

```
// Namespace variable
var Notifications = Windows.UI.Notifications;

// The delay in delivery time from the sample's control
var dueTimeInSeconds = parseInt(document.getElementById("futureTimeBox").value);

// The actual delivery date and time
var currentTime = new Date();
var dueTime = new Date(currentTime.getTime() + dueTimeInSeconds * 1000);

// In here we create the XmlDocument in the variable tileDOM

// Now create the update with the delivery date
var futureTile = new Notifications.ScheduledTileNotification(tileDOM, dueTime);
Notifications.TileUpdateManager.createTileUpdaterForApplication().addToSchedule(futureTile);
```

Other than these small changes, everything else about the tile update is the same as before.

It's certainly possible, as you can guess, to queue up many scheduled updates. At any time you can call [TileUpdater.getScheduledTileNotifications](#) to obtain a vector of active [Scheduled-TileNotification](#) objects. You can also remove any of those updates with [TileUpdater.-removeFromSchedule](#).

What happens if you schedule a series of updates that will end up being active at the same time? In the Scheduled notifications sample, for instance, issue a series of tile updates for 10 seconds from now, and then quickly switch to the Start screen to see the results. Those updates will appear in sequence, and one of them might actually be dropped if another update is scheduled right on its heels. And once you reach the last update, it just stays there until it expires, the tile is cleared, or some new update comes along.

In such cases it would be better to have the tile cycle through a series of tiles, thereby keeping the tile active with the more than just the last update.

Live tiles support cycling through up to five updates, where the duration of each is controlled by the system so that the whole Start screen has a consistent look and feel. To enable this you must first call `TileUpdater.enableNotificationQueue(true)`, and you can call it with `false` to disable cycling. The queue itself is first in, first out (FIFO; you cannot control the order otherwise), so the oldest notification is removed if a new one is added when the queue is already at maximum. In other words, if you enable the queue and issue updates as we've been doing, the five most recent updates will cycle.

You might want to selectively replace existing updates already in the queue rather than rely on the FIFO behavior (or calling `TileUpdater.clear` and reissuing the update you want to retain). This is the purpose of the `tag` property in `TileNotification` and `ScheduledTileNotification`. The `tag` is again just a maximum 16-character string that simply identifies a particular update. If the queue is enabled, a new update with any given `tag` will replace any update already in the queue with the same `tag`. If that tag doesn't exist, the update will replace the oldest in the queue. So, for example, a news app might have five tags for different categories of headlines such as world, local, politics, business, and health; a stock app would obviously use tags for different ticker symbols. Similarly, a weather app might tag updates with a zip code or other location identifier.

You can play with all this in Scenario 6 of the App tile and badges sample. In the process you might note that when activating an app from a cycling live tile, it does not receive an indication as to which update is currently shown. For this reason, it's currently recommended that activating a cycling live tile opens the app on a hub page that displays relevant content for all the updates together.

Badge Updates

The last bit you can use to update a live tile is a badge. I've kept this topic separate because it works through a separate API and is not part of the tile update XML payloads we've been using.

To review, badges are simply small glyphs—a one- or two-digit number, or one of a small number of symbols—that appear on a tile regardless of any other update activity. They are meant to indicate the status of the app rather than a piece of content, so like a logo or name they are not animated with other updates. However, badge changes are separately animated using `WinJS.UI.Animations.updateBadge`, as briefly noted in Chapter 11.

How you send badges to your tile is structurally similar to a tile update. Start by creating an `XmlDocument` payload for the badge update by using a template from the [Badge image catalog](#) or using the Notifications Extensions Library, which contains full support for badges. In the case of badges, it's really overkill to speak of an XML "document" because it contains only one element, `badge`, with one attribute, `value`, for which you can indicate a number from 1–99 (anything over that will display 99+) or one of 11 specific glyphs, as shown below. Note that although the glyphs are shown here against a blue background, the actual color will be the Background Color in your manifest:

Value	Glyph	XML
(none)	n/a	<badge value="none"/>
(number 1-99)	1	<badge value="1"/>
(number over 99)	99+	<badge value="123456"/>
activity		<badge value="activity"/>
alert		<badge value="alert"/>
available		<badge value="available"/>
away		<badge value="away"/>
busy		<badge value="busy"/>
newMessage		<badge value="newMessage"/>
paused		<badge value="paused"/>
playing		<badge value="playing"/>
unavailable		<badge value="unavailable"/>
error		<badge value="error"/>
attention		<badge value="attention"/>

If you like, you can use the [BadgeUpdateManager.getTemplateContent](#) function to obtain an [XmlDocument](#) with such contents; there's a bit of sample code on this method's reference page that shows how. But because the XML is so simple, it's just as easy to create the object from a string by using [new Windows.Data.Xml.Dom.XmlDocument](#) followed by a call to its [loadXml](#) method, as we've seen with tiles. The Notifications Extensions Library also has methods for this and doing updates. Both of these approaches are demonstrated in Scenario 6 of the App tiles and badges sample.

However you create it, the next step is to instantiate a [BadgeNotification](#) with that [XmlDocument](#):

```
var badge = new Windows.UI.Notifications.BadgeNotification(badgeDOM);
```

This notification object also supports an [expirationTime](#) property just like tiles do. That aside, the last step is to call [BadgeUpdateManager.createBadgeUpdaterForApplication](#) to obtain a [BadgeUpdater](#) for your app tile or—you can predict this one—[BadgeUpdateManager.create- BadgeUpdater- ForSecondaryTile](#) to obtain a [BadgeUpdater](#) for a secondary tile with a given [tileId](#). After you obtain your [BadgeNotification](#) (and again, the Notifications Extensions Library can help here), you then call [BadgeUpdate.update](#):


```
Windows.UI.Notifications.BadgeUpdateManager.createBadgeUpdaterForApplication()
    .update(badge);
```

or:

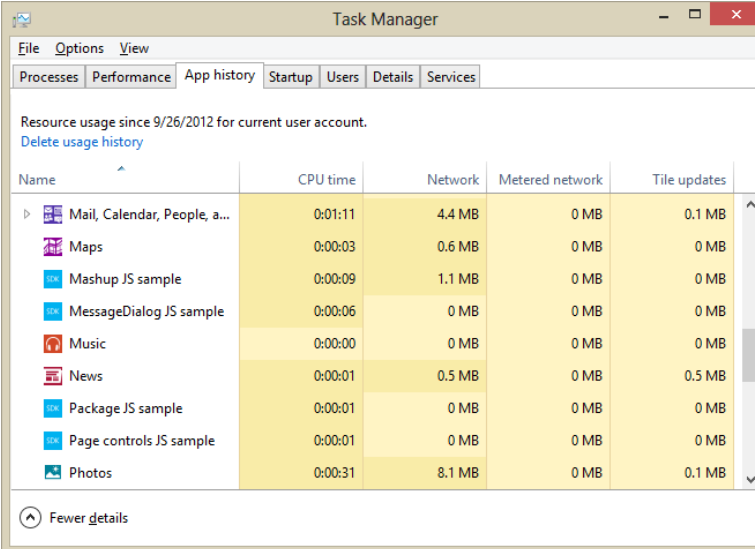
```
Windows.UI.Notifications.BadgeUpdateManager.createBadgeUpdaterForSecondaryTile(
    "SecondaryTile.LiveTile").update(badge);
```

And as with tiles, `BadgeUpdater.clear` removes the badge entirely, which is equivalent to sending an update with the value of none. That's it.

Beyond that, the `BadgeUpdater` class has just two additional methods, `startPeriodicUpdate` and `stopPeriodicUpdate`, which are also found on the `TileUpdater` class. And wouldn't you know it? Periodic updates just so happen to be our next topic—yes, I planned it this way too!

Sidebar: How Much Network Traffic for Tiles?

Included with the many improvements to Task Manager for Windows 8 is the ability to track your app's network traffic for tile updates. Run Task Manager, make sure you click More Details in the lower left, and then click the App History tab. Network traffic for tile updates is shown on the rightmost column. This number will typically be small, but it's a metric you can monitor to see whether updates become excessive.



The screenshot shows the Windows Task Manager window with the 'App history' tab selected. The window title is 'Task Manager'. Below the tabs, it says 'Resource usage since 9/26/2012 for current user account.' and has a link 'Delete usage history'. The table below lists several applications with columns for Name, CPU time, Network, Metered network, and Tile updates. The 'Tile updates' column shows values like 0.1 MB, 0 MB, and 0.5 MB.

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

At the bottom left of the table, there is a link 'Fewer details' with an upward arrow icon.

Periodic Updates

As described earlier in this chapter, periodic updates configure the system to automatically request updates from a web service on behalf of an app. Provided that the app has declared the *Internet (Client)* capability, this enables the app to continually keep its live tiles fresh without needing to run at all.

Periodic updates are great proof that Microsoft really does listen to developer feedback. When the first Developer Preview of Windows 8 was released in September 2011, many developers were very interested in implementing live tiles but it could only be done with push notifications and the Windows Push Notification Service, even if the tiles only needed low-frequency updates. In other words, push notifications were total overkill for apps that needed to get only a bit of data from a web service every once in a while to create their updates. So developers asked, “Is it possible to have my app just run in the background, periodically request data from my service, and then issue tile or badge updates?”

It was a completely legitimate request, but as described earlier, background tasks are very carefully controlled and allowed only for very specific scenarios. Hearing this feedback, the tiles and notifications team at Microsoft studied the problem and found that creating a new class of background task would also be overkill for low-frequency tile updates. Furthermore, they found that even if apps could use a background task for this purpose, they’d all pretty much do the same thing: poll data from a service and populate a tile or badge template. So instead of adding a background task, they added a new API for system-managed periodic updates.

This API consists of the following methods of the [TileUpdater](#) and [BadgeUpdater](#) classes:

- [TileUpdater.startPeriodicUpdate](#) and [BadgeUpdater.startPeriodicUpdate](#) Configure Windows to request an update from a given URI with a specified period (see below). These calls remove any previous URIs registered for the tile. An app can specify an optional date and time around which regular polling should begin, and in all cases the first request will happen immediately (very helpful for debugging!). A good time to call these is when the app is launched, when it’s resumed, and when a configuration changes that might alter the URI.
- [TileUpdater.stopPeriodicUpdate](#) and [BadgeUpdater.stopPeriodicUpdate](#) Cancels the current periodic update process but does not clear the tile of existing updates.
- [TileUpdater.startPeriodicUpdateBatch](#) For tile updates only, is identical to [startPeriodicUpdate](#) but accepts an array of up to five URIs, automatically creating an update queue with the results (replacing any previous URIs). Note that [TileUpdater.enableNotificationQueue](#) must be set to `true` prior to using this method, as described earlier in the “Cycling, Scheduled, and Expiring Updates” section.

In all these cases, each URI is represented by a [Windows.Foundation.Uri](#) object and the polling period is set with a value from the [PeriodicUpdateRecurrence](#) enumeration. Values are `halfHour`, `hour`, `sixHours`, `twelveHours`, and `daily` giving a clear indication that periodic updates are meant for content that changes relatively infrequently, like the weather, daily offers from local retailers (in which case

you'd use start time), or the phases of the moon. For anything that requires more timely delivery, like appointment reminders, traffic conditions, current sports scores, or online auction status, you'll need to use push notifications.

In addition, all these updates can employ tags and expiration times as with local updates. One detail is that at any given polling interval, a web service can return only a single update payload—hence the need for [startPeriodicUpdateBatch](#). That said, if the notification queue is enabled and the updates from a single web service contain different tags, the live tile will behave the same as if those updates were issued locally: the most recent update for each tag (up to five) will be cycled through in the queue.

Hint The periodic update API does not directly provide a means to authenticate with the service. This typically isn't necessary because periodic updates are not designed to be user-specific. However, you can certainly include encrypted credentials in the URI with a query string. You might also be able to use the *Enterprise Authentication* capability if the app is running on a domain-joined system.

The app side of the periodic update scene is demonstrated in the [Push and periodic notifications client-side sample](#). Specifically, see scenarios 4 and 5, which are somewhat general tools to employ the [TileUpdater](#) and [BadgeUpdater](#) methods with a given service URI just as I've just described. A few lines of that code (condensed from `js/scenario4.js`) appear as follows:

```
var notifications = Windows.UI.Notifications;
var updater = notifications.TileUpdateManager.createTileUpdaterForApplication();

updater.enableNotificationQueue(true);
updater.startPeriodicUpdate(urisToPoll[0], recurrence);
updater.startPeriodicUpdateBatch(urisToPoll, recurrence);
```

The real work of periodic updates, however, lies in the service itself, whose responsibility it is to return the appropriate XML from which Windows can create an update. Being able to even run the client-side sample requires some service to which we can make requests, and unfortunately the Windows SDK does not provide one. So let's remedy that situation with a service of our own.

Because you'll likely use the client-side sample to play around with your own update service, though, there are two changes you should make, specifically to clear existing updates in the functions that stop polling. In `js/scenario4.js`, change the [stopTilePolling](#) function to read as follows:

```
function stopTilePolling() {
    var updater = notifications.TileUpdateManager.createTileUpdaterForApplication();
    updater.clear();
    updater.stopPeriodicUpdate();
    WinJS.log && WinJS.log("Stopped polling.", "sample", "status");
}
```

Similarly, change `stopBadgePolling` in `js/scenario5.js` to read as follows:

```
function stopBadgePolling() {  
    var updater = notifications.BadgeUpdateManager.createBadgeUpdaterForApplication();  
    updater.clear();  
    updater.stopPeriodicUpdate();  
    WinJS.log && WinJS.log("Stopped polling.", "sample", "status");  
}
```

Without these changes, old updates will persist on the tile even after you stop the updates. If you then change your web service but it has an error in the XML, you won't see any change on the tile and might think that the update worked when it really didn't. Trust me: making these small changes will simplify your life!

Web Services for Updates

Creating a web service for periodic updates means creating a web page at some given URI whose sole purpose is to respond to an `XmlHttpRequest` with XML content for a `TileNotification` or `BadgeNotification` object. Ideally, such a page also handles the scaling, accessibility, and localization parameters provided in the query string described earlier in "Using Local and Web Images."

The page can be implemented using whatever language and tools you want, such as PHP or ASP.NET. In fact, unless you really enjoy programming in Notepad, you'll certainly want to utilize a good web development tool! Visual Studio Express for Windows 8 is not actually equipped for this task; the full version of Visual Studio 2012 is. You might also look into Visual Studio Express 2012 for Web as another option; more on this in a moment. If you use ASP.NET, remember again that you can again employ the [Notifications Extensions Library](#) for easily creating the tile XML.

Some examples of pages that provide tile updates are given in the [Creating a great tile experience \(Part 2\)](#) post on the Windows Developers Blog. Based on those examples, here is a trivial (but functional) one-liner PHP page that will post XML for a badge update with the current day of the month:

```
<?php echo "<badge value='".date("j")."'/>"; ?>
```

For proper XML we should also include a header element, which also works:

```
<?php echo '<?xml version="1.0" encoding="utf-8"?>';  
echo "<badge value='".date("j")."'/>"; ?>
```

Drop this code into a `.php` file (see `HelloTiles/dayofmonthservice.php` in the companion content for this chapter) on whatever web server you might have access to and voila! There's a very basic service that delivers badge updates. You can use this in Scenario 5 of the Push and periodic notifications client-side sample—enter your page's URI in the box, press the button to start polling, and then check the sample's tile on the Start screen tile. In a few seconds you should see the day of the month appear as a badge. (Of course, with this ultrasimplistic example the date will reflect the local time on the web server rather than the device, which could be completely mismatched. A real service would be sensitive to time zone and other locale-specific considerations.)

Tip The tile and badge updaters are very sensitive to properly formed XML. In the PHP code above, leaving off the closing / for the `badge` element will make the update fail to appear. Avoiding such trivial errors is again why the Notifications Extensions Library was created, at least for ASP.NET. I'm hoping that some enterprising reader might consider a similar project for PHP and other server-side languages!

Going back to the Windows Developer Blog post mentioned earlier, I want to point out that the ASP.NET example given there—the one that begins with `@{`—is using the ASP.NET Razor syntax (typically in a `.cshtml` file), introduced with [Microsoft WebMatrix](#). Razor/WebMatrix, along with tools such as Visual Studio Express 2012 for Web and a whole lot else, can be installed through the [Web platform installer](#). To familiarize yourself with Razor, which works much in the same way as PHP, start with [Walkthrough: Creating a Web Site using Razor Syntax in Visual Studio](#).

To make that long story short, here are the steps in Visual Studio Express 2012 for Web to create a simple tile update service based on the Razor code in the blog post:

- Select File > New Web Site from the menu.
- In the New Web Site dialog, select ASP.NET Web Site (Razor v1), give it a project name and folder, and press OK. Call this site *HelloTiles*.
- Once the project is created, you should see the Default.cshtml file opened.
- Copy and paste the following Razor code into that file, replacing the default contents. The little piece of C# code at the top for the `weekDay` variable is something we'll use in the next section to demonstrate debugging; it's not used in generating the XML. Here, note that I tested and generated the XML contents by using the tile designer in Scenario 5 of the [App tiles and badges sample](#) (see Figure 13-15); this saved me lots of time wondering whether my XML was correct.

```
@{
    //
    // This is where any other code would be placed to acquire the dynamic content
    // needed for the tile update. In this case we'll just return static XML to show
    // the structure of the service itself.
    //
    var weekDay = DateTime.Now.DayOfWeek;
}
<?xml version="1.0" encoding="utf-8" ?>
<tile>
    <visual lang="en-US">
        <binding template="TileSquarePeekImageAndText02" branding="none">
            <image id="1" src="http://www.kraigbrockschmidt.com/images/Liam07.png"/>
            <text id="1">Liam--</text>
```

```

        <text id="2">Giddy on the day he learned to sit up!</text>
    </binding>
    <binding template="TileWideSmallImageAndText04" branding="none">
        <image id="1" src="http://www.kraigbrockschmidt.com/images/Liam08.png"/>
        <text id="1">This is Liam</text>
        <text id="2">Exploring the great outdoors!</text>
    </binding>
</visual>
</tile>

```

- Run the website in Internet Explorer using the Debug > Start Debugging command or the Internet Explorer toolbar button (where you'd find the Local Machine or Simulator options in Visual Studio Express 2012 for Windows 8). This will launch Internet Explorer with a URI like *http://localhost:52568/HelloTiles/Default.cshtml* where the port number is what routes the URI to the site running in the debugger. If you need to set up your localhost server, see the next section, "Using the Localhost."
- Run the Push notifications app sample, switch to Scenario 4, paste the URI into the URI1 field, and press the Start periodic updates.
- If all is well, you should see the wide tile update as follows. (OK, so it's another shameless picture of my kid...what can I say? You have to give us fathers a break!)



- If you use the Start screen's app bar command to make the tile smaller, thereby using the square tile payload in the XML, you should see it switch between to another gratuitous picture of my kid and peek text:



The complete website code for all this can be found in the HelloTiles example in this chapter's companion content. As simple as it is, it provides the basic framework in which you can add code to generate more dynamic results. In the top part of the file, within the `@{ }`, you can write whatever code you need by using `C#`.

Generally speaking, a real service that provides tile and badge updates would probably be connected on the server side to some other useful source of information, perhaps to an always-running process that can monitor other sites, extract the desired data, and generate updates. Those updates can be

returned from page requests, as we’ve seen here, or fed directly to WNS so that they can be pushed directly to specific clients. We’ll come back to this subject in “Push Notifications and the Windows Push Notification Service” later in this chapter.

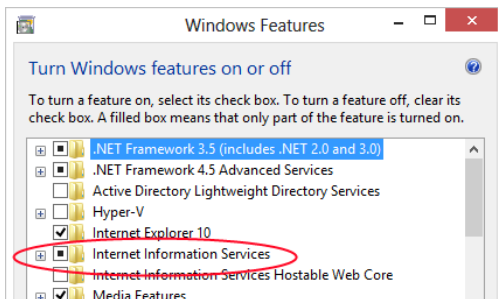
For now, a more pressing question is this: how does one actually debug such a service? Fortunately, the Visual Studio tools make this very straightforward through the localhost.

Using the Localhost

Debugging a tile and badge update service with periodic notifications can be a difficult proposition. You can just enter your service’s URI in a browser and use its View Source command to examine the XML, but how do you step through that server-side code to isolate problems?

The solution is to run your service on your local machine, as we just did in the previous section, where the URI references your localhost server, however you want to set it up. You can install a server like Apache, of course, or you can use the solution that’s built into Windows and integrated with the Visual Studio tools: Internet Information Services (IIS).

To turn on IIS in Windows, go to Control Panel > Turn Windows Features On Or Off. Check the Internet Information Services box at the top level, as shown below, to install the core features:



Once IIS is installed, the local site addressed by <http://localhost/> is found in the folder `c:\inetpub\wwwroot`. That’s where you drop something like the PHP page described in the last section so that you can use a URI like `http://localhost/dayofmonthservice.php` in the Push notifications sample (Scenario 5, in this case, for badge updates).

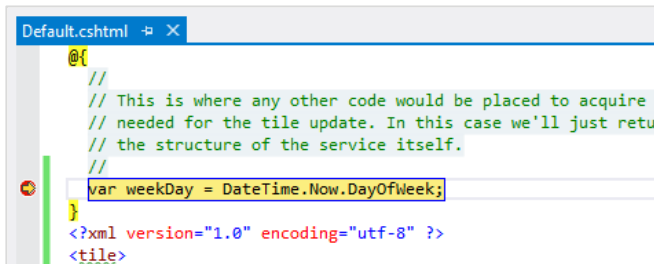
With the web page running on the local machine, you can hook it into whatever tools you have available for server-side debugging. Here it’s good to know that access to localhost URIs—also known as local loopback—is normally blocked for Windows Store apps unless you’re on a machine with a developer license, which you are if you’re been running Visual Studio or Blend. This won’t be true for your customer’s machines, though! In fact, the Windows Store will reject apps that attempt to do so.⁶⁷

⁶⁷ Visual Studio enables local loopback by default for a project. To change it, right-click the project in Solution Explorer, select Properties, select Configuration Properties > Debugging on the left side of the dialog, and set Allow Local Network Loopback to No. For more on the subject of loopback, see [How to enable loopback and troubleshoot network isolation](#).

To use PHP with IIS, you might need to install it through Microsoft's [Web platform installer](#) or the server-side code won't execute properly. After PHP installation, try entering the URI for the PHP page in your browser. If you get an error message that says "Handler PHP53_via_FastCGI has a bad module" (yeah, that's really helpful!), return to the Turn Windows Features On Or Off dialog shown earlier, navigate to Internet Information Services > World Wide Web Services > Application Development Features, check the box for CGI, and press OK. Once the CGI engine is installed, your PHP page should work.

If you plan to work in ASP.NET or Razor, I highly recommend you also install Visual Studio Express 2012 for Web through Web platform installer. When you run a website in its debugger, it assigns a port on localhost such as `http://localhost:53528` and launches Internet Explorer with that URI. The port links the browser to the debugger, so if you set a breakpoint in the page code, the debugger will stop at that point whenever there's a page request, allowing you to step through the code with the same features we've been enjoying in writing Windows Store apps.

For example, load up the HelloTiles example site with this chapter in Visual Studio Express 2012 for Web, set a breakpoint on the `var weekDay` line at the top, and start debugging. Once Internet Explorer has loaded `default.cshtml`, copy and paste its URI into Scenario 4 of the Push notifications sample. Press the Stop Periodic Updates button followed by the Start Periodic Update button to force a new request to the URI and—magic!—you should hit the breakpoint in the service:



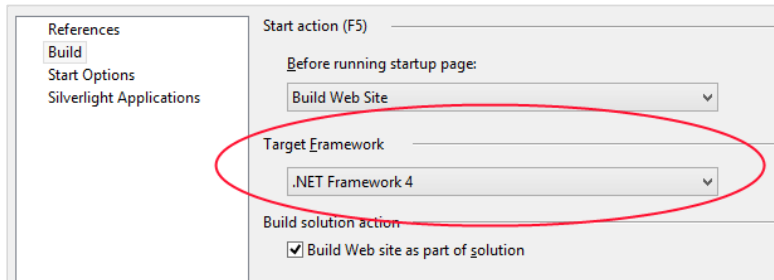
Windows Azure

When you're ready to upload an app to the Windows Store and have real customers using your web service, you'll need to consider where, exactly, you'll host that service so that it can scale to what hopefully becomes a very large customer base! During your development and testing process, of course, you can host the service anywhere you want because only a few instances of your app will ever call upon it. But if your app is acquired by many customers and each instance of that app starts banging on the host server for tile and badge updates, that host might soon become overloaded!

For this reason you should investigate services like Windows Azure, where more server power can be added when it's necessary and scaled back when it's not (and you pay only for what is actually used). To get started, visit <http://www.windowsazure.com> where you can set up a free 90-day trial for a hosted site. The Windows Azure site also provide direct support and SDKs for .NET, node.js, PHP, Java, and Python, along with Visual Studio Express 2012 with Web for Windows Azure SDK—all available again

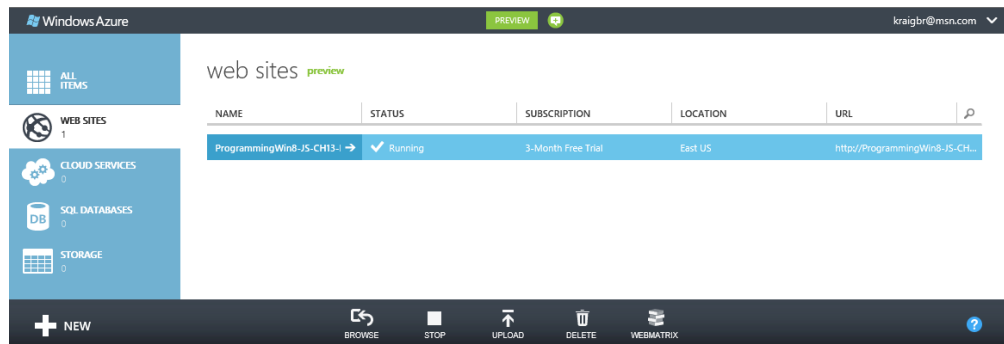
through the Web platform installer. You might also be interested in the [Windows Azure Toolkit](#) that provides project templates, samples, and other resources for creating services on Windows Azure.

Tip As of this writing, Windows Azure has support for .NET Framework 4.0 but not .NET Framework 4.5, so be sure to check the target framework in your project's Build settings before deployment. (Right-click your project in Visual Studio Web, select Property Pages, and click Build—see image below). When I deployed my service targeting version 4.5, pages like Default.cshtml produced errors.



As a brief walkthrough to get you started, I started my Windows Azure trial, installed the Windows Azure SDK for .NET, and deployed the HelloTiles example service with these steps:

- Go to the [Windows Azure Management Portal](#), and sign in with your account.
- Create a new website with some URI. I used *ProgrammingWin8-JS-CH13-HelloTiles*, making the full site URI <http://programmingwin8-js-ch13-hellotiles.azurewebsites.net/>. After this step, my portal looks like this:

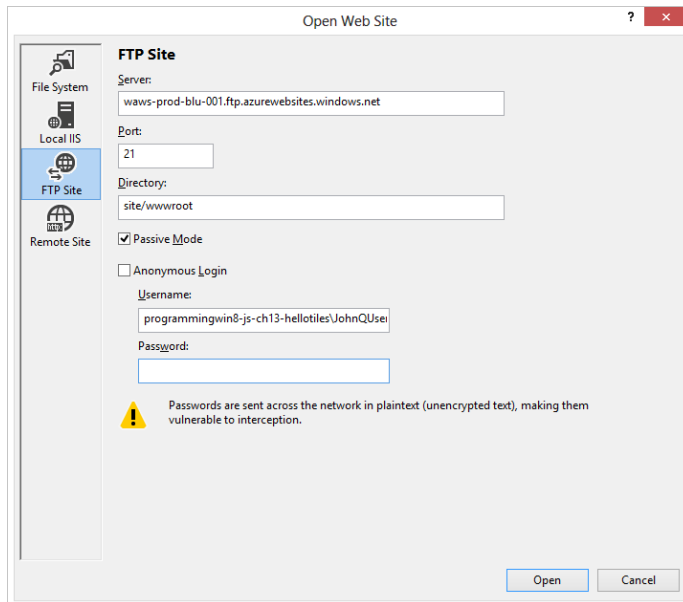


- To upload the site, you first need to set up FTP credentials. Click the site in the list, which takes you to its specific dashboard. Under Publish Your App, click Set Up Deployment Credentials and enter a username and password.
- Click Dashboard along the top of the window, and scroll down the right side to find the FTP hostname:

FTP HOSTNAME

<ftp://waws-prod-blu-001.azurewebsites.windows.net>

- In Visual Studio Express 2012 for Web, right-click the project and select Copy Web Site. In the dialog that appears, click Connect along the top to open the Open Web Site dialog (below). Select FTP site on the left, enter the FTP hostname on the top, enter *site/wwwroot* in Directory, and then enter your credentials at the bottom. Make sure to prefix your username with the site name and a backslash (for example, ProgrammingWin8-JS-CH13-HelloTiles\JohnQUser) or you'll be very confused by Azure's refusal to let you through the door! Finish by clicking Open.



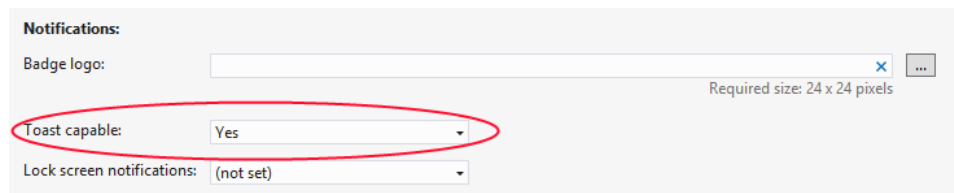
- Once Visual Studio Express 2012 for Web is connected to the Azure site, you can upload your files.
- Once that's complete, you should be able to use the site URI plus Default.cshtml as the URI for the tile update service. For instance, try *<http://programmingwin8-js-ch13-hellotiles.azurewebsites.net/dayofmonthservice.php>* with the Push notifications sample, Scenario 5, and *<http://ProgrammingWin8-JS-CH13-HelloTiles.azurewebsites.net/Default.cshtml>* with Scenario 4 and you'll see the same results as when we used localhost. You can also just visit these URIs directly to see the XML they're producing.

With this, you now have the ability to scale up your Windows Azure hosting, with your app supplying the hosted URI to the periodic update API. This also puts you in a good position to use push notifications, as we'll see later, but because those are also often used with toasts, let's see how toasts work next.

Toast Notifications

So far in this chapter we've exhausted the subject of tiles and tile updates, which is actually a great prelude to our next topic, toast notifications. This is because the process of creating and issuing toasts is quite similar to that for tiles and is simplified by the fact that there are no periodic updates for toasts: they either come from the running app, from background tasks, or through push notifications, as we'll see in "Push Notifications and the Windows Push Notification Service" below. Fortunately, the topic of toasts is considerably shorter than that of tiles. Here are the salient aspects of toasts:

- Toasts always use the app's Background Color and Foreground Text settings in the manifest for branding, along with the small logo. There are no means to override this; the `branding` attribute in the XML is ignored for toasts.
- An app must set the Toast Capable setting in its manifest for any toasts to appear on its behalf. This is found in the Application UI > Notifications area:



- As shown long ago in Figure 13-8, the user can disable toasts for a particular app or disable them globally (which I find helpful when recording a screencast!). System administrators can also disable toasts by policy. To check this status programmatically, look at the `ToastNotifier.setting` property, a value from the `NotificationSetting` enumeration that will be `enabled`, `disabledForApplication`, `disabledForUser`, or `disabledByGroupPolicy`.
- When enabled, toasts always appear in the upper right corner of the screen (left-to-right languages) or the upper left corner (right-to-left languages). This is not configurable.
- Toasts are managed through instances of the `Windows.UI.Notifications.ToastNotification` or `Windows.UI.Notifications.ScheduledToastNotification` classes. The first supports an `expirationTime` property; the scheduled toast supports `deliveryTime`, `snoozeInterval`, and `maximumSnoozeCount` properties.
- As with tiles, the content of toasts are created with an XML payload from one of four text-only and four image-plus-text templates, as shown on the [Toast template catalog](#). Toast templates are acquired from the `ToastNotificationManager` object's `getTemplateContent` method, can be created from strings, or can be created through the Notifications Extensions Library. Various options can be set in the XML:

- Toasts can include text and an image, where the image can come from the app package, app data, or a remote source (given *Internet Client* capability). Images have the same limits as with tiles: 1024x1024 maximum resolution and 200KB maximum file size. Unlike tiles, however, if the image exceeds the limits, the notification will still show but with a gray placeholder image instead.
- Toasts can specify a predefined sound to play when the toast appears, with a looping option. Custom sounds are not supported.
- By default, toasts appear for seven seconds (five seconds opaque plus a two-second fade) or until activated or dismissed. (The opaque duration is available through the [Windows.UI.ViewManagement.UISettings.messageDuration](#) property.) You can issue long-duration toasts, looping toasts, and recurring toasts that appear a given number of times with some interval in between.
- A toast is issued through the [ToastNotifier](#) class, namely the [show](#) and [addToSchedule](#) methods for immediate and scheduled toasts, respectively. The [ToastNotifier](#) also provides methods to manage previously scheduled toasts.
- Like secondary tiles, toast notifications can (and generally should) be created with specific arguments in the XML payload that will be passed to the app's [activated](#) event handler with the activation kind of [launch](#). Without such arguments, no [activated](#) event is raised, but otherwise an app handles toast notifications exactly as it would a secondary tile. Alternately, an app can listen to specific events that the toast itself will raise when it's activated or dismissed.

The following sections provide details on a number of these steps, using the [Toast notifications sample](#) and [Scheduled notifications sample](#) for reference. We also recommend you review the [Guidelines and checklist for toast notifications](#).

Tip As noted before, toasts are not enabled within the Visual Studio simulator; you must run these samples on the Local Machine or a Remote Machine to see the toasts.

Creating Basic Toasts

Let's start with Scenarios 1, 2 and 3 of the Toast notifications sample, which shows how to issue toasts from a running app using the text and text+image templates. As shown in Figure 13-16 and Figure 13-17 (for text-only and text+image toasts, respectively), up to three toasts can be visible at one time. Remember that you must have Toast Capable set to Yes in the app manifest for any of this to work.

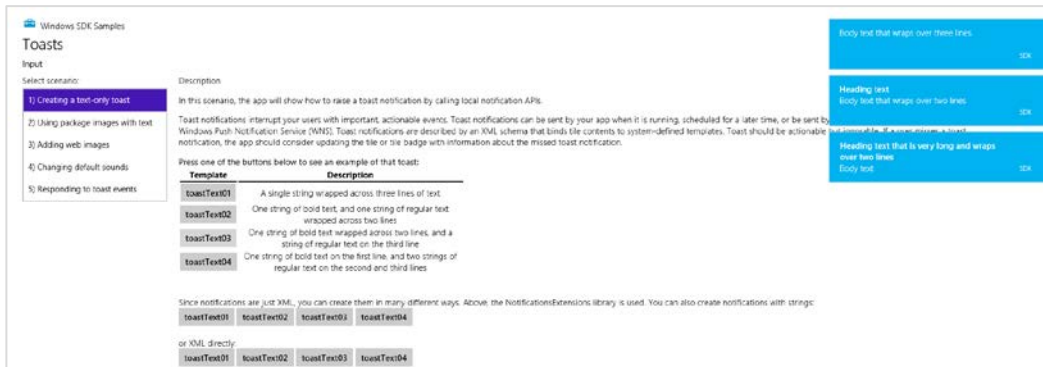


FIGURE 13-16 Issuing text toasts through Scenario 1 of the Toast notifications sample. (The bottom of the app is cropped.)

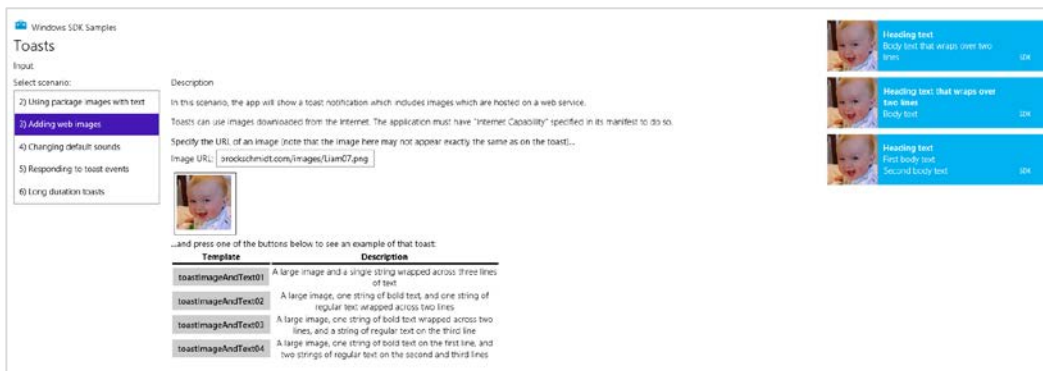


FIGURE 13-17 Issuing text+image toasts through Scenario 3 of the sample. (The bottom of the app is again cropped.) Scenario 2 does the same thing with in-package images that aren't nearly as interesting, in my paternal opinion, as my cute kid!

Just as we saw earlier with tiles, the sample shows how to create the XML payloads for toasts by using template content from [ToastNotificationManager.getTemplateContent](#), the Notifications Extensions Library, or XML strings. The resulting `XmlDocument` is then used to create a `ToastNotification` object that is then passed to the `ToastNotifier.show` method.

For example, here's how Scenario 1 (`js/scenario1.js`) issues a toast using the `toastText01` template (a value from the `ToastTemplateType` enumeration) through `getTemplateContent`:

```
var Notifications = Windows.UI.Notifications;

function displayToastUsingXmlManipulation(e) {
    // toastTemplateName is set according to the button you click

    var notificationManager = Notifications.ToastNotificationManager;
    var toastXml = notificationManager.getTemplateContent(
        Notifications.ToastTemplateType[toastTemplateName]);
```

```

// Populate the XmlDocument in toastXml (code omitted)

var toast = new Notifications.ToastNotification(toastXml);
notificationManager.createToastNotifier().show(toast);
}

```

The following code from Scenario 3 (`js/scenario3.js`) demonstrates creating a toast with XML strings (`toastImageAndText01`). As with tiles, you can use `ms-appx:///`, `ms-appdata:///local`, or `http://` URIs to refer to images (in-package, app data, and remote images, respectively):

```

function displayWebImageToastWithStringManipulation(e) {
    // toastTemplateName is set according to the button you click

    var notificationManager = Notifications.ToastNotificationManager;
    var toastXmlString;

    if (templateName === "toastImageAndText01") {
        toastXmlString = "<toast>"
            + "<visual version='1'>"
            + "<binding template='toastImageAndText01'>"
            + "<text id='1'>Body text that wraps over three lines</text>"
            + "<image id='1' src='" + urlBox.value + "' alt='" + altText + "'/>"
            + "</binding>"
            + "</visual>"
            + "</toast>";
    } else {
        // Other cases omitted
    }

    var toastDOM = new Windows.Data.Xml.Dom.XmlDocument();
    toastDOM.loadXml(toastXmlString);
    var toast = new Notifications.ToastNotification(toastDOM);
    notificationManager.createToastNotifier().show(toast);
}

```

Butter and Jam: Options for Your Toast

Beyond the properties you can assign when creating a `ToastNotification` object (or a `ScheduledToastNotification`), there are additional bits you can include within the XML, as described in the [Toast schema](#):

- The root `toast` element in the XML has optional `launch` and `duration` attributes. The `launch` attribute can be assigned a string that will be passed to the app's activated handler as `eventArgs.detail.arguments`, exactly as happens with a secondary tile. (See "App Activation From a Secondary Tile" earlier in this chapter.) The duration attribute can have values of `short` (five seconds or the value from PC Settings > Ease of Access) or `long` (25 seconds or the value from PC Settings > Easy of Access, whichever is longer; refer back to Figure 13-7).
- The `visual` and `binding` elements in the XML can have `branding` and `addImageQuery` attributes that act exactly like their counterparts for tiles. Refer back to the "Branding" and "Using Local and Web Images" sections under "Tiles, Secondary Tiles, and Badges." The `image` element also

supports `addImageQuery` for scale, language, and contrast settings.

- The `visual`, `binding`, and `text` elements support a `lang` attribute to identify the current app language.

The `toast` element can also have a child `audio` element through which you can add a sound to a toast notification provided that the user has not disabled notification sounds altogether in PC Settings > Notifications. (Refer to Figure 13-8.) The particular sound is set with the `src` attribute and must be a one of the following string values as described in the [Toast audio options catalog](#):⁶⁸

- `ms-winsoundevent:Notification.Default`
- `ms-winsoundevent:Notification.IM`
- `ms-winsoundevent:Notification.Mail`
- `ms-winsoundevent:Notification.Reminder`
- `ms-winsoundevent:Notification.SMS`
- `ms-winsoundevent:Notification.Looping.Alarm`
- `ms-winsoundevent:Notification.Looping.Alarm2`
- `ms-winsoundevent:Notification.Looping.Call`
- `ms-winsoundevent:Notification.Looping.Call2`

Separately, the `audio.silent` attribute controls whether audio plays at all (`false`, the default) or is muted (`true`). If the `toast.duration` attribute is set and you set `audio.src` to one of the latter four “Looping” sounds above, you can also set `audio.loop` to `true` (to repeat the sound) or `false` (to play the sound only once, the default).

Scenario 4 in the Toast notifications sample lets you play with the different notification sounds—different buttons choose different sounds. The text of each button (in a variable named `toast-SoundSource`) is appended to the `ms-winsoundevent:Notification`, as in this XML used to create the notification:

```
"<audio src='ms-winsoundevent:Notification.'" + toastSoundSource + "'/>"
```

Scenario 6 shows the use of the `loop` attribute in the XML as well:

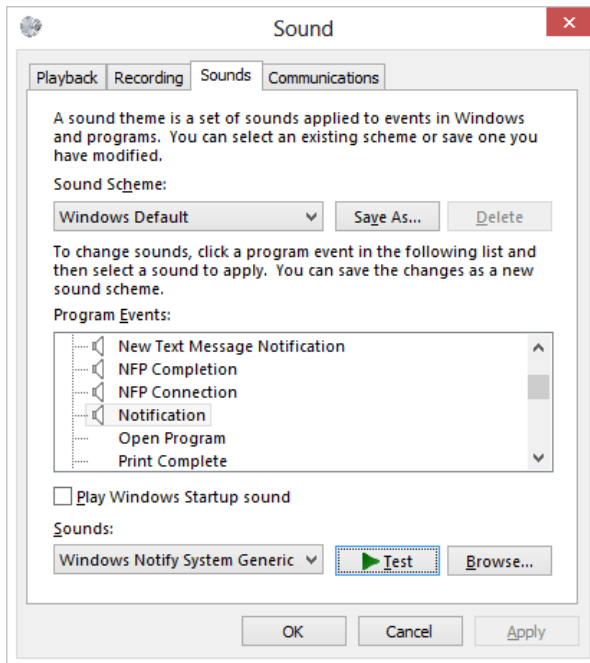
```
"<audio loop='true' src='ms-winsoundevent:Notification.Looping.Alarm'/">"
```

⁶⁸ With all the catalogs we’ve seen in this chapter, it feels like we’ve been shopping! More seriously, the fact that you must use audio from this list means that custom audio is not supported.

Did you actually hear any sounds? When I first ran these samples, I sure didn't! It took me a while to figure out why, so let me save you the trouble.

The values in the `audio.src` attribute simply map to various system sounds that are assigned in Control Panel > Hardware and Sounds > Change System Sounds, which displays the dialog box below. Having tired of all the beeps, boings, and dingalings that were once all the rage on personal computers, I routinely select the "No Sounds" option under Sound Scheme. As a result, there were no sounds assigned to anything in the Program Events list, so there were no sounds whatsoever for toast notifications. When I selected the Windows Default scheme, I then heard sounds with the toasts.

In short, the user does have ultimate control over the sounds, both generally in PC Settings and specifically in the dialog box below. So, if it's appropriate to use a sound at all, just choose the one that's closest to the nature of your toast and leave it at that.



Tea Time: Scheduled Toasts

Issuing toasts from a running app is all well and good, but it's not actually a common scenario because the user is already looking at the very same app. What's more interesting are cases where the app isn't necessarily running when a notification appears. This is why toasts are often used with push notifications as well as background tasks, as we'll see in the last two sections of this chapter, but the other means is a scheduled toast that will simply appear at some later time regardless of whether the app is running. This is a great way to invite the user to activate the app again.

A scheduled toast is created using [Windows.UI.Notifications.ScheduledToast-Notification](#) instead of the usual [ToastNotification](#) as we've been using. There are two forms of scheduled notification, as indicated by its pair of constructors:

- [ScheduledToastNotification\(content, deliveryTime\)](#) Creates a one-time scheduled toast with the toast's [XmlDocument](#) in [content](#) and the UTC [DateTime](#) when it should appear in [deliveryTime](#).
- [ScheduledToastNotification\(content, deliveryTime, snoozeInterval, maximumSnoozeCount\)](#) Creates a recurring scheduled toast whose content will appear at [deliveryTime](#). If the toast is dismissed either explicitly or by letting it disappear on its own, it will continue to appear a total of [maximumSnoozeCount](#) times at intervals defined by the number of milliseconds in [snoozeInterval](#). The [snoozeInterval](#) must be set between 60 seconds and 60 minutes; for longer intervals it's best to just schedule separate toasts altogether.

A [ScheduledToastNotification](#) also has an [id](#) property, a maximum 16-character string that's used to identify that toast. If you schedule a toast with the same [id](#) as an existing one, the new one will replace the old.

In all cases, the toast is scheduled by calling the [ToastUpdater.addToSchedule](#) method passing in the notification object. Here's the process in code, as found Scenario 1 of the [Scheduled notifications sample](#) (js/scenario1.js), where [toastDOM](#) is the [XmlDocument](#) containing the content and [dueTime](#) is determined by a UI control in the sample. First, for a one-time notification:

```
var Notifications = Windows.UI.Notifications;

toast = new Notifications.ScheduledToastNotification(toastDOM, dueTime);
Notifications.ToastNotificationManager.createToastNotifier().addToSchedule(toast);
```

Second, for a notification that will repeat five times at 60-second intervals (the option that's exercised if you check the Repeat checkbox in the sample's UI):

```
var Notifications = Windows.UI.Notifications;

toast = new Notifications.ScheduledToastNotification(toastDOM, dueTime, 60 * 1000, 5);
Notifications.ToastNotificationManager.createToastNotifier().addToSchedule(toast);
```

To enumerate currently scheduled toasts, call [ToastNotifier.getScheduledToast-Notifications](#). This returns a vector of [ScheduledToastNotification](#) objects, any of which can be canceled through [ToastNotifier.removeFromSchedule](#). These methods are demonstrated in Scenario 2 of the Scheduled notifications sample that I will leave you to examine more closely. Also, there are some debugging tips on the [Guidelines and checklist for scheduled notifications](#) topic in the documentation, mostly to note that the system has a limit of 4096 total notifications and to make sure you've set Toast Capable in the manifest to Yes.

Toast Events and Activation

As far as toasts are concerned, we have perhaps saved the best topic for last! The whole purpose of a toast is to get the user's attention and have them activate your app to take some kind of action. An app will commonly navigate to an appropriate page for whatever the content of the toast implies.

The most straightforward case of activation is with a scheduled toast or one that has been put up by a background task or through a push notification. In all of these cases the app won't be running, so Windows will start it with the activation kind of `launch`, where the value of the `toast.launch` attribute will be in the `activated` event's `eventArgs.detail.arguments` property. This is, once again, identical to the way secondary tiles work and you can process the arguments value however you wish.

If the app is not running when the toast is activated, it will still be launched even if the `toast.launch` attribute is empty. That is, toasts that occur under nonrunning conditions can be used to just launch the app, if desired. On the other hand, if a *running* app issues a toast with no `toast.launch` value, its `activated` event will *not* be fired at all. This is a way of saying that activation through a toast with no additional information would never cause the app to navigate in the first place, so what's the point of firing the `activated` event? None whatsoever. Thus, if a running app issues a toast with the intent that activating that toast will switch to a different part of the app, a `launch` value is essential. (Of all the scenarios in the Toast notifications sample, only Scenario 5 provides a `launch` value; set a breakpoint in the `activated` event of `js/default.js`, and you'll see that Scenario 5 is the only time that event will fire when you tap a toast.)

Still, a running app might want to know when the user interacts with a toast, launch arguments aside. For this purpose it can listen to a `ToastNotification` object's `activated`, `dismissed`, and `failed` events, a few of which are demonstrated in Scenario 5 of the sample. The `activated` event has no specific `eventArgs`, but `dismissed` comes with a `ToastDismissalReason` in `eventArgs.reason` (values are `userCanceled`, `applicationHidden`, and `timedOut`), and the `failed` event comes with an error code in `eventArgs.errorCode`. (These are all events from a WinRT object, so be sure to manage them with `removeEventListener` as appropriate. See the section "WinRT Events and `removeEventListener`" in Chapter 3.)

Note that a `ScheduledToastNotification` does not support any of these events because the assumption is that the app probably won't be running by that time anyway.

Push Notifications and the Windows Push Notification Service

We've now finally arrived in this chapter where we can leave running apps behind and look at more of the fun things that can happen behind the scenes. Earlier, in "Periodic Updates," we learned that the shortest interval you can use with that method is pretty darn long by a computer's reckoning: 30 minutes. That's even long by many human standards, especially those of a user who really wants to know what's happening with whatever information source your app is connected to.

To update a tile, set a badge, or issue a toast as quickly as the system allows, and to personalize the content (as with calendar reminders and email alerts), it's necessary to be a little pushy and use *push notifications*. These are notifications that come to a system from an outside agent, typically a service that is monitoring some other source of information and detects a condition for which a notification is appropriate. We saw the mechanism in the “The Four Sources of Updates and Notifications” section and Figure 13-14 early in this chapter. To summarize:

- When launched, an app requests a channel URI for each of its live tiles and then sends those URIs to its associated web service. An app should do this each time it's launched, as the expiration period for a WNS channel is 30 days.⁶⁹ Each channel URI is unique for the user, the tile, and the device.
- The web service stores the channel URI and associates it with a user to customize their notification content (as again with email and calendar alerts, notifications from friends' activities, etc.).
- When needed, the web service issues updates (XML payloads) to that channel.
- WNS then send the notification to the client devices where the app acquired the channel URI. Those notifications can update tiles, update badges, issue toasts, and update the lock screen (with appropriate lock screen apps and background tasks).

It's also possible for the service and WNS to send what is called a *raw notification*, which can contain any payload you want: there just needs to be someone listening as Windows won't know what to do with the data. A foreground app can listen through the `PushNotificationChannel.onpush-notificationreceived` event; a lock screen app can listen with a background task. In the latter case, raw notifications are generally used to deliver information to the background task and issues other notifications in response or updates app data.

Before you do anything in your app, however, you need to follow the instructions on [How to authenticate with the Windows Push Notification Service \(WNS\)](#) on the Windows Developer Center (which is part of a whole series on [Sending push notifications](#)). This will walk you through the steps on the Windows Store Dashboard to obtain a Package Security Identifier (SID) and a secret key that your web service must use to authenticate itself with WNS.

That done, let's go through each of the steps in turn, using Scenarios 1–3 of the same [Push and periodic notifications client-side sample](#) we used earlier for periodic updates.

Note Because channel URIs are unique for an app+user+device, using push notifications can become an expensive proposition for your web service, which must record and maintain a channel for every unique tile on every user's device and then figure out when to send which notifications to which channels. If your app becomes popular, this will require scaling up your service to potentially manage

⁶⁹ Usage of the app is also a pretty good indicator to the service as to whether it needs to continue supporting a particular user.

thousands or even millions of channel URIs. For this reason, seriously evaluate whether periodic notifications will be sufficient for your scenario, especially for updates that aren't user specific, because they will be much simpler on the service side of the picture.

Requesting and Caching a Channel URI (App)

Requesting a channel URI is done through the

`Windows.Networking.PushNotifications.PushNotificationChannelManager` object. This manager has only two methods: `createPushNotificationChannelForApplicationAsync` and `createPushNotificationChannelForSecondaryTileAsync`. The first is clearly linked to the app tile as well as toast notifications; the second is clearly for use with secondary tiles and takes a `tileId` argument to identify the specific one.

The result of both async operations is a `PushNotificationChannel` object that will be passed to your completed handler, as shown in Scenario 1 of the sample (start in `js/scenario1.js`, then go into `js/notifications.js`):

```
var channelOperation;

// Channel for the app tile
if (isPrimaryTile) {
    channelOperation = Windows.Networking.PushNotifications.PushNotificationChannelManager
        .createPushNotificationChannelForApplicationAsync();
} else {
    // Channel for a secondary tile
    channelOperation = Windows.Networking.PushNotifications.PushNotificationChannelManager
        .createPushNotificationChannelForSecondaryTileAsync(itemId);
}

channelOperation.done(function (newChannel) {
    // Send channel to web service
}, /* error handler */
);
```

The `PushNotificationChannel` object (`newChannel` in the code above) is a simple object with just a few members, but they are important ones:

- `expirationTime` A read-only property indicating when the channel expires—notifications sent to this channel after expiration will be rejected. Apps must be sure to refresh their channels when needed to avoid an interruption in notifications.
- `uri` A read-only URI to which the app's web service sends notifications to WNS.
- `close` A method that explicitly invalidates the channel.
- `pushnotificationreceived` An event that's fired when a notification is received on the client device from this notification channel. This will be fired only for apps that are in the foreground.

Your app should go through this short process to obtain the necessary channel URIs whenever it's launched as well as when it's resumed (especially if any channel's `expirationTime` has passed). It's unlikely that an app would stay suspended for that long, but it's still possible! Furthermore, if you're concerned that your app might not run for more than 30 days, you can implement a background task on a maintenance trigger for this purpose. See "Tasks for Maintenance Triggers" later in this chapter and Scenario 2 of the sample.

Again note that you may have more than one channel URI if you're also using push notifications for secondary tiles as well as your app tile. In this case you'll be managing a separate channel URIs for each tile.

Each time through the process, save the channel URI for each tile in your local app state. This is so that you can check on subsequent runs if the URI is the same as one you've already obtained and sent to your web service, in which case you can avoid unnecessary network traffic.

Sending the URI to your web service can be done with a simple call to `WinJS.xhr`, as in the sample (inside `channelOperation.done`). Here we also see the checks for whether the URI is the same as before:

```
channelOperation.done(function (newChannel) {
    // _urls[] is an array of channel ids for primary and secondary tiles
    var tileData = that._urls[itemId];

    // Upload the channel URI if the client hasn't recorded sending the same
    // uri to the server
    if (tileData && newChannel.uri === tileData.channelUri) {
        // This saves the URI to local app data
        that._updateUrl(url, newChannel.uri, itemId, isPrimaryTile);
        completed(newChannel);
    } else {
        WinJS.xhr({
            type: "POST",
            url: url,
            headers: { "Content-Type": "application/x-www-form-urlencoded" },
            data: "channelUri=" + encodeURIComponent(newChannel.uri) +
                "&itemId=" + encodeURIComponent(itemId)
        }).done(function (request) {

            // Only update the data on the client if uploading the channel URI succeeds.
            // If it fails, you may considered setting another background task, trying
            // again, etc. (An exception will be thrown if it fails, ending up in the
            // error handler instead.)
            that._updateUrl(url, newChannel.uri, itemId, isPrimaryTile);
            completed(newChannel);
        }, failed);
    }
}, failed);
```

Managing Channel URIs (Service)

If you use the code in the previous section, your web service that generates push notifications will receive HTTP POST requests with unique channel URIs for each and every tile. This isn't the only way to transport channel URIs, of course; in fact, because a channel URI might be used to transmit personal information through notifications, it should ideally be encrypted with a private key before it's sent to the server. Otherwise someone could possibly intercept that URI and use it to redirect user-specific notifications.

In any case, the service must expect to receive—and then manage—a unique URI for each app/user/device combination. This underscores the fact that push notifications are best used for user-specific notifications rather than broadcast notifications. In the latter case, setting up a service for periodic updates is a much easier solution.

Once the service receives a channel URI along with any data to identify the user and the purpose of the channel, it should securely save that information in persistent storage of some kind, such as a SQL Server database (for an ASP.NET service) or a MySQL (for a PHP service).

It's important that the service also removes obsolete channel URIs. If it receives a new URI for the same user and the same purpose, it should replace the old with the new. It should also remove any URIs from its data store if it receives an HTTP 404 or 410 error back from WNS, indicating an obsolete channel.

A simple ASP.NET service page that can receive a post from Scenario 1 of the [Push and periodic notifications client-side sample](#) can be found in the HelloTiles website project in this chapter's companion content, specifically `receiveuri.aspx`. To run this service, make sure you have the localhost established, as described earlier in the "Using the localhost" section for periodic updates. You may also need to install ASP.NET on your localhost. An easy way to do this is to obtain the [Background transfer sample](#), go into its Server folder, and then from an administrator command prompt run **powershell -ExecutionPolicy unrestricted -file serversetup.ps1**. If you then run the site in Visual Studio Express 2012 for Web as we did before, you should have a localhost port for the service (for instance, `http://localhost:52568/HelloTiles/receiveuri.aspx`).

You can then set a breakpoint in the service code, paste the service URI into Scenario 1 of the Push notifications sample, and press its Reopen Channel And Send To Server button. This should hit the breakpoint in the service and allow you to step through the code that processes the request. Here you'll find that the request contains *channelUri* and *itemId* values (along with *LOGON_USER*), which can be saved for when the service needs to send a notification to WNS. Something similar can be written in other server-side languages, of course, and a great place to find tools to help with writing services is the [Windows Azure Toolkit](#).

Sending Updates and Notifications (Service)

Before a service can send updates, it must to authenticate itself with WNS by sending the Package Security Identifier (SID) and client secret as obtained through the Windows Store Dashboard. This is a

matter of sending an XmlHttpRequest to WNS (via HTTPS) that looks like this:

```
POST /accesstoken.srf HTTP/1.1
Content-Type: application/x-www-form-urlencoded
Host: https://login.live.com
Content-Length: 211
grant_type=client_credentials&client_id=ms-app%3a%2f%2f5-1-15-2-2972962901-2322836549-3722629029-1345238579-3987825745-2155616079-650196962&client_secret=Vex8L9W0FZuj95euaLrvSH7XyoDhLJc7&scope=notify.windows.com
```

where you must make sure the values of `client_id` and `client_secret` match the package SID and client secret. If the authentication works, you'll receive a 200 OK response with the access token you need for sending notifications:

```
HTTP/1.1 200 OK
Cache-Control: no-store
Content-Length: 422
Content-Type: application/json
{
  "access_token": "EgAcAQMAAAALYAAY/c+Huiw3Fv4Ck10UrKNmtxRO6Njk2MgA=",
  "token_type": "bearer"
}
```

Code that accomplishes these steps for a service written in C# can be found on [How to authenticate with the Windows Push Notification Service](#), where your service would use the `GetAccessToken` method shown there to obtain an `OAuthToken` object with the information from the response. Services written in other languages will obviously need to use the appropriate means to send the request and receive the response.

Whatever the case, once you have the access token you're ready to start sending updates and notifications via XmlHttpRequests to the channel URIs maintained by the service.

For tile updates, badge updates, and toast notifications, sending a notification means generating the XML payload as for any other update or notification, and then sending it to WNS with the previously acquired access token. The only real difference between these requests, besides the specific XML, is the value of X-WNS-Type in the request header: `wns/badge`, `wns/tile`, `wns/toast`, or `wns/raw` (see the next section). Otherwise the code is the same.

Generic code for a C# service can be found on [Quickstart: Sending a tile push notification](#) and [Quickstart: Sending a toast push notification](#). I've included a badge update version in `sendBadgeToWNS.aspx` in the Hello Tiles example site with this chapter, where the SID and client secret are ones I obtained for the Push notifications sample. (You may need to re-create those yourself.) To test all this, run the Hello Tiles website in Visual Studio Express 2012 for Web and set a breakpoint at the beginning of `sendBadgeToWNS.aspx`. Assuming you're running Scenario 1 of the Push notifications sample in Visual Studio Express 2012 for Windows 8 to upload a channel URI to `receiveuri.aspx`, there should be a file called `channeluri.aspx.txt` in the website project that contains the uploaded data.

Now switch to Scenario 3 of the Push Notifications sample and press the button to start listening to the `pushnotificationreceived` event. In `js/scenario3.js` of that sample, set a breakpoint within the

`pushNotificationReceivedHandler` function. With all this in place, open a browser and enter the address of `sendBadgeToWNS.aspx` on the localhost: `http://localhost:52568/HelloTiles/sendBadgeToWNS.aspx`, for example. This should hit the breakpoint in Visual Studio Express 2012 for Web, where you can walk through that page's code and see it loading the channel URI from `channeluri.aspx.txt`, to which it then sends a badge update. When that happens, you should hit the breakpoint in the Push notifications app where you can walk through that code. Note that when you get to the line `e.cancel=true`, skip over it—right-click the line below it and select Set Next Statement. This will allow Windows to process the notification and update the badge for the Push notifications sample, which should now look like the following with a * badge on the lower right:



Note again that if you receive an HTTP 404 or 410 error back from WNS, the channel URI is no longer valid and you should remove it from your list. It's also good for the app to notify your service whenever it no longer needs updates for a particular channel (no point in paying for unproductive bandwidth). And if WNS returns an error, avoid posting the update again unless it makes sense for your scenario.

A 404 or 410 error is different, by the way, from an inability to deliver to notification because the client is offline. In this case WNS will cache the tile, badge, or raw notification until the client reconnects. In other words, it's not a condition that your service has to worry about. Send notifications as you always would, and let WNS handle the delivery details.

Raw Notifications (Service)

If you use `wns/raw` as the push notification type, the payload included with the push notification can be anything you want, not just XML, as long as it's under 5KB. Of course, Windows cannot do anything with this payload directly, so an app has to provide a handler for receipt of the notification, as the next section explains.

Receiving Notifications (App)

A running app receives push notifications through the `PushNotificationChannel.onpush-notificationreceived` event. This is again required to process `wns/raw` payloads but can be used for any type. If the app is not running, of course, it won't receive this event. Instead, the app must be on the lock screen with a `PushNotificationTrigger` background task for this purpose. (See "Lock Screen Dependent Tasks and Triggers" later in this chapter.) That piece of code will then receive the XML payload, process it, and issue whatever tile updates, badge updates, or toast notifications are necessary. Besides saving some state to the app data folders, this is really all that the background task can do, but it's enough to keep that sense of aliveness going as well as invite the user to launch the app in response.

In the running app, the `pushnotificationreceived` event is fired for the other notification types as well. Scenario 3 of the sample shows this in its event handler—I've modified this code a little bit for simplicity:

```
function startListening() {
    // Assume channel has been obtained and validated
    channel.addEventListener("pushnotificationreceived", pushNotificationReceivedHandler);
}

function pushNotificationReceivedHandler(e) {
    // Extract notification payload for each notification type

    var notificationPayload;
    switch (e.notificationType) {
        case pushNotifications.PushNotificationType.toast:
            notificationPayload = e.toastNotification.content.getXml();
            break;

        case pushNotifications.PushNotificationType.tile:
            notificationPayload = e.tileNotification.content.getXml();
            break;

        case pushNotifications.PushNotificationType.badge:
            notificationPayload = e.badgeNotification.content.getXml();
            break;

        case pushNotifications.PushNotificationType.raw:
            notificationPayload = e.rawNotification.content;
            break;
    }

    // Process the notification: set e.cancel to true to suppress automatic handling.
}
```

The last bit in the comment above is important. When you receive this event in a running app, it wouldn't be necessary to display a toast unless it pertains to some other part of the app that isn't visible. For example, if you have an app that handles both email and a calendar, you might want to show email toasts when the user is looking at the calendar and calendar toasts when the user is looking at email. In this case, setting `e.cancel` to `true` will suppress the toast.

With the `pushnotificationreceived` event, the running apps gets first crack at raw notifications. If the app doesn't process it, the notification will be sent to any lock screen background task configured for the `PushNotificationTrigger`. In either case, refer to the [Raw notifications sample](#) for more details.

Debugging Tips

When using push notifications, experience shows that if notifications aren't getting through, it's typically not a problem with WNS. Here's a list of things to check (thanks to Hans Andersen for this list):

- Check the return status of your HTTP POSTs to WNS. If it's returning an HTTP 200 response, check the X-WNS-NotificationStatus and other headers you get back. Look particularly for the status of "Received," which indicates that a notification has gone to the client.
- Lacking anything conclusive in the headers, run Event Viewer and check the events under *Application And Services Logs > Microsoft > Windows > Push Notifications Platform > Operational* to see the activity.
- Also look under *Application And Services Logs > Microsoft > Windows > Immersive-Shell > Microsoft-Windows-TWinUI > Operational* to see if there are error messages related to XML parsing about the same time you expected to receive a notification.
- Even if the XML is well-formed, an update might not show up if a referenced image is either too large (pixel dimensions or file size), if the image is the wrong format (for example, TIF), if the image is corrupt, if the server handling the image request can't handle the query parameters for the tile (scaling, contrast, language), or if the server is encountering other errors as might be revealed in its own logs.
- If updates appear but after a considerable delay, it could just mean internal timeouts or other network latency within the tile and notification infrastructure. If this happens, just accept that the world isn't always perfect and operations must sometimes be retried!

Windows Azure Toolkit and Windows Azure Mobile Services

Clearly, plenty of work is involved to create a service capable of receiving channel URIs and sending notifications through WNS. Recognizing this, the [Windows Azure Toolkit](#) once again provides some solutions. Going into all the details is beyond the scope of this book, but the link above will get you started. More specifically, check out the Azure [Notifications Samples](#) and the [Raw Notifications Sample](#), as well as the [Push Notification Worker sample](#). There is also a helpful video on the Channel 9 site: [Episode 73 – Nick Harris on Push Notifications for Windows 8](#). Additional resources have likely been published since this chapter was written, so a quick Internet search will likely turn up more.

Also check out [Windows Azure Mobile Services](#), which helps you create a scalable backend for an app, including structured cloud data, authentication, and push notifications. This is quite new as of the time of writing, so I don't have links to other resources, but it's certainly worth looking into.

Background Tasks and Lock Screen Apps

At the end of the introduction to this chapter, I described how everything we've talked about so far helps to "create an environment that is alive with activity while those apps are often not actually running or are *allowed to run just a little bit*." It's that last phrase—being allowed to run just a little bit through background tasks—that is our last topic for this chapter. And as described earlier as well, background tasks are related to the lock screen because the apps that are allowed to work on the lock screen must also employ certain background tasks.

Let me reiterate here that we've already seen a number of scenarios in which the user can experience app-related activity without the app having to run. Periodic tile updates, push notifications, scheduled toasts, and even sharing data through the Share contract all provide for activity when an app is suspended or not running. Apps can also configure background data transfers to occur while the app isn't running, as we'll see in Chapter 14, "Networking." And background audio provides for that specific class of apps that need to continue running to maintain VoIP sessions, audio playback, online meetings, and so forth.

What's left in the story are those little pieces of app code that Windows can run in response to specific *triggers*. Triggers in some cases can be further refined with optional *conditions* so that the background task runs only when it really needs to. This is all to minimize the amount of background activity that can drain a device's battery. Some types of triggers, in fact, require that the user has placed the app on the lock screen to specifically limit the number of apps that can respond to those triggers. Furthermore, Windows also limits the amount of CPU time that background tasks can consume:

- **Lock screen background tasks** Two seconds of cumulative CPU time per 15 minutes.
- **Other background tasks** One second of cumulative CPU time every two hours.

Consumption of network bandwidth is also limited on battery power. What that limit is, exactly, I cannot say, because the system analyzes energy usage more so than bytes transferred. A means of estimating the limit can be found on [Supporting your app with background tasks](#). (While we're at it, you might also be interested in [Guidelines and checklists for background tasks](#), the [Introduction to background tasks](#) whitepaper, and [Being productive in the background – background tasks](#) on the Windows Blog.)

"Whoa!" you're probably saying, "Is Windows 8 really that restrictive?"

The short answer is yes, because Windows not only wants to save battery power for the foreground app with which a user is engaged, but also wants to make sure the foreground app delivers the best user experience. This means it isn't having to compete with background apps that would, if allowed, take up as many resources as they possibly can. (Developers of background services will always find a justification for hogging up resources!)

It's likely that you've experienced situations like this directly, where you've started an app but it takes for-EV-er to start up because some other dark and mysterious services is chewing on the hard drive,

pounding the network, flaring up the CPU, and so forth. I, for one, have dug through Task Manager and the Resource Monitor to figure out—and kill off—whatever process is pouring molasses on my system, let come what will. This is the kind of user experience that Windows 8 is trying to avoid.

“OK,” you say (assuming that I’ve actually convinced you to some small degree), “does that mean that there isn’t any way to do some background work like indexing data, creating picture thumbnails, processing video, and so on?”

Actually, there are ways to do this. For one, when an app is in the foreground, it can do however much it wants of all these things because it’s ultimately responsible for its own user experience. (An app written in JavaScript can, for such purposes, employ web workers to move such work off the UI thread, as well as delegate tasks to WinRT components that do their work on other threads and return results asynchronously. We’ll look at this in Chapter 16.)

Second, when a device is on AC power instead of battery, Windows allows apps to run background tasks in response to *maintenance triggers* on 15-minute or longer intervals (whatever is appropriate for the app). These are still limited in the total amount of CPU time they can consume, but tasks that don’t involve UI—which background tasks are not allowed—can burn through a few billion instructions in one or two seconds on a gigahertz CPU!

What we have in this whole story, then, are three distinct classes of background tasks and their associated triggers:

- Tasks for maintenance triggers that run on AC power only
- Tasks for potentially conditioned system triggers that run on AC or battery and don’t require being on the lock screen
- Tasks for those privileged apps that the user had added to the lock screen

We’ll look at each of these in detail in the sections that follow, but first there are a few aspects that are shared by them all: declaring background tasks in the manifest, the general process of building the task with the WinRT API, and conditions.

Background Tasks in the Manifest

All background tasks for an app are declared in the manifest, where each declaration indicates the type of task as well as the code to execute for that task, as shown in Figure 13-18. We’ve seen this section of the manifest before in Chapter 10 where we checked Audio for a background audio app. As for the other options, System Event is used for the first two classes of background tasks in the list above, and Control Channel, Timer, and Push Notification are specifically for tasks reserved for lock screen apps.

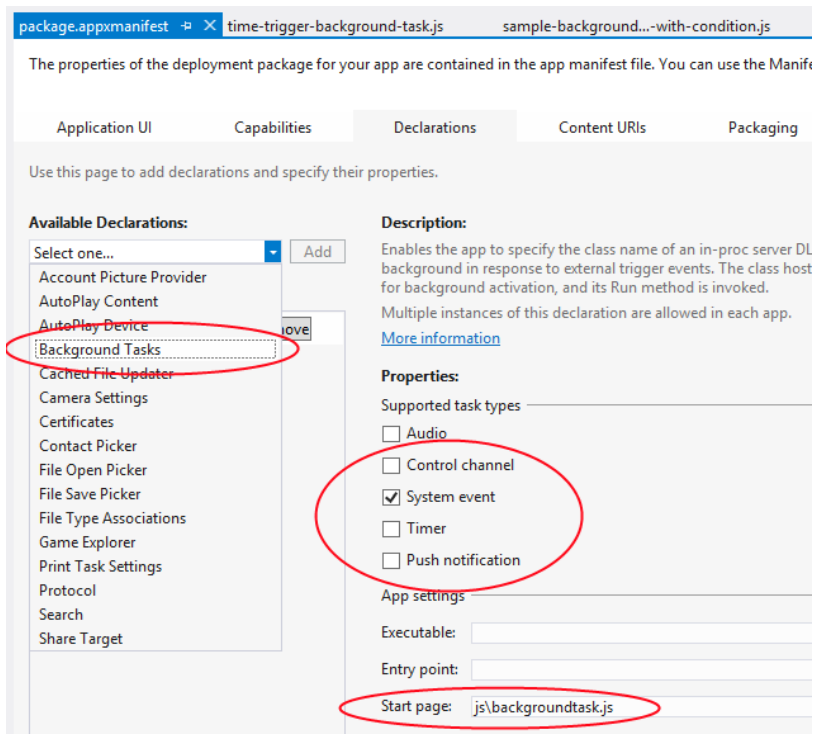


FIGURE 13-18 The manifest editor for declaring background tasks, showing the option for Background Tasks in the drop-down list of Available Declarations (left), the background task types (center), and the Start Page field to indicate the JavaScript code to run for the task (bottom). Background tasks can also be written in other languages, in which case the Executable and Entry Point fields are used.

In all cases, the Start Page field is where you indicate the JavaScript file to execute for the task, but do note that because background tasks execute independently of the app itself, sharing state only through app data, you can really choose whatever language you want. Given the quotas on CPU time, writing a background task in a language like C++ or C# will allow you to do some tasks more efficiently, in which case you'll use the Entry Point field (if the task is in a DLL in the package) and perhaps the Executable field (if the task is another EXE in the package) to identify the code module and specific function to call.

It's also good to note that even though the Start Page field for JavaScript suggests a *page*, it's really just code that you use for a background task—you'll get an error if you try to specify an HTML file here. (To be more specific, a background task in JavaScript is a web worker, plain and simple.) Such is why you can't do UI from a background task: you can't have Windows load any HTML or CSS, just JavaScript! Thus, issuing tile updates, badge updates, and toasts is as much UI work as a task is allowed. For anything else, the background task must write values to app data that the main app can pick up within its handlers for background task events, as we'll see shortly.

You might also notice that the System Event option in the manifest editor doesn't offer another field in which you indicate the task's specific trigger—this is done in code when we build the task, as we'll see next.

Building and Registering Background Task

The declaration of a background task in the manifest is only that—a declaration that tells the system that the app *intends* to use a background task. The app must still register the background task from code in order for it to execute at all, which is accomplished using the [Windows.ApplicationModel.Background.BackgroundTaskBuilder](#) (whose parent namespace contains everything we'll be referring to in the context of background tasks). Simply said, you create an instance of the builder, set its `name` and `taskEntryPoint` properties, call its `setTrigger` and `addCondition` methods to specify exactly when the task should run, and then call `register`.

Generic code for this is found in the [Background task sample](#) within `js/global.js`. This module declares a global object `BackgroundTaskSample` that contains a number of properties and methods. The one that concerns us here is a method called `registerBackgroundTask` that registers a given entry point (the name of a JavaScript file or the name of a class in C#, Visual Basic, or C++), with a given name, and applying some trigger and condition:

```
var BackgroundTaskSample = {
    // Properties with names and entry points of the sample's tasks are omitted

    //
    // Register a background task with the specified taskEntryPoint, taskName, trigger,
    // and condition (optional).
    //
    "registerBackgroundTask": function (taskEntryPoint, taskName, trigger, condition) {
        var builder = new Windows.ApplicationModel.Background.BackgroundTaskBuilder();

        builder.name = taskName;
        builder.taskEntryPoint = taskEntryPoint;
        builder.setTrigger(trigger);

        if (condition !== null) {
            builder.addCondition(condition);
        }

        var task = builder.register();
        BackgroundTaskSample.attachProgressAndCompletedHandlers(task);

        // [Sample-specific code omitted]

        // Remove previous completion status from local settings.
        var settings = Windows.Storage.ApplicationData.current.localSettings;
        settings.values.remove(taskName);
    },
};
```

In this code, the `BackgroundTaskBuilder.register` method returns a [BackgroundTask-Registration](#) object through which you manage a registered task. A registered task will have a `name` property and a system-assigned `taskId` property, the latter of which you can use to store app data that's unique to the task. It also has an `unregister` method, which you would call for obvious purpose, and two events: `completed` and `progress`. Handlers for those events are assigned in the usual manner with `addEventListener`, as seen within the `BackgroundTaskSample.attach-ProgressAndCompletedHandlers` function in the sample:

```
"attachProgressAndCompletedHandlers": function (task) {
    task.addEventListener("progress",
        new BackgroundTaskSample.progressHandler(task).onProgress);
    task.addEventListener("completed",
        new BackgroundTaskSample.completeHandler(task).onCompleted);
},
```

One of the key uses of these handlers is to perform UI update tasks in response to data left behind by the background tasks. Those tasks themselves cannot work with UI, but they can save data to the app data areas that they share with the main app. The `completed` and `progress` events, then, are how the main app—the one that can work with the UI—can pick up those events from the background task to read values from app data and do the necessary updates. The [Background task sample](#) does this in each of its scenarios.

There is also one static property, `BackgroundTaskRegistration.allTasks`, an `IMapView` through which you can retrieve your registered tasks and obtain the specific `BackgroundTask-Registration` object for each.

It's very important to note that Windows allows you to register the same background task twice and will assign unique `taskId` values to both, so be careful to avoid duplicate tasks. Notice too how the `registerBackgroundTask` code above makes a little use of the local settings container in app data. This is how the app communicates with each of its independent tasks because app data is the only means for such data exchange.

With this structure, the question now becomes: what do we provide for the triggers and the conditions? The answer is what differentiates the various kinds of background tasks.

Conditions

The specific conditions you can specify through `BackgroundTaskBuilder.addCondition` are instances of the `SystemCondition` class. A background task registered with one or more conditions—each call to `addCondition` is cumulative—will run only if the conditions are met. Each instance can be one of the following types as defined in the `SystemConditionType` enumeration: `internetAvailable`, `internetNotAvailable`, `sessionConnected` (the user is logged in), `sessionDisconnected` (the user is logged out), `userPresent` (the user has been recently active), and `userNotPresent` (the user has not been active for a time).

Clearly, each pair of these conditions is mutually exclusive: if you register a task with `internetAvailable` and `internetNotAvailable`, Windows will recognize that you never really wanted to run the task in the first place, so it will let it sit on the roadside, forever undisturbed! Otherwise, you can use these to make sure that your task is run only when needed. If you want to execute a background task that renews push notification channels, for example, there's no point in trying if there's no connectivity. (We'll see an example in the next section, "Tasks for Maintenance Triggers.") On the other hand, if you have a background task that you want to make sure never interferes with the overall user experience, you can add the `userNotPresent` condition.

Note that because the `sessionDisconnected` condition implies that the user has logged out, it's useful only for background tasks that require the lock screen.

Tasks for Maintenance Triggers

Background tasks that use a maintenance trigger are probably the most generic kind of task—they're really just any kind of code you want to run every now and then when the system is on AC power.⁷⁰ Such tasks are best for "checking up on something" or other activity that you want to run periodically but don't really care when. As such, maintenance triggers *aren't* appropriate for something like synchronizing data with a server because that should happen in a more timely manner and is best done with the background transfer API that we'll see in Chapter 14.

A maintenance trigger—what you pass to `BackgroundTaskBuilder.setTrigger`—is an instance of the `MaintenanceTrigger` class. When creating the instance, you provide two parameters. The first is basically the refresh period you need (the `freshnessTime` property, in minutes), and you should always use the longest period that's reasonable for your scenario; the system will always wait at least this long before first running the task. The second parameter is a flag that indicates if the task needs to be run only once (the `oneShot` property).

Scenario 2 of the [Push and periodic notifications client-side sample](#) demonstrates using a maintenance trigger to periodically refresh its WNS channels, as described earlier in "Requesting and Caching a Channel URI." The code here is condensed from `js/scenario2.js`, some of which is in an internal function called `registerTask`:

```
var background = Windows.ApplicationModel.Background;
var pushNotificationsTaskName = "UpdateChannels";
var maintenanceInterval = 10 * 24 * 60; // 10 days

var taskBuilder = new background.BackgroundTaskBuilder();
var trigger = new background.MaintenanceTrigger(maintenanceInterval, false);
taskBuilder.setTrigger(trigger);
taskBuilder.taskEntryPoint = "js\\backgroundTask.js";
taskBuilder.name = pushNotificationsTaskName;
```

⁷⁰ This is about the only API for which a clear distinction is made for battery vs. AC power; WinRT does not offer a specific API to detect the power source. What this means is that apps should design for running on the battery but assign AC-only tasks to maintenance triggers. It's another way that Windows 8 apps let the system manage power on a systemwide basis and not on a per-app basis.


```

var internetCondition = new
    background.SystemCondition(background.SystemConditionType.internetAvailable);
taskBuilder.addCondition(internetCondition);

taskBuilder.register();

```

Because the expiration period for channel URIs is 30 days, the sample creates a trigger on a recurring 10-day interval (10 days * 24 hours/day * 60 minutes/hour). It also wisely adds the `internetAvailable` condition because it's again pointless to attempt to renew channel URIs when there's no connectivity.

The task itself can be found in the `js/backgroundTask.js` file of the sample, as indicated in the `taskEntryPoint` property:

```

(function () {
    // Import the Notifier helper object
    importScripts("//Microsoft.WinJS.1.0/js/base.js");
    importScripts("notifications.js");

    var closeFunction = function () {
        close();
    };

    var notifier = new SampleNotifications.Notifier();
    notifier.renewAllAsync().done(closeFunction, closeFunction);
})();

```

This task code pulls in a couple of other script files using `importScripts`, the second of which, `notifications.js`, is the sample's set of helper functions for notifications where `renewAllAsync` refreshes the app's list of previously saved channel URIs.

Important You'll also notice that the completed and error handlers given to the promise from `renewAllAsync` both go to `closeFunction`, which makes this mysterious call to `close`. What `close` is this? Well, it's not `window.close` (as you might guess) but rather `WorkerGlobalScope.-close`. Background tasks in an app written in JavaScript run within the scope of a web worker, so the global scope within the code is `WorkerGlobalScope` rather than `window`. Calling this makes sure the independently running background task is shut down and guarantees that the resources that were allocated for the task are properly released.

Sidebar: The Task Instance and Background Task Deferrals

Within a JavaScript background task, the `Windows.UI.WebUI.WebUIBackgroundTask-Instance.current` contains a `WebUIBackgroundTaskInstanceRuntimeClass` object that provides additional details about the running task: its `instanceId`, its associated `Background-TaskRegistration` object in the `task` property, a `progress` property in which the task can store a percentage value, a `succeeded` flag to indicate that the task has completed, a `suspended` count (when the task is suspended due to the resource quota being exceeded), and a `canceled` event that informs the task that the app as a whole has been terminated.

This object also provides a [getDeferral](#) method that, once again, returns a deferral object whose [completed](#) method you call when the task is complete. As always, you employ the deferral if you need to perform asynchronous operations within the background task. Just be sure to always call [close](#) when everything is finished.

Tasks for System Triggers (Non-Lock Screen)

The next class of background tasks contains those tied to a variety of system triggers, specifically instances of the [SystemTrigger](#) class. You again create the trigger object with [new](#) and pass two parameters: a [SystemTriggerType](#) value (available afterwards as the [triggerType](#) property) and a [oneShot](#) Boolean flag. The triggers that operate independently of the lock screen are described in the following table:

SystemTriggerType ⁷¹	When Triggered and Usage Scenarios
internetAvailable	Internet becomes available. This is typically used for apps that need to start a synchronization process when connectivity is restored from an offline state. Note that this <i>trigger</i> is different from the <i>condition</i> with the same name.
lockScreenApplicationAdded	User has added the app to the lock screen, signaling that lock screen–dependent background tasks will now be executed.
lockScreenApplicationRemoved	User has removed the app from the lock screen, signaling that lock screen–dependent background tasks will no longer run.
networkStateChange	Change in network (cost, connectivity, etc.). A running app can detect the same event through Windows.Networking.Connectivity.NetworkInformation.onnetworkstatuschanged , and this provides a means for apps to execute a small piece of code when the app is suspended or otherwise not running. This trigger combined with the internetAvailable or internetNotAvailable <i>conditions</i> offers the app full awareness of connectivity states. We'll talk more of connectivity in Chapter 14; for now you can refer to the Network status background sample .
onlineIdConnectedStateChange	The user's Microsoft account has changed. This is a relatively rare occurrence, but the trigger is essential for any app that caches any part of the Microsoft account for its own user identity.
servicingComplete	App has been updated from the Windows Store. This trigger could be used, for example, to migrate app data from one version to another as soon as the update happens. Refer to Chapter 8, "State, Settings, Files, and Documents," for more on versioning app data.
timeZoneChange	A time zone or daylight savings time change has occurred. An app might refresh its locale settings at such a time, as well as adjust any internal timekeeping. This can be important to adjust scheduled notifications.

⁷¹ There is an additional trigger called [SmsReceived](#) that is only for apps provided by mobile operators.

The [Background task sample](#) provides a few examples of these triggers. In Scenario 1, `js/sample-background-task-with-condition.js`, we can see the use of `timeZoneChange` along with the `userPresent` condition (where `BackgroundTaskSample` is again a helper object in `global.js`):

```
BackgroundTaskSample.registerBackgroundTask(BackgroundTaskSample.sampleBackgroundTaskEntryPoint,  
    BackgroundTaskSample.sampleBackgroundTaskWithConditionName,  
    new Windows.ApplicationModel.Background.SystemTrigger(  
        Windows.ApplicationModel.Background.SystemTriggerType.timeZoneChange, false),  
    new Windows.ApplicationModel.Background.SystemCondition(  
        Windows.ApplicationModel.Background.SystemConditionType.userPresent));
```

This is clearly a case where I'd use another variable to not type the `Windows.ApplicationModel.Background` namespace out every time, but at least you can't make a mistake in reading this code! In any case, the same sample, in Scenario 4 and `js/global.js`, also shows use of the `servicing-Complete` trigger within a helper function `registerServicingCompleteTask`, which also checks if the task is already registered:

```
"registerServicingCompleteTask": function () {  
    // Check whether the servicing-complete background task is already registered.  
    var iter =  
        Windows.ApplicationModel.Background.BackgroundTaskRegistration.allTasks.first();  
    var hascur = iter.hasCurrent;  
    while (hascur) {  
        var cur = iter.current.value;  
        if (cur.name === BackgroundTaskSample.servicingCompleteTaskName) {  
            BackgroundTaskSample.updateBackgroundTaskStatus(  
                BackgroundTaskSample.servicingCompleteTaskName, true);  
            return;  
        }  
        hascur = iter.moveToNext();  
    }  
}  
  
// The servicing-complete background task is not already registered.  
BackgroundTaskSample.registerBackgroundTask(  
    BackgroundTaskSample.servicingCompleteTaskEntryPoint,  
    BackgroundTaskSample.servicingCompleteTaskName,  
    new Windows.ApplicationModel.Background.SystemTrigger(  
        Windows.ApplicationModel.Background.SystemTriggerType.servicingComplete, false),  
    null);  
},
```

In the sample, the tasks associated with these triggers are implemented in C#, within a WinRT component found in the `Tasks` project of the sample's solution. I won't show the code here because we'll be looking at the general structure of WinRT components in Chapter 16. What it does demonstrate, though, is that you can use a mixed-language approach for background tasks. In these cases, the Entry Point field for the tasks in the manifest point to the C# class/method that implements the background task, such as `Tasks.ServicingComplete`. If you go to the [Background task sample](#) page, you can also download the C# and C++ versions of the sample to see even more structural variants.

Lock Screen–Dependent Tasks and Triggers

The last group of background tasks are those that require the app is also added to the lock screen. For this there are four applicable `SystemTrigger` options from `SystemTriggerType`, along with the three other distinct types that are represented in the manifest editor: `TimeTrigger`, and `PushNotificationTrigger`, and `Windows.Networking.Sockets.ControlChannelTrigger`. These are described in the following table along with pointers to available samples that demonstrate their usage:

<code>SystemTriggerType</code>	When Triggered, Scenarios, and Samples
<code>SystemTriggerType.controlChannelReset</code>	See <code>ControlChannelTrigger</code> below.
<code>SystemTriggerType.sessionConnected</code>	User has logged in from the lock screen.
<code>SystemTriggerType.userAway</code>	Device has become inactive (e.g., blank screen) due to user inactivity.
<code>SystemTriggerType.userPresent</code>	User becomes present from an inactive state.
<code>Windows.Networking.Sockets.ControlChannelTrigger</code>	<p>Real-time notifications have been received through the <i>control channel</i>—that is, a networking channel typically using sockets or another networking transport, if it's not possible to use WNS and raw notifications for the same purpose. This trigger is used for real-time communication apps such as VoIP, IM, and Mail so that they are “always reachable” if the user places them on the lock screen. We'll look at the networking transports themselves in Chapter 14, but the whole of this subject is beyond the scope of this book. Please refer to the How to set background connectivity options in the documentation along with the following samples, all of which employ C# or C++ and are not available in JavaScript:</p> <ul style="list-style-type: none"> • ControlChannelTrigger StreamSocket sample • ControlChannelTrigger XmlHttpRequest sample • ControlChannelTrigger StreamWebSocket sample • ControlChannelTrigger HTTP client sample <p>The <code>SystemTriggerType.controlChannelReset</code> is used to manage a background task for changes in the control channel rather than events on the channel itself.</p>
<code>TimeTrigger</code>	A period of time as configured in the trigger has elapsed. Scenario 5 of the Background task sample demonstrates this (see below).
<code>PushNotificationTrigger</code>	A raw push notification for the app has arrived from WNS; because Windows cannot handle a raw notification directly, this kind of background task is necessary to take action on a raw notification when the app isn't running. A running app, on the other hand, can use the <code>pushnotificationreceived</code> event, as described earlier in “Receiving Notifications (App).” For an example, refer to the Raw notifications sample .

Note Working with the lock screen is not supported in the Visual Studio simulator. To debug lock screen apps and background tasks, you'll need to use the Local Machine or Remote Machine debugging options.

Background tasks for these triggers are created and registered as we've already seen. A [TimeTrigger](#), for example, is created with its [freshnessTime](#) interval (in minutes) and a [oneShot](#) flag, as seen in Scenario 5 of the Background tasks sample ([js/time-trigger-background-task.js](#)):

```
BackgroundTaskSample.registerBackgroundTask(  
    BackgroundTaskSample.sampleBackgroundTaskEntryPoint,  
    BackgroundTaskSample.timeTriggerTaskName,  
    new Windows.ApplicationModel.Background.TimeTrigger(15, false), null);
```

A [TimeTrigger](#) is also used in Scenario 3 of the [Geolocation sample](#) to allow the user to add a navigation app to the lock screen for more continuous tracking. Generally speaking, though, a navigation app isn't particularly useful on the lock screen in the first place, since it wouldn't be able to show a map! Better, then, to again use the [Windows.System.Display.DisplayRequest](#) API to prevent going to the lock screen at all.

Creating a [PushNotificationTrigger](#) is even simpler as there are no parameters. This can be seen in the [Raw notifications sample](#), Scenario 1 ([js/scenario1.js](#)):

```
function registerBackgroundTask() {  
    // Register the background task for raw notifications  
    var taskBuilder = new background.BackgroundTaskBuilder();  
    var trigger = new background.PushNotificationTrigger();  
    taskBuilder.setTrigger(trigger);  
    taskBuilder.taskEntryPoint = sampleTaskEntryPoint;  
    taskBuilder.name = sampleTaskName;  
  
    var task = taskBuilder.register();  
    task.addEventListener("completed", backgroundTaskComplete);  
}
```

Although the call to [BackgroundTaskBuilder.register](#) might succeed, the task itself will not execute until the user adds the app to the lock screen, as we saw earlier in Figure 13-10. This latter action is never under the app's control—all it can do is make sure it's *available* for the user to select on that section of PC Settings, which is what asking for access is all about.

The request is made through the method [Windows.ApplicationModel.Background.BackgroundExecutionManager.requestAccessAsync](#); this call should be made prior to registering the background task (see Scenario 5 of the Background task sample again):

```
Windows.ApplicationModel.Background.BackgroundExecutionManager.requestAccessAsync();
```

When this is called the first time in an app, it will generate a user consent prompt, as shown in Figure 13-19. If the user chooses Allow, the app will appear in PC Settings as an option for the lock screen, otherwise it won't. As with other permissions, users can change their minds later on through the Permissions settings, as shown in Figure 13-20.

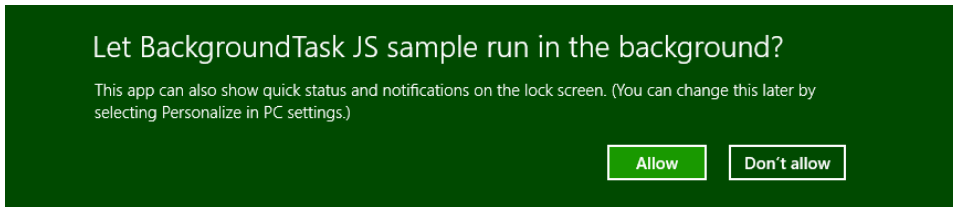


FIGURE 13-19 The user consent prompt when an app requests lock screen access.

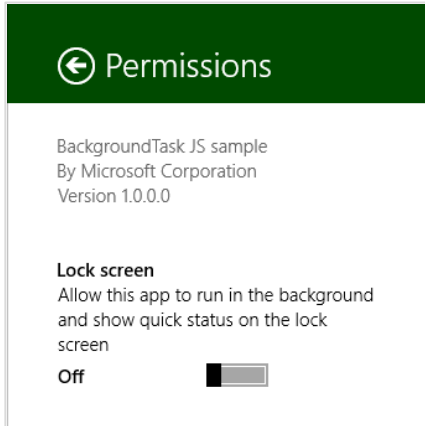


FIGURE 13-20 The lock screen option on the Permissions settings panel for apps that request access.

For a complete demonstration, refer to the [Lock screen apps sample](#). In Scenario 1 it shows how to again request access to the lock screen and check the result, which is a value from the [Background-AccessStatus](#) enumeration. It also shows querying for and removing that access with the [getAccessStatus](#) and [removeAccess](#) methods of [BackgroundExecutionManager](#).

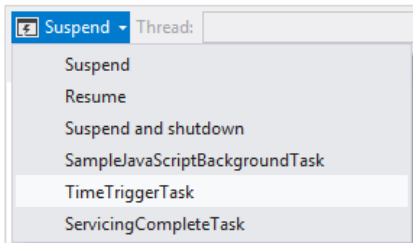
Scenario 2 then demonstrates sending badge updates to the lock screen, along with a text tile update if the app happens to be the single one selected for that privilege. There is nothing particular in this process where the lock screen is concerned, however: such updates happen exactly as they do for the primary app tile. It's just that those updates are also reflected on the lock screen. In the case of badges, the badge glyph will appear along with the app's Badge Logo in the Application UI > Notifications section of the manifest. (Refer back to Figure 13-10.) To repeat from earlier, this graphic must have white or transparent pixels, and the three scale sizes are 24x24 (100%), 33x33 (140%), and 43x43 (180%).

Scenario 3, finally, demonstrates that secondary tiles can be added to the lock screen as well, irrespective of the app tile. To make a secondary tile available for the lock screen, assuming that the app has requested lock screen access already, you need to set those two properties of [Windows.UI.StartScreen.SecondaryTile](#) that we mentioned long ago: [lockScreenBadgeLogo](#) and [lockScreenDisplayBadgeAndTileText](#). If the secondary tile is on the Start screen, these properties will also make it available on the PC Settings page for the lock screen.

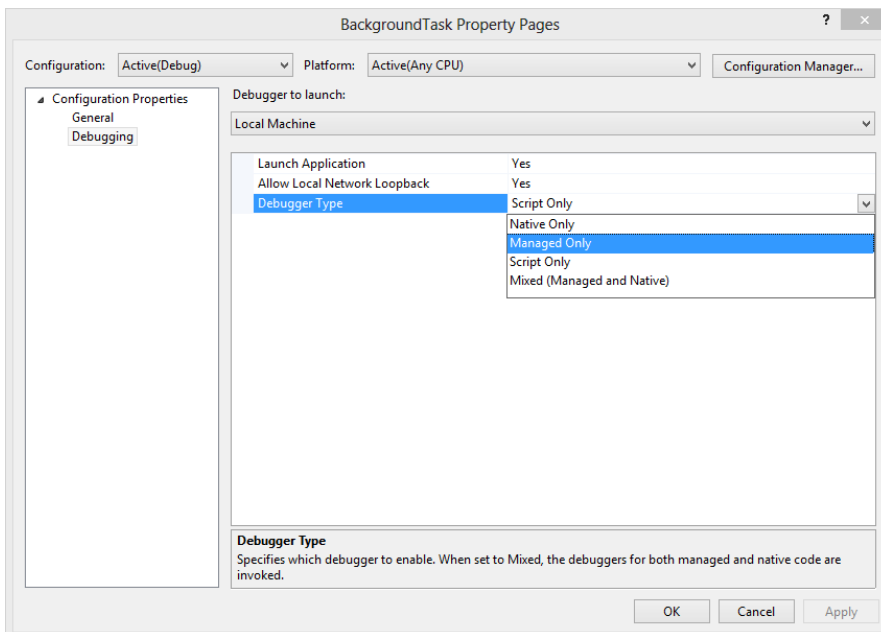
Debugging Background Tasks

By this time you might have run the `TimeTrigger` background task in Scenario 5 of the Background tasks sample, and unless it's been more than 15 minutes since that time (maybe up to 30 minutes if you just missed the 15-minute window when timers are coalesced), you might still be waiting for that period to elapse. Is this, then, your destiny for debugging background tasks: to wait, wait, wait?

Fortunately, the answer is no, no, and maybe! That is, Visual Studio's debugger is mostly aware of registered background tasks and provides a list of them on the Suspend drop-down menu on the toolbar:



Selecting one of these will immediately trigger the background task, so you won't have to wait or otherwise attempt to activate the trigger for real. One caveat is that if the trigger had `oneShot` set to `true` and already fired, it won't fire again. A second caveat with this is that if you're running a JavaScript app with background tasks written in other languages, you'll need to change the debugger type for the main app project from Script Only to one that supports Managed or Native as shown here, otherwise you can't set breakpoints in those other modules:



A third caveat is that background tasks using `ControlChannelTrigger`, `PushNotificationTrigger`, and the `SystemTriggerType.SmsReceived` will not appear on the Visual Studio drop-down menu.

So you might need to rely on the tried-and-true methods of outputting diagnostic information to figure out what's going on with your task and checking events in the Event Viewer for activation failures. More on these methods can be found on [How to debug a background task](#).

Finally, note one more time that background tasks are not supported in the Visual Studio simulator, as is also true of live tiles, notifications, and much else that we've covered in this chapter. You'll need to use the Local Machine or Remote Machine options instead.

What We've Just Learned (Whew!)

- Tile updates, on both the app's primary and secondary tile, along with badges, toast notifications, and background tasks—from which an app can issue tile updates and notifications—are how an app contributes to the overall aliveness of the system even while the app isn't running.
- Tile updates and notifications can be sent from a running app, of course, but there are other methods to deliver those updates when the app isn't running. Updates can be scheduled to appear at a later time, and the app can configure the system to periodically ask a service for tile/badge updates. Apps can also configure push notifications that are raised from a web service and sent to clients through the Windows Push Notification Service (WNS).
- Tile updates are issued using an XML payload based on predefined templates. Typical payloads include both square and wide tile updates so that the user can choose how the tile is displayed. The XML can reference images from both local and remote sources, so long as the images are 1024x1024 pixels or smaller and less than 200KB in size.
- A tile can cycle through up to five updates at any given time, each of which can be replaced separately.
- Apps that have specific content that is interesting to bookmark as secondary tiles to the Start screen provide Pin and Unpin commands to the user for that purpose. Secondary tiles, which launch the app with specific startup arguments, can also receive live tile updates.
- Badges are small glyphs or numbers that can appear on any given tile. Badge updates are sent through the same mechanism as tile updates, but they operate independently.
- Toasts are popup notifications that appear for a time to alert the user of new information, reminders, and so on. They can be configured to play sounds, set to recur on a given interval, and scheduled to appear in the future. Like secondary tiles, activating a toast launches the app with specific startup arguments.

- Periodic updates for tiles and badges means providing Windows the URIs of web services to call at selected intervals between 30 minutes and 24 hours. Periodic updates are the easiest and lower-cost means to update a tile from a running web service.
- Push notifications for tiles, badges, toast notifications, and raw notifications (whatever data an app wants to manage) can be used for higher-frequency, user-specific updates. This involves creating web services that communicate with WNS to issue those notifications to specific channel URIs, a process that is much more involved and expensive than periodic updates.
- Background tasks are small pieces of code that an app configures to run when certain triggers occur, such as changes in connectivity, timers, receipt of push notifications, and app updates. Apps should never depend on background tasks, however, because they are always under the user's control.
- Background task triggers can be refined through specific conditions to avoid running tasks when it's not necessary (such as when there is no connectivity).
- Some triggers require that the app has also been added to the lock screen. Such apps must first request access, which is subject to user consent, and the user must specifically add the app through PC Settings. Given that privilege, apps can issue badge updates and tile text to the lock screen.
- Through maintenance triggers, apps can also set up tasks to run periodically when a device is on AC power.

Chapter 14

Networking

Having moved twice with my family in the last year, I've been struck by the similarities between that process and how data moves around on our networks. Packing a truck for the first move—from Oregon to California—was certainly an exercise in compression! At times I didn't think we'd manage to get everything into our van and the 20-foot truck we'd rented, but somehow it all fit with just inches to spare. Then we had the long drive south before decompressing the data, so to speak, into a house where we lived for a time while the builders were finishing up our permanent home.

Moving into the new house was a different experience, mainly because it was only about 200 meters away and we were able to move bits and pieces at different times. When the final inspection was signed off and we could (legally) move all the furniture and boxes, the task was accomplished with the help of many friends and many random vehicles, with very little compression of our stuff along the way!

These two moves illustrate, in some loose way, the character of different networking transports, such as datagram sockets and stream sockets, one of the topics we'll visit here. In the first case we see a large (compressed) data packet moving *in toto* from one endpoint to another. In the second we instead have more of a data *streaming* experience, with some compression to optimize each trip between the endpoints but very different in nature from the first experience.

Along these same lines, consider how you might hire movers to do all this work—you show them your stuff, give them the destination address, write them a large check, and magically all that stuff shows up in another location. Such a process is reminiscent of the background transfer APIs in WinRT that will be one of the first subjects of this chapter. In moving one's household, there is also the concept of renting and packing "pods," in which case you do the packing yourself but the transportation (and possibly storage) is handled separately. But whether this has anything whatsoever to do with writing a Windows 8 app, I'm not so sure!

Lest I dig too deep a hole with such analogies, let's just say that networking is a rich and varied topic, all of which is based on the need to get data from one place to another in a variety of ways. The goal of this chapter, then, is to provide at least an overview of the networking capabilities in Windows 8. Topics will range from XMLHttpRequests, background transfers, authentication and credentials, and syndication, to connectivity and network information, offline functionality, and sockets. We'll focus most on areas of common concern for most apps, touching briefly on other areas that are more specific to certain scenarios and for which there are many additional resources to draw upon. One such resource is [Developing connected apps](#), which serves as a good overview of networking in general.

In any case, we'll begin here with the subject of connectivity, because without it, there isn't much to speak about with networking at all.

Network Information and Connectivity

In the previous chapter, I mentioned in a footnote how at the very time I was writing on the subject of live tiles and all the connections that Windows 8 apps can have to the Internet, my home and many thousands of others in Northern California were completely disconnected due to a fiber optic breakdown. The outage lasted for what seemed like an eternity by present standards—36 hours! Although I wasn't personally at a loss for how to keep myself busy, there was a time when I opened one of my laptops, found that our service was still down, and wondered for a moment just what the computer was really good for without connectivity! Clearly I've grown, as I suspect you have too, to take constant connectivity completely for granted.

As developers of great apps, however, we cannot afford to be so complacent. It's always important to handle errors when trying to make connections and draw from online resources, because any number of problems can arise within the span of a single operation. But it goes much deeper than that. It's our job to make our apps as useful as they can be when connectivity is lost, perhaps just because our customers got on an airplane and switched on airplane mode. That is, don't give customers a reason to wonder about the usefulness of their device in such situations! A great app will prove its worth through a great user experience even if it lacks connectivity.

Connectivity can also vary throughout an app session, where an app can often be suspended and resumed, or suspended for a long time. With mobile devices especially, one might switch between any number of networks without necessarily knowing about it. Windows 8, in fact, tries to make the transition between networks as transparent as possible, except where it's important to inform the user that there may be costs associated with the current network. It's required by Window Store policy, in fact, for apps to be aware of data transfer costs on metered networks and to prevent "bill shock" from not-always-generous mobile broadband providers. Just as there are certain things an app can't always do when the device is offline, the characteristics of the current network might also cause it to defer or avoid certain operations as well.

Let's now see how to retrieve and work with connectivity details, starting with the different types of networks represented in the manifest, followed by obtaining network information, dealing with metered networks, and providing for an offline experience.

Network Types in the Manifest

Nearly every sample we've worked with so far in this book has had the *Internet (Client)* capability declared in its manifest, thanks to Visual Studio turning that on by default. Once before I mentioned how this wasn't always the case: early app builders within Microsoft would occasionally scratch their heads wondering just why something really obvious—like making a simple `XmlHttpRequest` to a blog—failed outright. Without this capability, there just isn't any Internet!

Still, *Internet (Client)* isn't the only player in the capabilities game. Some networking apps will also want to act as a server to receive incoming traffic from the Internet, and not just make requests to other

servers. In those cases—such as file sharing, media servers, VoIP, chat, multiplayer/multicast games, and other bi-directional scenarios involving incoming network traffic, as with sockets—the app must declare the *Internet (Client & Server)* capability, as shown in Figure 14-1. This lets such traffic through the inbound firewall, though critical ports are always blocked.

There is also network traffic that occurs on a private network, as in a home or business, where the Internet isn't involved at all. For this there is also the *Private Networks (Client & Server)* capability, also shown in Figure 14-1, which is good for file or media sharing, line-of-business apps, HTTP client apps, multiplayer games on a LAN, and so on. What makes any given IP address part of this private network depends on many factors, all of which are described on [How to configure network isolation capabilities](#). For example, IPv4 addresses in the ranges of 10.0.0.0–10.255.255.255, 172.16.0.0–172.31.255.255, and 192.168.0.0–192.168.255.255 are considered private. Users can flag a network as trusted, and the presence of a domain controller makes the network private as well. Whatever the case, if a device's network endpoint falls into this category, the behavior of apps on that device is governed by this capability rather than those related to the Internet.

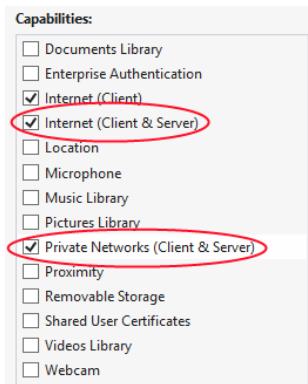


FIGURE 14-1 Additional network capabilities in the manifest.

Sidebar: Localhost Loopback

Regardless of the capabilities declared in the manifest, local loopback—that is, using `http://localhost` URIs—is blocked for Windows Store apps. An exception is made for machines on which a developer license has been installed, as described in Chapter 13, “Tiles, Notifications, the Lock Screen, and Background Tasks,” in the section “Using the Localhost.” This exception exists only to simplify debugging apps and services together, because they can all be running on a single machine during development.

Network Information (the Network Object Roster)

Regardless of the network involved, everything you want to know about that network is available through the [Windows.Networking.Connectivity.NetworkInformation](#) object. Besides a single

`networkstatuschanged` event that we'll discuss in "Connectivity Events" a little later, the interface of this object is made up of methods to retrieve more specific details in other objects.

Fulfilling my earlier promise to just touch on some specifics, below is the roster of the methods in `NetworkInformation` and the contents of the objects obtained through them. You can exercise the most common of these APIs through the indicated scenarios of the [Network information sample](#):

- `getHostNames` Returns a vector of [Windows.Networking.HostName](#) objects, one for each connection, that provides various name strings (`displayName`, `canonicalName`, and `rawName`), the name's `type` (from [HostNameType](#), with values of `domainName`, `ipv4`, `ipv6`, and `bluetooth`), and an `ipinformation` property (of type [IPInformation](#)) containing `prefixLength` and `networkAdapter` properties for IPV4 and IPV6 hosts. (The latter is a [NetworkAdapter](#) object with various low-level details.) The `HostName` class is used in various networking APIs to identify a server or some other endpoint.
- `getConnectionProfiles` (Scenario 3) Returns a vector of [ConnectionProfile](#) objects, one for each connection, among which will be the active Internet connection as returned by `getInternetConnectionProfile`. Also included are any wireless connections you've made in the past for which you indicated Connect Automatically. (In this way the sample will show you some details of where you've been recently!) See the next section for more on [ConnectionProfile](#).
- `getInternetConnectionProfile` (Scenario 1) Returns a single [ConnectionProfile](#) object for the currently active Internet connection. If there is more than one connection, this method returns the profile of the preferred one that is most likely to be used for Internet traffic.
- `getLanIdentifiers` (Scenario 4) Returns a vector of [LanIdentifier](#) objects, each of which contains an `infrastructureId` ([LanIdentifierData](#) containing a `type` and `value`), a `networkAdapterId` (a GUID), and a `portId` ([LanIdentifierData](#)).
- `getProxyConfigurationAsync` Returns a [ProxyConfiguration](#) object for a given URI and the current user. The properties of this object are `canConnectDirectly` (a Boolean) and `proxyUris` (a vector of [Windows.Foundation.Uri](#) objects for the configuration).
- `getSortedEndpointPairs` Sorts an array of [EndpointPair](#) objects according to [HostNameSortOptions](#). An `EndpointPair` contains a host and service name for local and remote endpoints, typically obtained when you set up specific connections like sockets. The two sort options are `none` and `optimizeForLongConnections`, which vary connection behaviors based on whether the app is making short or long duration connection. See the documentation for [EndpointPair](#) and [HostNameSortOptions](#) for more details.

The ConnectionProfile Object

Of all the information available through the [NetworkInformation](#) object, the most important for apps is found in [ConnectionProfile](#), most frequently that returned by [getInternetConnection-Profile](#) because that's the one through which an app's Internet traffic will flow. The profile is what contains all the information you need to make decisions about how you're using the network, especially for cost awareness. It's also what you'll typically check when there's a change in network status. Scenarios 1 and 3 of the Network information sample retrieve and display most of these details.

Each profile has a [profileName](#) property (a string), such as "Ethernet" or the SSID of your wireless access point, plus a [getNetworkNames](#) method that returns a vector of friendly names for the endpoint. The [networkAdapter](#) property contains a [NetworkAdapter](#) object for low-level details, should you want them, and the [networkSecuritySettings](#) property contains a [NetworkSecurity-Settings](#) object properties describing authentication and encryption types.

More generally interesting is the [getNetworkConnectivityLevel](#), which returns a value from the [NetworkConnectivityLevel](#) enumeration: [none](#) (no connectivity), [localAccess](#) (the level you hate to see when you're trying to get a good connection!), [constrainedInternetAccess](#) (captive portal connectivity, typically requiring further credentials as is often encountered in hotels, airports, etc.), and [internetAccess](#) (the state you're almost always trying to achieve). The connectivity level is often a factor in your app logic and something you typically watch with network status changes.

To track the inbound and outbound traffic on a connection, the [getLocalUsage](#) method returns a [DataUsage](#) object that contains [bytesReceived](#) and [bytesSent](#), either for the lifetime of the connection or for a specific time period. Similarly, the [getConnectionCost](#) and [getDataPlanStatus](#) provide the information an app needs to be aware of how much network traffic is happening and how much it might cost the user. We'll come back to this in "Cost Awareness" shortly, including how to see per-app usage in Task Manager.

Connectivity Events

It is very common for a running app to want to know when connectivity changes. This way it can take appropriate steps to disable or enable certain functionality, alert the user, synchronize data after being offline, and so on. For this, apps need only watch the [NetworkInformation.onnetworkstatus-changed](#) event, which is fired whenever there's a significant change within the hierarchy of objects we've just seen (and be mindful that this event comes from a WinRT object). For example, the event will be fired if the connectivity level of a profile changes. It will also be fired if the Internet profile itself changes, as when a device roams between different networks, or when a metered data plan is approaching or has exceeded its limit, at which point the user will start worrying about every megabyte of traffic. In short, you'll generally want to listen for this event to refresh any internal state of your app that's dependent on network characteristics and set whatever flags you use to configure the app's networking behavior. This is especially important for transitioning between online and offline and between unlimited and metered networks; Windows, for its part, also watches this event to adjust its own behavior, as with the Background Transfer APIs.

Note Windows Store apps written in JavaScript can also use the basic `window.navigator.ononline` and `window.navigator.onoffline` events to track connectivity. The `window.navigator.onLine` property is also `true` or `false` accordingly. These events, however, will not alert you to changes in connection profiles, cost, or other aspects that aren't related to the basic availability of an Internet connection.

You can play with `networkstatuschanged` in Scenario 5 of the Network information sample. As you connect and disconnect networks or make other changes, the sample will update its details output for the current Internet profile if one is available (code condensed from `js/network-status-change.js`):

```
var networkInfo = Windows.Networking.Connectivity.NetworkInformation;
// Remember to removeEventListener for this event from WinRT as needed
networkInfo.addEventListener("networkstatuschanged", onNetworkStatusChange);

function onNetworkStatusChange(sender) {
    internetProfileInfo = "Network Status Changed: \n\r";
    var internetProfile = networkInfo.getInternetConnectionProfile();

    if (internetProfile === null) {
        // Error message
    } else {
        internetProfileInfo += getConnectionProfileInfo(internetProfile) + "\n\r";
        // display info
    }

    internetProfileInfo = "";
}
```

Of course, listening for this event is useful only if the app is actually running, but what if it isn't? In that case an app needs to register a background task, as discussed at the end of Chapter 13, for the `networkStateChange` trigger, typically applying the `internetAvailable` or `internetNot-Available` conditions as needed. The [Network status background sample](#) provides a demonstration of this, declaring a background task in its manifest with a C# entry point of `NetworkStatusTask.-NetworkStatusBackgroundTask`. The task is registered in `js/network-status-with-internet-present.js` (using helpers in `js/global.js` as typical for the background task samples):

```
BackgroundTaskSample.registerBackgroundTask(BackgroundTaskSample.sampleBackgroundTaskEntryPoint,
    BackgroundTaskSample.sampleBackgroundTaskWithConditionName,
    new Windows.ApplicationModel.Background.SystemTrigger(
        Windows.ApplicationModel.Background.SystemTriggerType.networkStateChange, false),
    new Windows.ApplicationModel.Background.SystemCondition(
        Windows.ApplicationModel.Background.SystemConditionType.internetAvailable));
```

The background task in `BackgroundTask.cs` simply writes the Internet profile name and network adapter id to local app data in response to the trigger. These values are output to the display within the `completeHandler` in `js/global.js`. A real app would clearly take more meaningful action, such as activating background transfers for data synchronization when connectivity is restored. The basic structure is there in the sample nonetheless.

It's also very important to remember that network status might have changed while the app was suspended. Apps that watch the `networkstatuschanged` event should also refresh their connectivity-related state within their `resuming` handler.

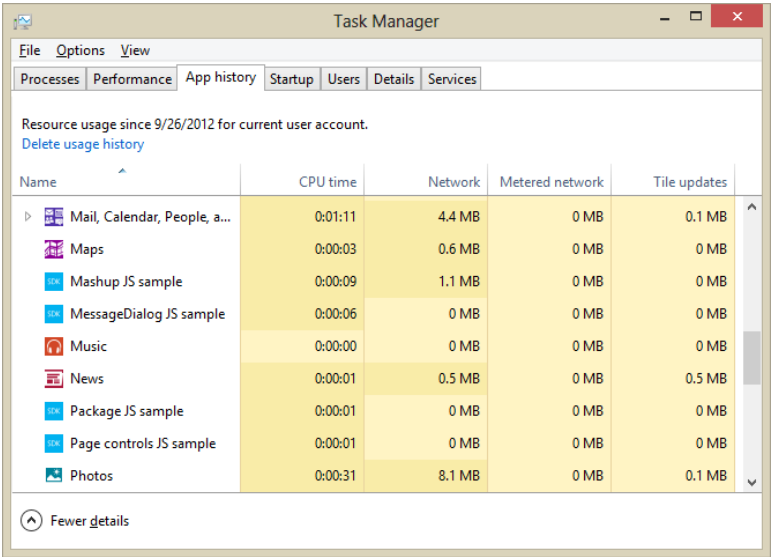
As a final note, check out the [Troubleshooting and debugging network connections](#) topic in the documentation, which has a little more guidance on responding to network changes as well as network errors.

Cost Awareness

If you ever crossed between 3G roaming territories with a smartphone that's set to automatically download email, you probably learned the hard way to disable syncing in such circumstances. I once drove from Washington State into Canada without realizing that I would suddenly be paying \$15/megabyte for the privilege of downloading large email attachments. Of course, since I'm a law-abiding citizen I did not look at my phone while driving (wink-wink!) to notice the roaming network. Well, a few weeks later I knew what "bill shock" was all about!

The point here is that if users conclude that *your* app is responsible for similar behavior, regardless of whether it's actually true, the kinds of rating and reviews you'll receive in the Windows Store won't be good! It's vital, then, to pay attention to changes in the cost of the connection profiles you're using, typically the Internet profile. Always check these details on startup, within your `networkstatuschanged` event handler, and within your `resuming` handler.

You—and all of your customers, I might add—can track your app's network usage in the App History tab of Task Manager, as shown below. Make sure you've expanded the view by tapping More Details on the bottom left if you don't see this view. You can see that it shows Network and Metered Network usage along with the traffic due to tile updates:



The screenshot shows the Windows Task Manager application with the 'App history' tab selected. The window title is 'Task Manager'. The menu bar includes 'File', 'Options', and 'View'. The tab bar shows 'Processes', 'Performance', 'App history' (selected), 'Startup', 'Users', 'Details', and 'Services'. Below the tabs, it says 'Resource usage since 9/26/2012 for current user account.' with a link 'Delete usage history'. A table displays resource usage for several applications. The table has five columns: 'Name', 'CPU time', 'Network', 'Metered network', and 'Tile updates'. The applications listed are Mail, Calendar, People, a...; Maps; Mashup JS sample; MatDialog JS sample; Music; News; Package JS sample; Page controls JS sample; and Photos. At the bottom left, there is a button labeled 'Fewer details' with an upward arrow icon.

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

Programmatically, as noted before, the profile provides usage information through its `get-ConnectionCost` and `getDataPlanStatus` methods. The first method returns a [ConnectionCost](#) object with four properties:

- `networkCostType` A [NetworkCostType](#) value, one of `unknown`, `unrestricted` (no extra charges), `fixed` (unrestricted up to a limit), and `variable` (charged on a per-byte or per-megabyte basis).
- `roaming` A Boolean indicating whether the connection is to a network outside of your provider's normal coverage area, meaning that extra costs are likely involved. An app should be very conservative with network activity when this is `true`.
- `approachingDataLimit` A Boolean that indicates that data usage on a fixed type network (see [networkCostType](#)) is getting close to the limit of the data plan.
- `overDataLimit` A Boolean indicating that a fixed data plan's limit has been exceeded and overage charges are definitely in effect. When this is `true`, an app should be very conservative with network activity, as when `roaming` is `true`.

The second method, `getDataPlanStatus`, returns a [DataPlanStatus](#) object with these properties:

- `dataPlanLimitInMegabytes` The maximum data transfer allowed for the connection in each billing cycle.
- `dataPlanUsage` A [DataPlanUsage](#) object with an all-important `megabytesUsed` property and a `lastSyncTime` (UTC) indicating when `megabytesUsed` was last updated.
- `maxTransferSizeInMegabytes` The maximum recommended size of a single network operation. This property reflects not so much the capacities of the metered network itself (as its documentation suggests), but rather an appropriate upper limit to transfers on that network.
- `nextBillingCycle` The UTC date and time when the next billing cycle on the plan kicks in and resets `dataPlanUsage` to zero.
- `inboundBitsPerSecond` and `outboundBitsPerSecond` Indicate the nominal transfer speed of the connection.

With all these properties you can make intelligent decisions about your app's network activity and/or warn the user about possible overage charges. Clearly, when the `networkCostType` is `unrestricted`, you can really do whatever you want. On the other hand, when the type is `variable` and the user is paying for every byte, especially when `roaming` is `true`, you'll want to inform the user of that status and provide settings through which the user can limit the app's network activity, if not halt that activity entirely. After all, the user might decide that certain kinds of data are worth having. For example, they should be able to set the quality of a streaming movie, indicate whether to download email messages or just headers, indicate whether to download images, specify whether caching of online data should occur, turn off background streaming audio, and so on.

Such settings, by the way, might include tile, badge, and other notification activities that you might have established, as those can generate network traffic. If you're also using background transfers, you can set the cost policies for downloads and uploads as well.

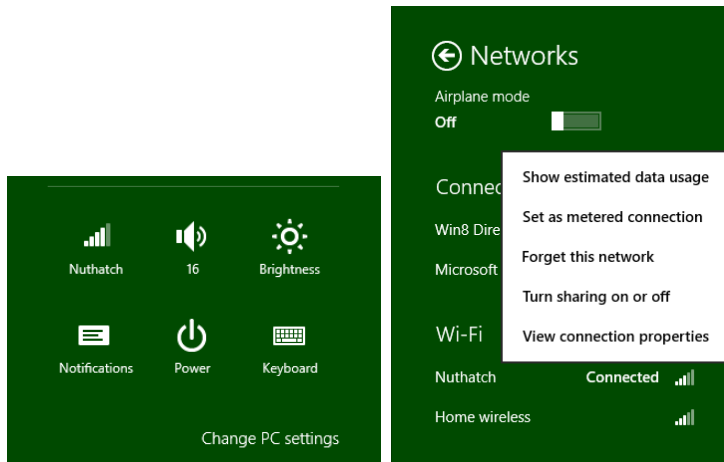
An app can, of course, ask the user's permission for any given network operation. It's up to you and your designers to decide when to ask and how often. [Windows Store policy](#), for its part (section 4.5), requires that you ask the user for any transfer exceeding one megabyte when `roaming` and `overDataLimit` are both `true`, and when performing any transfer over `maxTransferSizeInMegabytes`.

On a `fixed` type network, where data is unrestricted up to `dataPlanLimitInMegabytes`, we find cases where a number of the other properties become interesting. For example, if `overDataLimit` is already `true`, you can ask the user to confirm additional network traffic or just defer certain operations until the `nextBillingCycle`. Or, if `approachingDataLimit` is `true` (or even when it's not), you can determine whether a given operation might exceed that limit. This is where the connection profile's `getLocalUsage` method comes in handy to obtain a `DataUsage` object for a given period (see [How to retrieve connection usage information for a specific time period](#)). Call `getLocalUsage` with the time period between `lastSyncTime` and `DateTime.now()`. Then add that value to `megabytesUsed` and subtract the result from `dataPlanLimitInMegabytes`. This tells you how much more data you can transfer before incurring extra costs, thereby providing the basis for asking the user, "Downloading this file will exceed your data plan limit. Do you want to proceed?"

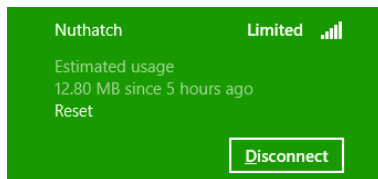
For simplicity's sake, you can think of cost awareness in terms of three behaviors: normal, conservative, and opt-in, which are described on [Managing connections on metered networks](#) and, more broadly, on [Developing connected apps](#). Both topics provide additional guidance on making the kinds of decisions described here already. In the end, saving the user from bill shock—and designing a great user experience around network costs—is definitely an essential investment.

Sidebar: Simulating Metered Networks

You may be thinking, "OK, so I get the need for my app to behave properly with metered networks, but how do I test such conditions without signing up with some provider and paying them a bunch of money (including roaming fees) while I'm doing my testing?" The simple answer is that you can simulate the behavior of metered networks with any Wi-Fi connection. First, invoke the Settings charm and tap on your network connection near the bottom (see below left, specifically the upper left icon, shown here as "Nuthatch"). In the Networks pane that then opens up (below right), right-click a wireless connection to open the menu and then select Set As Metered Connection:



Although this option will not set up [DataUsage](#) properties and all that a real metered network might provide, it will return a [networkCostType](#) of `fixed`, which allows you to see how your app responds. You can also use the Show Estimated Data Usage menu item to watch how much traffic your app generates during its normal operation, and you can reset the counter so that you can take some accurate readings:



Running Offline

The other user experience that is sure to earn your app a better reputation is how it behaves when there is no connectivity or when there's a change in connectivity. Ask yourself the following questions:

- What happens if your app starts without connectivity, both from tiles (primary and secondary) and through contracts such as search, share, and the file picker?
- What happens if your app runs the first time without connectivity?
- What happens if connectivity is lost while the app is running?
- What happens when connectivity comes back?

As described above in the “Connectivity Awareness” section, you can use the `networkstatus-changed` event to handle these situations while running and your `resuming` handler to check if connection status changed while the app was suspended. If you have a background task tied to the `networkStateChange` trigger, it would primarily save state that your `resuming` handler would then check.

It's perfectly understood that some apps just can't run without connectivity, in which case it's appropriate to inform the user of that situation when the app is launched or when connectivity is lost while the app is running. In other situations, an app might be partially usable, in which case you should inform the user more on a case-by-case basis, allowing them to use unaffected parts of the app. Better still is to cache data that might make the app even more useful when connectivity is lost. Such data might even be built into the app package so that it's always available on first launch.

Consider the case of an ebook reader app that would generally acquire new titles from an online catalog. For offline use it would do well to cache copies of the user's titles locally, rather than rely solely on having a good Internet connection. The app's publisher might also include a number of popular free titles directly in the app package such that a user could install the app while waiting to board a plane and have at least those books ready to go when the app is first launched at 30,000 feet. Other apps might include some set of preinstalled data at first and then add to that data over time (perhaps through in-app purchases) when unrestricted networks are available. By following network costs closely, such an app might defer downloading a large data set until either the user confirms the action or a different connection is available.

How and when to cache data from online resources is probably one of the fine arts of software development. When do you download it? How much do you acquire? Where do you store it? Should you place an upper limit on the cache? Do you allow changes to cached data that would need to be synchronized with a service when connectivity is restored? These are all good questions ask, and certainly there are others to ask as well. Let me at least offer a few thoughts and suggestions.

First, you can use any network transport to acquire data to cache such as [WinJS.xhr](#), the background transfer API, as well as the HTML5 [AppCache](#) mechanism, which works well for web content you load up in [iframe](#) elements. Note that using the AppCache requires that the URLs in question are declared in the manifest as `ApplicationContentUris` (see Chapter 3, "App Anatomy and Page Navigation"). Separately, other content acquired from remote resources, such as images, are also cached automatically like typical temporary Internet files. Even remote script downloaded within a web context [iframe](#) is cached this way. Both caching mechanisms are subject to the storage limits defined by Internet Explorer.

How much data you cache depends, certainly, on the type of connection you have and the relative importance of the data. On an unrestricted network, feel free to acquire everything you feel the user might want offline, but it would be a good idea to provide settings to control that behavior, such as overall cache size or the amount of data to acquire per day. I mention the latter because even though my own Internet connection appears to the system as unrestricted, I'm charged more as my usage reaches certain tiers (on the order of gigabytes). As a user, I would appreciate having a say in matters that involve significant network traffic.

Even so, if caching specific data will greatly enhance the user experience, separate that option to give the user control over the decision. For example, an ebook reader might automatically download a whole title while the reader is perhaps just browsing the first few pages. Of course, this would also mean consuming more storage space. Letting users control this behavior as a setting, or even on a per-book basis, lets them decide what's best. For smaller data, on the other hand—say, in the range of several

hundred kilobytes—if you know from analytics that a user that views one set of data is highly likely to view another, automatically acquiring and caching those additional data sets could be the right design.

The best place to store cached data is your app data folders, specifically the `LocalFolder` and `TemporaryFolder`. Avoid using the `RoamingFolder` to cache data acquired from online sources: besides running the risk of exceeding the roaming quota (see Chapter 8, “State, Settings, Files, and Documents”), it’s also quite pointless. Because the system would have to roam such data over the network anyway, it’s better to just have the app re-acquire it when it needs to. The same applies to in-app purchases: because the user can easily acquire those purchases through the Windows Store on another machine (where the app on that machine would find that those purchases are already paid for), they need not be roamed.

Whether you use the `LocalFolder` or `TemporaryFolder` depends on how essential the data is to the operation of the app. If the app cannot run without the cache—such as the cookbook app I mentioned earlier—use local app data. If the cache is just an optimization such that the user could reclaim that space with the Disk Cleanup tool, store the cache in the `TemporaryFolder` and rebuild it again later on. (Be aware once again that `IndexedDB`, as described in Chapter 8, has a per-app limit and an overall system limit. If this is a potential problem, you might want to choose a different storage mechanism.)

In all of this, also consider that what you’re caching really might be user data that you’d want to store outside of your app data folders. That is, be sure to think through the distinction between app data and user data!

Finally, you might again have the kind of app that allows offline activity (like processing email) where you will have been caching the results of that activity for later synchronization with an online resource. When connectivity is restored, then, check if the network cost is suitable before starting your sync process.

Sidebar: Connectivity and Remote Images in Live Tiles and Toasts

In Chapter 13 we looked at how an app appears “alive with activity” through features such as live tiles and notifications. Clearly, periodic updates and push notifications are completely dependent on connectivity and will not operate without it; a running app, on the other hand, can still issue updates when the device is offline. Under such a circumstance, the app must avoid referencing remote images in the update, because these will not be resolved without connectivity and the tile and toast systems do not presently support the use of local fallback images. An app should thus check connectivity status before issuing an update and should make sure to use local (`ms-appx:///` or `ms-appdata:///`) images instead of remote ones or opt for text-only tile and toast templates.

XmlHttpRequest

As we've seen a number of times already in this book, transferring data to and from web services with XMLHttpRequest is a common activity for Windows Store apps, especially those written in JavaScript for which handling XML and/or JSON is simple and straightforward. This is especially true when using the `WinJS.xhr` wrapper that turns the whole process into a simple promise.

To build on what we already covered in Chapter 3, in the section "Data from Services and WinJS.xhr," there are a few other points to make where such requests are concerned, most of which come from the section in the documentation entitled [Connecting to a web service](#).

First, [Downloading different types of content](#) provides the details of the different content types supported by XHR for Windows Store apps. These are summarized here:

Type	Use	responseText	responseXML
arraybuffer	Binary content as an array of Int8 or Int64, or another integer or float type.	undefined	undefined
Blob	Binary content represented as a single entity.	undefined	undefined
document	An XML DOM object representing XML content (MIME type of text/XML).	undefined	The XML content
json	JSON strings.	The JSON string	undefined
ms-stream	Streaming data; see XmlHttpRequest enhancements .	undefined	undefined
Text	Text (the default).	The text string	undefined

Second, know that XHR responses can be automatically cached, meaning that later requests to the same URI might return old data. To resend the request despite the cache, add an *If-Modified-Since* HTTP header as shown on [How to ensure that WinJS.xhr resends requests](#).

Along similar lines, you can wrap a `WinJS.xhr` operation in another promise to encapsulate automatic retries if there is an error in any given request. That is, build your retry logic around the core XHR operation, with the result stored in some variable. Then place that whole block of code within `WinJS.Promise.wrap` (or a new `WinJS.Promise`) and use that elsewhere in the app.

In each XHR attempt, remember that you can also use `WinJS.Promise.timeout` in conjunction with `WinJS.Xhr` as described on [Setting timeout values with WinJS.xhr](#), because `WinJS.xhr` doesn't have a timeout notion directly. You can, of course, set a timeout in the raw request, but that would mean rebuilding everything that `WinJS.xhr` already does.

Generally speaking, XHR headers are accessible to the app with the exception of cookies (the *set-cookie* and *set-cookie2* headers), as these are filtered out by design for XHR done from a local context. They are not filtered for XHR from the web context, so if you need cookies, try acquiring them in a web context `iframe` and pass them to a local context using `postMessage`.

Finally, avoid using XHR for large file transfers because such operations will be suspended when the app is suspended. Use the Background Transfer API instead (see the next section), which uses XHR under the covers, so your web services won't know the difference anyway!

And on that note, let's now look at that Background Transfer API in detail.

Sidebar: Debugging Network Traffic with Fiddler

If you're interested in watching the HTTP(S) traffic between your computer and the Internet—something that can be invaluable when working with XMLHttpRequests—check out the freeware tool known as Fiddler (<http://www.fiddler2.com/fiddler2/>). In addition to inspecting traffic, you can also set breakpoints on various events and fiddle with (that is, modify) incoming and outgoing data. It supports traffic from any app or browser, including Windows Store apps.

Background Transfer

One need with user data, especially, is to transfer potentially large files to and from an online repository. For even moderately sized files, however, this presents a challenge: very few users typically want to stare at their display to watch file transfer progress, so it's highly likely that they'll switch to another app to do something far more interesting while the transfer is taking place. In doing so, the app that's doing the transfer will be suspended in the background and possibly even terminated. This does not bode well for trying to complete such operations using a mechanism like [WinJS.xhr](#)!

One solution would be to provide a background task for this purpose, which was a common request with early previews of Windows 8 before this API was ready. However, there's little need to run app code for this common purpose. WinRT thus provides a specific background transfer API, [Windows.Networking.BackgroundTransfer](#), supporting up to 500 scheduled transfers systemwide. It offers built-in cost awareness and resiliency to changes in connectivity, reliving apps from needing to worry about such concerns themselves. Transfers continue when an app is suspended and will be paused if the app is terminated. When the app is resumed or launched again, it can then check the status of background transfers it previously initiated and take further action as necessary—processing downloaded information, noting successful uploads in its UI, and enumerating pending transfers, which will restart any that were paused or otherwise interrupted. On the other hand, if the user directly closes the app (through a gesture, Alt+F4, or Task Manager), all pending transfers for that app are canceled. This is also true if you stop debugging an app in Visual Studio.

Generally speaking, then, it's highly recommended that you use the background transfer API whenever you expect the operation to exceed your customer's tolerance for waiting. This clearly depends on the network's connection speed, and whether you think the user will switch away from your app while such a transfer is taking place. For example, if you initiate a transfer operation but the user can continue to be productive (or entertained) in your app while that's happening, then using [WinJS.xhr](#) with HTTP GET and POST/PUT might be a possibility, though you'll still be responsible for cost

awareness and handling connectivity. If, on the other hand, the user cannot do anything more until the transfer is complete, you might choose to use background transfer for perhaps any data larger than 10K or some other amount based on the current network speed.

In any case, when you're ready to employ background transfer in your app, the [BackgroundDownloader](#) and [BackgroundUploader](#) objects in the [Windows.Networking.BackgroundTransfer](#) namespace will become your fast friends. Both objects have methods and properties through which you can enumerate pending transfers as well as perform general configuration of credentials, HTTP request headers, transfer method, cost policy (for metered networks), and grouping. Each individual operation is then represented by a [DownloadOperation](#) or [UploadOperation](#) object, through which you can control the operation (pause, cancel, etc.) and retrieve status. With each operation you can also set its particular credentials, cost policy, and so forth, overriding the general settings in the [BackgroundDownloader](#) and [BackgroundUploader](#) classes.

Note In both download and upload cases, the connection request will be aborted if a new connection is not established within five minutes. After that, any other HTTP request involved with the transfer times out after two minutes. Background transfer will retry an operation up to three times if there's connectivity.

To see the basics of this API in action, let's start by looking at the [Background transfer sample](#). To run this sample you must first set up a localhost server along with a data file and an upload target page. So make sure you have Internet Information Services installed on your machine, as described in Chapter 13 in the section "Using the Localhost." Then, from an administrator command prompt, navigate to the sample's Server folder and run the command **powershell -file serversetup.ps1**. This will install the necessary server-side files for the sample on the localhost, and allow you to run an additional example in this chapter's companion content.

Basic Downloads

Scenario 1 of the Background transfer sample ([js/downloadFile.js](#)) lets you download an image file from the localhost server and save it to the Pictures library. By default the URI entry field is set to a specific localhost URI and the control is disabled. This is because the sample doesn't perform any validation on the URI, a process that you should always perform in your own app. If you'd like to enter other URIs in the same, of course, just remove `disabled="disabled"` from the `serverAddressField` element in [html/downloadFile.html](#). To see the downloader in action, it's also helpful to locate some large image files that will take a while to transfer; your favorite search engine can help you out, or you can copy one of your own to the localhost server.

The sample's UI also provides a handful of buttons to start, cancel, pause, and resume the async operation, an essential feature for apps with background transfers. Within its progress handler, which the transfer operations support, the sample demonstrates how to display as much of the image has been transferred. You can also start multiple transfers to observe how they are all managed simultaneously.

Starting a download transfer happens as follows. First create a [StorageFile](#) to receive the data (though this is not required as we'll see later in this section). Then create a [DownloadOperation](#) object for the transfer using [BackgroundDownloader.createDownload](#), at which point you can set its [method](#), [costPolicy](#), and [group](#) properties to override the defaults supplied by the [Background-Downloader](#). The [method](#) is a string that identifies the type transfer being used (normally GET for HTTP or RETR for FTP). We'll come back to the other two properties later in the "Setting Cost Policy" and "Grouping Multiple Transfers" sections.

Once the operation is configured as needed, the last step is to call its [startAsync](#) method with your completed, error, and progress handlers:⁷²

```
// Asynchronously create the file in the pictures folder (capability declaration required).
Windows.Storage.KnownFolders.picturesLibrary.createFileAsync(fileName,
    Windows.Storage.CreationCollisionOption.generateUniqueName)
    .done(function (newFile) {
        // Assume uriString is the text URI of the file to download
        var uri = Windows.Foundation.Uri(uriString);
        var downloader = new Windows.Networking.BackgroundTransfer.BackgroundDownloader();

        // Create a new download operation.
        var download = downloader.createDownload(uri, newFile);

        // Start the download
        download.startAsync().then(complete, error, progress);
    })
```

While the operation underway, the following properties provide additional information on the transfer:

- [requestedUri](#) and [resultFile](#) The same as those passed to [createDownload](#).
- [guid](#) A unique identifier assigned to the operation.
- [progress](#) A [BackgroundDownloadProgress](#) structure with [bytesReceived](#), [total-BytesToReceive](#), [hasResponseChanged](#) (a Boolean, see the [getResponseInformation](#) method below), [hasRestarted](#) (a Boolean set to [true](#) if the download had to be restarted), and [status](#) (a [BackgroundTransferStatus](#) value: [idle](#), [running](#), [pausedByApplication](#), [pausedCostedNetwork](#), [pausedNoNetwork](#), [canceled](#), [error](#), and [completed](#)).

A few methods of [DownloadOperation](#) can also be used with the transfer:

- [pause](#) and [resume](#) Control the download in progress. We'll talk more of these in the "Suspend, Resume, and Restart with Background Transfers" section below.
- [getResponseInformation](#) Returns a [ResponseInformation](#) object with properties named [headers](#) (a collection of response headers from the server), [actualUri](#), [isResumable](#), and

⁷² The code in the sample has more structure than shown here. It defines its own *DownloadOperation* class that unfortunately has the same name as the WinRT class, so I'm electing to omit mention of it.

`statusCode` (from the server). Repeated calls to this method will return the same information until the `hasResponseChanged` property is set to true.

- `getResultStreamAt` Returns an `IInputStream` for the content downloaded so far or the whole of the data once the operation is complete.

In Scenario 1 of the sample, the progress function—which is given to the promise returned by `startAsync`—uses `getResponseInformation` and `getResultStreamAt` to show a partially downloaded image:

```
var currentProgress = download.progress;

// ...

// Get Content-Type response header.
var contentType = download.getResponseInformation().headers.lookup("Content-Type");

// Check the stream is an image.
if (contentType.indexOf("image/") === 0) {
    // Get the stream starting from byte 0.
    imageStream = download.getResultStreamAt(0);

    // Convert the stream to a WinRT type
    var msStream = MSApp.createStreamFromInputStream(contentType, imageStream);
    var imageUrl = URL.createObjectURL(msStream);

    // Pass the stream URL to the HTML image tag.
    id("imageHolder").src = imageUrl;

    // Close the stream once the image is displayed.
    id("imageHolder").onload = function () {
        if (imageStream) {
            imageStream.close();
            imageStream = null;
        }
    };
}
```

All of this works because the background transfer API is saving the downloaded data into a temporary file and providing a stream on top of that, hence a function like `URL.createObjectURL` does the same job as if we provided it with a `StorageFile` object directly. Once the `DownloadOperation` object goes out of scope and is garbage collected, however, that temporary file will be deleted.

The existence of this temporary file is also why, as I noted earlier, it's not actually necessary to provide a `StorageFile` object in which to place the downloaded data. That is, you can pass `null` as the second argument to `createDownload` and work with the data through `DownloadOperation.getResultStreamAt`. This is entirely appropriate if the ultimate destination of the data in your app isn't a separate file.

There is also a variation of [createDownload](#) that takes a second [StorageFile](#) argument whose contents provide the body of the HTTP GET or FTP RETR request that will be sent to the server URI before the download is started. This accommodates some web sites that require you to fill out a form to start the download.

Sidebar: Where Is Cancel?

You might have already noticed that neither [DownloadOperation](#) nor [UploadOperation](#) has a cancellation method. So how is this accomplished? You cancel the transfer by canceling the [startAsync](#) operation—that is, call the [cancel](#) method of the *promise* returned by [startAsync](#). This means that you need to hold onto the promises for each transfers you initiate.

Basic Uploads

Scenario 2 of the Background transfer sample ([js/uploadFile.js](#)) exercises the background upload capability, specifically sending some file (chosen through the file picker) to a URI that can receive it. By default the URI points to [http://localhost/BackgroundTransferSample/upload.aspx](#), a page installed with the PowerShell script that sets up the server. As with Scenario 1, the URI entry control is disabled because the sample performs no validation, as you would again always want to do if you accepted any URI from an untrusted source (user input in this case). For testing purposes, of course, you can remove [disabled="disabled"](#) from the *serverAddressField* element in [html/uploadFile.html](#) and enter other URIs that will exercise your own upload services. This is especially handy if you run the server part of the sample in Visual Studio 2012 Express for Web where the URI will need a localhost port number as assigned by the debugger.

In addition to a button to start an upload and to cancel it, the sample provides another button to start a *multipart* upload; we'll talk more of this in the "Multipart Uploads" section below.

In code, an upload happens very much like a download. Assuming you have a [StorageFile](#) with the contents to upload, create an [UploadOperation](#) object for the transfer with [BackgroundUploader.createUpload](#). If, on the other hand, you have data in a stream ([IInputStream](#)), create the operation object with [BackgroundUploader.createUploadFromStreamAsync](#) instead. This can also be used to break up a large file into discrete chunks, if the server can accommodate it; see "Breaking Up Large Files" below.

With the operation object in hand you can customize a few properties of the transfer, overriding the defaults provided by the [BackgroundUploader](#). These are [method](#) (HTTP POST or PUT, or FTP STOR), [costPolicy](#), and [group](#). For the latter, again see "Setting Cost Policy" and "Grouping Multiple Transfers" below.

Once you're ready, then, calling the operation's [startAsync](#) will proceed with the upload:⁷³

⁷³ As with downloads, the code in the sample has more structure than shown here and again defines its own

```
// Assume uri is a Windows.Foundation.Uri object and file is the StorageFile to upload
var uploader = new Windows.Networking.BackgroundTransfer.BackgroundUploader();
var upload = uploader.createUpload(uri, file);
promise = upload.startAsync().then(complete, error, progress);
```

While the operation is underway, the following properties provide additional information on the transfer:

- **requestedUri** and **sourceFile** The same as those passed to **createUpload** (an operation created with **createUploadFromStreamAsync** supports only **requestedUri**).
- **guid** A unique identifier assigned to the operation.
- **progress** A **BackgroundUploadProgress** structure with **bytesReceived**, **totalBytesToReceive**, **bytesSent**, **totalBytesToSend**, **hasResponseChanged** (a Boolean, see the **getResponseInformation** method below), **hasRestarted** (a Boolean set to **true** if the upload had to be restarted), and **status** (a **BackgroundTransferStatus** value, again with values of **idle**, **running**, **pausedByApplication**, **pausedCostedNetwork**, **pausedNoNetwork**, **canceled**, **error**, and **completed**).

Unlike a download, an **UploadOperation** does not have pause or resume methods but does have the same **getResponseInformation** and **getResultStreamAt** methods. In the upload case, the response from the server is less interesting because it doesn't contain the transferred data, just headers, status, and whatever body contents the upload page cares to return. If that page returns some interesting HTML, though, you might use the results as part of your app's output for the upload.

As noted before, to cancel an **UploadOperation**, call the **cancel** method of the promise returned from **startAsync**.

Breaking Up Large Files

Because the outbound (upload) transfer rates of most broadband connections is significantly slower than the inbound (download) rates and might have other limitations, uploading a large file to a server is typically a riskier business than a large download. If an error occurs during the upload, it can invalidate the entire transfer—a very frustrating occurrence if you've already been waiting an hour for that upload to complete!

For this reason, a cloud service might allow a large file to be transferred in discrete chunks, each of which is sent as a separate HTTP request with the server reassembling the single file from those requests. This minimizes or at least reduces the overall impact of connectivity hiccups.

From the client's point of view, each piece would be transferred with an individual **Upload-Operation**; that much is obvious. The tricky part is breaking up a large file in the first place. With a lot of elbow grease—and what would likely end up being a complex chain of nested async

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

operations—it is possible to create a bunch of temporary files from the single source. If you're up to a challenge, I invite you to write such a routine and post it somewhere for the rest of us to see!

But there is an easier path using `createUploadFromStreamAsync`, through which you can create separate `UploadOperation` objects for different segments of the stream. Given a `StorageFile` for the source, start by calling its `openReadAsync` method, the result of which is an `IRandom-AccessStreamWithContentType` object. Through its `getInputStreamAt` method you then obtain an `IInputStream` for each starting point in the stream (that is, at each offset depending on your segment size). You then create an `UploadOperation` with each input stream by using `create-UploadFromStreamAsync`. The last requirement is to tell that operation to consume only some portion of that stream. You do this by calling its `setRequestHeader("content-length", <length>)` where `<length>` is the size of the segment plus the size of other data in the request; you'll also want to add a header to identify the segment for that particular upload. After all this, call each operation's `startAsync` method to begin its transfer.

Multipart Uploads

In addition to the `createUpload` and `createUploadFromStreamAsync` methods, the `BackgroundUploader` provides another method called `createUploadAsync` (with three variants) that handles what are called *multipart uploads*.

From the server's point of view, a multipart upload is a *single* HTTP request that contains various pieces of information (the parts), such as app identifiers, authorization tokens, and so forth, along with file content, where each part is possibly separated by a specific boundary string. Such uploads are used by online services like Flickr and YouTube, each of which accepts a request with a multipart Content-Type. (See [Content-type: multipart](#) for a reference.) For example, as shown on [Uploading Photos – POST Example](#), Flickr wants a request with the content type of `multipart/form-data`, followed by parts for `api_key`, `auth_token`, `api_sig`, `photo`, and finally the file contents. With YouTube, as described on [YouTube API v2.0 – Direct Uploading](#), it wants a content type of `multipart/related` with parts containing the XML request data, the video content type, and then the binary file data.

The background uploader supports all this through the `BackgroundUploader.createUploadAsync` method. (Note the `Async` suffix that separates these from the synchronous `createUpload`.) There are three variants of this method. The first takes the server URI to receive the upload and an array of `BackgroundTransferContentPart` objects, each of which represents one part of the upload. The resulting operation will send a request with a content type of `multipart/form-data` with a random GUID for a boundary string. The second variation of `createUploadAsync` allows you to specify the content type directly (through the sub-type, such as `related`), and the third variation then adds the boundary string. That is, assuming `parts` is the array of parts, the methods look like this:

```
var uploadOpPromise1 = uploader.createUploadAsync(uri, parts);
var uploadOpPromise2 = uploader.createUploadAsync(uri, parts, "related");
var uploadOpPromise3 = uploader.createUploadAsync(uri, parts, "form-data", "-----123456");
```

To create each part, first create a `BackgroundTransferContentPart` using one of its [three constructors](#):

- `new BackgroundContentPart()` Creates a default part.
- `new BackgroundContentPart(<name>)` Creates a part with a given name.
- `new BackgroundContentPart(<name>, <file>)` Creates a part with a given name and a local filename.

In each case you further initialize the part with a call to its `setText`, `setHeader`, and `setFile` methods. The first, `setText`, assigns a value to that part. The second, `setHeader`, can be called multiple times to supply header values for the part. The third, `setFile`, is how you provide the `StorageFile` to a part created with the third variant above.

Now, Scenario 2 of the original Background transfer sample shows the latter using an array of selected files, but probably few services would accept a request of this nature. Let's instead look at how we'd create the multipart request shown on [Uploading Photos – POST Example](#). For this purpose I've created the Multipart Upload example in this chapter's companion content. Here's the code from `js/uploadMultipart.js` that creates all the necessary parts using the `tinyimage.jpg` file in the app package:

```
// The file and uri variables are already set by this time. bt is a namespace shortcut
var bt = Windows.Networking.BackgroundTransfer;
var uploader = new bt.BackgroundUploader();
var contentParts = [];

// Instead of sending multiple files (as in the original sample), we'll create those parts that
// match the POST example for Flickr on http://www.flickr.com/services/api/upload.example.html
var part;

part = new bt.BackgroundTransferContentPart();
part.setHeader("Content-Disposition", "form-data; name=\"api_key\"");
part.setText("3632623532453245");
contentParts.push(part);

part = new bt.BackgroundTransferContentPart();
part.setHeader("Content-Disposition", "form-data; name=\"auth_token\"");
part.setText("436436545");
contentParts.push(part);

part = new bt.BackgroundTransferContentPart();
part.setHeader("Content-Disposition", "form-data; name=\"api_sig\"");
part.setText("43732850932746573245");
contentParts.push(part);

part = new bt.BackgroundTransferContentPart();
part.setHeader("Content-Disposition", "form-data; name=\"photo\"; filename=\"\" + file.name + "\"");
part.setHeader("Content-Type", "image/jpeg");
part.setFile(file);
contentParts.push(part);

// Create a new upload operation specifying a boundary string.
uploader.createUploadAsync(uri, contentParts,
```

```

        "form-data", "-----7d44e178b0434")
    .then(function (uploadOperation) {
        // Start the upload and persist the promise
        upload = uploadOperation;
        promise = uploadOperation.startAsync().then(complete, error, progress);
    })
);

```

The resulting request will look like this, very similar to what's shown on the Flickr page (just with some extra headers):

```

POST /website/multipartupload.aspx HTTP/1.1
Cache-Control=no-cache
Connection=Keep-Alive
Content-Length=1328
Content-Type=multipart/form-data; boundary="-----7d44e178b0434"
Accept=/*/*
Accept-Encoding=gzip, deflate
Host=localhost:60355
User-Agent=Mozilla/5.0 (compatible; MSIE 10.0; Windows NT 6.2; Win64; x64; Trident/6.0; Touch)
UA-CPU=AMD64
-----7d44e178b0434
Content-Disposition: form-data; name="api_key"

3632623532453245
-----7d44e178b0434
Content-Disposition: form-data; name="auth_token"

436436545
-----7d44e178b0434
Content-Disposition: form-data; name="api_sig"

43732850932746573245
-----7d44e178b0434
Content-Disposition: form-data; name="photo"; filename="tinysquare.jpg"
Content-Type: image/jpeg

{RAW JFIF DATA}
-----7d44e178b0434--

```

To run the sample and also see how this request is received, go to the MultipartUploadServer folder in this chapter's companion content. Load website.sln into Visual Studio 2012 Express for Web, open MultipartUploadServer.aspx, and set a breakpoint on the first `if` statement inside the `Page_Load` method. Then start the site in Internet Explorer to open that page on a localhost debugging port. Copy that page's URI for the next step.

In the Multipart Upload example, paste that URI into the URI field and click the Start Multipart Transfer. When the upload operation's `startAsync` is called, you should hit the server page breakpoint in Visual Studio for Web. You can step through that code if you want and examine the Request object; in the end, the code will copy the request into a file named multipart-request.txt on that server. This will contain the request contents as above, where you can see the relationship between how you set up the parts in the client and how they are received by the server.

Providing Headers and Credentials

Within the `BackgroundDownloader` and `BackgroundUploader` you have the ability to set values for individual HTTP headers by using their `setRequestHeader` methods. Both take a header name and a value, and you call them multiple times if you have more than one header to set.

Similarly, both the downloader and uploader objects have two properties for credentials: `serverCredential` and `proxyCredential`, depending on the needs of your server URI. Both properties are `Windows.Security.Credentials.PasswordCredential` objects. As the purpose in a background transfer operation is to provide credentials to the server, you'd typically create a `PasswordCredential` as follows:

```
var cred = new Windows.Security.Credentials.PasswordCredential(resource, userName, password);
```

where the `resource` in this case is just a string that identifies the resource to which the credentials applies. This is used to manage credentials in the credential locker, as we'll see in the "Authentication, Credentials, and the User Profile" section later. For now, just creating a credential in this way is all you need to authenticate with your server when doing a transfer.

Note At present, setting the `serverCredential` property doesn't work with URIs that specify an FTP server. To work around this, include the credentials directly in the URI with the form `ftp://<user>:<password>@server.com/file.ext` (for example, `ftp://admin:password1@server.com/file.bin`).

Setting Cost Policy

As mentioned earlier in the section "Cost Awareness," the Windows Store policy requires that apps are careful about performing large data transfers on metered networks. The Background Transfer API takes this into account, based on values from the `BackgroundTransferCostPolicy` enumeration:

- `default` Allow transfers on costed networks.
- `unrestrictedOnly` Do not allow transfers on costed networks.
- `always` Always download regardless of network cost.

To apply a policy to subsequent transfers, set the value of `BackgroundDownloader.costPolicy` and/or `BackgroundUploader.costPolicy`. The policy for individual operations can be set through the `DownloadOperation.costPolicy` and `UploadOperation.costPolicy` properties.

Basically, you would change the policy if you've prompted the user accordingly or allow them to set behavior through your settings. For example, if the user has a setting to disallow downloads or uploads on a metered network, your apps would set the general `costPolicy` to `unrestrictedOnly`. If you know you're on a network where roaming charges would apply and the user has consented to a transfer, you'd want to change the `costPolicy` of that *individual* operation to `always`. Otherwise the API would not perform the transfer because doing so on a roaming network is disallowed by default.

When a transfer is blocked by policy, the operation's `progress.status` property will contain `BackgroundTransferStatus.pausedCostedNetwork`.

Grouping Transfers

The `group` property that's found in `BackgroundDownloader`, `BackgroundUploader`, `DownloadOperation`, and `UploadOperation` is a simple string that tags a transfer as belonging to a particular group. The property can be set only through `BackgroundDownloader` and `BackgroundUploader`; you would set this prior to creating a series of individual operations. In those operations, the `group` property is available but read-only.

The purpose of grouping is so that you can selectively enumerate and control related transfers, as we'll see in the next section. For example, a photo app that organizes pictures into albums or album pages can present a UI through which the user can pause, resume, or cancel the transfer of an entire album, rather than working on the level of individual files. The `group` property makes the implementation of this kind of experience much easier, as the app doesn't need to maintain its own grouping structures.

The `group` has no bearing on the transfers themselves; it is not communicated to the server upload page.

Suspend, Resume, and Restart with Background Transfers

At the beginning of this section I mentioned that background transfers will continue while an app is suspended and paused if the app is terminated by the system. Because apps will be terminated only in low-memory conditions, it's appropriate to also pause background transfers.

When an app is resumed from the suspended state, it can check on the status of pending transfers by using the `BackgroundDownloader.getCurrentDownloadsAsync` and `BackgroundUploader.getCurrentUploadsAsync` methods. In both cases two variants of the methods exist: one that enumerates all transfers, and one that enumerates those belonging to a specific group (as matches the `group` properties in the operations).

The list that comes back from these methods is a vector of `DownloadOperation` and `UploadOperation` objects, and, as always, the vector can be addressed as an array. Code to iterate over the list looks like this:

```
Windows.Networking.BackgroundTransfer.BackgroundDownloader.getCurrentDownloadsAsync()
    .done(function (downloads) {
        for (var i = 0; i < downloads.size; i++) {
            var download = downloads[i];
        }
    });
```

```
Windows.Networking.BackgroundTransfer.BackgroundUploader.getCurrentUploadsAsync()
    .done(function (uploads) {
        for (var i = 0; i < uploads.size; i++) {
            var upload = uploads[i];
        }
    });
```

In each case, the `progress` property of each operation will tell you how far the transfer has come along. The `progress.status` property is especially important. Again, status is a [Background-TransferStatus](#) value and will be one of `idle`, `running`, `pausedByApplication`, `pausedCosted-Network`, `pausedNoNetwork`, `canceled`, `error`, and `completed`). These are clearly necessary to inform the user, as appropriate, and to give her the ability to restart transfers that are paused or experienced an error, to pause running transfers, and to act on completed transfers.

Speaking of which, when using the background transfer API, an app should always give the user control over pending transfers. Downloads can be paused through the `DownloadOperation.pause` method and resumed through `DownloadOperation.resume`. (There are no equivalents for uploads.) Download and upload operations are canceled by canceling the promises returned from `startAsync`.

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

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

```
promise = download.attachAsync().then(complete, error, progress);
```

and call `promise.cancel()` if needed. In short, when Windows restarts a background transfer and essentially calls `startAsync` on your app's behalf, it holds that promise internally. The `attachAsync` methods simply return that new promise.

A final question is whether a suspended app can be notified when a transfer is complete, perhaps to issue a toast to inform the user. Such a feature isn't supported in Windows 8 as there is no background task available for this purpose. At present, then, the user needs to switch back to the app to check on transfer progress.

Authentication, Credentials, and the User Profile

If you think about it, just about every online resource in the world has some kind of credentials or authentication associated with it. Sure, we can read many of those resources without credentials, but having permission to upload data to a website is more tightly controlled, as is access to one's account or profile in a database managed by a website. In many scenarios, then, apps need to manage credentials and handle other authentication processes, perhaps for accounts that you manage but perhaps also when using accounts from other sources such as Facebook, Twitter, and so on.

There are two basic approaches for dealing with credentials. First, you can collect credentials directly through the Credential Picker UI or a UI of your own. In either case, though, the next question is how to store those credentials securely, for which we have the Credential Locker API. The locker allows an app to retrieve those credentials in subsequent sessions such that it doesn't need to ask the user to enter those credentials again (which gets tiresome, as I'm sure you know).

It's very important to understand here that whenever an app acquires credentials as plain text, either from its own UI or from the Credential Picker with certain options, the app is fully responsible for protecting those credentials. For one thing, the app must always store and transmit those credentials with full encryption, but there are many subtleties here that are typically far more complicated than apps should worry about themselves.

For this reason it's a good idea to delegate those details to others. For example, the Credential Picker UI will, by default, encrypt passwords before they ever get back to your app. Or you can use the second approach to credentials where the app authenticates users through another provider altogether, such as Microsoft Live Connect, Facebook, Flickr, Yahoo, and so forth. In doing so, the provider does the heavy lifting of authentication and the app needs only to store the appropriate tokens or other access keys for these services. A primary benefit to this kind of integrated authorization is that the app never touches those credentials itself and thus does not need to concern itself with their security. (An should still encrypt tokens or access keys if it stores them. The credential locker can also be used for this purpose.)

In most cases this process involves an agent called the Web Authentication Broker, which specifically works with OAuth/OpenID protocols and providers as generally found on the web. Microsoft Live Connect is a special case because the Microsoft account used with it might also be the one used to log into Windows itself. (Authenticating through Live Connect also gives an app access to other data from Live services including Calendar, Messenger, and SkyDrive.)

One of the other significant benefits of this second approach is the ability to provide a *single sign on* experience. This means that once a user has signed in through a particular OAuth provider in one app, they often don't need to sign into *other* apps that use the same provider (unless the app deems it necessary). In the case of Live Connect, apps might never need to request credentials at all if that same Microsoft account is used to log in to Windows or is linked to the user's domain login.

In this section we'll also take a brief look—which is all that's needed—at the user profile information available through WinRT APIs, along with the API for encryption and decryption. Beyond this, I'll mention two other resources on the subject. The first is [How to secure connections and authenticate requests](#); the second is the [Banking with strong authentication sample](#), which demonstrates secure authentication and communication over the Internet. A full writeup on this sample is found on [Tailored banking app code walkthrough](#), so we won't be specifically looking at it here.

Design tip There are a number of design guidelines for different login scenarios, such as when an app requires a login to be useful and when a login is simply optional. These topics as well as where to place login and account/profile management UI are discussed in [Guidelines and checklist for login controls](#).

The Credential Picker UI

Just as WinRT provides a built-in UI for picking files, it also has a built-in UI for entering credentials: [Windows.Security.Credentials.UI.CredentialsPicker](#). This is provided as a convenience; again, you're free to implement your own UI if that works better for your app, but many features make the credential picker attractive.

When you instantiate this object and call its [pickAsync](#) method, as with the [Credential Picker sample](#), you'll see the UI shown in Figure 14-2. This UI provides for domain logins, supports smart cards (as you can see—I have two smart card readers on my machine), and it allows for various options such as authentication protocols and automatic saving of the credential (see the next section).

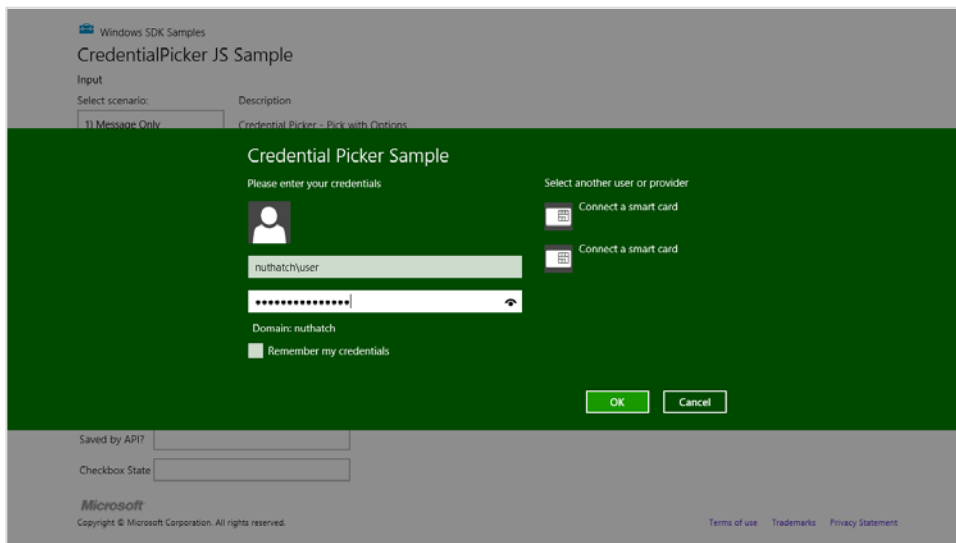


FIGURE 14-2 The credential picker UI appears like a message dialog over the app.

The result from `pickAsync`, as given to your completed handler, is a [CredentialPickerResults](#) object with the following properties—when you enter some credentials in the sample here, you’ll see these values reflected in the sample’s output:

- `credentialUserName` A string containing the entered username.
- `credentialPassword` A string containing the password (typically encrypted depending on the authentication protocol option).
- `credentialDomainName` A string containing a domain if entered with the username (as in `<domain>\<username>`).
- `credentialSaved` A Boolean indicating whether the credential was saved automatically; this depends on picker options, as discussed below.
- `credentialSavedOption` A [CredentialSavedOption](#) value indicating the state of the Remember My Credentials check box: `unselected`, `selected`, or `hidden`. We’ll see how to handle this shortly as well.
- `errorCode` Contains zero if there is no error, otherwise an error code.
- `credential` An `IBuffer` containing the credential as an opaque byte array. This is what you can save in your own persistent state if needs be and passed back to the picker at a later time; we’ll see how shortly.

The three scenarios in the sample demonstrate the different options you can use to invoke the credential picker. For this there are three separate variants of `pickAsync`. The first variant accepts a target name (ignored) and a message string that appears in the place of “Please enter your credentials” in Figure 14-2:

```
Windows.Security.Credentials.UI.CredentialPicker.pickAsync(targetName, message)
    .done(function (results) {
    })
```

The second variant accepts the same arguments plus a caption string that appears in the place of “Credential Picker Sample” in Figure 14-2:

```
Windows.Security.Credentials.UI.CredentialPicker.pickAsync(targetName, message, caption)
    .done(function (results) {
    })
```

The third variant accepts a [CredentialPickerOptions](#) object that has properties for the same `targetName`, `message`, and `caption` strings along with the following:

- `previousCredential` An `IBuffer` with the opaque credential information as provided by a previous invocation of the picker (see [CredentialPickerResults.credential](#) above).
- `alwaysDisplayDialog` A Boolean indicating whether the dialog box is displayed. The default value is `false`, but this applies only if you also populate `previousCredential` (with an exception

for domain-joined machines—see table below). The purpose here is to show the dialog when a stored credential might be incorrect and the user is expected to provide a new one.

- **errorCode** The numerical value of a [Win32 error code](#) (default is ERROR_SUCCESS) that will be formatted and displayed in the dialog box. You would use this when you obtain credentials from the picker initially but find that those credentials don't work and need to invoke the dialog again. Instead of providing your own message, you just choose an error code and let the system do the rest. The most common values for this are 1326 (login failure), 1330 (password expired), 2202 (bad username), 1907 or 1938 (password must change/password change required), 1351 (can't access domain info), and 1355 (no such domain). There are, in fact, over 15,000 Win32 error codes, but that means you'll have to search the reference linked here (or search within the winerror.h file typically found in your *Program Files (x86)\Windows Kits\8.0\Include\shared* folder). Happy hunting!
- **callerSavesCredential** A Boolean indicating that the app will save the credential and that the picker should not. The default value is `false`. When set to `true`, credentials are saved to a secure system location (not the credential locker) if the app has the *Enterprise Authentication* capability (see below).
- **credentialSaveOption** A value from the [CredentialSaveOption](#) enumeration indicating the initial state of the Remember My Credentials check box: `unselected`, `selected`, or `hidden`.
- **authenticationProtocol** A value from the [AuthenticationProtocol](#) enumeration: `basic`, `digest`, `ntlm`, `kerberos`, `negotiate` (the default), `credSsp`, and `custom` (in which case you must supply a string in the `customAuthenticationProtocol` property). Note that with `basic` and `digest`, the `CredentialPickerResults.credentialPassword` will *not* be encrypted and is subject to the same security needs as a plain text password you collect from your own UI.

Here's an example of invoking the picker with an `errorCode` indicating a previous failed login:

```
var options = new Windows.Security.Credentials.UI.CredentialPickerOptions();
options.message = "Please enter your credentials";
options.caption = "Sample App";
options.targetName = "Target";
options.alwaysDisplayDialog = true;
options.errorCode = 1326; // Shows "The username or password is incorrect."
options.callerSavesCredential = true;
options.authenticationProtocol =
    Windows.Security.Credentials.UI.AuthenticationProtocol.negotiate;
options.credentialSaveOption = Windows.Security.Credentials.UI.CredentialSaveOption.selected;

Windows.Security.Credentials.UI.CredentialPicker.pickAsync(options)
    .done(function (results) {
    })
```

To clarify the relationship between the `callerSavesCredential`, `credentialSaveOption`, and the `credentialSaved` properties, the following table enumerates the possibilities:

Enterprise Auth capability	callerSavesCredential	credentialSaveOption	Credential Picker saves credentials	Apps saves credentials to credential locker
No	true	Selected	No	Yes
		unselected or hidden	No	No
	false	Selected	No	Yes
		unselected or hidden	No	No
Yes	true	Selected	No	Yes
		unselected or hidden	No	No
	false	Selected	Yes (<code>credentialSaved</code> will be true)	Optional
		unselected or hidden	No	No

The first column refers to the *Enterprise Authentication* capability in the app's manifest, which indicates that the app can work with Intranet resources that require domain credentials (assuming that the app is running on the Enterprise Edition of Windows 8 as well). In such cases the credential picker has a separate secure location (apart from the credential locker) in which to store credentials, so the app need not save them itself. Furthermore, if the picker saves a credential and the app invokes the picker with `alwaysDisplayDialog` set to `false`, `previousCredential` can be empty because the credential will be loaded automatically. But without a domain-joined machine and this capability, the app must supply a `previousCredential` to avoid having the picker appear.

The Credential Locker

One of the reasons that apps might repeatedly ask a user for credentials is simply because they don't have a truly secure place to store and retrieve those credentials that's also isolated from all other apps. This is entirely the purpose of the credential locker, a function that's also immediately clear from the name of this particular API: [Windows.Security.Credentials.PasswordVault](#).

With the locker, any given credential itself is represented by a [Windows.Security.Credentials.PasswordCredential](#) object, as we saw briefly with the background transfer API. You can create an initialized credential as follows:

```
var cred = new Windows.Security.Credentials.PasswordCredential(resource, userName, password);
```

Another option is to create an uninitialized credential and populate its properties individually:

```
var cred = new Windows.Security.Credentials.PasswordCredential();
cred.resource = "userLogin";
cred.userName = "username";
cred.password = "password";
```

A credential object also contains an [IPropertySet](#) value named [properties](#), through which the same information can be managed.

In any case, when you collect credentials from a user and want to save them, create a [PasswordCredential](#) and pass it to [PasswordVault.add](#):

```
var vault = new Windows.Security.Credentials.PasswordVault();
vault.add(cred);
```

Note that if you add a credential to the locker with a [resource](#) and [userName](#) that already exist, the new credential will replace the old. And if at any point you want to delete a credential from the locker, call the [PasswordVault.remove](#) method with that credential.

Furthermore, even though a [PasswordCredential](#) object sees the world in terms of usernames and passwords, that password can be anything else you need to store securely, such as an access token. As we'll see in the next section, authentication through OAuth providers might return such a token, in which case you might store something like "Facebook_Token" in the credential's [resource](#) property, your app name in [userName](#), and the token in [password](#). This is a perfectly legitimate and expected use.

Once a credential is in the locker, it will remain there for subsequent launches of the app until you call the [remove](#) method or the user explicitly deletes it through Control Panel > User Accounts and Family Safety > Credential Manager. On a trusted PC (which requires sign-in with a Microsoft account), Windows will also automatically and securely roam the contents of the locker to the user's other devices (which can be turned off in PC Settings > Sync Your Settings > Passwords). This help to create a seamless experience with your app as the user moves between devices.⁷⁴

So, when you launch an app—even when launching it for the first time—always check if the locker contains saved credentials. There are several methods of the [PasswordVault](#) class for doing this:

- [findAllByResource](#) Returns an array (vector) of credential objects for a given resource identifier. This is how you can obtain the username and password that's been roamed from another device, because the app would have stored those credentials in the locker on the other machine under the same resource.
- [findAllByUserName](#) Returns an array (vector) of credential objects for a given username. This is useful if you know the username and want to retrieve all the credentials for multiple resources that the app connects to.
- [retrieve](#) Returns a single credential given a resource identifier and a username. Again, there will only ever be a single credential in the locker for any given resource and username.
- [retrieveAll](#) Returns a vector of all credentials in the locker for this app. The vector contains a snapshot of the locker and will not be updated with later changes to credentials in the locker.

⁷⁴ Such roaming will not happen, however, if a credential is *first* stored in the locker on a domain joined machine. This protects domain credentials from leaking to the cloud.

There is one subtle difference between the `findAll` and `retrieve` methods in the list above. The `retrieve` method will provide you with fully populated credentials objects. The `findAll` methods, on the other hand, will give you objects in which the `password` properties are still empty. This avoids performing password decryption on what is potentially a large number of credentials. To populate that property for any individual credential, call the `PasswordCredential.retrievePassword` method.

For further demonstrations of the credential locker—the code is very straightforward—refer to the [Credential locker sample](#). This shows variations for single user/single resource (Scenario 1), single user/multiple resources (Scenario 2), multiple users/multiple resources (Scenario 3), and clearing out the locker entirely (Scenario 4).

The Web Authentication Broker

Although apps can acquire and manage user credentials of their own, supplying users perhaps with the ability to create app-specific or service-specific accounts (typically through the Settings charm, as discussed in Chapter 8 and also on [Guidelines and checklist for login controls](#)), you might want to simply leverage an account that the user has already created through another OAuth provider, especially when you want to use that provider's resources. You've likely experienced this on many websites already, where you log in through another site like Facebook. Of course, that typically means navigating away from the original website to the provider's site—a process that flows well enough in a web browser but isn't quite so attractive in the context of an app!

For this purpose, Windows provides the Web Authentication Broker, which essentially does the same job without leaving the context of the app itself. An app provides the URI of the authenticating page of the external site (which must use the `https://` URI scheme, otherwise you get an invalid parameter error). The broker then creates a new web host process in its own app container, into which it loads the indicated web page. The UI for that process is displayed as an overlay dialog on the app, as shown in Figure 14-3, for which I'm using Scenario 1 of the [Web authentication broker sample](#).

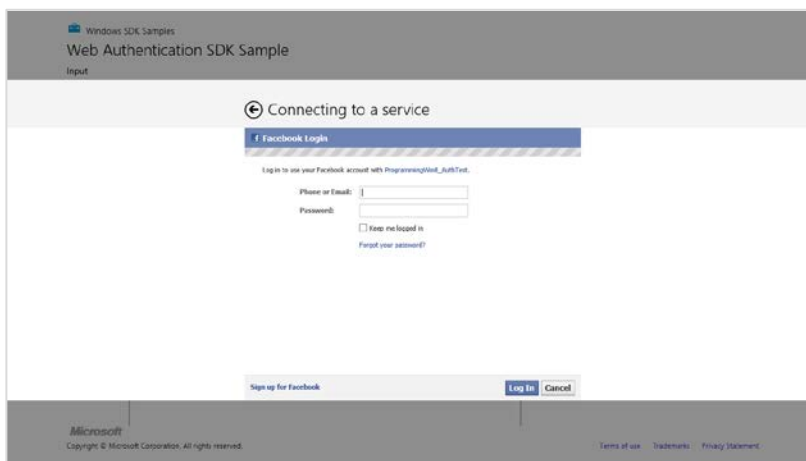


FIGURE 14-3 The Web authentication broker sample using a Facebook login page.

Note To run the sample you'll need an app ID for each of the authentication providers in the various scenarios. For Facebook in Scenario 1, visit <http://developers.facebook.com/setup> and create an App ID/API Key for a test app.

In the case of Facebook, the authentication process involves more than just checking the user's credentials. It also needs to obtain permission for other capabilities that the app wants to use (which the user might have independently revoked directly through Facebook). As a result, the authentication process might navigate to additional pages, each of which still appears within the web authentication broker, as shown in Figure 14-4. In this case the app identity, *ProgrammingWin8_AuthTest*, is just one that I created through the Facebook developer setup page for the purposes of this demonstration.

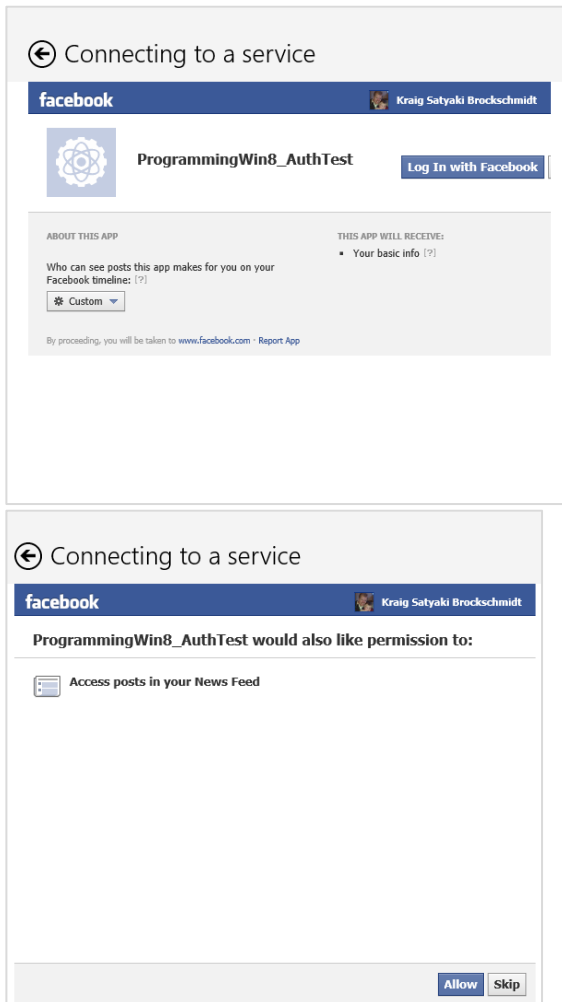


FIGURE 14-4 Additional authentication steps for Facebook within the web authentication broker.

Within the web authentication broker UI, the user might be taken through multiple pages on the provider's site (but note that the back button next to the "Connecting to a service" title dismisses the dialog entirely). But this begs a question: how does the broker know when authentication is actually complete? On the right side of Figure 14-4, clicking the Allow button is the last step in the process, after which Facebook would normally show a login success page. In the context of an app, however, we don't need that page to appear—we'd rather have the broker's UI taken down so that we return to the app with the results of the authentication. What's more, many OAuth providers don't even have such a page—so what do we do?

Fortunately, the broker takes this into account. As we'll see in a moment, the app simply provides the URI of that final page of the provider's process. When the broker detects that it's navigated to that page, it removes its UI and gives the response to the app.

As part of this process, Facebook saves these various permissions in its own back end for each particular user, so even if the app started the authentication process again, the user would not see the same pages shown in Figure 14-4. The user can, of course, manage these permissions when visiting Facebook through a web browser; if the user deletes the app information there, these additional authentication steps would reappear.

In WinRT, the broker is represented by the [Windows.Security.Authentication.Web.WebAuthenticationBroker](#) class. Authentication happens through its `authenticateAsync` methods. I say "methods" here because there are two variations. We'll look at one here and return to the second in the next section, "Single Sign On."

This first variant of `authenticateAsync` method takes three arguments:

- `options` Any combination of values from the [WebAuthenticationOptions](#) enumeration (combined with bitwise OR). Values are `none` (the default), `silentMode` (no UI is shown), `useTitle` (returns the window title of the webpage in the results), `useHttpPost` (returns the body of the page with the results), and `useCorporateNetwork` (to render the web page in an app container with the *Private Networks (Client & Server)*, *Enterprise Authentication*, and *Shared User Certificates* capabilities; the app must have also declared these).
- `requestUri` The URI ([Windows.Foundation.Uri](#)) for the provider's authentication page along with the parameters required by the service; again, this must use the `https://` URI scheme.
- `callbackUri` The URI ([Windows.Foundation.Uri](#)) of the provider's final page in its authentication process. The broker again uses this to determine when to take down its UI.⁷⁵

⁷⁵ As described on [How the web authentication broker works](#), `requestUri` and `callbackUri` "correspond to an Authorization Endpoint URI and Redirection URI in the OAuth 2.0 protocol. The OpenID protocol and earlier versions of OAuth have similar concepts."

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

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

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

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

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

This token is what you then save in the credential locker for later use when the app is relaunched after being closed or terminated. (With Facebook, you don't need to worry about the expiration of that token because the API generally reports that as an error and has a built-in renewal process.) You'd do something similar with other authentication providers, referring, of course, to their particular documentation on what information you'll receive with the response and how to use and/or renew keys or tokens when necessary.

All in all, a key benefit to web authentication is that the user never actually gives credentials to an app—the user gives them only to a much more trusted provider. From the app's point of view as well, it never needs to ask for or manage those credentials, only the tokens returned by the provider. For this same reason, invoking the broker as we've seen here will always show the login page with blank fields, irrespective of the Keep Me Logged In check box, because the calling app doesn't retain any of that information, and any cookies and session state created within the broker's hosting environment will have been discarded. So, if the app wants to have the user log in again with different credentials, it would just invoke the broker as before and replace whatever tokens or keys it saved from the last authentication.

Speaking of providers, the [OAuth page on Wikipedia](#) lists current authentication providers. The Web authentication broker sample, for its part, shows how to work specifically with Facebook (Scenario 1), Twitter (Scenario 2), Flickr (Scenario 3), and Google/Picasa (Scenario 4), and it also provides a generic interface for any other service (Scenario 5).

It's instructive to look through these various scenarios. Because Facebook and Google use the OAuth 2.0 protocol, the `requestUri` for each is relatively simple (ignore the word wrapping):

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

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

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

Tip Web authentication events are visible in the Event Viewer under *Application and Services Logs > Microsoft > Windows > WebAuth > Operational*. This can be helpful for debugging because it brings out information that is otherwise hidden behind the opaque layer of the broker.

Single Sign On

What we've seen so far with the credential locker and the web authentication broker works very well to minimize how often the app needs to pester the user for credentials. Where a single app is concerned, it would ideally only ask for credentials once until such time as the user explicitly logs out. But what about multiple apps? Imagine over time that you acquire some dozens, or even hundreds, of apps from the Windows Store that use external authentication providers. It could mean that you'd have to enter your Facebook, Twitter, Google, LinkedIn, Tumblr, Yahoo, or Yammer credentials in each app that uses them. Sure, you might need to do that only once in each individual app, but the compound effect is still tedious and annoying!

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

The concept of *single sign on* is exactly this: authenticating the user in one app (or the system in the case of a Microsoft account) effectively logs the user in to other apps that use the same provider. At the same time, each app must often acquire its own access keys or tokens, because these should not be shared between apps. So the real trick is to effectively perform the same kind of authentication we've already seen, only to do it without showing any UI unless it's really necessary.

This is provided for in the web authentication broker through the variation of `authenticateAsync` that takes only the `options` and `requestUri` arguments. In this case `options` is often set to `Web-AuthenticationOptions.silentMode` to prevent the broker's UI from appearing (but this isn't required).

To make `silentMode` work the broker still needs to know when the process is complete. So what `callbackUri` does it use for comparison, and how does the provider know that itself? It sounds like a situation where the broker would just sit there, forever hidden, while the provider patiently waits for input to a web page that's equally invisible! What actually happens is that `authenticateAsync` watches

for the provider to navigate to a special *callbackUri* that looks something like *ms-app://<app_package>/<secret_sauce>*, at which point it will pass the provider's response data as the async result.

Of course, that URI won't mean a thing to the provider...*unless* it's told about it beforehand and is expecting such a URI to appear in its midst.

This brings us to the fact that single sign on will work only if a provider has a means (an API or such) through which an app can communicate its intent along these lines. To understand this, let's follow the entire flow of the silent authentication process:

1. An app that wants to use single sign on obtains its particular *ms-app://* URI—also called an SID URI—through one of two means. First is by calling the static method `WebAuthenticationBroker.GetCurrentApplicationCallbackUri`. This returns a `Windows.Foundation.Uri` object whose `absoluteUri` property is the string you need. The second means is through the Windows Store Dashboard > Manage Your Cloud Services > Advanced Features > Application Authentication page, where you should see a string that looks like this:
ms-app://s-1-15-2-477157379-2961032073-432767880-3229792171-202870256-1369967874-2241987394/.
2. If necessary, the app then calls the provider's API to register the SID URI (typically a provider will have a page to define an app where you'd enter this).
3. When constructing the `requestUri` argument for `authenticateAsync`, the app inserts its SID URI as the value of the `&redirect_uri=` parameter.
4. The app calls `authenticateAsync` with the `silentMode` option.
5. When the provider processes the `requestUri` parameters, it checks whether the `redirect_uri` value has been registered, responding with a failure if it hasn't.
6. Having validated the app, the provider then silently authenticates (if possible) and navigates to the `redirect_uri`, making sure to include things like access keys and tokens in the response data.
7. The web authentication broker will detect this navigation and match it to its special *callbackUri*. Finding a match, the broker can complete the async operation and provide the response data to the app.

Again, the provider must have a way for the developer or app to register its SID URI, must check that SID URI when it appears in an authentication request, and must write appropriate response data to that page when authentication is complete. The developer or app is then responsible for registering that SID URI in the first place and including it in the `requestUri`. (Whew, that's a lot of URIs!)

With all of this, it's still possible that the authentication might fail for some other reason. For example, if the user has not set up permissions for the app in question (as with Facebook), it's not possible to silently authenticate. So, an app attempting to use single sign on would call this form of `authenticateAsync` first and, failing that, would then revert to calling its longer form (with UI), as described in the previous section.

Single Sign On with Live Connect

Because various Microsoft services, such as Hotmail, are OAuth providers, it is possible to use the web authentication broker with a Microsoft account (such as Hotmail, Live, and MSN accounts). (I still have the same @msn.com email account I've had since 1996!) Details can be found on the [OAuth 2.0 page](#) on the Live Connect Developer Center.

However, Live Connect accounts are in a somewhat more privileged position because they can also be used to sign in to Windows or can be connected to a domain account used for the same purpose. Many of the built-in apps such as Mail, Calendar, SkyDrive, People, and for that matter the Windows Store itself work with this same account. Thus, it's something that many other apps might want to take advantage of as well, because such authentication provides access to the same Live services data that those built-in apps draw from themselves.

The Live Services API for signing in this way is called `WL.login`, which is available when you install the Live SDK and add the appropriate references to your project. To get started with that process, visit the [Live Connect documentation](#) and check out the following references:

- [Live Connect \(Windows Store apps\) home page](#)
- [Live Connect Developer Center \(Windows Store Apps\)](#)
- [Guidelines for single sign-on and connected accounts](#)
- [Guidelines for the Microsoft account sign-in experience](#)
- [Single sign-on with Microsoft accounts](#)
- [Quickstart: Accessing Live services data](#)
- [Windows account authorization sample](#)
- [Bring single sign-on and SkyDrive to your Windows 8 apps with the Live SDK](#) and [Best Practices when adding single sign-on to your app with the Live SDK](#) on the Windows 8 Developer Blog.

As you can imagine, working with Live Services is an extensive topic, so I'll defer to the resources above. One key point, though, is that it's possible for a user to log in to Windows with a domain account that has not been connected to a Microsoft account through PC Settings > Users. In this case, the first call to `WL.login` from any app will display the Microsoft account login dialog, as shown in Figure 14-5. Once the user enters credentials here, they're logged in to all other apps that use the Microsoft account.

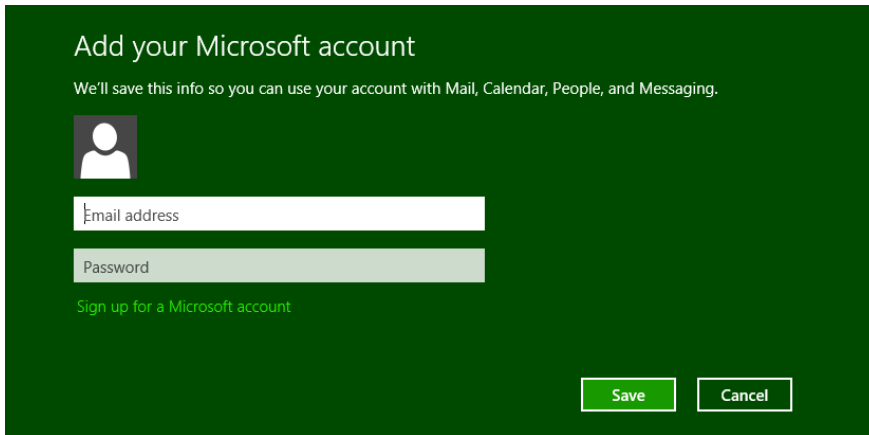


FIGURE 14-5 The Microsoft account login dialog.

The User Profile (and the Lock Screen Image)

Any discussion about user credentials brings up the question of accessing additional user information. What is available to Windows Store apps is provided through the [Windows.System.UserProfile](#) API, where we find three classes of interest.

The first is the [LockScreen](#) class through which you can get or set the lock screen image (and nothing more). The image is available through the [originalImageFile](#) property (returning a [StorageFile](#)) and the [getImageStream](#) method (returning an [IRandomAccessStream](#)). Setting the image can be accomplished through [setImageFileAsync](#) (using a [StorageFile](#)) and [setImageStreamAsync](#) (using an [IRandomAccessStream](#)). This would be utilized in a photo app that has a command to use a picture for the lock screen. See the [Lock screen personalization sample](#) for a demonstration.

The second is the [GlobalizationPreferences](#) object, which we'll return to in Chapter 17, "Apps for Everyone."

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

- **User name** If the [nameAccessAllowed](#) property is [true](#), an app can then call [getDisplaynameAsync](#), [getFirstNameAsync](#), and [getLastNameAsync](#), all of which provide a string to your completed handler. If [nameAccessAllowed](#) is false, these methods will complete but provide an empty result. Also note that the first and last names are available only from a Microsoft account.
- **User picture** Retrieved through [getAccountPicture](#), which returns a [StorageFile](#) for the image. The method takes a value from [AccountPictureKind](#): [smallImage](#), [largeImage](#), and [video](#).
- If the [accountPictureChangeEnabled](#) property is [true](#), you can use one of four methods to set the image(s): [setAccountPictureAsync](#) (for providing one image from a [StorageFile](#)),

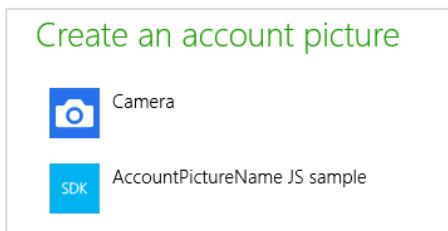
`setAccountPicturesAsync` (for providing small and large images as well as a video from `StorageFile` objects), and `setAccountPictureFromStreamAsync` and `setAccount-PicturesFromStreamAsync` (which do the same given `IRandomAccessStream` objects instead). In each case the async result is a `SetAccountPictureResult` value: `success`, `failure`, `changeDisabled` (`accountPictureChangeEnabled` is `false`), `large-OrDynamicError` (the picture is too large), `fileSizeError` (file is too large), or `video-FrameSizeError` (video frame size is too large),

- The `accountpicturechanged` event signals when the user picture(s) have been altered. Remember that because this event originates within WinRT you should call `removeEvent-Listener` if you aren't listening for this event for the lifetime of the app.

These features are demonstrated in the [Account picture name sample](#). Scenario 1 retrieves the display name, Scenario 2 retrieves the first and last name (if available), Scenario 3 retrieves the account pictures and video, and Scenario 4 changes the account pictures and video and listens for picture changes.

Tip To obtain the profile picture from Live Connect, the exact API call is as follows:
`https://apis.live.net/v5.0/me/picture?access_token=<ACCESS_TOKEN>`.

One other bit that this sample demonstrates is the Account Picture Provider declaration in its manifest, which causes the app to appear within PC Settings > Personalize under Create an Account Picture:



In this case the sample doesn't actually provide a picture directly but activates into Scenario 4. A real app, like the Camera app that's also in PC Settings by default, will automatically set the account picture when one is acquired through its UI. How does it know to do this? The answer lies in a special URI scheme through which the app is activated. That is, when you declare the Account Picture Provider declaration in the manifest, the app will be activated with the activation kind of `protocol` (see Chapter 12, "Contracts"), where the URI scheme specifically starts with `ms-accountpictureprovider`. You can see how this is handled in the sample's `js/default.js` file:

```
if (eventObject.detail.kind === Windows.ApplicationModel.Activation.ActivationKind.protocol) {  
    // Check if the protocol matches the "ms-accountpictureprovider" scheme  
    if (eventObject.detail.uri.schemeName === "ms-accountpictureprovider") {
```

```

    // This app was activated via the Account picture apps section in PC Settings.
    // Here you would do app-specific logic for providing the user with account
    // picture selection UX
}

```

Returning to the [UserInformation](#) class, it also provides a few more details for domain accounts provided that the app has declared the *Enterprise Authentication* capability in its manifest:

- [getDomainNameAsync](#) Provides the user's fully qualified domain name as a string in the form of <domain>\<user> where <domain> is the full name of the domain controller, such as *mydomain.corp.ourcompany.com*.
- [getPrincipalNameAsync](#) Provides the principal name as a string. In Active Directory parlance, this is an Internet-style login name (known as a user principal name or UPN) that is shorter and simpler than the domain name, consolidating the email and login namespaces. Typically, this is an email address like *user@ourcompany.com*.
- [getSessionInitiationProtocolUriAsync](#) Provides a *session initiation protocol URI* that will connect with this user; for background, see [Session Initiation Protocol](#) (Wikipedia).

The use of these methods is demonstrated in the [User domain name sample](#).

Encryption, Decryption, Data Protection, and Certificates

Authorization and credentials are a security matter, so it's appropriate to end this section with a quick rundown of the other APIs grouped under the [Windows.Security](#) namespace, where we found the web authentication broker already.

First is [Windows.Security.Cryptography](#). Here you'll find the [CryptographicBuffer](#) class that can encode and decode strings in hexadecimal and base64 (UTF-8 or UTF-16) and also provide random numbers and a byte array full of such randomness. Refer to Scenario 1 of the [CryptoWinRT sample](#) for some demonstrations, as well as Scenarios 2 and 3 of the Web authentication broker sample. WinRT's base64 encoding is fully compatible with the JavaScript [atob](#) and [btoa](#) functions.

Next is [Windows.Security.Cryptography.Core](#), which is truly about encryption and decryption according to various algorithms. See the [Encryption](#) topic, Scenarios 2-8 of the [CryptoWinRT sample](#), and again Scenarios 2 and 3 of the Web authentication broker sample.

Third is [Windows.Security.Cryptography.DataProtection](#), whose single class, [Data-ProtectionProvider](#), deals with protecting and unprotecting both static data and a data stream. This applies only to apps that declare the *Enterprise Authentication* capability. For details, refer to [Data protection API](#) along with Scenarios 9 and 10 of the [CryptoWinRT sample](#).

Fourth, [Windows.Security.Cryptography.Certificates](#) provides several classes through which you can create certificate requests and install certificate responses. Refer to [Working with certificates](#) and the [Certificate enrollment sample](#) for more.

And lastly it's worth at least listing the API under [Windows.Security.ExchangeActive-SyncProvisioning](#) for which there is the [EAS policies for mail clients sample](#). I'm assuming that if you know why you'd want to look into this, well, you'll know!

Syndication

When we first looked at doing XMLHttpRequests with `WinJS.XHR` in Chapter 3, we grabbed the RSS feed from the Windows 8 Developer Blog with the URI <http://blogs.msdn.com/b/windowsappdev/rss.aspx>. We learned then that `WinJS.xhr` returned a promise, the result of which contained a `responseXML` property, which is itself a `DomParser` through which you can traverse the DOM structure and so forth.

Working with syndicated feeds like this is completely supported for Windows Store apps. In fact, the [How to create a mashup topic](#) in the documentation describes exactly this process, components of which are demonstrated in the [Integrating content and controls from web services sample](#).

That said, WinRT offers additional APIs for dealing with syndicated content. One, `Windows.Web.Syndication`, offers a more structured way to work with RSS feeds. The other, `Windows.Web.AtomPub`, provides a means to publish and manage feed entries. Both are provided in WinRT for languages that don't have another means of accomplishing the same ends, but as a developer working JavaScript, you have the choice.

Reading RSS Feeds

The primary class within `Windows.Web.Syndication` is the `SyndicationClient`. To work with any given feed, you create an instance of this class and set any necessary properties. These are `serverCredential` (a `PasswordCredential`), `proxyCredential` (another `PasswordCredential`), `timeout` (in milliseconds, default is 30000 or 30 seconds), `maxResponseBufferSize` (a means to protect from potentially malicious servers), and `bypassCacheOnRetrieve` (a Boolean to indicate whether to always obtain new data from the server). You can also make as many calls to its `setRequestHeader` method (passing a name and value) to configure the XMLHttpRequest header.

The final step is to then call the `SyndicationClient.retrieveFeedAsync` method with the URI of the desired RSS feed (a `Windows.Foundation.Uri`). Here's an example derived from the [Syndication sample](#), which retrieves [the RSS feed for the Building Windows 8 blog](#):

```
uri = new Windows.Foundation.Uri("http://blogs.msdn.com/b/b8/rss.aspx");
var client = new Windows.Web.Syndication.SyndicationClient();
client.bypassCacheOnRetrieve = true;
client.setRequestHeader("User-Agent",
    "Mozilla/5.0 (compatible; MSIE 10.0; Windows NT 6.2; WOW64; Trident/6.0)");

client.retrieveFeedAsync(uri).done(function (feed) {
    // feed is a SyndicationFeed object
})
```

The result of `retrieveFeedAsync` is a [Windows.Web.Syndication.SyndicationFeed](#) object; that is, the `SyndicationClient` is what you use to talk to the service, and when you retrieve the feed you get an object through which you can then process the feed itself. If you take a look at `SyndicationFeed` using the link above, you'll see that it's wholly stocked with properties that represent all the parts of the feed, such as `authors`, `categories`, `items`, `title`, and so forth. Some of these are represented themselves by other classes in `Windows.Web.Syndication`, or collections of them, where simpler types aren't sufficient: `SyndicationAttribute`, `SyndicationCategory`, `SyndicationContent`, `SyndicationGenerator`, `SyndicationItem`, `SyndicationLink`, `SyndicationNode`, `SyndicationPerson`, and `SyndicationText`. I'll leave the many details to the documentation.

We can see some of this in the sample, picking up from inside the completed handler for `retrieveFeedAsync`. Let me offer a more annotated version of that code:

```
client.retrieveFeedAsync(uri).done(function (feed) {
    currentFeed = feed;

    var title = "(no title)";

    // currentFeed.title is a SyndicationText object
    if (currentFeed.title) {
        title = currentFeed.title.text;
    }

    // currentFeed.items is a SyndicationItem collection (array)
    currentItemIndex = 0;
    if (currentFeed.items.size > 0) {
        displayCurrentItem();
    }
}

// ...

function displayCurrentItem() {
    // item will be a SyndicationItem

    var item = currentFeed.items[currentItemIndex];

    // Display item number.
    document.getElementById("scenario1Index").innerText = (currentItemIndex + 1) + " of "
        + currentFeed.items.size;

    // Display title (item.title is another SyndicationText).
    var title = "(no title)";
    if (item.title) {
        title = item.title.text;
    }
    document.getElementById("scenario1ItemTitle").innerText = title;

    // Display the main link (item.links is a collection of SyndicationLink objects).
    var link = "";
    if (item.links.size > 0) {
        link = item.links[0].uri.absoluteUri;
    }
}
```

```

}

var scenario1Link = document.getElementById("scenario1Link");
scenario1Link.innerText = link;
scenario1Link.href = link;

// Display the body as HTML (item.content is a SyndicationContent object, item.summary is
// a SyndicationText object).
var content = "(no content)";
if (item.content) {
    content = item.content.text;
}
else if (item.summary) {
    content = item.summary.text;
}
document.getElementById("scenario1WebView").innerHTML = window.toStaticHTML(content);

// Display element extensions. The elementExtensions collection contains all the additional
// child elements within the current element that do not belong to the Atom or RSS standards
// (e.g., Dublin Core extension elements). By creating an array of these, we can create a
// WinJS.Binding.List that's easily displayed in a ListView.
var bindableNodes = [];
for (var i = 0; i < item.elementExtensions.size; i++) {
    var bindableNode = {
        nodeName: item.elementExtensions[i].nodeName,
        nodeNamespace: item.elementExtensions[i].nodeNamespace,
        nodeValue: item.elementExtensions[i].nodeValue,
    };
    bindableNodes.push(bindableNode);
}
var dataList = new WinJS.Binding.List(bindableNodes);
var listView = document.getElementById("extensionsListView").winControl;
WinJS.UI.setOptions(listView, { itemDataSource: dataList.dataSource });
}

```

It's probably obvious that the API, under the covers, is probably just using the [XmlDocument](#) API to retrieve all the feed these properties. In fact, its [getXmlDocument](#) returns that [XmlDocument](#) if you want to access it yourself.

You can also create a [SyndicationFeed](#) object around the XML for a feed you might already have. For example, if you obtain the feed contents by using [WinJS.xhr](#), you can create a new [SyndicationFeed](#) object and call its [load](#) method with the XHR [responseXML](#). Then you can work with the feed through the class hierarchy. When using the [Windows.Web.AtomPub](#) API to manage a feed, you also create a new or updated [SyndicationItem](#) to send across the wire, settings its values through the other objects in its hierarchy. We'll see this shortly.

One last note: if [retrieveFeedAsync](#) throws an exception, which would be picked up by an error handler you provide to the promise's [done](#) method, you can turn the error code into a [SyndicationErrorStatus](#) value. Here's how it's used in the sample's error handler:

```
function onError(err) {
    // Match error number with a SyndicationErrorStatus value. Use
    // Windows.Web.WebErrorStatus.getStatus() to retrieve HTTP error status codes.
    var errorStatus = Windows.Web.Syndication.SyndicationError.getStatus(err.number);
    if (errorStatus === Windows.Web.Syndication.SyndicationErrorStatus.invalidXml) {
        displayLog("An invalid XML exception was thrown. Please make sure to use a URI that"
            + "points to a RSS or Atom feed.");
    }
}
```

Using AtomPub

On the flip side of reading an RSS feed, as we've just seen, is the need to possibly manage entries on a feed: adding, removing, and editing entries. This would be used for an app that lets the user maintain a specific blog, not just read entries from others.

The API for this is found in [Windows.Web.AtomPub](#) and demonstrated in the [AtomPub sample](#). The main class is the [AtomPubClient](#) that encapsulates all the operations of the AtomPub protocol. It has methods like [createResourceAsync](#), [retrieveResourceAsync](#), [updateResourceAsync](#), and [deleteResourceAsync](#) for working with those entries, where each resource is identified with a URI and a [SyndicationItem](#) object, as appropriate. Media resources for entries are managed through [createMediaResourceAsync](#) and similarly named methods, where the resource is provided as an [IInputStream](#).

The [AtomPubClient](#) also has [retrieveFeedAsync](#) and [setRequestHeader](#) methods that do the same as the [SyndicationClient](#) methods of the same names, along with a few similar properties like [serverCredential](#), [timeout](#), and [bypassCacheOnRetrieve](#). Another method, [retrieve-ServiceDocumentAsync](#), provides the workspaces/service documents for the feed (in the form of a [Windows.Web.AtomPub.ServiceDocument](#) object).

Again, the [AtomPub sample](#) demonstrates the different operations: retrieve (Scenario 1), create (Scenario 2), delete (Scenario 3), and update (Scenario 4). Here's how it first creates the [AtomPub-Client](#) object (see `js/common.js`), assuming there are credentials:

```
function createClient() {
    client = new Windows.Web.AtomPub.AtomPubClient();
    client.bypassCacheOnRetrieve = true;

    var credential = new Windows.Security.Credentials.PasswordCredential();
    credential.userName = document.getElementById("userNameField").value;
    credential.password = document.getElementById("passwordField").value;
    client.serverCredential = credential;
}
```

Updating an entry (`js/update.js`) then looks like this, where the update is represented by a newly created [SyndicationItem](#):

```
function getCurrentItem() {
    if (currentFeed) {
```

```

        return currentFeed.items[currentItemIndex];
    }
    return null;
}

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

var currentItem = getCurrentItem();

if (!currentItem) {
    return;
}

// Update the item
var updatedItem = new Windows.Web.Syndication.SyndicationItem();
var title = document.getElementById("titleField").value;
updatedItem.title = new Windows.Web.Syndication.SyndicationText(title,
    Windows.Web.Syndication.SyndicationTextType.text);
var content = document.getElementById("bodyField").value;
updatedItem.content = new Windows.Web.Syndication.SyndicationContent(content,
    Windows.Web.Syndication.SyndicationTextType.html);

client.updateResourceAsync(currentItem.editUri, updatedItem).done(function () {
    displayStatus("Updating item completed.");
}, onError);

```

Error handling in this case works with the [Window.Web.WebError](#) class (see js/common.js):

```

function onError(err) {
    displayError(err);

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

```

Sockets

Sockets are a fundamental network transport. Unlike HTTP requests, where a client sends a request to a server and the server responds—essentially an isolated transaction—sockets are a connection between client and server IP ports such that either one can send information to the other at any time. Certainly we’ve seen a mechanism like this earlier—namely, using the Windows Push Notification Service (WNS). WNS, however, is limited to notifications and is specifically designed to issue tile updates or notifications for apps that aren’t running. Sockets, on the other hand, are for data exchange between a server and a running client.

Sockets are generally used when there isn't a higher-level API or other abstraction for your particular scenario, when there's a custom protocol involved, when you need two-way communication, or when it makes sense to minimize the overhead of each exchange. Consider HTTP, a protocol that is itself built on lower-level sockets. A single HTTP request generally includes headers and lots of other information beyond just the bit of data involved, so it's an inefficient transport when you need to send lots of little bits. It's better to connect directly with the server and exchange data with a minimized custom protocol. VoIP is another example where sockets work well, as are multicast scenarios like multiplayer games. In the latter, one player's machine, acting as a server within a local subnet, can broadcast a message to all the other players, and vice versa, again with minimal overhead.

In the world of sockets, exchanging data can happen two ways: as discrete packets/messages (like water balloons) or as a continuous stream (like water running through a hose). These are called datagram and stream sockets, respectively, and both are supported through the WinRT API. WinRT also supports both forms of exchange through the WebSocket protocol, a technology originally created for web browsers and web servers that has become increasingly interesting for general purpose use within apps. All of the applicable classes can be found in the [Windows.Networking.Sockets](#) API, as we'll see in the following sections. Note that because there is some overlap between the different types of sockets, these sections are meant to be read in sequence so that I don't have to repeat myself too much!

Datagram Sockets

In the language of sockets, a water balloon is called a datagram, a bundle of information sent from one end of the socket to the other—even without a prior connection—according to the User Datagram Protocol (UDP) standard. UDP, as I summarize here from its [description on Wikipedia](#), is simple, stateless, unidirectional, and transaction-oriented. It has minimal overhead and lacks retransmission delays, and for these reasons it cannot guarantee that a datagram will actually be delivered. Thus, it's used where error checking and correction isn't necessary or is done by the apps involved rather than at the network interface level. In a VoIP scenario, for example, this allows data packets to just be dropped if they cannot be delivered, rather than having everything involved wait for a delayed packet. As a result, the quality of the audio might suffer, but it won't start stuttering or make your friends and colleagues sound like they're from another galaxy. In short, UDP might be unreliable, but it minimizes latency. Higher-level protocols like the Real-time Transport Protocol (RTP) and the Real Time Streaming Protocol (RTSP) are built on UDP.

A Windows Store app works with this transport—either as a client or a server—using the [Windows.Networking.Sockets.DatagramSocket](#) class, an object that you need to instantiate with the `new` operator to set up a specific connection and listen for messages:

```
var listener = new Windows.Networking.Sockets.DatagramSocket();
```

On either side of the conversation, the next step is to listen for the object's [messagereceived](#) event:

```
// Event from WinRT: remember to call removeEventListener as needed
listener.addEventListener("messagereceived", onMessageReceived);
```


When data arrives, the handler receives a—wait for it!—[DatagramSocketMessageReceived-EventArgs](#) object (that’s a mouthful). This contains [LocalAddress](#) and [remoteAddress](#) properties, both of which are a [Windows.Networking.HostName](#) that contains the IP address, a display name, and a few other bits. See the “Network Information (the Network Object Roster)” section earlier in this chapter for details. The event args also contains a [remotePort](#) string. More importantly, though, are the two methods through which you extract the data. One is [getDataStream](#), which returns an [IInputStream](#) through which you can read sequential bytes. The other is [getDataReader](#), which returns a [Windows.Storage.Streams.DataReader](#) object, a higher-level abstraction built on top of the [IInputStream](#) that helps you read specific data types directly. Clearly, if you know the data structure you expect to receive in the message, using the [DataReader](#) will relieve you from doing type conversions yourself.

Of course, to get any kind of data from a socket, you need to connect it to something. For this purpose there are a few methods in [DatagramSocket](#) for establishing and managing a connection:

- [connectAsync](#) Starts a connection operation given a [HostName](#) object and a service name (or UDP port, a string) of the remote network destination. This is used to create a one-way client to server connection.
- Another form of [connectAsync](#) takes a [Windows.Networking.EndpointPair](#) object that specifies host and service names for both local and remote endpoints. This is used to create a two-way client/server connection, as the local endpoint implies a call to [bindEndpointAsync](#) as below.
- [bindEndpointAsync](#) For a one-way server connection—that is, to only listen to but not send data on the socket—this method just binds a local endpoint given a [HostName](#) and a service name/port. Binding the service name by itself can be done with [bindServiceNameAsync](#).
- [joinMulticastGroup](#) Given a [HostName](#), connects the Datagram socket to a multicast group.
- [close](#) Terminates the connection and aborts any pending operations.

Tip To open a socket to a localhost port for debugging purposes, use [connectAsync](#) as follows:

```
var socket = new Windows.Networking.Sockets.DatagramSocket();
socket.connectAsync(new Windows.Networking.Sockets.DatagramSocket("localhost",
    "12345", Windows.Networking.Sockets.SocketProtectionLevel.plainSocket)
    .done(function () {
        // ...
    }, onError);
```

Note that any given socket can be connected to any number of endpoints—you can call [connect-Async](#) multiple times, join multiple multicast groups, and bind multiple local endpoints with [bindEnd-pointAsync](#) and [bindServiceNameAsync](#). The [close](#) method, mind you, closes everything at once!

Once the socket has one or more connections, connection information can be retrieved with the `DatagramSocket.information` property (a [DatagramSocketInformation](#)). Also, note that the static [DatagramSocket.getEndpointPairsAsync](#) method provides (as the async result) a vector of available [EndpointPair](#) objects for a given remote hostname and service name. You can optionally indicate that you'd like the endpoints sorted according to the [optimizeForLongConnections](#) flag. See the documentation page linked here for details, but it basically lets you control which endpoint is preferred over others based on whether you want to optimize for a high-quality and long-duration connection that might take longer to connect to initially (as for video streaming) or for connections that are easiest to acquire (the default).

Control data can also be set through the `DatagramSocket.control` property, a [Datagram-SocketControl](#) object with [qualityOfService](#) and [outputUnicastHopLimit](#) properties.

All this work, of course, is just a preamble to sending data on the socket connection. This is done through the `DatagramSocket.outputStream` property, an [IOutputStream](#) to which you can write whatever data you need using its `writeAsync` and `flushAsync` methods. This will send the data on every connection within the socket. Alternately, you can use one of the variations of [getOutput-StreamAsync](#) to specify a specific [EndpointPair](#) or `HostName`/port to which to send the data. The result of both of these async operations is again an [IOutputStream](#). And in all cases you can create a higher-level [DataWriter](#) object around that stream:

```
var dataWriter = new Windows.Storage.Streams.DataWriter(socket.outputStream)
```

Here's how it's all demonstrated in the [DatagramSocket sample](#), a little app in which you need to run each of the scenarios in turn. Scenario 1, for starters, sets up the server-side listener of the relationship on the localhost, using port number 22112 (the service name) by default. To do this, it creates the sockets, adds the listener, and calls [bindServiceNameAsync](#) (`js/startListener.js`):

```
socketsSample.listener = new Windows.Networking.Sockets.DatagramSocket();
// Reminder: call removeEventListener as needed; this can be common with socket relationships
// that can come and go through the lifetime of the app.
socketsSample.listener.addEventListener("messagereceived", onServerMessageReceived);

socketsSample.listener.bindServiceNameAsync(serviceName).done(function () {
    // ...
}, onError);
```

When a message is received, this server-side component takes the contents of the message and writes it to the socket's output stream so that it's reflected in the client side. This looks a little confusing in the code, so I'll show the core code path of this process with added comments:

```
function onServerMessageReceived(eventArgument) {
    // [Code here checks if we already got an output stream]

    socketsSample.listener.getOutputStreamAsync(eventArgument.remoteAddress,
        eventArgument.remotePort).done(function (outputStream) {
        // [Save the output stream with some other info, omitted]
        socketsSample.listenerOutputStream = outputStream;
    })
}
```

```

        // This is a helper function
        echoMessage(socketsSample.listenerOutputStream, eventArgument);
    });
}

// eventArgument here is a DatagramSocketMessageReceivedEventArgs with a getDataReader method function
echoMessage(outputStream, eventArgument) {
    // [Some display code omitted]

    // Get the message stream from the DataReader and send it to the output stream
    outputStream.writeAsync(eventArgument.getDataReader().detachBuffer()).done(function () {
        // Do nothing - client will print out a message when data is received.
    });
}
}

```

In most apps using sockets, the server side would do something more creative with the data than just send it back to the client! But this just changes what you do with the data in the input stream.

Scenario 2 sets up a listener to the localhost on the same port. On this side, we also create a [DatagramSocket](#) and set up a listener for `messagereceived`. Those messages—such as the one written to the output stream on the server side, as we’ve just seen—are picked up in the event handler below (`js/connectToListener.js`), which uses the [DataReader](#) to extract and display the message:

```

function onMessageReceived(eventArgument) {
    try {
        var messageLength = eventArgument.getDataReader().unconsumedBufferLength;
        var message = eventArgument.getDataReader().readString(messageLength);
        socketsSample.displayStatus("Client: receive message from server \"" + message + "\"");
    } catch (exception) {
        status = Windows.Networking.Sockets.SocketError.getStatus(exception.number);
        // [Display error details]
    }
}
}

```

Note in the code above that when an error occurs on a socket connection, you can pass the error number to the `getStatus` method of the [Windows.Networking.Sockets.SocketError](#) object and get back a more actionable [SocketErrorStatus](#) value. There are many possible errors here, so see its reference page for details.

Even with all the work we’ve done so far, nothing has yet happened because we’ve sent no data! So switching to Scenario 3, pressing its Send ‘Hello’ Now button does the honors from the client side (`js/sendData.js`):

```

// [This comes after a check on the socket's validity]
socketsSample.clientDataWriter =
    new Windows.Storage.Streams.DataWriter(socketsSample.clientSocket.outputStream);

var string = "Hello World";
socketsSample.clientDataWriter.writeString(string);

socketsSample.clientDataWriter.storeAsync().done(function () {

```

```
socketsSample.displayStatus("Client sent: " + string + ".");
}, onError);
```

The `DataWriter.storeAsync` call is what actually writes the data to the stream in the socket. If you set a breakpoint here and on both `messagereceived` event handlers, you'll then see that `storeAsync` generates a message to the server side, hitting `onServerMessageReceived` in `js/startListener.js`. This will then write the message back to the socket, which will hit `onMessage-Received` in `js/connectToListener.js`, which displays the message. (And to complete the process, Scenario 4 gives you a button to call the socket's `close` method.)

The sample does everything with the same app on localhost to make it easier to see how the process works. Typically, of course, the server will be running on another machine entirely, but the steps of setting up a listener apply just the same. As noted in Chapter 13, localhost connections work only on a machine with a developer license and will not work for apps acquired through the Windows Store.

Stream Sockets

In contrast to datagram sockets, streaming data over sockets uses the [Transmission Control Protocol](#) (TCP). The hallmark of TCP is accurate and reliable delivery—it guarantees that the bytes received are the same as the bytes that were sent: when a packet is sent across the network, TCP will attempt to retransmit the packet if there are problems along the way. This is why it's part of TCP/IP, which gives us the World Wide Web, email, file transfers, and lots more. HTTP, SMTP, and the Session Initiation Protocol (SIP) are also built on TCP. In all cases, clients and servers just see a nice reliable stream of data flowing from one end to the other.

Unlike datagram sockets, for which we have a single class in WinRT for both sides of the relationship, stream sockets are more distinctive to match the unique needs of the client and server roles. On the client side is [Windows.Networking.Sockets.StreamSocket](#); on the server it's [StreamSocketListener](#).

Starting with the latter, the `StreamSocketListener` object looks quite similar to the `DatagramSocket` we've just covered in the previous section, with these methods, properties, and events:

- **information** Provides a [StreamSocketListenerInformation](#) object containing a `localPort` string.
- **control** Provides a [StreamSocketListenerControl](#) object with a `qualityOfService` property.
- **connectionreceived** An event that's fired when a connection is made to the listener. Its event arguments are a [StreamSocketListenerConnectionReceivedEventArgs](#) that contains a single property, `socket`. This is the `StreamSocket` for the client, in which is an `outputStream` property where the listener can obtain the data stream.
- **bindEndpointAsync** and **bindServiceNameAsync** Binds the listener to a `HostName` and service name, or binds just a service name.
- **close** Terminates connections and aborts pending operations.

On the client side, `StreamSocket` again looks like parts of the `DatagramSocket`. In addition to the `control` (`StreamSocketControl`) and `information` properties (`StreamSocketInformation`) and the ubiquitous `close` method, we find a few other usual suspects and one unusual one:

- `connectAsync` Connects to a `HostName`/service name or to an `EndpointPair`. In each case you can also provide an optional `SocketProtectionLevel` object that can be `plainSocket`, `ssl`, or `sslAllowNullEncryption`. There are, in other words, four variations of this method.
- `inputStream` The `IInputStream` that's being received over the connection.
- `outputStream` The `IOutputStream` into which data is written.
- `upgradeToSslAsync` Upgrades a `plainSocket` connection (created through `connectAsync`) to use SSL as specified by either `SocketProtectionLevel.ssl` or `sslAllowNullEncryption`. This method also required a `HostName` that validates the connection.

For more details on using SSL, see [How to secure socket connections with TLS/SSL](#).

In any case, you can see that for one-way communications over TCP, an app creates either a `StreamSocket` or a `StreamSocketListener`, depending on its role. For two-way communications an app will create both.

The [StreamSocket sample](#), like the `DatagramSocket` sample, has four scenarios that are meant to be run in sequence on the localhost: first to create a listener (to receive a message from a client, Scenario 1), then to create the `StreamSocket` (Scenario 2) and send a message (Scenario 3), and then to close the socket (Scenario 4). With streamed data, the app implements a custom protocol for how the data should appear, as we'll see.

Starting in Scenario 1 (`js/startListener.js`), here's how we create the listener and event handler. Processing the incoming stream data is trickier than with a datagram because we need to make sure the data we need is all there. This code shows a good pattern of waiting for one async operation to finish before the function calls itself recursively. Also note how it creates a `DataReader` on the input stream for convenience:

```
socketsSample.listener = new Windows.Networking.Sockets.StreamSocketListener(serviceName);
// Match with removeEventListener as needed
socketsSample.listener.addEventListener("connectionreceived", onServerAccept);

socketsSample.listener.bindServiceNameAsync(serviceName).done(function () {
    // ...
}, onError);
}

// This has to be a real function; it will "loop" back on itself with the call to
// acceptAsync at the very end.
function onServerAccept(eventArgument) {
    socketsSample.serverSocket = eventArgument.socket;
    socketsSample.serverReader =
        new Windows.Storage.Streams.DataReader(socketsSample.serverSocket.inputStream);
    startServerRead();
}
```

```

}

// The protocol here is simple: a four-byte 'network byte order' (big-endian) integer that
// says how long a string is, and then a string that is that long. We wait for exactly 4 bytes,
// read in the count value, and then wait for count bytes, and then display them.
function startServerRead() {
    socketsSample.serverReader.loadAsync(4).done(function (sizeBytesRead) {
        // Make sure 4 bytes were read.
        if (sizeBytesRead !== 4) { /* [Show message] */ }

        // Read in the 4 bytes count and then read in that many bytes.
        var count = socketsSample.serverReader.readInt32();
        return socketsSample.serverReader.loadAsync(count).then(function (stringBytesRead) {
            // Make sure the whole string was read.
            if (stringBytesRead !== count) { /* [Show message] */ }

            // Read in the string.
            var string = socketsSample.serverReader.readString(count);
            socketsSample.displayOutput("Server read: " + string);

            // Restart the read for more bytes. We could just call startServerRead() but in
            // the case subsequent read operations complete synchronously we start building
            // up the stack and potentially crash. We use WinJS.Promise.timeout() to invoke
            // this function after the current call unwinds.
            WinJS.Promise.timeout().done(function () { return startServerRead(); });
        }); // End of "read in rest of string" function.
    }, onError);
}

```

This code is structured to wait for incoming data that isn't ready yet, but you might have situations in which you want to know if there's more data available that you haven't read. This value can be obtained through the [DataReader.unconsumedBufferLength](#) property.

In Scenario 2, the data-sending side of the relationship is simple: create a [StreamSocket](#) and call [connectAsync](#) (js/connectToListener.js; note that [onError](#) uses [StreamSocketError.getStatus](#) again):

```

socketsSample.clientSocket = new Windows.Networking.Sockets.StreamSocket();
socketsSample.clientSocket.connectAsync(hostName, serviceName).done(function () {
    // ...
}, onError);

```

Sending data in Scenario 3 takes advantage of a [DataWriter](#) built on the socket's output stream (js/sendData.js):

```

var writer = new Windows.Storage.Streams.DataWriter(socketsSample.clientSocket.outputStream);
var string = "Hello World";
var len = writer.measureString(string); // Gets the UTF-8 string length.
writer.writeInt32(len);
writer.writeString(string);

writer.storeAsync().done(function () {
    writer.detachStream();
}, onError);

```

And closing the socket in Scenario 4 is again just a call to `StreamSocket.close`.

As with the `DatagramSocket` sample, setting breakpoints within `openClient` (js/connectTo-Listener.js), `onServerAccept` (js/startListener.js), and `sendHello` (js/sendData.js) will let you see what's happening at each step of the process.

Web Sockets: MessageWebSocket and StreamWebSocket

Having now seen both `Datagram` and `Stream` sockets in action, we can look at their equivalents on the `WebSocket` side. As you might already know, `WebSockets` is a standard created to use HTTP (and thus TCP) to set up an initial connection after which the data exchange happens through sockets over TCP. This provides the simplicity of using HTTP requests for the first stages of communication and the efficiency of sockets afterwards.

As with regular sockets, the `WebSocket` side of WinRT supports both water balloons and water hoses: the `MessageWebSocket` class provides for discrete packets as with `datagram` sockets (though it uses TCP and not UDP), and `StreamWebSocket` clearly provides for stream sockets. Both classes are very similar to their respective `DatagramSocket` and `StreamSocket` counterparts, so much so that their interfaces are very much the same (with distinct secondary types like `MessageWebSocket-Control`):

- Like `DatagramSocket`, `MessageWebSocket` has `control`, `information`, and `outputStream` properties, a `messagereceived` event, and methods of `connectAsync` and `close`. It adds a `closed` event along with a `setRequestHeader` method.
- Like `StreamSocket`, `StreamWebSocket` has `control`, `information`, `inputStream`, and `outputStream` properties, and methods of `connectAsync` and `close`. It adds a `closed` event and a `setRequestHeader` method.

You'll notice that there isn't an equivalent to `StreamSocketListener` here. This is because the process of establishing that connection is handled through HTTP requests, so such a distinct listener class isn't necessary. This is also why we have `setRequestHeader` methods on the classes above: so that you can configure those HTTP requests. Along these same lines, you'll find that the `connectAsync` methods take a `Windows.Foundation.Uri` rather than hostnames and service names. But otherwise we see the same kind of activity going on once the connection is established, with streams, `Data-Reader`, and `DataWriter`.

Sidebar: Comparing W3C and WinRT APIs for WebSockets

Standard WebSockets, as they're defined in the W3C API, are entirely supported for Windows Store apps. However, they support only a transaction-based UDP model like [DatagramSocket](#) and only text content. The [MessageWebSocket](#) in WinRT, however, supports both text and binary, plus you can use the [StreamWebSocket](#) for a streaming TCP model as well. The WinRT APIs also emit more detailed error information and so are generally preferred over the W3C API.

Let's look more closely at these in the context of the [Connecting with WebSockets sample](#). This sample is dependent upon an ASP.NET server page running in the localhost, so you must first go into its Server folder and run **powershell.exe -ExecutionPolicy unrestricted -file setupserver.ps1** from an Administrator command prompt. (For more on setting up Internet Information Services and the localhost, refer to the "Using the localhost" section in Chapter 13.) If the script succeeds, you'll see a WebSocketSample folder in *c:\inetpub\wwwroot* that contains an EchoWebService.ashx file. Also, as suggested in Chapter 13, you can run the [Web platform installer](#) to install Visual Studio 2012 Express for Web that will allow you to run the server page in a debugger. Always a handy capability!

Within EchoWebService.ashx you'll find an [EchoWebSocket](#) class written in C#. It basically has one method, [ProcessRequest](#), that handles the initial HTTP request from the web socket client. With this request it acquires the socket, writes an announcement message to the socket's stream when the socket is opened, and then waits to receive other messages. If it receives a text message, it echoes that text back through the socket with "You said" prepended. If it receives a binary message, it echoes back a message indicating the amount of data received.

Going to Scenario 1 of the Connecting with WebSockets sample, we can send a message to that server page, using [MessageWebSocket](#), and get back a message of our own; see Figure 14-6. In this case the output in the sample reflects information known to the app and nothing from the service itself.

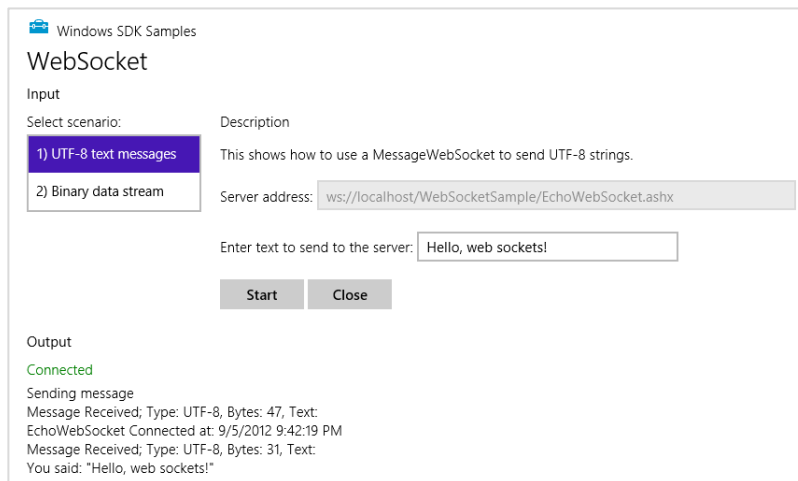


FIGURE 14-6 Output of Scenario 1 of the Connecting with WebSockets sample.

In the sample, we first create a `MessageWebSocket`, call its `connectAsync`, and then use a `DataWriter` to write some data to the socket. It also listens for the `messagereceived` event to output the result of the send, and it listens to the `closed` event from the server so that it can do the `close` from its end. The code here is simplified from `js/scenario1.js`:

```
var messageWebSocket;
var messageWriter;

var websocket = new Windows.Networking.Sockets.MessageWebSocket();
websocket.control.messageType = Windows.Networking.Sockets.SocketMessageType.utf8;
websocket.onmessagereceived = onMessageReceived;
websocket.onclosed = onClosed;

// The server URI is obtained and validated here, and stored in a variable named uri.

websocket.connectAsync(uri).done(function () {
    messageWebSocket = websocket;
    // The default DataWriter encoding is utf8.
    messageWriter = new Windows.Storage.Streams.DataWriter(websocket.outputStream);
    sendMessage(); // Helper function, see below
}, function (error) {
    var errorStatus = Windows.Networking.Sockets.WebSocketError.getStatus(error.number);
    // [Output error message]
});

function onMessageReceived(args) {
    var dataReader = args.getDataReader();
    // [Output message contents]
}

function sendMessage() {
    // Write message in the input field to the socket
    messageWriter.writeString(document.getElementById("inputField").value);
    messageWriter.storeAsync().done("", sendError);
}

function onClosed(args) {
    // Close our socket if the server closes [simplified from actual sample; it also closes
    // the DataWriter it might have opened.]
    messageWebSocket.close();
}
```

Similar to what we saw in previous sections, when an error occurs you can turn the error number into a `SocketErrorStatus` value. In the case of WebSockets you do this with the `getStatus` method of `Windows.Networking.Sockets.WebSocketError`. Again, see its reference page for details.

Scenario 2, for its part, uses a `StreamWebSocket` to send a continuous stream of data packets, a process that will continue until you close the connection; see Figure 14-7.

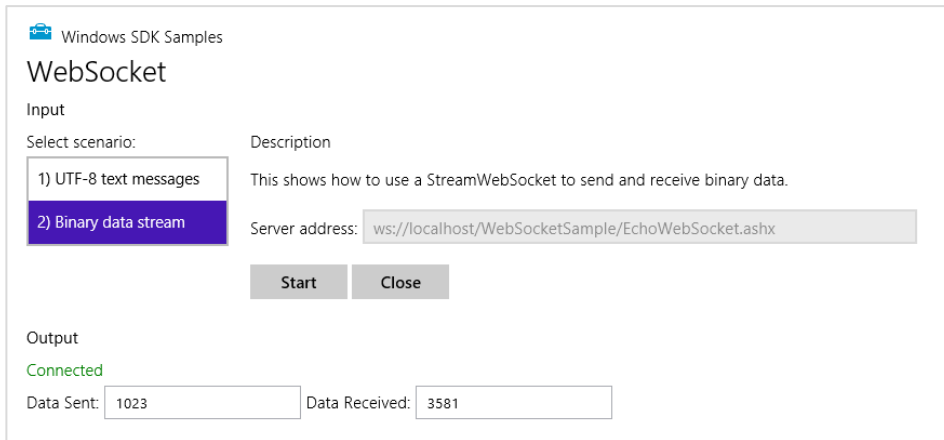


FIGURE 14-7 Output of Scenario 2 of the Connecting with WebSockets sample (cropped).

Here's the process in code, simplified from `js/scenario2.js`, where we see a similar pattern to what we just saw for `MessageWebSocket`, only sending a continuous stream of data:

```
var streamWebSocket;
var dataWriter;
var dataReader;
var data = "Hello World";
var countOfDataSent;
var countOfDataReceived;

var websocket = new Windows.Networking.Sockets.StreamWebSocket();
websocket.onclosed = onClosed;

// The server URI is obtained and validated here, and stored in a variable named uri.

websocket.connectAsync(uri).done(function () {
    streamWebSocket = websocket;
    dataWriter = new Windows.Storage.Streams.DataWriter(websocket.outputStream);
    dataReader = new Windows.Storage.Streams.DataReader(websocket.inputStream);
    // When buffering, return as soon as any data is available.
    dataReader.inputStreamOptions = Windows.Storage.Streams.InputStreamOptions.partial;
    countOfDataSent = 0;
    countOfDataReceived = 0;

    // Continuously send data to the server
    writeOutgoing();

    // Continuously listen for a response
    readIncoming();
}, function (error) {
    var errorStatus = Windows.Networking.Sockets.WebSocketError.getStatus(error.number);
    // [Output error message]
});

function writeOutgoing() {
```

```

try {
    var size = dataWriter.measureString(data);
    countOfDataSent += size;
}
dataWriter.writeString(data);
dataWriter.storeAsync().done(function () {
    // Add a 1 second delay so the user can see what's going on.
    setTimeout(writeOutgoing, 1000);
}, writeError);
}
catch (error) {
    // [Output error message]
}
}

function readIncoming(args) {
    // Buffer as much data as you require for your protocol.
    dataReader.loadAsync(100).done(function (sizeBytesRead) {
        countOfDataReceived += sizeBytesRead;
        // [Output count]

        var incomingBytes = new Array(sizeBytesRead);
        dataReader.readBytes(incomingBytes);

        // Do something with the data. Alternatively you can use DataReader to
        // read out individual booleans, ints, strings, etc.

        // Start another read.
        readIncoming();
    }, readError);
}

function onClose(args) {
    // [Other code omitted, including closure of DataReader and DataWriter]
    streamWebSocket.close();
}

```

As with regular sockets, you can exercise additional controls with WebSockets, including setting credentials and indicating supported protocols through the control property of both [MessageWeb-Socket](#) and [StreamWebSocket](#). For details, see [How to use advanced WebSocket controls](#) in the documentation. Similarly, you can set up a secure/encrypted connection by using the `wss://` URI scheme instead of `ws://` as used in the sample. For more, see [How to secure WebSocket connections with TLS/SSL](#).

The ControlChannelTrigger Background Task

In Chapter 13, in the “Lock Screen Dependent Tasks and Triggers” section we took a brief look at the [Windows.Networking.Sockets.ControlChannelTrigger](#) class that can be used to set up a background task for real-time notifications as would be used by VoIP, IM, email, and other “always reachable” scenarios. To repeat, working with the control channel is not something that can be done from JavaScript, so refer to [How to set background connectivity options](#) in the documentation along with the following C#/C++ samples:

- [ControlChannelTrigger StreamSocket sample](#)
- [ControlChannelTrigger XmlHttpRequest sample](#)
- [ControlChannelTrigger StreamWebSocket sample](#)
- [ControlChannelTrigger HTTP client sample](#)

Loose Ends (or Some Samples To Go)

Although we've covered quite a bit of territory in this chapter, you might find some additional samples helpful in your networking efforts. I won't address these topics further in this book, but this list will at least help you be aware of their existence.

Sample	Description (from the Windows Developer Center)
Check if current session is remote sample	This sample demonstrates the use of Windows.System.RemoteDesktop API. Specifically, this sample demonstrates how to use the InteractiveSession.IsRemote property to determine if the current session is a remote session.
HomeGroup app sample	Demonstrates how to use a HomeGroup to open, search, and share files. This sample uses some of the HomeGroup options. In particular, it uses Windows.Storage.-Pickers.PickerLocationId enumeration and the Windows.Storage.-KnownFolders.homeGroup property to select files contained in a HomeGroup.
Remote desktop app container client sample	Demonstrates how to use the Remote Desktop app container client objects in an app.
RemoteApp and desktop connections workspace API sample	Demonstrates how to use the WorkspaceBrokerAx object in a Windows 8 app.
SMS message send, receive, and SIM management sample	Demonstrates how to use the Windows 8 Mobile Broadband SMS API (Windows.Devices.Sms). This API can be used only from mobile broadband device apps and is not available to apps generally.
SMS background task sample	Demonstrates how to use the Windows 8 Mobile Broadband SMS API (Windows.-Devices.Sms) with the Background Task API (Windows.ApplicationModel.-Background) to send and receive SMS text messages. This API can be used only from mobile broadband device apps and is not available to apps generally.
USSD message management sample	Demonstrates network account management using the USSD protocol with GSM-capable mobile broadband devices. USSD is typically used for account management of a mobile broadband profile by the Mobile Network Operator (MNO). USSD messages are specific to the MNO and must be chosen accordingly when used on a live network. [That sample is applicable only to those building mobile broadband device apps; it draws on the API in Windows.Networking.NetworkOperators .]

What We've Just Learned

- Networks come in a number of different forms, and separate capabilities in the manifest specifically call out *Internet (Client)*, *Internet (Client & Server)*, and *Private Networks (Client & Server)*. Local loopback within these is normally blocked for apps but may be used for debugging purposes on machines with a developer license.
- Rich network information is available through the `Windows.Networking.Connectivity.-NetworkInformation` API, including the ability to track connectivity, be aware of network costs, and obtain connection profile details.
- Connectivity can be monitored from a background task by using the `networkStateChange` trigger and conditions such as `internetAvailable` and `internetNotAvailable`.
- The ability to run offline can be an important consideration that can make an app much more attractive to customers. Apps need to design and implement such features themselves, using local or temporary app data folders to store the necessary caches.
- `Windows.Networking.BackgroundTransfer` provides for cost-aware downloads and up-loads that continue to run while an app is suspended and that are easy to resume if an app is restarted after termination. Using this API is highly recommended over doing the same with `XmlHttpRequests`; the API supports credentials, multipart uploads, cost policy, and grouping.
- The Credential Picker UI provides a built-in UI for collecting credentials, and the credential locker provides a secure means for storing and retrieving those credentials (that can also be roamed to the user's other trusted devices if they allow it).
- Apps can authenticate through OAuth providers using the web authentication broker API. This allows apps to obtain necessary access keys and tokens to work with those providers while never having to manage user credentials directly.
- For authentication providers that support it, apps can use single sign on so that authenticating the user in one app will authenticate them in all others using the same provider. The Live SDK/Live Connect provides for this with the user's Microsoft account.
- Apps can obtain and manage some of the user's profile data, including the user image and the lock screen image.
- WinRT provides APIs for encryption and decryption, along with certificates.
- The `Windows.Web.Syndication` API provides a structured way to consume RSS feeds, and `Windows.Web.AtomPub` provides a structured way to post, edit, and manage entries.
- Socket support in WinRT includes datagram and stream sockets, as well as message and stream WebSockets. The capabilities of the latter expand on the capabilities of W3C WebSockets by supporting both a streaming (TCP) model and binary content.

Chapter 15

Devices and Printing

I sometimes marvel at all the stuff that's hanging off the humble laptop with which I've been writing this book. Besides the docking station that is currently also serving well as a monitor stand and rather efficient dust collector, there's a mouse (wired), a keyboard (wireless), two large monitors, speakers, a USB thumb drive, a headset, an Xbox game controller, and the occasional external hard drive. Add to that a couple of printers and media receivers hanging off my home network and, well, I probably don't come close to what the majority of my readers probably have around their home and workplace!

For all that might be going on within one computer itself, and for all the information it might be obtaining from online sources, the world of external devices is another great realm, especially those that apps can work with directly. Indeed, we've spent most of our time in this book talking about what's going on in the app and its host system and about using networks and services to gather data. It's time, then, that we take an introductory look at the other hardware we can draw on to make a great app experience.

We've already encountered a few of these areas that I'll just mention again here:

- In Chapter 8, "State, Settings, Files, and Documents," we learned about the *Removable Storage* capability (in the manifest) that enables an app to work with files on USB sticks and other pluggable media. When a device is connected, those folders become available through [Windows.Storage.KnownFolders.removableDevices](#), which is just a [StorageFolder](#) object whose subfolders are each connected to a particular storage device. See the [Removable storage sample](#).
- In Chapter 2, "Quickstart," along with Chapter 10, "Media," we took full advantage of the [Windows.Media.Capture](#) API to effortlessly obtain audio, images, and video from an attached camera (see the [CameraCaptureUI Sample](#)). This included the ability to select a specific capture device through Scenario 2 of the [Media capture using capture device sample](#), for which we used the API in [Windows.Devices.Enumeration](#).
- Also in Chapter 10 we looked at the [Windows.Media.PlayTo](#) API to connect media to a PlayTo capable receiver, as demonstrated in the [Media Play To sample](#).

Beyond these there is much more, far more than this book and chapter will allow—hardware truly is a world unto itself! But at least we'll understand where some of the resources are and spend a little time on those areas that are likely to be interesting to apps themselves. These are:

- Using devices through an API not directly available to JavaScript, such as Win32. Apps can also enumerate additional devices with a certain class interface. (The ability to interact with those devices is limited, at present, for Windows Store apps.)

- Connecting with devices in the vicinity of the one your app is running on through means such as WiFi Direct, Bluetooth, and near-field communications (NFC). In the latter case, NFC can connect apps running on devices or be used to acquire information from an inexpensive RFID tag.
- Last but not least is printing, which is an easy feature to add to a Windows Store app.

With that, let me also mention a class of apps that we won't be dealing with in this book: Windows Store Device Apps, as they're called. These are the ones that can be automatically acquired from the Windows Store when their associated devices are plugged into a Windows machine. The variety in this space is quite amazing, as we see everything from the most prosaic headsets, monitors, printers, cameras, mice, and keyboards to the newest smart TVs, remotes, home audio systems, health sensors, scientific devices, toys, point of sale systems, musical instruments, and more. It's the responsibility of device apps—the ones you always used to get on a CD that you then employed as a coaster—to light up the functionality of the device, and at present they are the only apps that can actually do so. Windows Store apps in general are not able to work with specialized devices unless there is some other public API that allows for it, as we'll see in the first section below.

Writing device apps is well beyond the scope of this book, but if you're interested you can refer to the [Windows Store Device App Workshop](#) (Channel 9 videos), along with [Windows 8 Device Experience: Windows Store Device Apps](#), along with [Windows Store device Apps for Specialized Connected Devices](#). There are also a few samples to draw on, such as the [Windows 8 device app for camera sample](#) and the [Device apps for printers sample](#). A much more specific one is the [Custom driver access sample](#), which works with a piece of hardware called FX2 in the [OSR USB FX2 Learning Kit](#) (from Open System Resources). This is a piece of hardware meant for people learning how to develop device drivers to use to understand the intricacies.

Using Devices

As mentioned earlier, Windows Store apps are not generally given access to specialized hardware and whatever interfaces exist in device drivers: this is the special privilege of device apps. However, if the device and its driver happen to plug into a system API of some kind, then there are ways for other apps to work with them, as illustrated in Figure 15-1.

The system APIs that are available to Windows Store apps are somewhat varied. As we've seen in previous chapters, WinRT itself enables access to cameras, PlayTo receivers, storage devices (including USB drives, cameras, and media players), and input devices, where in the latter case the hardware is hidden behind abstraction layers like pointers. WinRT also provides an abstraction through which an app can work with any number of printers, as we'll see in "Printing Made Easy" later on.

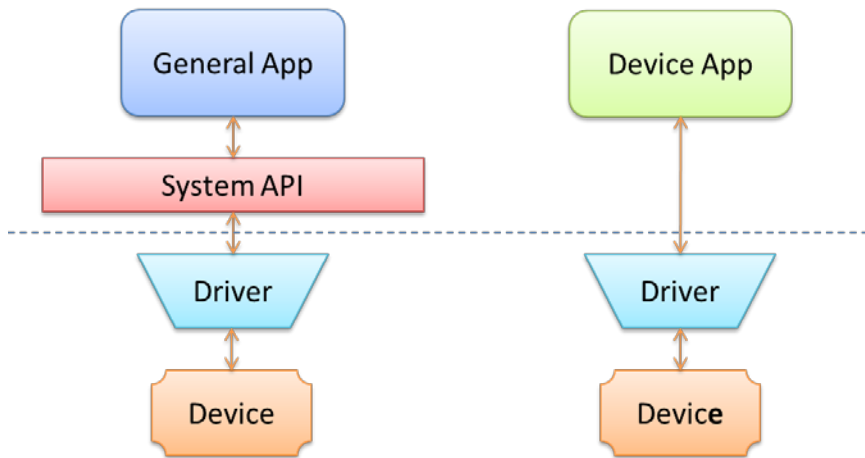


FIGURE 15-1 Store apps in Windows 8 can work only with devices that have a representative system API; Windows Store device apps have the privilege of working directly with a specific device through its driver.

In some cases, as we'll see in our first example of an Xbox controller, there are APIs in Win32/COM that are built on a certain device interface. In the case of the controller, the XInput API (part of DirectX) does exactly that. XInput isn't directly available to an app written in JavaScript, but a WinRT component can perform that job on the app's behalf. Such a component, as we'll cover in more depth in Chapter 16, "WinRT Components," extends the WinRT API for those apps that include it in their package. The APIs provided by such components look and feel like those in WinRT. They just start with some namespace other than `Windows`!

Another capability that WinRT does provide for is enumerating devices with a particular *device interface class* (a GUID). This is useful for building and displaying a list of devices when more than one exists, and works for common devices (like cameras) and uncommon devices alike.

The other class of devices that apps can work with is called Windows Portable Devices, which includes removable storage as well as a host of Bluetooth gizmos. This is one place where being able to enumerate the devices is helpful, because there's an ActiveX control, of all things—which is essentially a COM API—through which you can talk to such devices.

The XInput API and Game Controllers

The [XInput API](#), part of DirectX, is a Win32 API that specifically works with game controllers and is on the list of Win32/COM APIs that can be used from a Windows Store app. The most commonly used function within this group is probably [XInputGetState](#), which returns an [XINPUT_STATE](#) structure that describes the position of the various thumb controllers, how far throttle or other triggers are depressed, and the on/off states of all the buttons. It's basically meant to be polled with every animation frame in something like a game; the API doesn't itself raise events when the controller state changes.

The [XInput and JavaScript controller sketch sample](#) in the Windows SDK demonstrates exactly this. Because the XInput API is not accessible directly through JavaScript, it's necessary to create a WinRT component for this purpose. Put simply, you create a component with public classes inside a namespace that matches the component's filename, and then you add a reference to that component to the JavaScript app's project in Visual Studio. This imports the namespace and makes it available within JavaScript. We'll see more details on this in Chapter 16, but basically the component's C++ code looks like the following—first in the header (Controller.h) file within the GameController project:

```
namespace GameController
{
    public value struct State
    {
        // [Omitted--just contains the same values as the Win32 XINPUT_STATE structure
    };

    public ref class Controller sealed
    {
        ~Controller();

        uint32          m_index;
        bool             m_isControllerConnected; // Do we have a controller connected
        XINPUT_CAPABILITIES m_xinputCaps;        // Capabilites of the controller
        XINPUT_STATE      m_xinputState;         // The current state of the controller
        uint64           m_lastEnumTime;        // Last time a new controller connection
                                                // was checked

    public:
        Controller(uint32 index);

        void SetState(uint16 leftSpeed, uint16 rightSpeed);
        State GetState();
    };
}
```

The implementation of `GetState` in `Controller.cpp` then just calls `XInputGetState` and copies its properties to an instance of the component's public `State` structure:

```
State Controller::GetState()
{
    // defaults to return controllerState that indicates controller is not connected
    State controllerState;
    controllerState.connected = false;

    // An app should avoid calling XInput functions every frame if there are
    // no known devices connected as initial device enumeration can slow down
    // app performance.
    uint64 currentTime = ::GetTickCount64();
    if (!m_isControllerConnected && currentTime - m_lastEnumTime < EnumerateTimeout)
    {
        return controllerState;
    }

    m_lastEnumTime = currentTime;
```

```

auto stateResult = XInputGetState(m_index, &m_xinputState);

if (stateResult == ERROR_SUCCESS)
{
    m_isControllerConnected = true;
    controllerState.connected = true;
    controllerState.controllerId = m_index;
    controllerState.packetNumber = m_xinputState.dwPacketNumber;
    controllerState.LeftTrigger = m_xinputState.Gamepad.bLeftTrigger;
    controllerState.RightTrigger = m_xinputState.Gamepad.bRightTrigger;

    // And so on [copying all the other properties omitted.]
}
else
{
    m_isControllerConnected = false;
}

return controllerState;
}

```

The constructor for a Controller object is also very simple:

```

Controller::Controller(uint32 index)
{
    m_index = index;
    m_lastEnumTime = ::GetTickCount64() - EnumerateTimeout;
}

```

In a JavaScript app—once the reference to the component has been added—the `GameController` namespace contains the component’s public API, and we can utilize it as if it were built right into Windows. In the case of the sample, it first instantiates a Controller object (with index of zero) and then kicks off animation frames (program.js):

```

app.onactivated = function (eventObj) {
    if (eventObj.detail.kind ===
        Windows.ApplicationModel.Activation.ActivationKind.launch) {
        // [Other setup omitted]

        // Instantiate the Controller object from the WinRT component
        controller = new GameController.Controller(0);

        // Start rendering loop
        requestAnimationFrame(renderLoop);
    }
};

```

The `renderLoop` function then just calls the component’s `getState` method and applies the results to a canvas drawing before repeating the loop (also in program.js, though much code omitted):

```

function renderLoop() {
    var state = controller.getState();

    if (state.connected) {
        controllerPresent.style.visibility = "hidden";

        // Code added to the sample to extend its functionality
        if (state.leftTrigger) {
            context.clearRect(0, 0, sketchSurface.width, sketchSurface.height);
            requestAnimationFrame(renderLoop);
            return;
        }

        if (state.a) {
            context.strokeStyle = "green";
        } else if (state.b) {
            context.strokeStyle = "red";
        } else if (state.x) {
            context.strokeStyle = "blue";
        } else if (state.y) {
            context.strokeStyle = "orange";
        }

        // Process state and draw the canvas [code omitted]
    };

    // Repeat with the next frame
    requestAnimationFrame(renderLoop);
};

```

The output of this sample is shown in Figure 15-2, reflecting the features I added to the sample within the code above to make it more interesting to my young son: changing colors with the A/B/X/Y buttons and clearing the canvas with the left trigger. As you can see, my own artwork with this app isn't a whole lot different from his!

In the end, even though WinRT doesn't surface APIs like XInput, an app can do this for itself with a simple component implementation. Note that various aspects of the component's interface, like the casing of method names, will change when it's projected into JavaScript. Again, we'll see more details in Chapter 16. For now, it shows that getting access to such specialized devices is a straightforward task.

Javascript XInput game controller sample

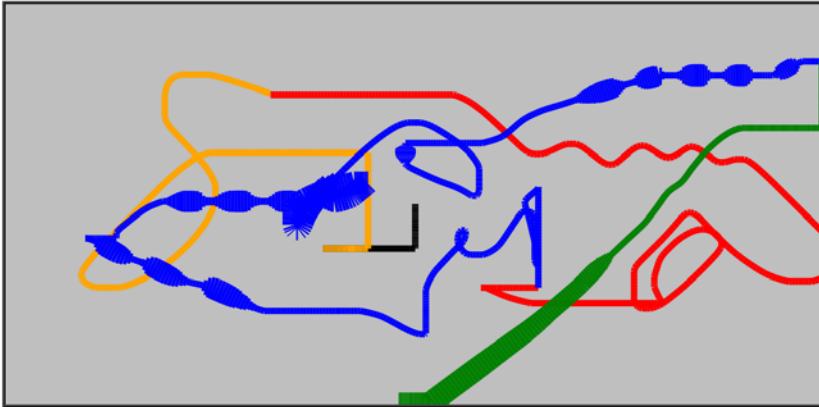


FIGURE 15-2 The XInput and JavaScript controller sketch sample with some modifications to change colors. The varying line width is controlled by the position of the right trigger.

Enumerating Devices in a Class

If you happen to know the device class interface GUID for a certain group of devices, you can use the static [Windows.Devices.Enumeration.DeviceInformation.findAllAsync](#) method to retrieve detailed information about those devices. Many details can be found in the [Enumerating Devices](#) topic in the documentation, but let me give a quick overview.

What you give to [findAllAsync](#) is something called a *selector*, specifically an [Advanced Query Syntax \(AQS\)](#) string, as we encountered in Chapter 8 in the section “Rich Enumeration with File Queries.” A device selector typically looks something like this:

```
System.Devices.InterfaceClassGuid="{E5323777-F976-4F5B-9B55-B94699C46E44}" AND  
System.Devices.InterfaceEnabled:=System.StructuredQueryType.Boolean#True
```

where the interface class GUID shown here is the particular one for webcams.

The result of [findAllAsync](#) is a [DeviceInformationCollection](#), which in JavaScript can basically be treated as an array of [DeviceInformation](#) objects. In Scenario 1 of the [Device enumeration sample](#) we see how to use this array to display details for each device:

```
Windows.Devices.Enumeration.DeviceInformation.findAllAsync(selector, null).done(  
    function(devinfoCollection) {  
        var numDevices = devinfoCollection.length;  
        for (var i = 0; i < numDevices; i++) {  
            displayDeviceInterface(devinfoCollection[i], id("scenario1Output"), i);  
        }  
    });
```

Some results from this are shown in Figure 15-3 and Figure 15-4.



FIGURE 15-3 Sample device enumeration output for a webcam—which perfectly represents the one attached to my monitor.



FIGURE 15-4 Sample device enumeration output for a printer—which also looks exactly like the one sitting next to my desk.

Scenario 2 of the sample executes the same process (with plain text output) for Plug and Play (PnP) object types using [Windows.Devices.Enumeration.Pnp.PnpObject.findAllAsync](#). This API lets you enumerate devices by interface, interface class, and container (the visible and localized aspects of a piece of hardware, like manufacturer and model name):

```
Windows.Devices.Enumeration.Pnp.PnpObject.findAllAsync(deviceContainerType,
    propertiesToRetrieve).done(function (containerCollection) {
    var numContainers = containerCollection.length;
    for (var i = 0; i < numContainers; i++) {
        displayDeviceContainer(containerCollection[i], id("scenario2Output"));
    }
});
```

In the call above, the `propertiesToRetrieve` variable contains an array of strings that identify the [Windows properties](#) you're interested in. The sample uses these:

```
var propertiesToRetrieve = ["System.ItemNameDisplay", "System.Devices.ModelName",
    "System.Devices.Connected"];
```

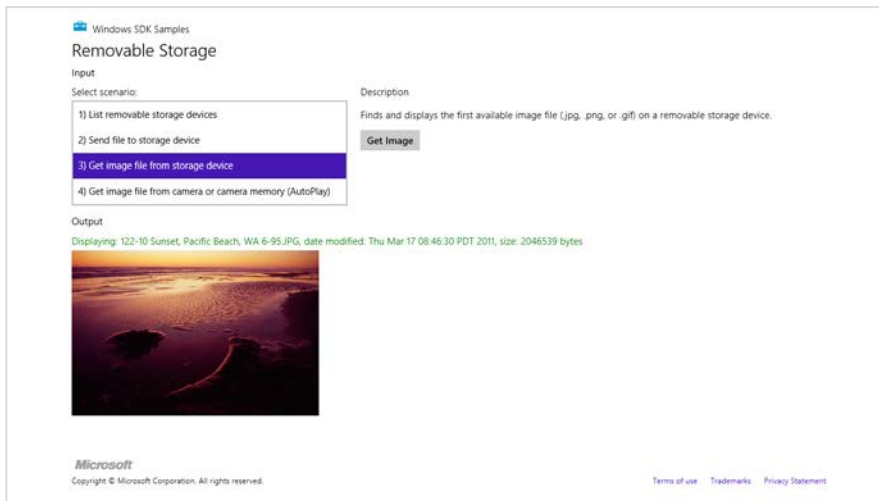
The result of the enumeration—the `containerCollection` variable in the code above—is a [PnpObjectCollection](#) that contains [PnpObject](#) instances. The sample just takes the information from these and displays a text output for each.

Note that there is a variant of [findAllAsync](#) that accepts an AQS string as a filter. This is a string that you obtain from APIs like [Windows.Devices.Portable.StorageDevice.getDeviceSelector](#) that makes enumeration of those particular devices easier.

Windows Portable Devices and Bluetooth Capabilities

In addition to enumerating printers and webcams (because they have standard interface class GUIDs), Scenario 1 of the [Device enumeration sample](#) also works with *portable devices*, as does the [Portable device service sample](#). This opens the doors to the subject of [Windows Portable Devices](#) or WPD, a driver technology that supports things like phones, digital cameras, portable media players, and so on but also a growing array of Bluetooth devices, where the need is primarily to transfer data between the device and the system. WPD supplies an infrastructure for this.

In WinRT, [Windows.Devices.Portable](#) API provides direct interaction with WPD. Here you'll find the [ServiceDevice](#) and [StorageDevice](#) classes. Both of which simply provide methods that return selector strings and `id` properties. In the former case, such information is meaningful only to the device app associated with the hardware. In the latter case, however, the [StorageDevice.fromId](#) method provides a [Windows.Storage.StorageFolder](#) through which you can enumerate its contents. This is demonstrated in Scenario 3 of the [Removable storage sample](#) that we noted in Chapter 8, where it will create a list of removable storage devices to choose from and then display the first image found on the one you select:



The Removable Storage sample, in Scenario 4, also demonstrates how to use AutoPlay to automatically launch the app when you plug in a suitable device. This involves declarations in the manifest for AutoPlay Content (inserting a storage medium) and/or AutoPlay Device (inserting a device). See [Auto-launching with AutoPlay for details](#).

As for Bluetooth devices, the [Windows.Media.Devices.CallControl](#) API gives you the ability to work with a telephony-related device. See [How to manage calls on the default Bluetooth communications device](#) for more along with the [Bluetooth call control sample](#).

Another group of Bluetooth devices includes those that collect information about one's physical health, such as heart rate, blood pressure, and temperature. See [Bluetooth low energy on Wikipedia](#) to learn more; working with these is demonstrated in the [Bluetooth low energy health profiles sample](#) as we'll briefly see below, but I'm told that access to these is presently limited to device apps.

Another specific sample is the [Bluetooth simple key service sample](#), which works with the [CC2540 Mini Development Kit controller](#). This little gizmo is something Texas Instruments created to assist development around their CC2540 chip; in the case of the sample, the buttons on the device control buttons in the app.

What's interesting in these latter two samples is how the app connects with the particular device using an ActiveX control called *PortableDeviceAutomation.Factory* (one of the few such ActiveX objects available). For example, to connect with a thermometer we first enumerate devices with its particular class GUID, *{00001809-0000-1000-8000-00805f9b34fb}* (see *js/thermometer.js* in the Bluetooth low energy health profiles sample):

```
Windows.Devices.Enumeration.DeviceInformation.findAllAsync(  
    "System.Devices.InterfaceClassGuid:={00001809-0000-1000-8000-00805f9b34fb}\\", null)
```

With the results from this async operation in a variable called *devices*, here's how it gets to a specific device and sets up a listener for its particular events:

```
// Use WPD Automation to initialize the device objects  
var deviceFactory = new ActiveXObject("PortableDeviceAutomation.Factory");  
  
// For the purpose of this sample we will initialize the first device  
deviceFactory.getDeviceFromIdAsync(devices[0].id, function (device) {  
    // The 'device' variable will have the device object.  
    // Initialize the temperature service and listen for measurements  
    tempService = device.services[0];  
    tempService.onTemperatureMeasurement =  
        function (timestamp, thermometerMeasurementValue) {  
            // ...  
        };  
});
```

Near Field Communication and the Proximity API

Connecting with devices that are near to the one on which your app is running is one area that I suspect will see much creative innovation in the coming years as PCs are increasingly equipped with the requisite hardware. In this case we're really speaking of "devices" more generally than we have been. In some cases there will be a separate discrete device, most notably Bluetooth devices or RFID tags. But then we're also speaking of an app running on one machine connecting with itself or another that's running on a different machine. In this sense, apps can communicate with each other as if they were themselves separate "devices."

Caveat Though it is possible for different apps to know about each other and communicate, the Store certification requirements do not allow them to be interdependent. Approach such communication scenarios as a way to extend the functionality of the app, but be sure to provide value when the app is run in isolation.

Near Field Communication (NFC) is one of the key ways for apps to connect to devices and across devices. NFC works with electromagnetic sensors (including unpowered RFID tags) that resonate with each other when they get close, within 3–4 centimeters. Practically speaking, this means that the devices actually make physical contact, a tap that effectively initiates a digital handshake that opens the conversation. When this happens between the same app running on both devices, a process known as pairing, those apps can have an ongoing conversation.

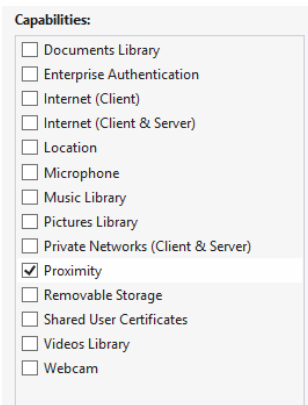
When you think about "devices" in this context, though, they can vary tremendously. That is, the devices that are making the connection don't need to be at all similar. One device might be your tablet PC, for example, and the other might be anything from a large all-in-one PC display to a simple RFID tag mounted in a poster or a name badge.

Apps can also learn about each other on different devices through WiFi Direct (if the wireless adapter supports it) and Bluetooth. In these cases it's possible for one app to browse for available (advertised) connections on other devices, which might or might not be coming from the same app.

Whatever the case, working with *proximity*—as all of this is collectively referred to—is useful for many scenarios such as sharing content, setting up multiplayer experience, broadcasting activity, and really anything else where some kind of exchange might happen, including both one-time data transfers and setting up more persistent connections.

There are three main conditions for using proximity (see [Guidelines for developing using proximity](#)):

- The app must declare the *Proximity* capability in its manifest.



- Communications are supported only for foreground apps (there are no background tasks to maintain a conversation).
- The app must ask for user consent to enter into a multiuser relationship. An app should show waiting connections, connections being established, and connections that are active, and it should allow the user to disconnect at any time. Note that using the APIs to make a connection will automatically prompt the user.

The API for working with proximity is in the appropriately named [Windows.Networking.-Proximity](#) namespace, as is the [Proximity sample](#) that we'll be working with here. It almost goes without saying that doing any deep exploration of proximity will require two machines that are suitably equipped (unless you're just working with RFID tags). For NFC between two machines there is also a driver sample in the Windows Driver Kit that simulates NFC over a network connection. To use it, note that you'll also need Visual Studio 2012 Ultimate Edition; the Express version does not support driver development. It might make more sense to just acquire some NFC-capable tablets!

Anyway, to install the Windows Driver Kit, follow the instructions on [How to get the WDK](#) after installing Visual Studio 2012 Ultimate. When you start the download, don't be put off by the indicated 1GB size—start the installer (it's a 937 K download), and then select the option to acquire the kit for use on another computer (which is actually a 307 MB download). Then, according to the Proximity sample's description page:

After you have installed the WDK and samples, you can find the proximity driver sample in the src\nfp directory in the location where you installed the WDK samples. See the NetNfpProvider.html file in the src\nfp\net directory for instructions on building and running the simulator. [Note: be sure to specify a Windows 8 target when you build.] After you start the simulator, it runs in the background while your proximity app is running in the foreground. Your app must be in the foreground for the tap simulation to work.

Assuming, then, that you have an environment in which proximity can at least be simulated, let's look at the two mainline scenarios in the following sections. The first is using the [PeerFinder](#) class to create a

socket connection between two peer apps for ongoing communication. The second uses the [ProximityDevice](#) class to send data packets between two devices.

Finding Your Peers (No Pressure!)

To find peers, an app on one machine can advertise its availability such that apps on other devices can browse advertised peers and initiate a connection over WiFi Direct or Bluetooth. The second way to find a peer is through a direct NFC tap. Both methods are shown in Scenario 1 of the [Proximity sample](#), but these three distinct functions—advertise, browse, and tap to connect—are somewhat intermixed, because they each use some distinct parts of the [Windows.Networking.Proximity.PeerFinder](#) class and some parts in common.

One commonality is the static property [PeerFinder.supportedDiscoveryTypes](#), which indicates how connections can be made. This contains a combination of values from the [PeerDiscoveryTypes](#) enumeration and depends on the available hardware in the device. Those values are [browse](#) (WiFi Direct is available), [triggered](#) (NFC tapping is available), and [none](#) ([PeerFinder](#) can't be used). You can use these values to selectively enable or disable certain capabilities in your app as needed. Scenario 1 of the Proximity sample, for instance, checks the discovery types to set some flags and enable buttons for NFC activities. Otherwise, it just shows disappointing messages (this code is condensed somewhat from `js/PeerFinder.js`, and note the namespace variable at the top):

```
var ProxNS = Windows.Networking.Proximity;

var supportedDiscoveryTypes = ProxNS.PeerFinder.supportedDiscoveryTypes;

// Enable triggered (tap) related UI only if the hardware support is present
if (supportedDiscoveryTypes & ProxNS.PeerDiscoveryTypes.triggered) {
    triggeredConnectSupported = true;
} else {
    peerFinderErrorMsg = "Tap based discovery of peers not supported \n";
}

// Enable browse related buttons only if the hardware support is present
if (supportedDiscoveryTypes & ProxNS.PeerDiscoveryTypes.browse) {
    browseConnectSupported = true;
    // [Add listeners to buttons, code omitted]
} else {
    // [Show messages, code omitted]
}

if (triggeredConnectSupported || browseConnectSupported) {
    // [Set up additional UI]
}

// ...
}
```

Now let's tease apart the distinct areas.

Advertising a Connection

Making yourself available to others through advertising has two parts: putting out the word and listening for connections that are made.

Assuming that some form of communication is possible, the first step in all of this is to configure the `PeerFinder` with the `displayName` that will appear to other devices when you advertise and to set `allowBluetooth`, `allowInfrastructure`, and `allowWiFiDirect` as desired to allow discovery over additional networks (infrastructure refers to TCP/IP). Setting none of these flags will still enable connections through NFC tapping, which is always enabled.

Next, set up a handler for the `PeerFinder.onconnectionrequested` event, followed by a call to the static method `PeerFinder.start` (again, `ProxNS` is a namespace variable):

```
ProxNS.PeerFinder.onconnectionrequested = connectionRequestedEventHandler;  
ProxNS.PeerFinder.start();
```

Note `connectionrequested` is an event that originates within WinRT. Because this is perhaps an event you might only listen to temporarily, be sure to call `removeEventListener` or assign `null` to the event property to prevent memory leaks. See the “WinRT Events and removeEventListener” section in Chapter 3, “App Anatomy and Page Navigation.”

The `connectionrequested` event is triggered when other devices pick up your advertisement and call your toll-free hotline, so to speak, specifically over WiFi Direct or Bluetooth. The event receives a `ConnectionRequestedEventArgs` object that contains a single property, `peerInformation`, which is an instance of—not surprisingly—the `PeerInformation` class. This object too is simple, containing nothing but a `displayName`, but that is enough to make a connection.

```
function connectionRequestedEventHandler(e) {  
    requestingPeer = e.peerInformation;  
    ProximityHelpers.displayStatus("Connection Requested from peer: "  
        + requestingPeer.displayName);  
    // Enable Accept button (and hide Send and Message) [some code omitted]  
    ProximityHelpers.id("peerFinder_AcceptRequest").style.display = "inline";  
}
```

A connection is established by passing that `PeerInformation` object to `PeerFinder.-connectAsync`. This will prompt the user for consent, and given that consent, your completed handler will receive a `Windows.Networking.Sockets.StreamSocket`, which we’ve already encountered in Chapter 14, “Networking.”

```
function peerFinder_AcceptRequest() {  
    ProxNS.PeerFinder.connectAsync(requestingPeer).done(function (proximitySocket) {  
        startSendReceive(proximitySocket);  
    });  
}
```

From this point on, you're free to send whatever data with whatever protocols you'd like, on the assumption, of course, that the app on the other end will understand what you're sending. This is clearly not a problem when it's the same app on both ends of the connection; different apps, of course, will need to share a common protocol. In the sample, the "protocol" exchanges only some basic values, but the process is all there.

If at any time you want to stop advertising, call `PeerFinder.stop`. To close a specific connection, call the socket's `close` method.

Making a Connection

On the other side of a proximity relationship, an app can look for peers that are advertising themselves over WiFi Direct or Bluetooth. In the Proximity sample, a Browse Peers button is enabled if the `browse` discovery type is available. This button triggers a call to the following function (`js/PeerFinder.js`) that uses `PeerFinder.findAllPeersAsync` to populate a list of possible connections, including those from different apps:

```
function peerFinder_BrowsePeers() {
  // Empty the current option list [code omitted]

  ProxNS.PeerFinder.findAllPeersAsync().done(function (peerInfoCollection) {
    // Add newly found peers into the drop down list.
    for (i = 0; i < peerInfoCollection.size; i++) {
      var peerInformation = peerInfoCollection[i];
      // Create and append option element using peerInformation.displayName
      // to the peerFinder_FoundPeersList control [code omitted]
    }
  });
}
```

When you select a peer to connect to, the sample takes its `PeerInformation` object and calls `PeerFinder.connectAsync` as before (during which the user is prompted for consent):

```
function peerFinder_Connect() {
  var foundPeersList = ProximityHelpers.id("peerFinder_FoundPeersList");
  var peerToConnect = discoveredPeers[foundPeersList.selectedIndex];

  ProxNS.PeerFinder.connectAsync(peerToConnect).done(
    function (proximitySocket) {
      startSendReceive(proximitySocket);
    });
}
```

Once again, this provides a `StreamSocket` as a result, which you can use as you will. To terminate the connection, call the socket's `close` method.

Tap to Connect and Tap to Activate

To detect a direct NFC tap—which again works to connect apps running on two devices—listen to the [PeerFinder.ontriggeredConnectionStateChanged](#) (a WinRT event that I spell out in camel casing so that it's readable!). In response, start the [PeerFinder](#):

```
ProxNS.PeerFinder.ontriggeredconnectionstatechanged = triggeredConnectionStateChangedEventHandler;  
ProxNS.PeerFinder.start();
```

The process of connecting through tapping will go through a series of state changes (including user consent), where those states are described in the [TriggeredConnectState](#) enumeration: [listening](#), [connecting](#), [peerFound](#), [completed](#), [canceled](#), and [failed](#). Each state is included in the event args sent to the event (a [TriggeredConnectionStateChangedEventArgs](#)...some of these names sure get long!), and when that state reaches [completed](#), the [socket](#) property in the event args will contain the [StreamSocket](#) for the connection:

```
function triggeredConnectionStateChangedEventHandler(e) {  
    // [Other cases omitted]  
  
    if (e.state === ProxNS.TriggeredConnectState.completed) {  
        startSendReceive(e.socket);  
    }  
}
```

Again, from this point on, it's a matter of what data is being exchanged through the socket—the NFC tap is just a means to create the connection. And once again, call the socket's [close](#) when you're done with it.

When tapping connects the same app across devices, it's possible to have the tap launch an app on one of those devices. That is, when the app is running on one of the devices and has started the [PeerFinder](#), Windows will know the app's identity and can look for it on the other device. If it finds that app, it will launch it (or activate it if it's already running). The app's [activated](#) handler is then called with an activation kind of [launch](#), where [eventArgs.detail.arguments](#) will contain the string *"Windows.Networking.Proximity.PeerFinder:StreamSocket"* (see [js/default.js](#)):

```
var tapLaunch = ((eventObject.detail.kind ===  
    Windows.ApplicationModel.Activation.ActivationKind.launch) &&  
    (eventObject.detail.arguments ===  
        "Windows.Networking.Proximity.PeerFinder:StreamSocket"));  
if (tapLaunch) {  
    url = scenarios[0].url; // Force scenario 0 if launched by tap to start the PeerFinder.  
}  
return WinJS.Navigation.navigate(url, tapLaunch);
```

The code in Scenario 1 picks up this condition (the [tapLaunch](#) parameter to [WinJS.Navigation.Navigate](#) is [true](#)) and calls [PeerFinder.start](#) automatically instead of waiting for a button press. In the process of startup, the app also registers its own [triggeredConnection-StateChanged](#) handler so that it will also receive a socket when the connection is complete.

Sending One-Shot Payloads: Tap to Share

Although the [PeerFinder](#) sets up a [StreamSocket](#) and is good for scenarios involving ongoing communication, other scenarios—like sharing a photo, a link, or really any kind of information including data from an RFID tag—need only send some data from one device to another and be done with it. For such purposes we have the [Windows.Networking.Proximity.ProximityDevice](#) class, which you obtain as follows:

```
var proximityDevice = Windows.Networking.Proximity.ProximityDevice.getDefault();
```

An app that has something to share “publishes” that something as a *message* in the form of a string, a URI, or a binary buffer. RFID tags publish their messages passively; an app, on the other hand, uses the [ProximityDevice](#) class and its [publishMessage](#), [publishUriMessage](#), and [publish-BinaryMessage](#) methods (and a matching [stopPublishing](#) method). For example, drawing from Scenario 2 of the Proximity sample (js/ProximityDevice.js, there [publishText](#) contains the contents of an edit control):

```
var publishedMessageId = proximityDevice.publishMessage("Windows.SampleMessageType",
    publishText);
```

On the other side, an app that would like to receive such a message calls [ProximityDevice.-subscribeForMessage](#), passing the name of the message it expects along with a handler for when messages arrive:

```
var subscribedMessageId = proximityDevice.subscribeForMessage("Windows.SampleMessageType",
    messageReceived);

function messageReceived(receivingDevice, message) {
    // Process the message
}
```

If the app is no longer interested in messages, it calls [stopSubscribingForMessage](#).

With this simple protocol, you can see that an app that supports “tap to share” (as it’s called) would probably publish messages whenever it has appropriate content in hand. It can also use the [ProximityDevice](#) object’s [devicearrived](#) and [devicedeparted](#) events to know when other tap-to-share peers are in proximity such that it’s appropriate to publish (these are WinRT events). The [devicearrived](#) event is also what you use to discover that an RFID tag has come into proximity (see below).

What’s interesting to think about, though, is what kind of data you might share. Consider a travel app in which you can book flights, hotels, rental cars, and perhaps much more. It can, of course, publish messages with basic details but could also publish richer binary messages that would allow it to transfer an entire itinerary to the same app running on another device, typically to another user. This would enable one person to set up such an itinerary and then share it with a second person, who could then just tap a Book It button and be done! This would be far more efficient than emailing that itinerary as text and having the second person re-enter everything by hand.

On a simpler note, publishing a URI makes it super-simple for one person to tap-and-share whatever they're looking at with another person, again avoiding the circuitous email route or other forms of sharing. A quick tap, and you're seeing what I'm seeing. Again, though, there's so much more than can be shared that it's a great thing to consider in your design, especially if you're targeting mobile devices. "What do people near each other typically do together?" That's the question to ask—and to answer in the form of a great proximity app.

Do note that the URIs you share don't have to be [http://](#) references to websites but can contain any URI scheme. If there's an app associated with that URI scheme, tap-to-share also becomes tap-to-activate, because Windows will launch the default app for that association. And if there's no association, Windows will ask if you want to acquire a suitable app from the Store. You can also consider using a Windows Store URI that will lead a user to directly install an app. Those URIs are described on [Creating links with the Windows Store protocol](#).

Such URIs make it possible for RFID tags, whose messages are basically hardcoded into the device, to support tap-to-share and tap-to-activate scenarios. When you tap an RFID tag to an NFC-capable device, the `ProximityDevice` object will fire a `devicearrived` event. An app can then receive the tag's message through `ProximityDevice.subscribeForMessage`. This means that the app will need to know what type of message might be sent from that tag—it might be a standard type, or the app might be written specifically for tags with specific programming. For example, an art gallery could place tags near every piece it displays and then make an app available in the Windows Store for that gallery (or any other galleries that work in cooperation) that knows what messages those tags will send. If the message has an appropriate URI scheme in it, tapping on an RFID tag can help the user acquire an app and enjoy a rich experience.

For more on this topic, look for an NFC-related post on the [Windows Developer Blog](#). (One was in the publishing queue at the time of writing but not yet available.)

Printing Made Easy

An embarrassingly long time ago, when I was first working in the computer industry, I remember hearing excited talk about the "paperless office" and how very soon now we wouldn't need things like printers because everything would be shuttled around digitally.

Decades later, we do find ourselves shuttling around plenty of digital content, and yet printing still seems alive and well (except for this present book, of course, where early on we decided on an ebook format so that we could use extensive hyperlinks and color!). Maybe we still like paper for how it feels, how it uses our eyes differently, how it's cheap and disposable (unlike your Windows 8 tablet), how it can be used to start fires in a pinch or make airplanes, and how it makes good use of all the small trees that get thinned out of commercial tree farms (at least here in the western United States). Maybe too it's just part of the human experience—after all, as much as we play with our computers, we do still live in a physical world with physical objects, so it makes total sense that we continue to appreciate placing information onto physical media.

Sometimes I wonder whether the idea of the paperless office wasn't fueled in part by the fact that many apps didn't implement printing very well, an artifact of it being a difficult task to begin with. (And then there were printer drivers of dubious quality, connection difficulties, and many other challenges.) But gradually the whole world of printing has improved, both for consumers and for developers.

Of course, printing isn't always about going to paper either. I frequently use a PDF "printer" to create read-only copies of documents that are more suitable to sharing in many cases than my originals. Occasionally I print to a fax machine (which sends a fax), and more occasionally I'll print an email or web page directly to Microsoft OneNote for filing. In fact, I highly recommend setting your default printer to a digital target of some kind when working on printing features in your own app. That way you'll avoid producing copious amounts of scratch paper in the process, unless you happen to own a tree farm that you'll be thinning in a couple of years!

Get the backstory If you want to know more about how printing as a whole has been reimaged, check out [Simplifying printing in Windows 8](#) on the Building Windows 8 Blog, a post that provides deep soul satisfaction knowing that there are fewer drivers in your printing future.

To understand how to implement printing in an app, let's first see what it looks like to the user. Then we'll see how to ready content for printing and how to handle the printing-related events from Windows.

Note A Windows Store app written in JavaScript can use the `window.print` method to print with default settings. It's not recommended, however, because it doesn't work with the print UI and doesn't always produce the best output. Windows Store apps should give the user the full Windows 8 experience as described here.

The Printing User Experience

Printing typically starts in an app where the user is looking at something they want to print and invokes an appropriate command. In Scenario 2 of the [Print sample](#), for instance, whose code we'll be looking at in the next section, we see a big block of content along with a Print button, as shown in Figure 15-5. Note that such a Print button would normally be on the app bar and not on the app canvas, but this is a sample.

To start printing, the user can either tap this Print button or open the Charms bar and select Devices. Either way, if the app is registered for printing—that is, it's listening for the event that's raised from the Devices charm and provides suitable content—the user will see a list of print targets, as shown on the left side of Figure 15-6. If the app doesn't have printable content (that is, it doesn't listen for the event or provides no content in response), the user will see a panel like that on the right side of Figure 15-6. This is very much the same experience that a user sees with the Share charm depending on whether the app provides data for that feature. You've likely seen the epic fail message of "This app can't share." Printing supplies a similar disappointment for apps that lack the capability. Don't let your app be one of them.

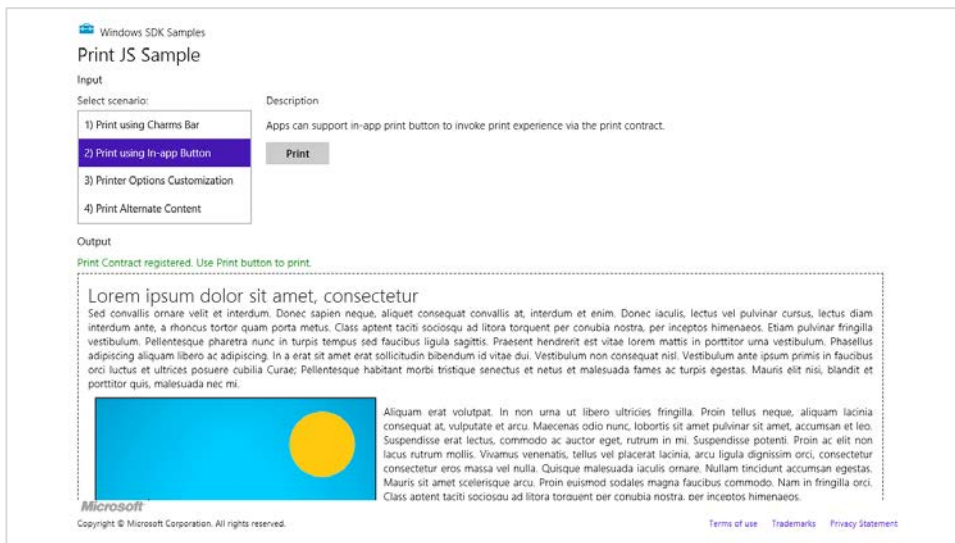


FIGURE 15-5 Scenario 2 of the Print sample shows a typical app with something ready to print.

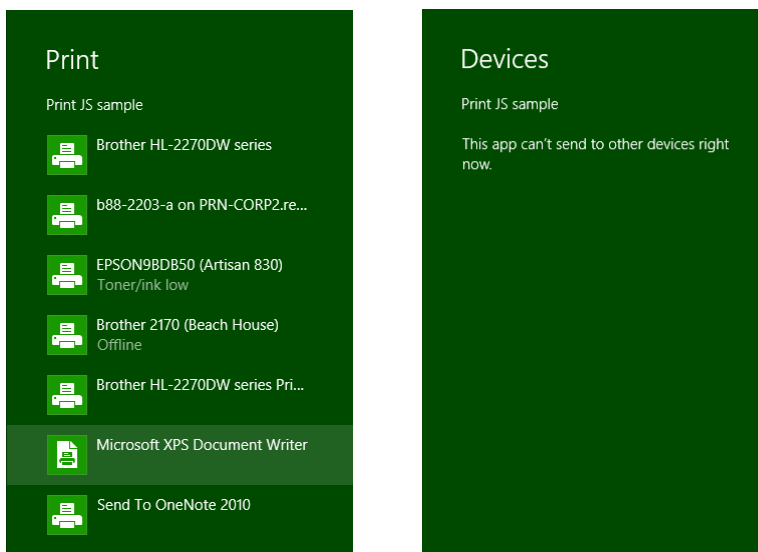


FIGURE 15-6 The Devices charm when an app has available print content (left) and when it doesn't (right).

From this point on, the system is really just taking whatever content the app provides and displays UI based on the capabilities of the printer driver, as shown in Figure 15-7. From the app's point of view, it thankfully gets all of this for free! The app can also indicate additional options to customize the UI, such as paper size and duplex printing, as shown in Figure 15-8, which comes from Scenario 3 of the sample.

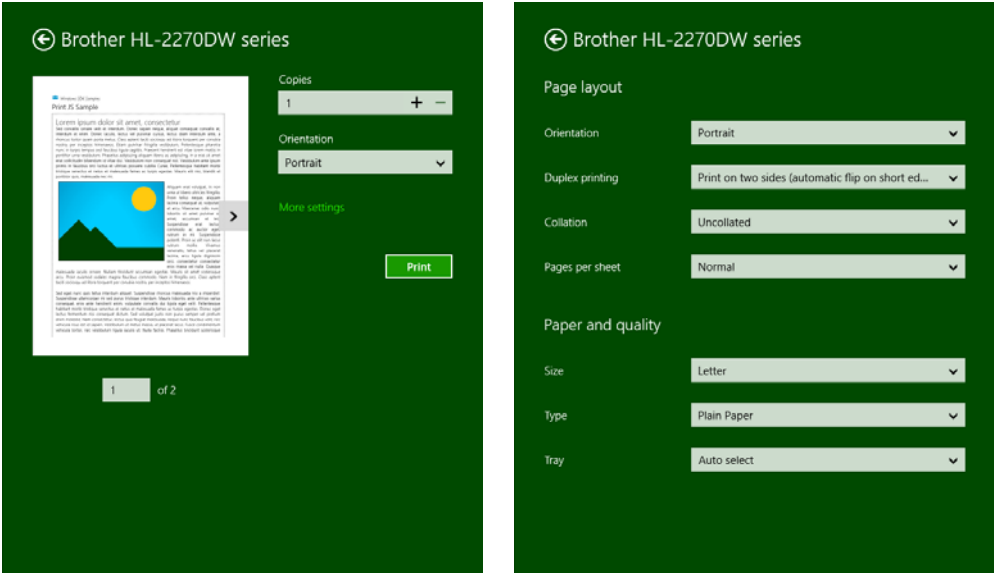


FIGURE 15-7 Print preview and printer options are shown once the user selects a printer. The More Settings link on the left is what opens the options pane on the right.



FIGURE 15-8 The Print pane reflecting customization options indicated by the app.

Print Document Sources

No matter where the user might want to print content, the important thing is to make that content ready for printing. The key function you need to know about here is not found in WinRT but in the `MSApp` object: `MSApp.getHtmlPrintDocumentSource`. I like the way the documentation put it: “This method is used as the bridge between HTML and [Windows 8 app] style printing. In other words, this is how an app dev says ‘give me some stuff to print.’” What you give it is an HTML *document* that contains your content.

I emphasize the word *document* here because what you pass to `getHtmlPrintDocumentSource` cannot be any arbitrary element in the DOM. It must be the same kind of thing that the `document` variable always points to, or else you’ll see a run-time exception with “no such interface supported.”

So where do you get such an object?

If what your app is showing on the screen is exactly what you want to print, you can just use the `document` object directly. This is what Scenarios 1–3 of the [Print sample](#) do:

```
MSApp.getHtmlPrintDocumentSource(document);
```

Of course, you don’t necessarily want to print everything on the screen; you can see that what’s on the screen in Figure 15-4 and what appears in the print preview of Figure 15-6 and Figure 15-7 is different. This is where the `print` media query in CSS comes into play:

```
@media print {  
    /* Print-only styles */  
}
```

Simply said, if there’s anything you don’t want to show up in the printed output, set the `display: none` style within this media query. An alternate strategy, one that the sample employs, is to create a separate CSS file, such as `css/print.css`, and link it in your HTML file with the `media` attribute set to *print* (see `html/scenario1.html`):

```
<link rel="stylesheet" type="text/css" href="/css/print.css" media="print" />
```

Print styles need not be limited to visibility of content: you can also use it however you like to arrange that content for more printer-friendly output. In a way, printing is like another view state where you’re not adding to or changing the content; you’re simply changing the visibility and layout. There are also some events you can use to do more specific formatting before and after printing has happened, as we’ll see later.

But what if the content you want to print isn’t your `document` object at all? How do you create another? There are several options here:

- In the `document.body.onbeforeprint` event handler, append additional child elements to the document and use the `document.body.onafterprint` event to remove them (the structure of such handlers is shown in Scenario 2 of the Print sample). If your print CSS leaves only those newly added elements visible, that’s all that gets printed. This very effectively controls the entire

print output, such as adding additional headers and footers that aren't visible in the app. You might have a place in the app, in fact, where the user can configure those headers and footers.

- Call `document.createDocumentFragment` to obtain a document fragment and then populate it with whatever elements you want to print. `getHtmlPrintDocumentSource` accepts such a fragment.
- If you have an `iframe` whose `src` is set to an SVG document (one of the tips we discussed for SVG's in Chapter 10), obtain that SVG document directly through the `iframe` element's `contentDocument` property. This too can be passed directly to `getHtmlPrintDocument-Source` and will print just that SVG, for example:

```
<!-- in HTML -->
<iframe id="diagram" src="/images/diagram.svg"></iframe>

//In JavaScript
var frame = document.getElementById("diagram");
args.setSource(MSApp.getHtmlPrintDocumentSource(frame.contentDocument));
```

- If you want to print the contents of an altogether different HTML page, create a `link` element in the document `head` that points to that other page for print media (see below). This will redirect `getHtmlPrintDocumentSource` to process that page's content instead.

The latter is demonstrated in Scenario 4 of the Print sample, where a `link` element is added to the document with the following code (`js/scenario4.js`):

```
var alternateLink = document.createElement("link");
alternateLink.setAttribute("id", "alternateContent");
alternateLink.setAttribute("rel", "alternate");
alternateLink.setAttribute("href", "http://go.microsoft.com/fwlink/?LinkId=240076");
alternateLink.setAttribute("media", "print");
document.getElementsByTagName("head")[0].appendChild(alternateLink);
```

Here the `rel` attribute indicates that this is alternate content, the `media` attribute indicates that it's only for print, and `href` points to the alternate content (`id` is optional). Note that if the target page has any print-specific media queries, those are certainly applied when creating the print source.

Providing Print Content and Configuring Options

Now that we know how to get a source for print content, it's very straightforward to provide that content to Windows for printing.

First, obtain the `Windows.Graphics.Printing.PrintManager` object as follows:

```
var printManager = Windows.Graphics.Printing.PrintManager.getForCurrentView();
```

and then listen for its `printtaskrequested` event (a WinRT event), either through `addEventListener` or by assigning a handler as done in the sample:

```
printManager.onprinttaskrequested = onPrintTaskRequested;
```

If you don't add a handler for this event, the user will see the message on the right side of Figure 15-5 when invoking the Devices charm unless you've also registered for other device-related events such as [Windows.Media.PlayTo.PlayToManager.sourceRequested](#), as we saw at the end of Chapter 10.

If you want to directly invoke printing from an app command, such as the Print button in Scenario 2 of the sample, call the [PrintManager.showPrintUIAsync](#) method. This is equivalent to the user invoking the Devices charm when the app has registered for the `printtaskrequested` event.

The `printtaskrequested` event is fired when the Devices charm is invoked. In response, your handler creates a [PrintTask](#) object with a callback function that will provide the content document when needed. Here's how that works. First, your handler receives a [PrintTaskRequest](#) object that has just three members:

- `deadline` The date and time that indicates how long you have to fulfill the request.
- `getDeferral` Returns a [PrintTaskRequestedDeferral](#) object in case you need to perform any async operations to fulfill the request. As with all deferrals, you call its `complete` method when the async operation has finished.
- `createPrintTask` Creates a [PrintTask](#) with a given title and a function that provides the source document when requested.

The structure of `createPrintTask` is slightly tricky. While it returns a [PrintTask](#) object through which you can set options and listen to task-related events, as we'll see shortly, its `source` property is read-only. So, instead of creating a task and storing your content document in this property, you instead provide a callback function that does the job when requested. The function itself is simple: it just receives a [PrintTaskSourceRequestedArgs](#) object whose `setSource` method you call with what you get back from `MSApp.getHtmlDocumentPrintSource`.

This is typically where you can also do other work to configure the task, so let's take an example from Scenario 3 of the Print sample (where I've added a namespace variable for brevity):

```
function onPrintTaskRequested(printEvent) {
    var printTask = printEvent.request.createPrintTask("Print Sample", function (args) {
        args.setSource(MSApp.getHtmlPrintDocumentSource(document));

        // Choose the printer options to be shown. The order in which the options are
        // appended determines the order in which they appear in the UI
        var options = Windows.Graphics.Printing.StandardPrintTaskOptions;
        printTask.options.displayedOptions.clear();
        printTask.options.displayedOptions.append(options.copies);
        printTask.options.displayedOptions.append(options.mediaSize);
        printTask.options.displayedOptions.append(options.orientation);
        printTask.options.displayedOptions.append(options.duplex);

        // Preset the default value of the printer option
        printTask.options.mediaSize =
            Windows.Graphics.Printing.PrintMediaSize.northAmericaLegal;

        // Register the handler for print task completion event
```

```

        printTask.oncompleted = onPrintTaskCompleted;
    });
}

```

Note that [PrintTaskSourceRequestedArgs](#) also contains a [getDeferral](#) method, should you need it, along with a [deadline](#).

Tip If you step through the code in your [printtaskrequested](#) handler but you pass the deadline, the print UI will time out and say there's nothing available to print. This might not be an error in the app at all—take off the breakpoints and run again to check.

You can exercise some control over the appearance of the print UI through [PrintTask.options](#), in which context you should review [Guidelines for print-capable Windows Store apps](#). The [options](#) object here, of type [PrintTaskOptions](#), has a number of properties. A few obvious numerical ones are [maxCopies](#), [minCopies](#), and [numberOfCopies](#). You can also call [getPageDescription](#) with a page number to obtain a [PrintPageDescription](#) with resolution information for that page.

Then there is a host of properties whose values come from various printing enumerations:

PrintTaskOptions Property	Windows.Graphics.Printing Enumeration
binding	PrintBinding
collation	PrintCollation
colorMode	PrintColorMode
duplex	PrintDuplex
holePunch	PrintHolePunch
mediaSize	PrintMediaSize
mediaType	PrintMediaType
orientation	PrintOrientation
printQuality	PrintQuality
staple	PrintStaple

[PrintTaskOptions.displayedOptions](#), for its part, is a vector of strings that must come from the [StandardPrintOptions](#) class, as shown in the code above. Each of these controls the visibility of the option in the print UI if, of course, the printer supports it (otherwise the option will not be shown). The full list of options is [binding](#), [collation](#), [colorMode](#), [copies](#), [duplex](#), [holePunch](#), [inputBin](#), [mediaSize](#), [mediaType](#), [nUp](#), [orientation](#), [printQuality](#), and [staple](#).

Take special note of the [mediaSize](#) property, for which there are literally 172 different values in the [PrintMediaSize](#) enumeration that reflect all the sizes of paper, envelopes, and so forth that we find around the world. When you intend to market a print-capable Windows Store app in different regions, you might want to include [mediaSize](#) in [displayedOptions](#) and set its value to something that's applicable to the region (as the code above is doing for legal size paper). Even so, the media size is typically available in the More Settings panel in the print UI, depending on what the printer in question supports, so users will have access to it.

The final bit to mention in the code above is that a `PrintTask` has a `completed` event, along with `previewing`, `progressing`, and `submitting`. You can use these to reflect the status of print tasks in your app should you choose to do so. More information about the task itself is also available through its `properties`, which will typically contain the title you gave to the print job along with a unique ID. In all of this, however, you might have noticed a conspicuous absence of any method in `PrintTask` that would cancel a print job—in fact, there is none. This is because the HTML print model, as presently used by Windows Store apps written in JavaScript, is an all-or-nothing affair: once the job gets into the print engine, there's no programmatic means to stop it. The user can still go to the printer control panel on the desktop and cancel the job there, or revert to the old-school method of yanking out the paper tray, but at present an app isn't able to provide such management functions itself.

What We've Just Learned

- Although Windows Store apps in Windows 8 cannot access arbitrary hardware, they do have access to a fair number of devices through both the WinRT API and Win32 APIs like Xinput and those supporting Windows Portable Devices (WPD). In cases of Win32 APIs, a WinRT component provides a bridge to apps written in JavaScript.
- The `Windows.Devices.Enumeration` API allows an app to discover what hardware is installed on a machine. If an app can access specific types of devices, such as Bluetooth devices, it can use the enumeration to present a list from which the user can select a device to use.
- The `Windows.Networking.Proximity` API supports peer browsing (over WiFi Direct and Bluetooth) as well as *tap-to-connect* and *tap-to-share* scenarios with near field communication (NFC)–capable machines.
- Proximity connections can employ sockets for ongoing communication (like a multiplayer game) or can simply send messages from one device to another through a publish-and-subscribe mechanism, as is typical with tap-to-share scenarios, including RFID tags.
- Printing, having been reimagined for Windows 8 as a whole, is relatively easy to implement in a Windows Store app. It involves listening for the printing event when the Devices charm is invoked and providing HTML content to Windows.
- Printable content can come from the app's document, a document fragment, an SVG document, or a remote source. Such content can be customized using a CSS media query for print, and Windows takes care of the layout and flow of the information on the target printer.

Chapter 16

WinRT Components: An Introduction

At the very beginning of this book, in the first two pages of Chapter 1, “The Life Story of a Windows Store App,” we learned that apps can be written in a variety of languages and presentation technologies. For the better part of this book we’ve spoken of this as a somewhat singular choice: you choose the model that best suits your experience and the needs of your app and go from there.

At the same time, we’ve occasionally encountered situations where some sort of mixed language approach is possible, even necessary. In Chapter 1, in “Sidebar: Mixed Language Apps,” I introduced the idea that an app can actually be written in multiple languages. In Chapter 8, “State, Settings, Files, and Documents,” I mentioned that gaining access to database APIs beyond IndexedDB could be accomplished with a WinRT component. In Chapter 10, “Media,” we saw that JavaScript might not necessarily be the best language in which to implement a pixel crunching routine. And in Chapter 13, “Tiles, Notifications, the Lock Screen, and Background Tasks” we encountered the Notifications Extensions Library, a very useful piece of code written in C# that made the job of constructing an XML payload much easier from JavaScript. We also saw that background tasks can be written in languages that differ from that of the main app.

With the primary restriction that an app that uses HTML and CSS for its presentation layer cannot share a drawing surface with WinRT components written in other languages, the design of WinRT really makes the mixed language approach possible. As discussed in Chapter 1, WinRT components written in any language are made available to other languages through a projection layer that translates the component’s interface into what’s natural in the target language. All of WinRT is written this way, and custom WinRT components take advantage of the same mechanism. (We’ll see the core characteristics of the JavaScript projection later in this chapter.)

What this means for you—writing an app with HTML, CSS, and JavaScript—is that you can implement various parts of your app in a language that’s best suited to the task or technically necessary. As a dynamic language, JavaScript shows its strength in easily gluing together functionality provided by other components that do the heavy lifting for certain kinds of tasks (like camera capture, background transfers, etc.). Those heavy-lifting components are often best written in other language such as C++, where the compiled code runs straight on the CPU instead of going through runtime layers like JavaScript.

Indeed, when we speak of *mixed language apps*, you truly can use a diverse mix.⁷⁶ You can write a C# component that's called from JavaScript, but that C# component might then turn right around and invoke a component written in C++. Again, it always means that you can use the language for any particular job, including those where you need to create your own asynchronous operations (that is, to run code on concurrent threads that don't block the UI thread). In this context it's helpful to also think through what this means in relationship to web workers, something a Windows Store app can employ if desired.

In this penultimate chapter we'll first look at the different reasons why you might want to take a mixed language approach in your app. We'll then go through a couple of quickstarts for C# and C++ so that we understand the structure of these components, how they appear in JavaScript, and the core concepts and terminology. The rest of the chapter will then primarily give examples of those different scenarios, which means we won't necessarily be going deep into the many the mechanical details. I've chosen to do this because there is very good documentation on those mechanics, especially the following:

- [Creating Windows Runtime Components in C++](#)
- [Walkthrough: Creating a basic Windows Runtime component in C++](#)
- [Creating Windows Runtime Components in C# and Visual Basic](#) (and its subsidiary topics)
- [Walkthrough: creating a basic Windows Runtime component in C# or Visual Basic](#)

Don't let the word "basic" in the walkthrough titles deter you: all these topics are comprehensive cookbooks that cover the fine details of working with data types like vectors, maps, and property sets; declaring events; creating async operations; and how all this shows up in JavaScript. We'll see some of these things in the course of this introduction, but with great topics covering the *what*, we'll be spending our time here on *why* we'd want to use such components in the first place and the problems they can help solve. Plus, I want you to save some energy for the book's finale in the next chapter (a rather healthy one, by all reckoning), where we'll talk about getting your app out to the world once you solve those problems!

Note By necessity I have to assume in this chapter that you have a little understanding of the C# and C++ languages, as this is not the place to cover the basics. If these languages are entirely new to you, spending a few hours familiarizing yourself with them will improve your mileage with this chapter.

⁷⁶ The documentation on the Windows Developer Center along with samples in the Windows SDK sometimes refer to mixed language apps as "hybrid apps." I've chosen to avoid the latter term because it already has a meaning in the context of other client platforms, namely apps that employ a fair amount of web content as you can do with Windows Store apps and `iframe` elements.

Choosing a Mixed Language Approach (and Web Workers)

There are many reasons to take a mixed language approach in your app, which can again include any number of WinRT components written in C#, Visual Basic (hereinafter just VB), and/or C++:

- You can accomplish certain tasks faster with higher performance code. This can reduce memory overhead and also consume less CPU cycles and power, an important consideration for background tasks for which a CPU quota is enforced—you might get much more done in 2 CPU seconds in C++ than with JavaScript, because there's no language engine involved.
- C#, Visual Basic, and C++ have access to a sizable collection of additional APIs that are not available to JavaScript. These include .NET, Win32, and COM (Component Object Model) APIs, including the non-UI features of DirectX such as XAudio and Xinput. We'll see a number of examples in the "Access to Additional APIs" section later in this chapter.
- Access to other APIs might also be necessary for utilizing third-party .NET/Win32/COM libraries and also gives you the ability to reuse existing code that you might have in C#, VB, or C++. The [Developing Bing Maps Trip Optimizer, a Windows Store app in JavaScript and C++](#) topic shows a complete scenario along these lines, specifically that of migrating an ActiveX control to a WinRT component so that it can be used from an app, because ActiveX isn't directly supported. (We won't cover this scenario further in this chapter.)
- It can be easier to work with routines involving many async operations by using the `await` keyword in C# and Visual Basic, because the structure is much cleaner than with promises. An example of this can be found in the Here My Am! app of Chapter 17, "Apps for Everyone," where the `transcodeImage` function written in JavaScript for Chapter 13 is rewritten in C# (see [Utilities.ImageConverter.TranscodeImageAsync](#) in the Utilities project).
- A WinRT component written in C++ is more effective at obfuscating sensitive code than JavaScript and .NET languages. Although it won't obfuscate the interface to that component, its internal workings are more painstaking to reverse-engineer.
- A WinRT component is the best way to write a non-UI library that other developers can use in their chosen language or that you can just use in a variety of your own projects, like the Notifications Extensions library we saw in earlier chapters. In this context, see [How to: Create a software development kit](#), which includes details how the component should be structured to integrate with Visual Studio.
- Although you can use [web workers](#) in JavaScript to execute code on different threads, a WinRT component can be much more efficient for custom asynchronous APIs. Other languages can also make certain tasks more productive, such as using language-integrated queries (LINQ) from C#/VB, creating parallel loops in C#/C++, using C++ [Accelerated Massive Parallelism \(AMP\)](#) to take advantage of the GPU, and so on.

Again, components can also make use of one another—the component system has no problem with that. I reiterate this point because it becomes relevant in the context of the last bullet above—web workers—and running code off the UI thread.

You are again wholly free to use web workers (or just *workers*) in a Windows Store app. In fact, Visual Studio provides an item template for exactly this purpose: right-click your project in Visual Studio's solution explorer, select Add > New Item, and then choose Dedicated Worker. I'll also show an example later on in the section "JavaScript Workers." The real drawback here, compared with WinRT components, is that communication between the worker and the main app thread is handled entirely through the singular `postMessage` mechanism; data transferred between the app and the worker must be expressed as properties of a message. In other words, workers are set up for a client-server architecture more so than one of objects with properties, methods, and events, though you can certainly create structures for such things through messages.

What this means is that using workers with multiple asynchronous operations can get messy. By comparison, the methods we've seen for working with async WinRT operations—WinJS promises—is much richer, especially when you need to chain or nest async operations, raise exceptions, and report progress. Fortunately, there is a way to wrap a worker within a promise, as we'll see again in the "JavaScript Workers" section.

What also interesting to think about is how you might use a worker written in JavaScript to act as an agent that delegates work to WinRT components. The JavaScript worker then serves as the glue to bring all the results from those components together, reporting a combined result to the main app through `postMessage`.

Along these same lines, if you have some experience in .NET languages like C# and Visual Basic along with C++, you'll know that their respective programming models have their own strengths and weaknesses. Just as you can take advantage of JavaScript's dynamic nature where it serves best, you can use the managed nature of .NET where it relieves you from various burdens like managing memory and reference counts and then use C++ where you really want the most performant code.

In short, you can have your main app's UI thread in JavaScript delegate a task to a worker in JavaScript, which then delegates some tasks to a WinRT component written in C#, which might then delegate its most intensive tasks to still other components written in C++. Truly, the combination of workers with WinRT components gives you a great deal of flexibility in your implementation.

Note One potential disadvantage to using WinRT components in C++ in your app is that while JavaScript and .NET languages (C#/VB) are architecture-neutral and can target any CPU, C++ components must be separately compiled for x86, x64, and ARM. This means that your app will potentially have three separate packages in the Windows Store. However, a package for x86 will also work on x64, which would eliminate one of the specific targets if creating a specific x64 package doesn't really buy you any performance gains.

Quickstarts: Creating and Debugging Components

When you set out to add a WinRT component to your project, the easiest place to start is with a Visual Studio item template. Right-click your *solution* (not your project) in Visual Studio's solution explorer, select Add > New Project, and then choose the Windows Runtime Component item listed under the Visual Basic, Visual C#, or Visual C++ > Windows Store nodes as shown in Figure 16-1 (for a C# project).

In the sections below, we'll look at both the C# and C++ options here as we fulfill a promise made in Chapter 10 to improve the performance of the Image Manipulation example in this book's companion content. That earlier sample performed a grayscale conversion on the contents of an image file and displayed the results on a [canvas](#). We did the conversion in a piece of JavaScript code that performs quite well, actually, but it's good to ask if we can make it better. Our first two WinRT components, then, are equivalent implementations written in C# and C++. Together, all three give us the opportunity to compare relative performance, as we'll do in "Comparing the Results."

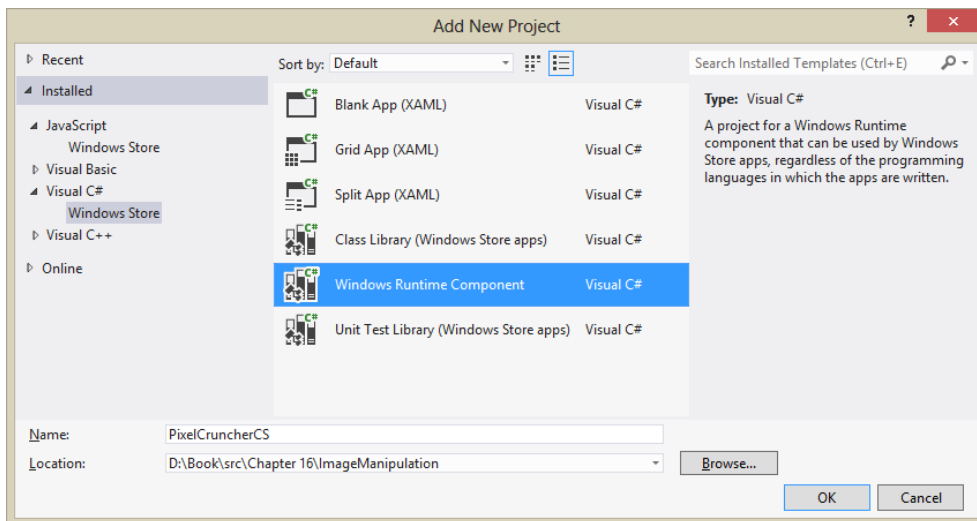


FIGURE 16-1 Visual Studio's option for creating a C# WinRT Component project; similar options appear under Visual Basic > Windows Store and Visual C++ > Windows Store.

One problem with all these implementations is that they still run on the UI thread, so we'll want to look at making the operations asynchronous. We'll come back to that later, however, in the "Key Concepts and Details" section so as to avoid making these quickstarts less than quick!

I've written the C# and C++ sections below assuming that you're read through them both in order. I will specifically be introducing terminology and tooling considerations in the first section that apply to the second.

Sidebar: WinRT Components vs. Class Libraries (C#/VB) and Dynamic-Link Libraries

In the Add New Project dialog of Visual Studio, you'll notice that an option for a Class Library is shown for Visual C# and Visual Basic and an option for a DLL (dynamic-link library) is shown for C++. These effectively compile into assemblies and DLLs, respectively, which bear resemblances to WinRT components. The difference, however, is that these types of components can be used only from those same languages: a Class Library (.NET assembly) can be used by apps written in .NET languages but not from JavaScript. Similarly, a DLL can be called from C++ and .NET languages (the latter only through a mechanism called P/Invoke) but is not available to JavaScript. A WinRT component is the only choice for this purpose.

Sometimes a simple DLL is required, as with media extensions that provide custom audio/video effects or a codecs. These are not WinRT components because they lack metadata that would project them into other languages, nor is such metadata needed. Details on DLLs for the media platform can be found in [Using media extensions](#) and [Media extensions sample](#).

Quickstart #1: Creating a Component in C#

As shown in Figure 16-1, I've added a WinRT Component in C# to the Image Manipulation example, calling it PixelCruncherCS. Once this project has been added to the solution, we'll have a file called class1.cs in that project that contains a namespace `PixelCruncherCS` with one class:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace PixelCruncherCS
{
    public sealed class Class1
    {
    }
}
```

Not particularly exciting code at this point but enough to get us going. You can see that the class is marked `public`, meaning it will be visible to apps using the component. It is also marked `sealed` to indicate that other classes cannot be derived from it (due to current technical constraints). Both keywords are required for WinRT components in Windows 8. (These two keywords are `Public` and `NotInheritable` in Visual Basic.)

To test the interaction with JavaScript, I'll give the class and its file a more suitable name (*Tests* and *grayscale.cs*, since we'll be adding more to it) and create a test method and a test property::

```

public sealed class Tests
{
    public static string TestMethod(Boolean throwAnException)
    {
        if (throwAnException)
        {
            throw new System.Exception("Tests.TestMethod was asked to throw an exception.");
        }

        return "Tests.TestMethod succeeded";
    }

    public int TestProperty { get; set; }
}

```

If you build the solution at this point (Build > Build Solution), you'll see that the result of the PixelCruncherCS project is a file called PixelCruncher.winmd. The .winmd extension stands for Windows Metadata: a WinRT Component written in C# is a .NET assembly that includes extra metadata referred to as the component's Application Binary Interface or ABI. This is what tells Windows about everything the component makes available to other languages (those `public` classes), and it's also what provides IntelliSense for that component in Visual Studio and Blend.

In the app you must add a reference to the component so that it becomes available in the JavaScript namespace, just like the WinRT APIs. To do this, right-click References within the JavaScript app project and select Add Reference. In the dialog that appears, select Solution on the side and then check the box for the WinRT component project as shown in Figure 16-2.

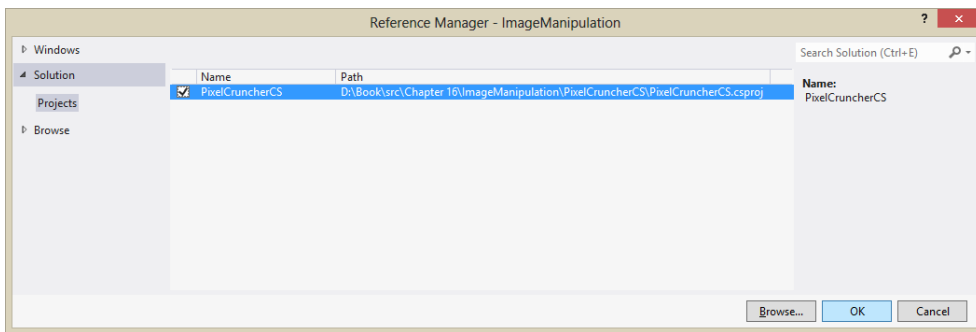
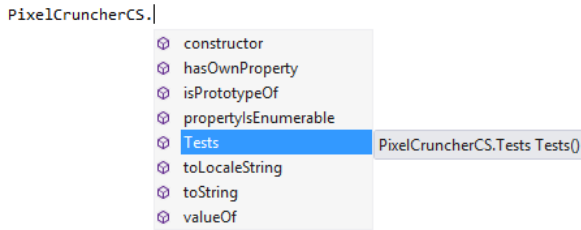
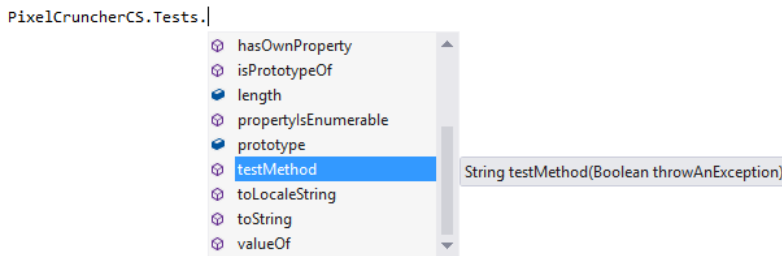


FIGURE 16-2 Adding a reference to a WinRT component within the same solution as the app.

When writing code that refers to the component, you always start with the namespace, PixelCruncherCS in our case. As soon as you enter that name and a dot, IntelliSense will appear for available classes in that namespace:



Once you add the name of a class and type another dot, IntelliSense appears for its methods and properties:



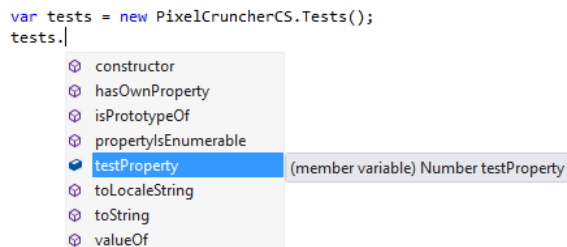
Note If you've made changes to namespace, class, and other names in your WinRT component project, you'll need to run Build > Build Solution to see the updated names within IntelliSense.

Here you can see that although the method name in the C# code is `TestMethod`, it's projected into JavaScript as `testMethod`, matching typical JavaScript conventions. This casing conversion is done automatically through the JavaScript projection layer for all WinRT components, including those in your own app.

Notice also that IntelliSense is showing only `testMethod` here but not `testProperty` (whose casing is also converted). Why is that? It's because in C#, `TestMethod` is declared as `static`, meaning that it can be executed without first instantiating an object of that class:

```
var result = PixelCruncherCS.Tests.testMethod(false);
```

On the other hand, `testProperty`, but not `testMethod`, is available on a specific instance:



I've set up `TestMethod`, by the way, to throw an exception when asked so that we can see how it's handled in JavaScript with a `try/catch` block:

```
try {  
    result = PixelCruncherCS.Tests.testMethod(true);  
} catch (e) {  
    console.log("PixelCruncherCS.Tests.testMethod threw: '" + e.description + "'.");  
}
```

Let's try this code. Attaching it to some button (see the `testComponentCS` function in the example's `js/default.js` file), set a breakpoint at the top and run the app in the debugger. When you hit that breakpoint, step through the code using Visual Studio's Step Into feature (F11 or Debug > Step Into). Notice that you do not step into the C# code, an effect of the fact that Visual Studio can debug either script or managed (C#)/native (C++) code in one debugging session but unfortunately not both. Clearly, using some console output will be your friend with such a limitation.

To set the debug mode, right-click your app project in solution explorer, select Debugging on the left, and then choose a Debugger Type on the right as shown in Figure 16-3. For debugging C#, select Managed Only or Mixed (Managed and Native). Then set breakpoints in your component code and restart the app (you have to restart for this change to take effect). When you trigger calls to the component from JavaScript, you'll hit those component breakpoints.

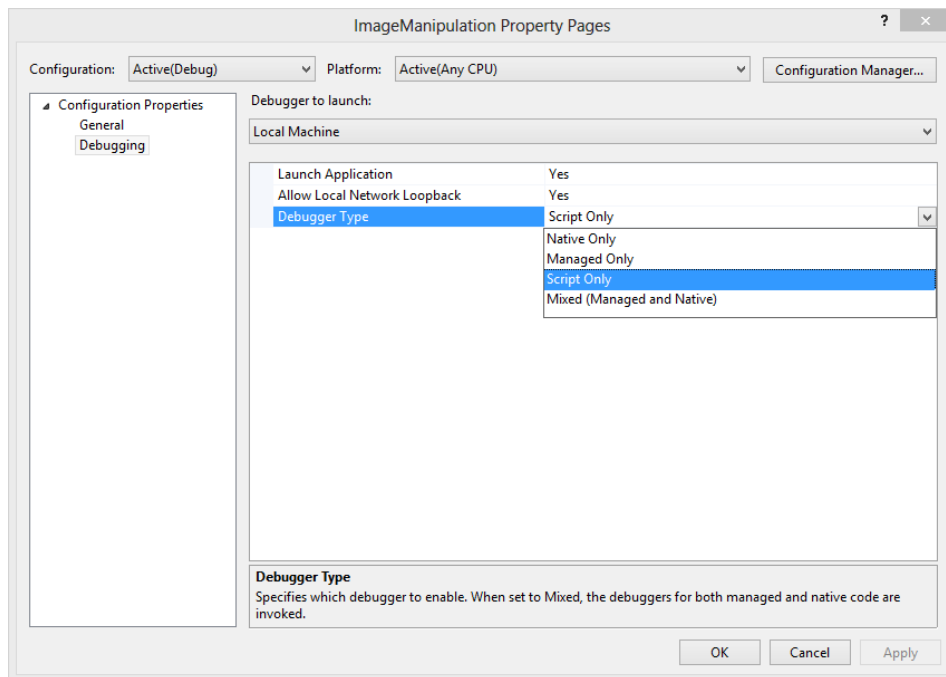


FIGURE 16-3 Debugger types in Visual Studio.

With the basic mechanics worked out, we're now ready to add our real functionality. The first step is to understand how to get the canvas pixel data arrays in and out of the WinRT component. In the JavaScript code (within the `copyGrayscaleToCanvas` method) we have an array named `pixels` with the original pixel data and another empty one in `imgData.data`, where `imgData` is obtained as follows:

```
var imgData = ctx.createImageData(canvas.width, canvas.height);
```

We can pass both these arrays into a component directly. A limitation here is that arrays passed to a WinRT component can be used for input *or* output, but not both—a component cannot just manipulate an array in place. The topic [Passing arrays to a Windows Runtime component](#) has the fine details. To make the story short, we fortunately already have an input array, `pixels`, and an *output* array, `imgData.data`, that we can pass to a method in the component:

```
var pc = new PixelCruncherCS.Grayscale();  
pc.convert(pixels, imgData.data); //Note casing on method name
```

Note The techniques shown here and in the article linked above apply only to *synchronous* methods in the WinRT component; arrays *cannot* be used with asynchronous operations. See “Key Concepts for WinRT Components” below for more on this topic.

To receive this array in C#, both parameters must to be appropriately marked with their directions. Such marks in C# are called attributes, not to be confused with those in HTML, and they appear in `[]`'s before the parameter name. In this particular case, the attributes appear as `[ReadOnlyArray()]` and `[WriteOnlyArray()]` preceding the parameters (the `ReadOnlyArray` and `WriteOnlyArray` methods are found in the `System.Runtime.InteropServices.WindowsRuntime` namespace). So the declaration of the method in the component, which again must be public, looks like this, just using a Boolean as a return type for the time being:

```
public Boolean Convert([ReadOnlyArray()] Byte[] imageDataIn,  
    [WriteOnlyArray()] Byte[] imageDataOut)
```

With this in place, it's a simple matter to convert the JavaScript grayscale code to C#:

```
public Boolean Convert([ReadOnlyArray()] Byte[] imageDataIn,  
    [WriteOnlyArray()] Byte[] imageDataOut)  
{  
    int i;  
    int length = imageDataIn.Length;  
    const int colorOffsetRed = 0;  
    const int colorOffsetGreen = 1;  
    const int colorOffsetBlue = 2;  
    const int colorOffsetAlpha = 3;  
  
    Byte r, g, b, gray;  
  
    for (i = 0; i < length; i += 4)  
    {  
        r = imageDataIn[i + colorOffsetRed];  
        g = imageDataIn[i + colorOffsetGreen];
```

```

        b = imageDataIn[i + colorOffsetBlue];

        //Assign each rgb value to brightness for grayscale
        gray = (Byte)(.3 * r + .55 * g + .11 * b);

        imageDataOut[i + colorOffsetRed] = gray;
        imageDataOut[i + colorOffsetGreen] = gray;
        imageDataOut[i + colorOffsetBlue] = gray;
        imageDataOut[i + colorOffsetAlpha] = imageDataIn[i + colorOffsetAlpha];
    }

    return true;
}

```

Sidebar: Back and Forth Debugging

Despite the fact that you can debug only one side of the JavaScript-component relationship within any given debugging session, it's not something you'll spend too long lamenting. After all, there's much work to be done; one must accept the state of affairs as they are and still be productive in those constraints.

When I was working on the examples in this chapter, I found that I developed a pattern wherein I set breakpoints in both the JavaScript and C# sources, after which I would routinely switch debugging modes and run the app again. That is, I'd run the app with script debugging and step through the JavaScript as far as I could (restarting when I found and corrected errors), until I clearly saw a problem in the component. Then I switched debugger mode to Managed and re-ran the app. With all my breakpoints set in the component, I could exercise the app in the same way and then step through the component's code, fixing errors, and restarting to repeat the process. At some later point, then, I'd find another problem in the JavaScript side, so I'd switch debug modes again and restart. It's not as convenient as being able to step from one piece of code into the other, but in the end it seemed to work reasonably well.

One suggestion that might speed things along is to write some bits of test code in C# or VB that will allow you to step from there into the component code directly. You might just build such test routines directly into the component, such that you can easily wire a button or two in JavaScript to those methods. This way you can probably tighten your iterations on the component code, being reasonably sure it works properly before invoking it from JavaScript with your real data.

Quickstart #2: Creating a Component in C++

To demonstrate a WinRT component written in C++, I've also added a project to the Image Manipulation example, calling it PixelCruncherCPP. The core code is the same as the C# example—manipulating pixels is a rather universal experience! The necessary code structure for the component, on the other hand, is unique to the language—C++ has ceremony all its own.

As we did with C#, let's start by adding a new project using the Visual C++ > Windows Runtime Component template, using the PixelCruncherCPP name. After renaming *Class1* to *Tests* in the code and renaming the files, we'll have the following code in the header (which I call grayscale.h, and omitting a compiler directive):

```
namespace PixelCruncherCPP
{
    public ref class Tests sealed
    {
    public:
        Tests();
    };
}
```

where we see that the class must be `public ref` and `sealed`, with a public constructor. These together make the object instantiable as a WinRT component. In *Tests.cpp* we have the following:

```
#include "pch.h"
#include "Grayscale.h"

using namespace PixelCruncherCPP;
using namespace Platform;

Tests::Tests()
{
}
```

Again, not too much to go on, but enough. (Documentation for the [Platform namespace](#), by the way, is part of the Visual C++ Language Reference.) To follow the same process we did for C#, let's add a static test method and a test property. The class definition is now:

```
public ref class Tests sealed
{
public:
    Tests();

    static Platform::String^ TestMethod(bool throwAnException);
    property int TestProperty;
};
```

and the code for `TestMethod` is this:

```
String^ Tests::TestMethod(bool throwAnException)
{
    if (throwAnException)
    {
        throw ref new InvalidArgumentException;
    }

    return ref new String(L"Tests.TestMethod succeeded");
}
```

When you build this project (Build > Build Solution) you'll see that we now get PixelCruncherCPP.dll and PixelCruncherCPP.winmd files. Whereas a C# assembly can contain both the code and the metadata, a C++ component compiles into separate code and metadata files. The metadata is again used to project the component's ABI into other languages and provides IntelliSense data for Visual Studio and Blend. If you now add a reference to this component in your app project—right-click the project > Add Reference > Solution, and then choose PixelCruncherCPP, as in Figure 16-2—you'll find that IntelliSense works on the class when writing JavaScript code.

You'll also find that the casing of the component's property and method names have also been changed. In fact, with the exception of the namespace, PixelCruncherCPP, everything we did to use the C# component in JavaScript looks exactly the same, as it should be: the app consuming a WinRT component does not need to concern itself with the language used to implement that component. The debugging experience, moreover, is also the same, except that you need to choose Native Only or Mixed (Managed And Native) in the debugger types dialog shown earlier in Figure 16-3.

Now we need to do the same work to accept arrays into the component, for which we can use [Array and WriteOnlyArray](#) as a reference. In C++, an input array is declared with `Platform::Array<T>^` and an output array as `Platform::WriteOnlyArray<T>^`, where we use `uint8` as the type here instead of `Byte` in C#:

```
bool Grayscale::Convert(Platform::Array<uint8>^ imageDataIn,
    Platform::WriteOnlyArray<uint8>^ imageDataOut)
```

The rest of the code is identical except for this one type change and for how we obtain the length of the input array, so we don't need to spell it out here. The code to invoke this class from JavaScript is also the same as for C#:

```
var pc2 = new PixelCruncherCPP.Grayscale();
pc2.convert(pixels, imgData.data);
```

Sidebar: The Windows Runtime C++ Template Library (WRL)

Visual Studio includes what is called the [Windows Runtime C++ Template Library](#), or WRL, that helps you write low-level WinRT components in C++. It's really a bridge between the raw COM level and what are called the C++/CX component extensions that we've actually been using in this section. If you have any experience with the Active Template Library (ATL) for COM, you'll find yourself right at home with WRL. For more information, see the documentation linked above along with the [Windows Runtime Component using WRL sample](#).

Comparing the Results

The Image Manipulation example in this chapter's companion content contains equivalent code in JavaScript, C#, and C++ to perform a grayscale conversion on image pixels. Taking a timestamp with `new Date()` around the code of each routine, I've compiled a table of performance numbers below.⁷⁷

	Average Milliseconds (five samples; dual-core 2.5GHz processor)		
Image Size	JavaScript	C#	C++
14.8K	8.4	7.2	6.4
231K	45.2	40	33.8
656K	76.6	65.8	54.4
1.98MB	798	728	598
4.57MB	796	750	637

A couple of notes and observations about these numbers and measuring them:

- When doing performance tests like this, be sure to set the build target to Release instead of Debug. This makes a tremendous difference in the performance of C++ code, because the compiler inserts all kinds of extra run-time checks in a debug build.
- When taking measurements, also be sure to run the Release app outside of the debugger (in Visual Studio select Debug > Start Without Debugging). If you've enabled script debugging, JavaScript will run considerably slower even with a Release build and could lead you to think that the language is far less efficient than it really is.
- If you run similar tests in the app itself, you'll notice that the time reported for the conversion is considerably shorter than the time it takes for the app to become responsive again. This is because the `canvas` element's `putImageData` method takes a long time to copy the converted pixels. Indeed, the majority of the time for the whole process is spent in `putImageData` and not the grayscale conversion.
- Assuming the CPU load for the grayscale conversion is roughly identical between the implementations, you can see that a higher performance component reduces the amount of time that the CPU is carrying that load. Over many invocations of such routines, this can add up to considerable power savings.
- The first time you use a WinRT component for any reason, it will take a little extra time to load the component and its metadata. The numbers above do not include first-run timings. Thus, if you're trying to optimize a startup process in particular, this extra overhead could mean that it's best to just do the job in JavaScript.

With these numbers we can see that C# runs between 6–21% faster than the equivalent JavaScript and C++ 25–46% faster. C++ also runs 13–22% faster than C#. This shows that for noncritical code,

⁷⁷ You might be interested in a series of [blog posts](#) by David Rousset about building a camera app using HTML and JavaScript. In [Part 4](#) of that series he offers much more in-depth performance analysis for a variety of devices, using pixel manipulation components much like we've been working with here.

writing a component won't necessarily give you a good return on the time you invest; it will be more productive to just stay in JavaScript. But using a component in a few places where performance really matters might pay off handsomely.

Tip There is much more to measuring and improving app performance than just offloading computationally intensive routines to a WinRT component. The [Analyzing the performance of Windows Store apps](#) and [Analyzing the code quality of Windows Store apps with Visual Studio code analysis](#) topics in the documentation will help you make a more thorough assessment of your app.

I also want to add that when I first ran these tests with the example program, I was seeing something like 100% improvements in C#/C++ over JavaScript. The reason for that had more to do with the nature of the `canvas` element's `ImageData` object (as returned by the canvas's `createImageData` method) than with JavaScript. In my original JavaScript code (since corrected in Chapter 10 as well), I was dereferencing the `ImageData.data` array to set every *r*, *g*, *b*, and *a* value for each pixel. When I learned how dreadfully slow that dereference operation actually is, I changed the code to cache the array reference in another variable and suddenly the JavaScript version became amazingly faster. Indeed, minimizing identifier references is generally a good practice for better performance in JavaScript. For more on this and other performance aspects, check out *High Performance JavaScript*, by Nicholas C. Zakas (O'Reilly, 2010).

Sidebar: Managed vs. Native

As shown in the previous section, going from JavaScript to C# buys you one level of performance gain, and going from C# to C++ buys another. Given that C++ is often more complex to work with, it's good to ask whether the extra effort will be worth it. In very critical situations where that extra 13–22% really matters, the answer is clearly yes. But there is another factor to consider: the difference between the managed environment of .NET languages (along with JavaScript, for that matter) and the native environment of C++.

Put simply, the reason why C#/VB code is often easier to write than C++ is that the .NET Common Language Runtime (CLR) provides a host of services like garbage collection so that you don't have to worry about every little memory allocation. What this means, however, is that your activities in C#/VB can also trigger extra processing within that runtime that could change the performance characteristics of your components.

For example, in the Image Manipulation example with this chapter—which really expanded into a test app for components!—I added a simple counting function in JavaScript, C#, and C++ (they all look about the same):

```
function countFromZero(max, increment) {  
    var sum = 0;  
  
    for (var x = 0; x < max; x += increment) {  
        sum += x;  
    }  
}
```

```
    return sum;
}
```

Running a count with a max of 1000 and an increment of 0.000001 (only use this increment outside the debugger—otherwise you might be waiting a while!), the timings I got averaged 2112ms for JavaScript, 1568ms for C#, and 1534ms for C++. Again, the difference between JavaScript and the other languages is significant (35–38% gain), but it's hardly significant between C# and C++.

However, I've occasionally found that after loading a number of images and running the grayscale tests that counting in JavaScript and/or C# can take considerably longer than before, most likely due to garbage collection, which can impact the performance of the JavaScript runtime and the CLR. This does not happen with C++, though of course high CPU demands elsewhere will slow down any process.

Be clear, though, that I said *occasionally*. You want to be aware of this, but I don't think you need worry about it for anything but the most critical pieces of code.

Key Concepts for WinRT Components

The WinRT components we've just seen in the "Quickstarts" section demonstrate the basic structure and operation of such components, but clearly there is much more to the subject. Again, because exploring all the nuances is beyond the scope of this chapter, I'll refer you again to the references given at the end of this chapter's introduction. Here I simply want to offer a summary of the most relevant points, followed by separate sections on asynchronous methods and the projection of WinRT into JavaScript.

Component Structure

- The output of a C#/VB component project is a .NET assembly with Windows metadata in a .winmd file; the output of a C++ component is a DLL with the code and a .winmd file with the metadata.
- Apps that use components must include the .winmd/DLL files in their projects and add a reference to them; it's not necessary to include the component source.
- Component classes that can be used by other languages are known as *activatable* classes. In C# these must be marked as `public sealed`, in Visual Basic as `Public NotInheritable`, and in C++ as `public ref sealed`. A component must have one activatable class to be usable from other languages.
- Classes can have `static` members (methods and properties) that can be used without instantiating an object of that class.

- A component can contain multiple public activatable classes as well as additional classes that are internal only. All public classes must reside in the same root namespace, which has the same name as the component metadata file.
- By default, all public classes in a component are visible to all other languages. A class can be hidden from JavaScript by applying the [WebHostHiddenAttribute](#) (that is, prefix the class declaration with `[Windows.Foundation.Metadata.WebHostHidden]` in C# or `[Windows::Foundation::Metadata::WebHostHidden]` in C++. This is appropriate for classes that work with UI (that cannot be shared with JavaScript, such as the whole of the `Windows.Xaml` namespace in WinRT) or others that are redundant with JavaScript intrinsics (such as `Windows.Data.Json`).
- For some additional structural options, see the following samples in the Windows SDK (all of which use the WRL; see “Sidebar: The Windows Runtime C++ Template Library” under “Quickstart #2”):
 - [Creating a Windows Runtime in-process component sample \(C++/CX\)](#)
 - [Creating a Windows Runtime in-process component sample \(C#\)](#)
 - [Creating a Windows Runtime EXE component with C++ sample](#)
 - [Creating a Windows Runtime DLL component with C++ sample](#)

Types

- Within a component, you can use native language types (that is, .NET types and C++ runtime types). At the level of the component interface (the Application Binary Interface, or ABI), you must use WinRT types or native types that are actually implemented with WinRT types. Otherwise those values cannot be projected into other languages. In C++, WinRT types exist in the [Platform](#) namespace, and see [Type System \(C++/CX\)](#); in C#/VB, they exist in the `System` namespace, and see [.NET Framework mappings of Windows Runtime types](#).
- A component can use structures created with WinRT types, which are projected into JavaScript as objects with properties that match the `struct` members.
- Collections must use specific WinRT types found in [Windows.Foundation.Collections](#), such as `IVector`, `IMap` (and `IMapView`), and `IPropertySet`. This is why we’ve often encountered vectors throughout this book.
- Arrays are a special consideration because they can be passed only in one direction as we saw in the quickstarts; each must therefore be marked as read-only or write-only. See [Passing arrays to a Windows Runtime component](#). Arrays also have a limitation in that they cannot be effectively used with async methods, because an output array will not be transferred back to the caller when the async operation is complete. We’ll talk more of this in “Implementing Asynchronous Methods” below.

Component Implementation

- When creating method overloads, make sure the *arity* (the number of arguments) is different for each one because JavaScript cannot resolve overloads only by type. If you do create multiple overloads with the same arity, one of them must be marked with the [DefaultOverloadAttribute](#) so that the JavaScript projection knows which one to use.
- A *delegate* (an anonymous function in JavaScript parlance) is a function object. Delegates are used for events, callbacks, and asynchronous methods. Declaring a delegate defines a function signature.
- The `event` keyword marks a public member of a specific delegate type as an event. Event delegates—the signature for a handler—can be typed (that is, `EventHandler<T>`), which means that the `EventArgs` to that handler will be an object of that type. Do note that typed event handlers like this, to support projection into JavaScript, require a COM proxy/stub implementation; see the four samples linked to above in the “Component Structure” section. Also see the topic [Custom events and event accessors in Windows Runtime Components](#) for .NET languages.
- Throwing exception: use the `throw` keyword in C#, VB, and C++. In C#/VB, you throw a new instance of an [exception type in the System namespace](#). In C++, you use `throw ref new` with one of the exception types within the Platform namespace, such as [Platform::InvalidArgumentException](#). These appear in JavaScript with a stack trace in the message field of the exception; the actual message from the component will appear in Visual Studio’s exception dialog.

Implementing Asynchronous Methods

For as fast as the C# and C++ routines that we saw in the quickstarts might be, fact of the matter is that they still take more than 50ms to complete while running on the UI thread. This is the recommended threshold at which you should consider making an operation asynchronous. This means running that code on other threads such that the UI thread isn’t blocked at all. To see the basics, the following sections show how to implement asynchronous versions of the simple `countFromZero` function we saw earlier in the “Sidebar: Managed vs. Native” section. We’ll do it first with a worker and then in C# and C++.

For C#/VB and C++ there is quite extensive documentation on creating async methods. The cookbook topics we’ve referred to already cover this in the subsections called [Asynchronous operations](#) and [Exposing asynchronous operations](#) for C#, and the “Adding asynchronous public methods to the class” section in the [C++ walkthrough](#). There is also [Creating Asynchronous Operations in C++ for Windows Store apps](#), along with a series of comprehensive posts on the Windows 8 Developer Blog covering both app and component sides of the story: [Keeping apps fast and fluid with asynchrony in the Windows Runtime](#), [Diving Deep with Await and WinRT](#), and [Exposing .NET tasks as WinRT asynchronous operations](#). Matching the depth of these topics would be a pointless exercise in

repetition, so the sections that follow focus on creating async versions of the pixel-crunching methods from the quickstarts and the lessons we can glean from that experience.

This turns out to be a fortuitous choice. The particular scenario that we've worked with—performing a grayscale conversion on pixel data and sending the result to a canvas—just so happens to reveal a number of complications that are instructive to work through and are not addressed directly in the other documentation. These include troubles with passing arrays between the app and a component, which introduces an interesting design pattern that is employed by some WinRT APIs. Even so, the solution brings us to something of a stalemate because of the limitations of the HTML `canvas` element itself. This forces us to think through some alternatives, which is a good exercise because you'll probably encounter other difficulties in your own component work.

JavaScript Workers

For pure JavaScript, workers are the way you offload code execution to other threads. A key point to understand here is that communication between the main app (the UI thread) and workers happens through the singular `postMessage` method and the associated `message` events. That is, workers are not like components in which you can just call methods and get results back. If you want to invoke methods inside that worker with specific arguments, you must make those calls through `postMessage` with a message that contains the desired values. On the return side, a function that's invoked inside the worker sends its results to the main app through its own call to `postMessage`.

For example, in the Image Manipulation example in this chapter—which is growing beyond its original intent for sure!—I placed the `countFromZero` function into `js/worker_count.js` along with a message handler that serves as a simple method dispatcher:

```
onmessage = function (e) {
    switch (e.data.method) {
        case "countFromZero":
            countFromZero(e.data.max, e.data.increment);
            break;

        default:
            break;
    }
};

function countFromZero(max, increment) {
    var sum = 0;
    max = 10;

    for (var x = 0; x < max; x += increment) {
        sum += x;
    }

    postMessage({ method: "countFromZero", sum: sum });
}
```

When this worker is started, the only code that executes is the `onmessage` assignment. When that handler receives the appropriate message, it then invokes `countFromZero`, which in turn posts its results. In other words, setting up a worker just means converting method calls and results into messages.

Invoking this method from the app now looks like this:

```
var worker = new Worker('worker_count.js');
worker.onmessage = function (e) { //e.data.sum is the result }

//Call the method
worker.postMessage({ method: "countFromZero", max: 1000, increment: .00005 });
```

Keep in mind with all of this that `postMessage` is itself an asynchronous operation—there's no particular guarantee about how quickly those messages will be dispatched to the worker or the app. Furthermore, when a worker is created, it won't start executing until script execution yields (as when you call `setImmediate`). This means that workers are not particularly well suited for async operations that you want to start as soon as possible or for those where you want to get the results as soon as they are available. For this reason, workers are better for relatively large operations and ongoing processing; small, responsive, and high-performance routines are better placed within WinRT components.

The `postMessage` mechanism is also not the best for chaining multiple async operations together, as we're easily able to do with promises that come back from WinRT APIs. To be honest, I don't even want to start thinking about that kind of code! I prefer instead to ask whether there's a way that we can effectively wrap a worker's messaging mechanism *within* a promise, such that we can treat async operations the same regardless of their implementation.

To do this, we somehow need a way to get the result from within the `worker.onmessage` handler and send it to a promise's completed handler. To do that, we use a bit of code in the main app that's essentially what the JavaScript projection layer uses to turn an async WinRT API into a promise itself:

```
// This is the function variable we're wiring up.
var workerCompleteDispatcher = null;

var promiseJS = new WinJS.Promise(function (completeDispatcher, errorDispatcher,
    progressDispatcher) {
    workerCompleteDispatcher = completeDispatcher;
});

// Worker would be created here and stored in the 'worker' variable

// Listen for worker events
worker.onmessage = function (e) {
    if (workerCompleteDispatcher != null) {
        workerCompleteDispatcher(e.data.sum);
    }
}

promiseJS.done(function (sum) {
    // Output for JS worker
});
```

The first things to understand here are what a promise actually does and how the promise is separate from the async operation itself. (It has to be, because WinRT APIs and components know nothing of promises.) A promise is really just a tool to manage a bunch of listener functions on behalf of an async operation, like our worker here. That is, when an async operation detects certain events—namely, completed, error, and progress—it wants to notify whomever has expressed an interest in those events. Those whomevers have done so by calling a promise’s `then` or `done` methods and providing one or more handlers.

Within `then` or `done`, all a promise really does is save those functions in a list (unless it knows the async operation is already complete, in which case it might just call the completed or error function immediately). This is why you can call `then` or `done` multiple times on the same promise—it just adds your completed, error, and progress handlers to the appropriate list within the promise. Of course, those lists are useless without some way to invoke the handlers they contain. For this purpose, a promise has three functions of its own that go through each list and invoke the registered listeners. That’s really the core purpose of a promise: maintain lists of listeners and call those listeners when asked.

The code that starts up an async operation, then, will want to use a promise to manage those listeners, hence the call to `new WinJS.Promise`. But it also needs to get to those functions in the promise that it should call to notify its listeners. This is the purpose of the anonymous function provided to the promise’s constructor. When the promise is initialized, this function is called with references to the functions that notify the listeners. The async operation code then saves whichever of these it needs for later use. In our worker’s case, we’re interested only in notifying the completed handlers, so we save the appropriate function reference in the `workerCompleteDispatcher` variable.

When we then detect that the operation is complete—when we receive the appropriate message from the worker—we check to make sure `workerCompleteDispatcher` is a good reference and then call it with the result value. That dispatcher will again loop through all the registered listeners and call them with that same result. In the code above, the only such listener is the anonymous function we gave to `promiseJS.done`.

Truth be told, it’s really just mechanics. To handle errors and progress, we’d simply save those dispatchers as well, add more specific code inside the `onmessage` event handler that would check `e.data` for other status values from the worker, and invoke the appropriate dispatcher in turn. Such relationships are illustrated in Figure 16-4.

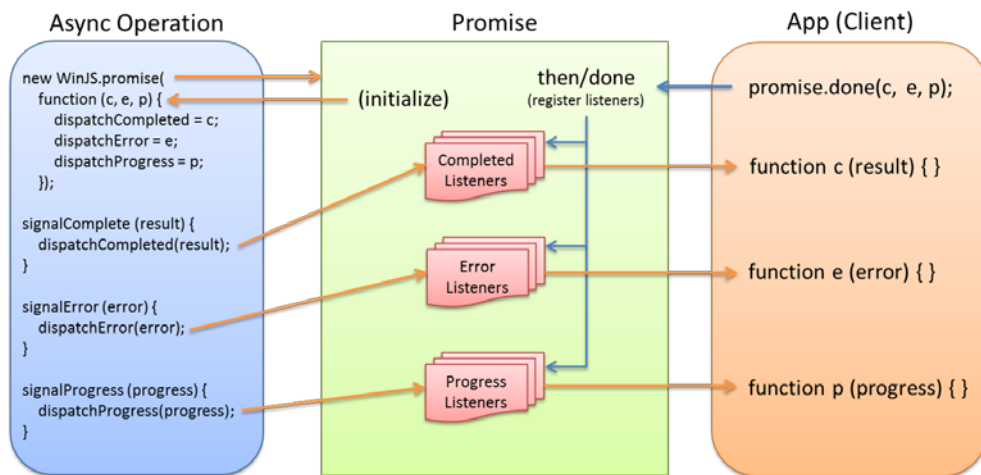


FIGURE 16-4 A promise manages and invokes listeners on behalf of an async operation.

Again, everything you see here with the exception of the call to `done` (which is client code and not part of the async operation) is what the JavaScript projection layer does for an async operation coming from WinRT. In those cases the async operation is represented by an object with an `IAsync*` interface instead of a worker. Instead of listening for a worker's `message` event, the projection layer just wires itself up through the `IAsync*` interface and creates a promise to manage connections from the app.

The code above is included in the Image Manipulation example accompanying this chapter. It's instructive to set breakpoints within all the anonymous functions and step through the code to see exactly when they're called, even to step into WinJS and see how it's working. In the end, what's meaningful is that this code gives us a promise (in `promiseJS`) that looks, feels, and acts like any other promise. This will come in very handy when we have promises from other async operations, as explained later in "Sidebar: Joining Promises." It means that we can mix and match async operations from WinRT APIs, WinRT components, and workers alike.

Async Basics in WinRT Components

Within a WinRT component, there are three primary requirements to make any given method asynchronous.

First, append *Async* to the method name, a simple act that doesn't accomplish anything technically but clearly communicates to callers that their need to treat the method differently from synchronous ones.

Second, the return value of the method must be one of the following [interfaces](#), shown in the table below, each one representing a particular combination of async behaviors, namely whether the method produces a result and whether the method is capable of reporting progress:

Interface (in Windows.Foundation)	Use Case
IAsyncAction	Use for an async method that produces no results (no arguments are sent to the completed handler) and reports no progress.
IAsyncActionWithProgress<TProgress>	Use for an async method that produces no results but does report progress, where <TProgress> is the data type of the argument sent to the progress handler.
IAsyncOperation<TResult>	Use for an async method that produces results of type <TResult> but reports no progress.
IAsyncOperationWithProgress<TResult, TProgress>	Use for an async method that produces results of type <TResult> and reports progress with type <TProgress> to a progress handler.

Having chosen the type of async method we're creating, we now have to run the method's code on another thread. It is possible here to utilize threads directly, using the thread pool exposed in the [Windows.System.Threading](#) API, but there are higher level constructs in both C#/VB and C++ that make the job much easier.

Async Methods in C#/Visual Basic In C# and Visual Basic we have the [System.Threading.Tasks.Task](#) class for this purpose. A [Task](#) is created through one of the static [Task.Run](#) methods. To this we give an anonymous function (called a *delegate* in .NET, defined with a *lambda operator* =>) that contains the code to execute. To then convert that [Task](#) into an appropriate WinRT async interface, we call the task's [AsAsyncAction](#) or [AsAsyncOperation](#) extension method. Here's what this looks like in a generic way:

```
public IAsyncOperation<string> SomeMethodAsync(int id)
{
    var task = Task.Run<string>( () => // () => in C# is like function () in JS
    {
        return "Here is a string.";
    });

    return task.AsAsyncOperation();
}
```

If the code inside the task itself performs any asynchronous operations (for which we use the C# [await](#) keyword as described in the blog posts linked earlier), the delegate must be marked with [async](#):

```
public IAsyncOperation<string> SomeMethodAsync(int id)
{
    var task = Task.Run<string>(async () =>
    {
        var idString = await GetMyStringAsync(id); // await makes an async call looks sync
        return idString;
    });

    return task.AsAsyncOperation();
}
```

Note that `Task.Run` does not support progress reporting and the `AsAsyncAction` and `AsAsyncOperation` extension methods don't support cancellation. In these cases you need to use the `System.Runtime.InteropServices.WindowsRuntime.AsyncInfo` class and one of its `Run` methods as appropriate to the chosen async behavior. The `Task.AsAsyncOperation` call at the end is unnecessary here because `AsyncInfo.Run` already provides the right interface:

```
public IAsyncOperation<string> SomeMethodAsync(int id)
{
    return AsyncInfo.Run<string>(async (token) =>
    {
        var idString = await GetMyStringAsync(id);
        token.ThrowIfCancellationRequested();
        return idString;
    });
}
```

In this code, `AsyncInfo.Run` provides the delegate with an argument of type `System.Threading.CancellationToken`. To support cancellation, you must periodically call the token's `ThrowIfCancellationRequested` method. This will pick up whether the original caller of the async method has canceled it (for example, calling a promise's `cancel` method). Because cancellation is typically a user-initiated action, there's no need to call `ThrowIfCancellationRequested` inside a very tight loop; calling it every 50 milliseconds or so will keep the app fully responsive.

Alternately, if a method like `GetMyStringAsync` accepted the `CancellationToken`, you could just pass the token to it. One strength of the `CancellationToken` model is that it's highly composable: if you receive a token in your own async call, you can hand it off to any number of other functions you call that also accept a token. If cancellation happens, the request will automatically be propagated to all those operations.

Note that WinRT methods can accept a token because of an `AsTask` overload. Instead of this:

```
await SomeWinRTMethodAsync();
```

you can use this:

```
await SomeWinRTMethodAsync().AsTask(token);
```

Anyway, given these examples, here's a noncancellable async version of `CountFromZero`:

```
public static IAsyncOperation<double> CountFromZeroAsync(double max, double increment)
{
    var task = Task.Run<double>(() =>
    {
        double sum = 0;

        for (double x = 0; x < max; x += increment)
        {
            sum += x;
        }

        return sum;
    });
}
```

```

});

return task.AsAsyncOperation();
}

```

The `IAsyncOperation` interface returned by this method, like all the async interfaces in `Windows.Foundation`, gets projected into JavaScript as a promise, so we can use the usual code to call the method and receive its results (`asyncVars` is just an object to hold the variables):

```

asyncVars.startCS = new Date();
var promiseCS = PixelCruncherCS.Tests.countFromZeroAsync(max, increment);
promiseCS.done(function (sum) {
    asyncVars.timeCS = new Date() - asyncVars.startCS;
    asyncVars.sumCS = sum;
});

```

With code like this, which is in the Image Manipulation example with this chapter, you can start the async counting operations (using the Counting Perf (Async) button) and then immediately go open an image and do grayscale conversions at the same time.

Async Methods in C++ To implement an async method in C++, we need to produce the same end result as in C#: a method that returns one of the `IAsync*` interfaces and runs its internal code on another thread.

The first part is straightforward—we just need to declare the method with the C++ types (shown here in the C++ code; the class declaration in `Grayscale.h` is similar):

```

using namespace Windows::Foundation;
IAsyncOperation<double>^ Tests::CountFromZeroAsync(double max, double increment)

```

The C++ analogue of the `AsyncInfo` class is a `task` found in what's called the Parallel Patterns Library for C++, also known as PPL or the Concurrency Runtime, whose namespace is `concurrency` (use a `#include <ppltasks.h>` and `using namespace concurrency;` in your C++ code you're good to go). The function that creates a task is called `create_async`. All we need to do is wrap our code in that function as follows:

```

IAsyncOperation<double>^ Tests::CountFromZeroAsync(double max, double increment)
{
    return create_async([max, increment]()
    {
        double sum = 0;

        for (double x = 0; x < max; x += increment)
        {
            sum += x;
        }

        return sum;
    });
}

```


As with C#, there are additional structures for when you're nesting async operations, supporting cancellation, and reporting progress. I will leave the details to the documentation. See [Asynchronous Programming in C++](#) and [Task Parallelism](#).

Sidebar: Joining Promises

There's one detail from the Image Manipulation example that takes advantage of having all the async operations managed through promises. In the app, we show a horizontal progress indicator before starting all the async operations with the Counting Perf (Async) button:

```
function testPerfAsync() {  
    showProgress("progressAsync", true);  
    //...  
}
```

We want this control to stay visible while any of the async operations are still running, something that's easily done with `WinJS.Promise.join`. What's interesting to point out here is that we can already have called `then` or `done` on those individual promises, which simply means that we've wired up separate handlers for those individual operations. The handlers we give to `join`, then, are just wired up to the fulfillment of all those promises together:

```
promiseJS.done(function (sum) {  
    // Output for JS worker  
})  
  
promiseCS.done(function (sum) {  
    // Output for C# component  
})  
  
promiseCPP.done(function (sum) {  
    // Output for C++ component  
});  
  
WinJS.Promise.join([promiseJS, promiseCS, promiseCPP]).done(function () {  
    // Hide progress indicator when all operations are done  
    showProgress("progressAsync", false);  
});
```

In this code you can see how much we simplify everything by wrapping a worker's message mechanism within a promise! Without doing so, we'd need to maintain one flag to indicate whether the promises were fulfilled (set to `true` inside the `join`) and another flag to indicate if the worker's results had been received (setting that one to `true` inside the worker's message handler). Inside the `join`, we'd need to check if the worker was complete before hiding the progress indicator; the worker's message handler would do the same, making sure the `join` was complete. This kind of thing is manageable on a small scale but would certainly get messy with many parallel async operations—which is the reason promises were created in the first place!

Arrays, Vectors, and Other Alternatives

Now that we've seen the basic structure of asynchronous methods in WinRT components, let's see how we might create an asynchronous variant of the synchronous `Convert` methods we implemented earlier. For the purpose of this exercise we'll just stick with the C# component.

It would be natural with `Convert` to consider `IAsyncAction` as the method's type, because we already return results in an output array. This would, in fact, be a great choice if we were using types *other* than an array. However, arrays present a variety of problems with asynchronous methods. First, although we can pass the method both the input and output arrays, and the method can do its job and populate that output array, its contents won't actually be transferred back across the async task boundary at present. So the completed handler in the app will be called as it would expect, but the output array passed to the async method will still be empty.

The next thing we can try is to turn the async action into an operation that produces a result. We might consider a return type of `IAsyncOperation<Byte[]>` (or an equivalent one using progress), where the method would create and populate the array it returns. The problem here, however, is that the app receiving this array wouldn't know how to release it—clearly some memory was allocated for it, but that allocation happened inside a component and not inside JavaScript, so there's no clear rule on what to do. Because this is a sure fire recipe for memory leaks, returning arrays like this isn't supported.

An alternative is for the async method to return a specific WinRT collection type (where there are clear rules for deallocation), such as an `IList<Byte>`, which will be converted to a vector in JavaScript that can also be accessed as an array. (Note that `IList` is specific to .NET languages; the C++ walkthrough topic shows how to use a vector directly with the `concurrent_vector` type.) Here's a simple example of such a method:

```
public static IAsyncOperation<IList<Byte>> CreateByteListAsync(int size)
{
    var task = Task.Run<IList<Byte>>>(() =>
    {
        Byte [] list = new Byte[size];

        for (int i = 0; i < size; i++)
        {
            list[i] = (Byte)(i % 256);
        }

        return list.ToList();
    });

    return task.AsAsyncOperation();
}
```

Applying this approach to the grayscale routine, we get the following `ConvertPixelArrayAsync` (see `PixelCruncherCS > ConvertGrayscale.cs`), where the `DoGrayscale` is the core code of the routine broken out into a separate function, the third parameter of which is a periodic callback that we can use to handle cancellation):

```

public IAsyncOperation<IList<Byte>> ConvertPixelFormatAsync([ReadOnlyArray()]
    Byte[] imageDataIn)
{
    //Use AsyncInfo to create an IAsyncOperation that supports cancellation
    return AsyncInfo.Run<IList<Byte>>>((token) => Task.Run<IList<Byte>>(() =>
    {
        Byte[] imageDataOut = new Byte[imageDataIn.Length];
        DoGrayscale(imageDataIn, imageDataOut, () =>
            {
                token.ThrowIfCancellationRequested();
            });

        return imageDataOut.ToList();
    }, token));
}

```

A fourth approach is to follow the pattern used by the [Windows.Graphics.Imaging.PixelData-Provider](#) class, which we're already using in the Image Manipulation example. In the function `setGrayscale` (`js/default.js`), we open a file obtained from the file picker and then decode it with `BitmapDecoder.getPixelDataAsync`. The result of this operation is the `PixelDataProvider` that has a method called `detachPixelData` to provide us with the pixel array (some code omitted for brevity):

```

function setGrayscale(componentType) {
    imageFile.openReadAsync().then(function (stream) {
        return Imaging.BitmapDecoder.createAsync(stream);
    }).then(function (decoderArg) {
        //Configure the decoder ... [code omitted]
        return decoder.getPixelDataAsync();
    }).done(function (pixelProvider) {
        copyGrayscaleToCanvas(pixelProvider.detachPixelData(),
            decoder.pixelWidth, decoder.pixelHeight, componentType);
    });
}

```

A similar implementation of our grayscale conversion routine is in `PixelCruncherCS` > `ConvertGrayscale.cs` in the function `ConvertArraysAsync`. Its type is `IAsyncAction` because it operates against the `Grayscale.inputData` array (which must be set first). The output is accessed from `Grayscale.detachOutputData()`. Here's how the JavaScript code looks:

```

pc1.inputData = pixels;
pc1.convertArraysAsync().done(function () {
    var data = pc1.detachOutputData()
    copyArrayToImgData(data, imgData);
    updateOutput(ctx, imgData, start);
});

```

You might be wondering about that `copyArrayToImgData` function in the code above. I'm glad you are, because it points out a problem that really forces us to take a different approach altogether, one that leads us to an overall better solution!

All along in this example we've been loading image data from a file, using the `BitmapDecoder`, and then converting those pixels to grayscale into an array provided by the `canvas` element's `createImageData` method. Once the data is inside that image data object, we can call the `canvas.putImageData` method to render it. All of this was originally implemented to show interaction with the `canvas`, including how to save `canvas` contents to a file. That was fine for Chapter 10, where graphics were our subject. But if we're really looking to just convert an image file to grayscale, using a `canvas` isn't the best road to follow.

The key issue that we're encountering here is that the `canvas`'s `putImageData` method accepts *only* an `ImageData` object created by the `canvas`'s `createImageData` method. The `canvas` does not allow you to create and render a separate pixel array, nor insert a different array in the `ImageData.data` property. The only way it works is to write data directly into the `ImageData.data` array.

In the synchronous versions of our component methods, it was possible to pass `ImageData.data` as the output array so that the component could perform a direct write. Unfortunately, this isn't possible with the async versions. Those methods can provide us with the converted data all right, but because we can't point `ImageData.data` to such an array, we're forced to use a routine like `copyArrayToImageData` function to copy those results into `ImageData.data`, byte by byte. Urk. That pretty much negates any performance improvement we might have realized by creating components in the first place!

Let me be clear that this is a limitation of the `canvas` element, not of WinRT or components in general. Moving arrays around between apps and components, as we've seen, works perfectly well for other scenarios. Still, the limitation forces us to ask whether we're even taking the right approach at all.

Taking a step back, the whole purpose of the demonstration is to convert an image file to grayscale and show that conversion on the screen. Using a `canvas` is just an implementation detail—we can achieve the same output in other ways. For example, instead of converting the pixels into a memory array, we could create a temporary file using the `Windows.Graphics.Image.BitmapEncoder` class instead, just like we use in the `SaveGrayscale` function that's already in the app. We'd just give it the converted pixel array instead of grabbing those pixels from the `canvas` again. Then we can use `URL.createObjectURL` or an `ms-appdata:///` URI to display it in an `img` element. This would likely perform much faster because the `canvas`'s `putImageData` method actually takes a long time to run, much longer than the conversion routines in our components.

Along these same lines, there's no reason that we couldn't place more of the whole process inside a component. Only those parts that deal with UI need to be in JavaScript, but the rest can be written in another language. For example, why bother shuttling pixel arrays between JavaScript and a WinRT component? Once we get a source `StorageFile` from the file picker we can pass that to a component method directly. It could then use the `BitmapDecoder` to obtain the pixel stream, convert it, and then create the temporary file and write the converted pixels back out using the `BitmapEncoder`, handing back a `StorageFile` to the temp file from which we can set an `img src`. The pixels, then, never leave the component and never have to be copied between memory buffers. This should result in both higher performance as well as a smaller memory footprint.

To this end the PixelCruncherCS project in the Image Manipulation example has another async

method called `ConvertGrayscaleFileAsync` that does exactly what I'm talking of here:

```
public static IAsyncOperation<StorageFile> ConvertGrayscaleFileAsync(StorageFile file)
{
    return AsyncInfo.Run<StorageFile>((token) => Task.Run<StorageFile>(async () =>
    {
        StorageFile fileOut = null;

        try
        {
            //Open the file and read in the pixels
            using (IRandomAccessStream stream = await file.OpenReadAsync())
            {
                BitmapDecoder decoder = await BitmapDecoder.CreateAsync(stream);
                PixelDataProvider pp = await decoder.GetPixelDataAsync();
                Byte[] pixels = pp.DetachPixelData();

                //We know that our own method can convert in-place,
                //so we don't need to make a copy
                DoGrayscale(pixels, pixels);

                //Save to a temp file.
                ApplicationData appdata = ApplicationData.Current;

                fileOut = await appdata.TemporaryFolder.CreateFileAsync(
                    "ImageManipulation_GrayscaleConversion.png",
                    CreationCollisionOption.ReplaceExisting);

                using (IRandomAccessStream streamOut =
                    await fileOut.OpenAsync(FileAccessMode.ReadWrite))
                {
                    BitmapEncoder encoder = await BitmapEncoder.CreateAsync(
                        BitmapEncoder.PngEncoderId, streamOut);

                    encoder.SetPixelData(decoder.BitmapPixelFormat, decoder.BitmapAlphaMode,
                        decoder.PixelWidth, decoder.PixelHeight,
                        decoder.DpiX, decoder.DpiY, pixels);

                    await encoder.FlushAsync();
                }
            }
        }
        catch
        {
            //Error along the way; clear fileOut
            fileOut = null;
        }

        //Finally, return the StorageFile we created, which makes it convenient for the
        //caller to copy it elsewhere, use in a capacity like URL.createObjectURL, or refer
        //to it with "ms-appdata:///temp" + fileOut.Name
        return fileOut;
    }));
}
```

One thing we can see comparing with this the equivalent JavaScript code is that the C# `await` keyword very much simplifies dealing with asynchronous methods—making them appear like they’re synchronous. This is one potential advantage to writing code in a component! The other important detail is to note the `using` statements around the streams. Streams, among other types, are *disposable* (they have an `IDisposable` interface) and must be cleaned up after use or else files will remain open and you’ll see access denied exceptions or other strange behaviors. The `using` statement encapsulates that cleanup logic for you.

In any case, with this method now we need only a few lines of JavaScript to do the job:

```
PixelCruncherCS.Grayscale.convertGrayscaleFileAsync(imageFile).done(function (tempFile) {  
    if (tempFile != null) {  
        document.getElementById("image2").src = "ms-appdata:///temp/" + tempFile.name;  
    }  
});
```

The line with the URI could be replaced with these as well:

```
var uri = URL.createObjectURL(tempFile, { oneTimeOnly: true });  
document.getElementById("image2").src = uri;
```

Running tests with this form of image conversion, the app shows a much better response, so much so that the progress ring that’s normally shown while the operation is running doesn’t even appear!

All this illustrates the final point of this whole exercise. If you’re looking for optimizations, think beyond just the most computationally intensive operations, especially if it involves moving lots of data around. As we’ve seen here, challenging our first assumptions can lead to a much better overall solution.

Projections into JavaScript

Already in this chapter we’ve seen some of the specific ways that a WinRT component is projected into JavaScript. In this section I wanted to offer a fuller summary of how this world of WinRT looks from JavaScript’s point of view.

Let’s start with naming. We’ve seen that a JavaScript app project must add a component as a reference, at which point the component’s namespace becomes inherently available in JavaScript; no other declarations are required. The namespace and the classes in the component just come straight through into JavaScript. What does change, however, are the names of methods, properties, and events. Although namespaces and class names are projected as they exist in the component, method and property names (including members of `struct` and `enum`) are converted to camel casing: *TestMethod* and *TestProperty* in the component become *testMethod* and *testProperty* in JavaScript. This casing change can have some occasional odd side effects, as when the component’s name starts with two capital letters such as *UIProperty*, which will come through as *uiProperty*.

Event names, on the other hand, are converted to all lowercase as befits the JavaScript convention. An event named *SignificantValueChanged* in the component becomes *significantvaluechanged* in JavaScript. You'd use that lowercase name with `addEventListener`, and the class that provides it will also be given a property of that name prefixed with *on*, as in *onsignificantvaluechanged*. An important point with events is that it's sometimes necessary to explicitly call `removeEventListener` to prevent memory leaks. For a discussion, refer back to Chapter 3, "App Anatomy and Page Navigation," in the section "WinRT Events and removeEventListener." In the context of this chapter, WinRT events include those that come from your own WinRT components.

Static members of a class, as we've seen, can just be referred to directly using the fully qualified name of that method or property using the component's namespace. Nonstatic members, on the other hand, are accessible only through an instance created with `new`.

Next are two limitations that we've mentioned before but are worth repeating in this context. First is that a WinRT component cannot work with the UI of an app written in JavaScript. This is because the app cannot obtain a drawing surface of any kind that the component could use. Second is that JavaScript can resolve only overloaded methods by arity (number of parameters) and not by type. If a component provides overloads distinguished only by type, JavaScript can access only whichever one of those is marked as the default.

Next we come to the question of data types, which is always an interesting subject where interoperability between languages is concerned. Generally speaking, what you see in JavaScript is naturally aligned with what's in the component. A WinRT `DateTime` becomes a JavaScript `Date`, numerical values become a `Number`, `bool` becomes `Boolean`, strings are strings, and so on. Some WinRT types, like `IMapView` and `IPropertySet`, just come straight through to JavaScript as an object type because there are no intrinsic equivalents. Then there are other conversions that are, well, more interesting:

- Asynchronous operations in a component that return interfaces like `IAsyncOperation` are projected into JavaScript as promises.
- Because JavaScript doesn't have a concept of `struct` as does C#, VB, and C++, structs from a WinRT component appear in JavaScript as objects with the struct's fields as members. Similarly, to call a WinRT component that takes a `struct` argument, a Javascript app constructs an object with the fields as members and passes that instead. Note that the casing of `struct` members is converted to camel casing in JavaScript.
- Some collection types, like `IVector`, appear in JavaScript as an array but with different methods. That is, the collection can be accessed using the array operator `[]`, but its methods are different. Be careful, then, passing these to JavaScript manipulation functions that assume those methods exist.
- Enums are translated into objects with camel-cased properties corresponding to each enum value, where those values are JavaScript `Number` types.

- WinRT APIs sometimes return `Int64` types (alone or in `structs`) for which there is no equivalent in JavaScript. The 64-bit type is preserved in JavaScript, however, so you can pass it back to WinRT in other calls. However, if you modify the variable holding that value, even with something as simple as a `++` operator, it will be converted into a JavaScript `Number`. Such a value will not be accepted by methods expecting an `Int64`.
- If a component method provides multiple output parameters, these show up in JavaScript as an object with those different values. There is no clear standard for this in JavaScript; it's best to avoid in component design altogether.

The bottom line is that the projection layer tries to make WinRT components written in any other language look and feel like they belong in JavaScript, without introducing too much overhead.

Scenarios for WinRT Components

Earlier in the “Choosing a Mixed Language Approach” section I briefly outlined a number of scenarios where WinRT components might be very helpful in the implementation of your app. In this section we'll think about these scenarios a little more deeply, and I'll point you to demonstrations of these scenarios in the samples, where such are available.

Higher Performance

Increasing the performance of a Windows Store app written in HTML, CSS, and JavaScript is one of the primary scenarios for offloading some work to a WinRT component.

When evaluating the performance of your app, keep an eye open for specific areas that are computationally intensive or involve moving a lot of data around. For example, if you found it necessary to implement an extended splash screen for those exact reasons, perhaps you can reduce the time the user has to wait (especially on first launch) before the app is active. Any other situation where the user might have to wait—while they might have anything better to do than watch a progress indicator!—is a great place to use a high performance component if possible. Clearly there are scenarios where the performance of the app isn't so much the issue as is network latency, but once you get data back from a service you might be able to pre-process it faster in a component than in JavaScript.

Another great place to insert a high-performance component is in an app's startup sequence, especially if it has extra work to do on first run. For example, an app package might include large amounts of compressed data to minimize the size of its download from the Store, but it needs to decompress that data on first run. A WinRT component might significantly shorten that initialization time. If the component uses WinRT APIs to write to your app data folders, all that data will also be accessible from JavaScript through those same APIs.

One challenge, as we saw in the quickstarts, is that writing a component to chew on a bunch of data typically means you want to pass JavaScript arrays into that component and get an array back out. As we saw in the quickstarts, this works just fine with synchronous operations but is not presently supported for async, which is how you'd often want to implement potentially long-running methods. Fortunately, there are ways around this limitation, either by transferring results for an async operation through synchronous properties or by using other collection types such as vectors.

One place where performance is very significant is with background tasks. As explained in Chapter 13, background tasks are limited to a few seconds of CPU time every 15 minutes. Because of this, you can get much more accomplished in a background task written in a higher-performance language than one written in JavaScript.

The structure of a component with such tasks is no different than any other, as demonstrated in the C# Tasks component included with the [Background task sample](#). Each of the classes in the Tasks namespace is marked as `public` and `sealed`, and because the component is brought into the JavaScript project as a reference, those class names (and their public methods and properties) are in the JavaScript namespace. As a result, their names can be given to the `BackgroundTaskBuild.taskEntryPoint` property without any problems.

Another example of the same technique can be found in the [Network status background sample](#).

Something that we didn't discuss in Chapter 13, but which is appropriate now, is that when you create a WinRT component for this purpose, the class that implements the background task must derive from `Windows.ApplicationModel.Background.IBackgroundTask` and implement its singular `Run` method. That method is what gets called when the background task is triggered. We can see this in the Network status background sample where the whole C# implementation of the component comprises just a few dozen lines of code (NetworkStatusTask project > BackgroundTask.cs; some comments and debug output omitted):

```
namespace NetworkStatusTask
{
    public sealed class NetworkStatusBackgroundTask : IBackgroundTask
    {
        ApplicationDataContainer localSettings = ApplicationData.Current.LocalSettings;

        // The Run method is the entry point of a background task.
        public void Run(IBackgroundTaskInstance taskInstance)
        {
            // Associate a cancellation handler with the background task.
            taskInstance.Canceled += new BackgroundTaskCanceledEventHandler(OnCanceled);

            try
            {
                ConnectionProfile profile =
                    NetworkInformation.GetInternetConnectionProfile();
                if (profile == null)
                {
                    localSettings.Values["InternetProfile"] = "Not connected to Internet";
                    localSettings.Values["NetworkAdapterId"] = "Not connected to Internet";
                }
            }
        }
    }
}
```

```

    }
    else
    {
        localSettings.Values["InternetProfile"] = profile.ProfileName;

        var networkAdapterInfo = profile.NetworkAdapter;
        if (networkAdapterInfo == null)
        {
            localSettings.Values["NetworkAdapterId"] =
                "Not connected to Internet";
        }
        else
        {
            localSettings.Values["NetworkAdapterId"] =
                networkAdapterInfo.NetworkAdapterId.ToString();
        }
    }
}
catch (Exception e)
{
    // Debug output omitted
}

// Handles background task cancellation.
private void OnCanceled(IBackgroundTaskInstance sender,
    BackgroundTaskCancellationReason reason)
{
    // Debug output omitted
}
}
}

```

You can see that the Run method receives an argument through which it can register a handler for canceling the task. The code above doesn't do anything meaningful with this because the task itself executes only a small amount of code. The C# tasks in the Background tasks sample, on the other hand, simulate longer-running operations, in which case it uses the handler to set a flag that will stop those operations.

Access to Additional APIs

Between the DOM API, WinJS, third-party libraries, and JavaScript intrinsics, JavaScript developers have no shortage of APIs to utilize in their apps. At the same time, there is a whole host of [.NET](#) and [Win32/COM](#) APIs that are available to C#, VB, and C++ apps that are not directly available to JavaScript, including the DirectX and Media Foundation APIs.

With the exception of APIs that affect UI or drawing surfaces (namely Direct2D and Direct3D), WinRT components can make such functions—or, more likely, higher-level operations built with them—available to apps written in JavaScript.

The [Building your own Windows Runtime components to deliver great apps](#) post on the Windows 8 developer blog gives some examples of this. It shows how to use the [System.IO.Compression](#) API in .NET to work with ZIP files and the XAudio APIs (part of DirectX) to bypass the HTML [audio](#) element and perform native audio playback. In the latter case, you might remember from Chapter 10, in the section “Playing Sequential Audio,” that no matter how hard we tried to smooth the transition between tracks with the [audio](#) element, there is always some discernible gap. This is due to the time it takes for the element to translate all of its operations into the native XAudio APIs. By going directly to those same APIs, you can circumvent the problem entirely. (That said, Microsoft knows about the behavior of the [audio](#) element and will likely improve its performance in the future.)

This approach can also be used to communicate with external hardware that’s not represented in the WinRT APIs but is represented in Win32/COM. We saw this with the [XInput and JavaScript controller sketch sample](#) in Chapter 15, “Devices and Printing” (also discussed in the blog post above).

Another very simple example would be creating a component to answer an oft-heard question: “How do I create a GUID in JavaScript?” Although you can implement a routine to construct a GUID string from random numbers, it’s not a proper GUID in that there is no guarantee of uniqueness (GUID stands for Globally Unique Identifier, after all). To do the job right, you’d want to use the Win32 API [CoGreatGuid](#), for which you can create a very simple C++ wrapper.

Overkill? Some developers have commented that going to all the trouble to create a WinRT component just to call one method like [CoCreateGuid](#) sounds like a heavyweight solution. However, considering the simplicity of a basic WinRT component as we’ve seen in this chapter, all you’re really doing with a component is setting up a multilanguage structure through which you can use the full capabilities of each language. The overhead is really quite small: a Release build of the C++ component in “Quickstart #2” produces a 39K DLL and a 3K .winmd file, for example.

Using a WinRT component in this way applies equally to COM DLLs that contain third-party APIs like code libraries. You can use these in a Windows Store app provided they meet three requirements:

- The DLL is packaged with the app.
- The DLL uses only those [Win32/COM APIs](#) that are allowed for Windows Store apps. Otherwise the app will not pass Store certification.
- The DLL must implement what is called *Regfree COM*, meaning that it doesn’t require any registry entries for its operation. (Windows Store apps do not have access to the registry and thus cannot register a COM library.) The best reference I’ve found for this is the article [Simplify App Deployment with ClickOnce and Registration-Free COM](#) in MSDN Magazine.

If all of these requirements are met, the app can then use the [CoCreateInstanceFromApp](#) function from a component to instantiate objects from that DLL.

Obfuscating Code and Protecting Intellectual Property

Back in Chapter 1, in the section “Playing in Your Own Room: The App Container,” we saw how apps written in HTML, CSS, and JavaScript exist on a consumer’s device as source files, which the app host loads and executes. By now you’ve probably realized that for as much as Windows tries to hide app packages from casual access, all that code is there on their device where a determined user can gain access to it. In other words, assume your code is just as visible in an app package as it is on the web through a browser’s View Source command.

It’s certainly possible—and common, in fact—to draw on web services that aren’t so exposed for much of an app’s functionality, in the same way that web apps take advantage of server-side processing. Still, there will be parts of an app that must exist and run on the client, so you are always running the risk of someone taking advantage of your generosity!

Ever since developers started playing with Windows 8 apps written in HTML, CSS, and JavaScript, they’ve been asking about how to protect their code. In fact, developers using C# and Visual Basic ask similar questions because although those languages are compiled to IL (intermediate language), plenty of decompilers exist to produce source code from that IL just as other tools circumvent JavaScript minification. Neither JavaScript nor .NET languages are particularly good at hiding their details.

Code written in C++, being compiled down to machine code, is significantly harder to reverse-engineer, although it’s still not impossible for someone to undertake such a task (in which case you have to ask why they aren’t just writing their own code to begin with!). Nevertheless, it’s the best protection you can provide for code that lives on the client machine. The only real way to protect an algorithm is to keep it on a remote server.

If the rest of the app is written in a language other than C++, especially JavaScript, know that it’s a straightforward manner to also reverse-engineer the interface to a component. The issue here, then, is whether a malicious party could use the knowledge of a component’s interface to employ that component in their own apps. The short answer is yes, because your code might show them exactly how. In such cases, a more flexible and nearly watertight solution would be for the component vendor to individually manage licenses to app developers. The component would have some kind of initialization call to enable the rest of its functionality. Within that call, it would compare details of the app package obtained through the `Windows.ApplicationModel.PackageId` class against a known and secure registry of its own, knowing that the uniqueness of app identity is enforced by the Windows Store. Here are some options for validation:

- **Check with an online service** This would require network connectivity, which might not be a problem in various scenarios. Just remember to encrypt the data you send over the network to avoid snooping with a tool like Fiddler!
- **Check against information encrypted and compiled into the component itself** That is, the component is compiled for each licensee uniquely. This is the most difficult to hack.

- **Check against an encrypted license file distributed with the component that is unique to the licensee** (contains the app name and publisher, for instance) Probably the easiest solution, because even if the license file is copied out of the licensed app's package, the info contained would not match another app's package info at run time. The encryption algorithm would be contained within the compiled component, so it would be very difficult to reverse-engineer in order to hack the license file—not impossible, but very difficult. Another app could use that component only if it used the same package information, which couldn't be uploaded to the Store but could still possibly be side-loaded by developers or an unscrupulous enterprise.

In the end, though, realize that Windows itself cannot guarantee the security of app code on a client device. Further protections must be implemented by the app itself.

Library Components

A library component is a piece of code that's written to be used by any number of other apps written in the language of their choice. You might be looking to share such a library with other developers (perhaps as a commercial product), or you might just be looking to modularize your own apps.

A great example of this is the Notifications Extensions Library that we saw in various samples of Chapter 13: [App tiles and badges sample](#), [Lock screen apps sample](#), [Scheduled notifications sample](#), [Secondary tiles sample](#), and [Toast notifications sample](#). This library contains a number of classes with their respective properties and methods, and because all those methods are small and fast, they're all designed to be synchronous.

In the case of the samples, the Notifications Extensions Library is included as source code in every project that uses it. More likely though, especially if you create a commercial project and follow the [How to: Create a software development kit](#) documentation, you'll be providing only the compiled DLL and/or WinMD file to your customers. Customers will add these libraries to their projects, so they're packaged directly with the app.

In these cases, be sure you provide separate components that are compiled for x86, x64, and ARM targets for components written in C++.

Concurrency

We've already seen that web workers and WinRT components with asynchronous methods can work hand in hand to delegate tasks to different threads. If you really want to exercise such an option, you can architect your entire app around concurrent execution by using these mechanisms, spinning up multiple async operations at once, possibly across multiple web workers. A WinRT component can also use multiple `Task.Run` calls, not just the single ones we've seen.

Deeper still, a WinRT component can utilize the APIs in [Windows.System.Threading](#) to get at the *thread pool*, along with those APIs in [Windows.System.Threading.Core](#) that work with semaphores and other threading events. The details of these are well beyond the scope of this book, but I wanted to mention them because many of the built-in WinRT APIs make use of these and your components can

do the same. And although it doesn't demonstrate components, necessarily, the [Thread pool sample](#) provides a good place to start on this topic.

What We've Just Learned

- Windows Store apps need not be written in just a single language; with WinRT components, apps can effectively use the best language for any given problem. Components, however, cannot work with UI on behalf of an app written in HTML, CSS, and JavaScript.
- Reasons for using a mixed language approach include improving performance, gaining access to additional APIs (including third-party libraries) that aren't normally available to JavaScript, obfuscating code to protect intellectual property, creating modular library components that can be used by apps written in any other language, and effectively managing concurrency.
- For computationally intensive routines, a component written in C#/VB can realize on the order of a 15% improvement over JavaScript and a component written in C++ an improvement on the order of 25%. When testing performance, be sure to build a Release version of the app and run outside of the debugger, otherwise you'll see very different results for the different languages.
- Windows Store apps can employ web workers for creating asynchronous routines that run separately from the UI thread, and you can wrap that worker within a WinJS promise to treat it like other async method in WinRT.
- Async methods can also be implemented in WinRT components by using task, concurrency, and thread pool APIs. Compared to web workers, such async methods are more responsive because they are directly structured as methods.
- No matter what language a component is written in, the JavaScript projection layer translates some of its structures into forms that are natural to JavaScript, including the casing of names and conversion of data types.
- Due to the nature of event handlers that cross the JavaScript/WinRT component boundary, apps must be careful to prevent memory leaks by calling `removeEventListener` for events originating from the WinRT API or WinRT components when a listener is only temporary.

Chapter 17

Apps for Everyone: Accessibility, World-Readiness, and the Windows Store

The title of this chapter, “Apps for Everyone”—especially the “Everyone” part—has several shades of meaning. First is the vitally central role that the Windows Store plays in the whole Windows 8 experience. As first mentioned in Chapter 1, “The Life Story of a Windows Store App,” the Store is *the* place where you distribute apps to customers (outside of the enterprise and sharing with other developers). Everyone, in other words, gets their apps from the Store.

In this same way, everyone who does business with apps does business with the Store. To define your app’s relationship to the Store is in many ways to define your business itself, and that relationship affects all stages of the app lifecycle from planning and development to distribution and servicing. Thinking about the Store, then, is not something you want to do only when you’ve completed an app: you want to be thinking about it when you start thinking about the apps you’d like to build. You might have come here directly from Chapter 1, in fact, where I recommended reading the first section below even before starting your first coding experiments! Truly, the Windows Store is like a pair of bookends to the whole app development process: you think about the Store when planning the business of your app, and when all is said and done, you go to the Store’s developer portal itself to make your app available to others.

Those “others” are the context for the additional meanings of “everyone.” In general, when you set out to offer a product to customers, you want to broaden your reach to include as many potential customers as you can. There are, of course, cases where you might want to specifically limit your audience, but for most apps, being able to reach more customers is certainly an attractive opportunity. And if you don’t, your competitors will!

One way to broaden your reach is to cover your bases where *accessibility* is concerned. Though accessibility has its origins in serving people with serious disabilities, research has shown that a majority of people—nearly 60%—use accessibility features in some capacity, even though there’s no disability involved. For one, being able to accommodate limited input models—like keyboard only or mouse/pointer only—is inherent in dealing with touch-only devices. Resolution-scaling, similarly, serves the needs of the visually impaired alongside the desires of the financially *unimpaired* (that is, those customers who are willing to splurge for a high-DPI device just to get sharper graphics). An app that works well with a screen reader for the visually impaired can also work rather well for the mobile customer whose otherwise sound eyes need to be focused elsewhere—like the road they’re ostensibly driving on! And providing for high-contrast color schemes helps not only those whose eyes don’t do

well with subtle colorations but also those who might be working with a mobile device in bright sunlight.

It therefore behooves app developers to take accessibility concerns seriously, especially as the Store will specifically mark fully accessible apps. As we will see, this primarily involves adding the appropriate *aria-** attributes to your HTML markup, adapting your layout to different screen sizes, and making sure to provide image variations for different contrast settings.

The second way to extend your reach is to make your app world-ready—that is, to utilize localized resources within the app so that it adapts itself to each user’s language, regional conventions, date and time formats, currency formats, and so on. Fortunately, Windows 8 enables you to structure your resources—images and strings, primarily—so that the right variations show up automatically, just like they do for resolution scales and contrasts. The Windows Runtime also contains a number of APIs to help an app be world-ready, and the app itself can take additional steps to localize the web services from which it’s drawing data, how it works with live tiles and notifications, and so on. Furthermore, some additional tools such as the Multilingual App Toolkit are available to make it all the easier to translate your resources.

The reward for all this effort, of course, is that users who search in the Windows Store for apps in their regional language will see *your* app and not those that are available only in a single language like English. Those users will also be more likely to express their appreciation in your app’s reviews and ratings.

One of the great things about the Windows Store is the access it gives you to global markets from wherever you happen to be working. In the past, learning to do business in many countries around the world has been a tedious and expensive process, sometimes requiring that you understand local tax laws, manage currency conversions, and so forth. No longer—this is really what the Windows Store is doing on your behalf. Once the Store becomes available in a market, it means that Microsoft has done the work to embed local policies into the Store itself. Put another way, whatever small fee you pay to upload apps to the Store has made it possible for you to business in those markets with little or no effort! This is good. The Store also lets you vary your app pricing for regional markets—if you charge for the app or in-app purchases—because standards of living do vary widely around the world. This is also good.

In this chapter then, we’ll begin by looking at the relationship between your app and the business of your app as supported by the Store, regardless of whether you seek to monetize your app in any way. We’ll then take a tour of accessibility followed by another through the world of world-readiness. The last section—of this chapter and of this book!—will bring us full circle to where we started in Chapter 1: uploading your app to the Store and what you can expect there. Now that you’ve brought your app this far, let’s get it ready for everyone to enjoy!

Your App, Your Business

If you check in with your local psychologist or philosopher, they'd probably agree with the idea that just about all people, across all professions, cultures, and capabilities, are driven by a small number of fundamental motivations: fear, lust, power, love, service to others, and just plain ol' joy. Indeed, the wisest among them will even say that the last one—the quest for joy or happiness—is actually the root of all the others.

Leaving all that aside, and assuming that you're not programming under threat of death or working on apps that are going to be rejected from the Windows Store as a matter of policy, let's take a simpler view and identify the few basic reasons why you might be interested in writing apps:

- **Fortune** You want to make money.
- **Fame** You want social recognition.
- **Philanthropy** You want to contribute to a cause.
- **Fun** You just want to enjoy yourself through coding—an activity that, alas, nonprogrammers just don't understand!

Wherever you land in this list—and with whatever combination—your motivations essentially define your “business” as a developer. I use the term loosely here. In English, at least, there are about a dozen different definitions of this word, only a third of which relate to commercial activities, organizations, practices, and commerce. The other definitions have to do with concerns that are important to you, as when we say “It's none of your business” or “I make it my business to know about such things.” In short, apps can reflect the nature of your “business,” whatever it is, and that nature is reflected in how you share apps with others. Again, with the exception of side loading (see the next section), sharing your app means distributing it through the Windows Store. For that reason, your app's relationship to the Store effectively defines your business with that app, and that relationship spans the entire app lifecycle:

- **Planning** Determining whether the app can actually be a Store app, meet Store certification requirements, and be suitably monetized (if desired).
- **Development** Implementing Store-related features and using the APIs for trial versions, in-app purchases, etc.
- **Testing** Using precertification tools prior to onboarding the app to the Store, and checking the app against certification requirements.
- **Availability** Making the app available in various markets through the Store developer portal.

- **Marketing, sales, and support** Promoting your app, increasing its visibility, working with your customers (responding to ratings and reviews), linking it to your website (if applicable), and using Store analytics through the developer portal.
- **Updates and growth** Improving your app over time, or removing it from the market.

We'll explore some of these topics in more detail shortly, especially those areas that affect planning and development. This includes the APIs that allow you to simulate Store interactions when debugging. We'll also review the available monetization models—from completely free apps, ad-supported apps, and paid apps, to trial versions and those with in-app purchases. After all, your choices here are fundamental to how you're going to fulfill your goals in writing applications, whatever your motivations.

The remaining topics we'll return to in the last section of this chapter, as it's appropriate to first discuss accessibility and localization.

Side Loading

A question that's arisen with just about every mobile platform is the ability for developers to load and run apps without going through the associated store. Because store-only distribution is generally inconvenient for developers who want to experiment with a platform—especially with apps that don't have the fit-and-finish needed for store certification—they end up hacking that platform to allow for some kind of side loading anyway.

Such is not an issue with Windows. First, as explained on [Get a developer license](#), "A developer license for Windows 8 lets you install, develop, test, and evaluate Windows Store apps before the Windows Store tests and certifies them." In addition to enabling some technical capabilities (like using local loopback, as we've seen in earlier chapters), a developer license means you can share app packages (.appx files) with other developers for whatever purposes you need, such as testing.

Second, enterprise administrators can allow side loading on machines running the Enterprise edition of Windows 8. This is typically useful for distributing line-of-business (LOB) apps within that organization, as such apps clearly have no need to be uploaded to the public Store. Details and requirements can be found on [How to Add and Remove Apps](#) (Microsoft TechNet) as well as [Deploying Windows Store apps to business](#) (Windows Store blog).

In both cases, the side loading process is the same. You first create the app package through the Store > Create App Packages command in Visual Studio and selecting the No radio button, as shown in Figure 17-1. You then provide some details for the package, as in Figure 17-2, and press the Create button.

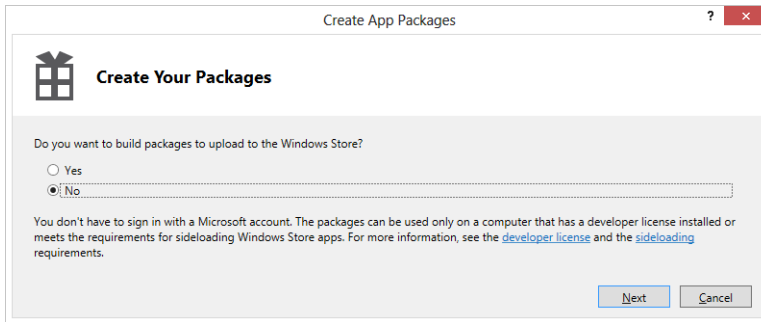


FIGURE 17-1 Creating an app package for side loading. The actual dialog box is much taller; I've compressed it to save space.

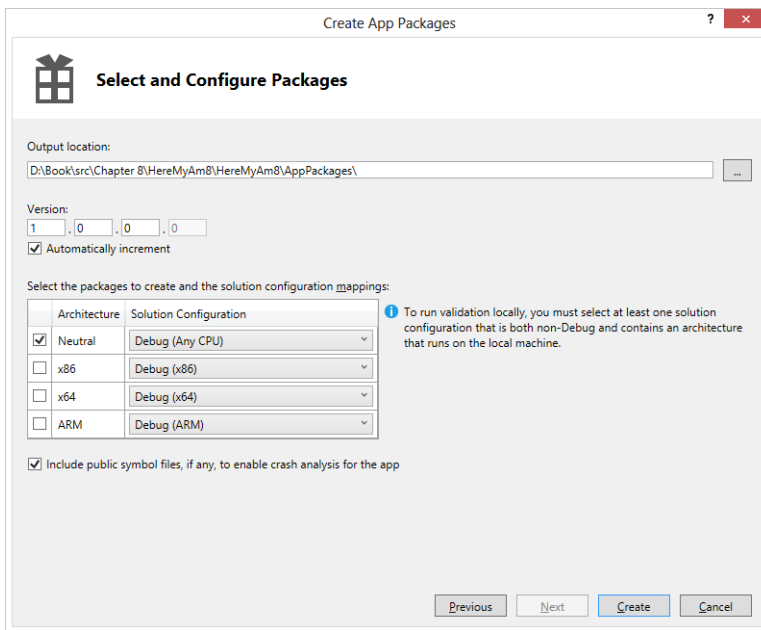


FIGURE 17-2 The second step of creating an app package for side loading.

When the process is complete, Visual Studio will give you a link to the folder where the package was created. Going there you'll see a file with the extension *.appxupload*, which is what you'd be using for the Store. For side loading, you want to look at the folder whose name ends with *_Test*. In that folder you'll see the following:

- The app package (.appx file).
- A temporary certificate (.cer file).
- A Dependencies folder that contains any libraries that would normally be provided by the Store; WinJS is one such library.

- Most importantly, a PowerShell script named *Add-AppDevPackage.ps1* (and a folder with associated resources) that will install the app on a side load-capable machine.

This process makes it easy to share your app with others without sharing your source code project. A good case for using this is if you hire others to thoroughly test your app. Running the PowerShell script installs the app very much like it would be from the Store, so if testers side-load on a machine where your app has not been installed before, it closely approximates a typical user's environment. This way you can truly test the first-run experience of your app on a variety of devices.

Planning: Can the App Be a Windows Store App?

In a slight contradiction to this chapter's title, the idea of "apps for everyone" doesn't necessarily mean that every app can, in fact, be a Windows Store app. There are two sides to this: technical feasibility and meeting Store certification requirements.

Technically, as we covered in Chapter 3, "App Anatomy and Page Navigation" and a few other places, Windows Store apps run under certain conditions and restrictions. Here's a summary:

- Windows Store apps always run in the app container and have no access to APIs that can openly access the file system or any other sensitive resource.
- Sharing data between Windows Store apps always goes through the Share contract, the clipboard, or web services; local interprocess communication is not supported. (Local loopback is supported only on machines with developer licenses and will cause the app to be rejected from Store certification.)
- Windows Store apps can use only the WinRT APIs along with a subset of Win32 and .NET APIs; apps written in HTML and JavaScript can also use the intrinsic HTML and JavaScript APIs provided by the app host. Any third-party libraries you use in the app must also use only these APIs.
- Apps cannot install custom device drivers or anything else that affects the system, nor can apps customize their install process.
- Only certain apps can run in the background, and for specific purposes, as we've seen in Chapter 10, "Media," and Chapter 13, "Live Tiles, Notifications, the Lock Screen, and Background Tasks."
- Some UI interaction models aren't appropriate for touch input, such as high precision CAD. Because Windows Store apps must support all forms of input, high precision apps either need to be redesigned for touch or should be implemented as a desktop app.
- Windows Store apps run in one of four view states and cannot utilize overlapping windows.

If any of these technical aspects would prevent you from writing the kind of app you want to write, then working with the Windows Store as your business location, if you will, is not really possible. For example, many development tools, network administration tools, file system utilities, antimalware utilities (that scan the whole hard drive), and database management systems must be implemented as

desktop applications and distributed through the Internet or other retail channels.⁷⁸

More generally, because Windows Store apps run full screen or with at most one other visible app, they are intended to be much more specifically focused on certain tasks. Apps that try to do too much—Swiss Army Knife apps, if you will—may end up feeling cumbersome or confusing. It's good to hone the purpose of the app, as discussed in [Planning Windows Store apps](#); otherwise you should probably implement a desktop app instead (for which there is still a very large market, mind you!).

When planning any app, be sure to review the [Windows 8 app certification requirements](#) to understand whether the app you're thinking about will be summarily rejected during the onboarding process. Examples include apps that contain gratuitous violence, hate, adult content, solicitations, and so forth, as well as apps that consume an inordinate amount of battery power, attempt to just repackage a website as an app, or clearly don't add value to the Store as a whole. So, if you're thinking to submit the next great bodily-functions-sound-effects app, you might think again.

Also be aware that the Store policies can change over time, so be sure to check them anew before you start any new project.

A final consideration is whether the Windows Store is itself available in a target market when you plan to release your app—the Store will be rolled out to different markets over time. Unfortunately, there is not a published schedule for that rollout; you'll need to watch for announcements. There might also be restrictions on whether you can submit an app to certain locales based on where you operate as a developer. This information is again best found on the [Store developer portal](#).

Planning for Monetization (or Not)

Just as there are a number of reasons why you're interested in creating apps in the first place, there are also a number of ways to fulfill your business goals. Will your app be completely free? Will it be free but supported by ads? Will it be paid, with or without a trial version? Will it involve in-app purchases? Each of these business models has its place, especially if you plan on releasing multiple apps. Furthermore, it's likely that your business model or models will change over time as you improve your apps and respond to competition.

In this section, we'll explore at these different models and better understand how they relate to one another.⁷⁹ Remember in this whole context that licenses for apps and in-app purchases are granted to the user and will apply across up to five devices. This isn't typically a concern for apps because the details are automatically handled by the Store—if the user attempts to install the app on a sixth device, Windows will instruct him or her to remove the app from a machine.

⁷⁸ With developer tools, it's feasible that a Windows Store app could itself provide an interpreted runtime environment for developing apps that would always run inside that tool. A Windows Store app cannot, however, directly produce another Windows Store app because the necessary packaging and deployment APIs are only available to desktop apps.

⁷⁹ Another general overview that includes details on pricing can be found on [Making money with your apps through the Windows Store](#) on the Windows 8 Developer Blog.

Free Apps

You don't need to do anything special to create a free app so far as the Store is concerned. You write it, upload it to the Store, have it certified, and then get the word out.

Free apps can serve several purposes:

- Earn you praise and glory from users and possibly other developers.
- Give you experience producing apps (otherwise known as resumé items!).
- Provide a space for marketing your own products and/or services (as opposed to hosting third-party ads, as discussed in the next section).

The first purpose here is self-explanatory and doesn't need any elaboration, I hope! If this is your motivation, I imagine you're already doing daily or hourly web searches on your name and will be watching your app's ratings and reviews like a floor trader watches stock tickers.

As for gaining experience, that's a great exercise, of course, but be aware that every app you make available through the Store—along with its ratings and reviews—becomes a permanent part of your developer reputation. Because of this, uploading apps to the Store before they're ready—or before you're really ready as a developer—could backfire over the long term. You don't want your reputation to be weighed down by early experiments when you finally have the idea that's really going to take off!

To manage this risk, you could start by only sharing apps with other developers who can side-load whatever packages you make available through some means other than the Store. You might also consider creating a personal developer account just for your experimental work, keeping it separate from the account through which you'd want to post your real apps. This way, any negative reputation from your experiments doesn't accrue to your serious work; neither does positive reputation, of course, but that's a balance you have to find for yourself. Also, creating an extra account will require an additional annual fee, but that might be well worth it in the end.

As for marketing, what I mean here again differs greatly from ad-supported apps (see the next section). Here I'm specifically referring to promoting your own business or causes (such as a charity) through the functioning of the app where you have complete control over the content.

Be aware, however, that the Store certification requirements are somewhat strict where this sort of thing is concerned. Section 2.1 states, "Your app must not display only ads." Section 2.3 says, "Your app must not use its tiles, notifications, app bar, or swipe-from-edge interactions to display ads." That said, it's recognized that the very purpose of some apps is to provide offers, for example, in which case it's not really displaying ads, per se, but content for which the user has expressed interest through the act of installing the app. The policies are targeted more toward apps that pick up ads from an ad provider, such that the user would get random content showing up on their tiles and other key areas of the user interface.

The other point to these policies reflects Section 1.1 of the requirements: "Your app must offer customers unique, creative value or utility in all languages and markets that it supports." What this

means is that if you want to promote a cause or business through the app, do it in a way that delivers value to consumers. For example, an app for a nonprofit organization could help its users understand and be inspired by the organization's activities, keep up to date on current projects, and be directed to a place where perhaps they can make a donation. There's a fine line to walk here, because apps that simply ask for donations won't be accepted to the store (no creative value). Their primary purpose must be to inform, educate, inspire, entertain, and so on, with links to websites for any kind of charitable transactions. (Making donations directly through the Windows Store is not presently supported, plus it would incur revenue sharing, which you'd want to avoid anyway.)

More generally, free apps can also provide some useful functions in themselves but otherwise be a demonstration of features of any number of other apps—something like a tour of your paid offerings (so long as there's again real value in the app by itself). When related to only a single app, such a demo or "lite" version is different from a trial version of a full app. As we'll see shortly, a trial version should look and operate as if you had acquired a license to a full version, but it would be hobbled in some key ways or time-limited to place a restriction on its use. A demo app, on the other hand, is meant more to showcase features rather than provide a complete experience.

For example, let's say you have a game with five distinct "worlds" through which a player would normally progress in the app's full version. A trial version would allow a player to start working through those worlds but would cease to operate completely after some short period of time, say 30 or 60 minutes. In that time, a player might not progress past the first few levels in the first world, so the experience of the overall game is incomplete. A free demo/lite version, on the other hand, could be played as much as one wished but would contain only one level from, say, three of the five worlds. This gives the user a broader taste of the app and, because it can be played many times, serves as a continual advertisement for the full experience without giving anything more away. In other words, a demo app is like a teaser trailer: enough to create but not satisfy a hunger. (And, yes, while there may be some people that only ever watch free trailers and never go see a full movie, those are a rare breed. The same is true with apps.)

Great free apps can also fit well into an overall business model without asking for anything: they can help build a great reputation for your business, thereby supporting other paid offerings. Each app in the store can include links to your website and support information, so each one is a doorway to the rest of your business. In this way, free apps are like the giveaways (or loss leaders) that many businesses offer to get you in the door so that you can look at their full line of products without distraction.

Ad-Supported Apps

Ad-supported apps, which are typically free but can also be paid, are those that deliver some clear value in themselves and use that value to sell advertising space to others. Such advertising, while hopefully well-directed to the user's interests, typically isn't integral to the app's own function. Many free games, for instance, place interstitial (gap-filling) ads between levels or boards, ostensibly to keep you entertained while the next level is loading but in truth to take full advantage of your captured attention! The bottom line is that a user's attention, focused on something of value, has real value to advertisers who are willing to pay you for a bit of that focus.

As a user of the web, you're undoubtedly familiar with how ads can appear in an app's overall layout: filling gaps in space rather than in time. Typically, an app will place a control in such a space, which is itself connected to an ad service and pretty much manages itself. The control will acquire ads to display and track click-throughs, which is typically how you get paid: clicking is a sign that the user actually did pay a little attention to the ad, so you receive the value for that attention; users who ignore ads and never click them don't register.

Either way, many developers have found that selling ad space may be the most lucrative means of monetizing an app and building a business, but of course you have to understand your target audience and whether they're the sorts who will care for advertising at all.

The advertising control you use depends on the ad provider. For Windows 8, you can use the [Microsoft Advertising SDK](#), an extension that you incorporate into your app. For details on how this works, see the [Developer Walkthrough – HTML 5 JavaScript](#) documentation (part of the [Microsoft Advertising SDK for Windows 8 documentation](#)). I expect that other ad providers will make similar controls available in time.

Paid Apps and Trial Versions

Producing an app and charging for a license is certainly the one of the oldest means of monetizing, and it still works quite well. Value received for value delivered: that's the simple equation on which many successful products are built. Generally speaking, paid apps are free of advertising and are not advertisements themselves (again enforced by policy), hence customers' willingness to pay money for the apps in the first place. It could be, however, that we'll gradually see some creative means of ad insertion even into paid apps: after all, you pay for issues of a magazine and yet that magazine contains ads (unless you pay for premium magazines that contain none). Think too how we once balked at the idea of advertising on cable television or in movie theaters, but all that's just a matter-of-course now. The simple truth is that wherever there is a focus of customer attention, as already mentioned, there is a value to advertisers and to the businesses that can sell them access to that attention. You just have to be careful not to abuse those customers!

An important consideration for paid apps especially (but really for all apps) is the need for marketing. The existence of a place—the Windows Store—where customers can acquire your app doesn't eliminate the need for finding your customers and making them aware of your product. With every such store, there is a brief window of time where the total number of apps is still relatively small, meaning that users have a good chance of finding the app through casual browsing. But as soon as the store contains more apps, and users tire of browsing as a primary means of discovery, either users have to find you in a search (assuming you even show up in the top of the results list) or you have to generate interest through other means. This is again one of the functions of other free apps or demo versions that you might produce: if one of your free apps gets featured in some category, every user who downloads that free app at least has an opportunity to learn about your other products. And then, of course, there are all the other means to market your product: the social web, your company website and SEO, advertising in traditional media, and so forth.

You should also strongly consider making a trial version of the paid app available. A trial is typically free and is subject to an expiration date. As noted before, a trial app looks, feels, and operates like the real thing but is simply time-limited or hobbled. For example, a picture editor might allow you to edit but not save your work, meaning that you get the full experience of using the product without the full benefits of owning a license. A video converter app, as another example, might place a logo or watermark (that is, an advertisement) on the output video, so the functionality is all there, but the result isn't as useful. A trial version might also just disable in-app purchases, thereby limiting its extensibility until the full app is acquired.

Whether the trial is hobbled is your choice as a developer—if an app creates something and saves it in a particular format, such that you could not re-open those files without the same app (unlike pictures), there may be no reason to disable a save feature at all. In such cases, the strategy is to get the trial user heavily invested in continued use of the app, such that purchasing the full license is a better choice than letting go of that investment. Personal finance and contact management are good examples: in a 30-day trial period (or whatever period the app sets), users of such apps could amass quite a bit of useful data that they would not want to re-enter into another app. (Such a trial might also quietly disable any exporting features.)

A trial version is typically quite adept at reminding the user (that is, nagging them) about their trial status and that, hey, really, don't you want to get the real thing? The APIs for working with the Windows Store are such that checking for trial status (and its pending expiration) is quite simple. APIs also exist to initiate a streamlined purchase flow through which the user can acquire a full license with minimal disruption, all within the context of the app itself. In short, trial versions are an important monetization model that are, fortunately, quite easy to implement in Windows 8.

A technical stipulation of a trial version is that all the bits of the full version are actually already present on the user's machine: purchasing a full license from a trial version is simply an act of setting the license information in the Windows Store, and such a purchase will not initiate any new downloading. For a user, this means that to download and install a trial is to effectively download and install the full version, with the Store simply indicating that the user doesn't have full rights. If such a full download would be an obstacle, however, such as when the app is large, it may be a better strategy to create a much smaller demo version that will take the user to the appropriate page in the Store to buy the full app.

All of this is really about creating a smooth and painless experience for users to try new software. One of the primary motivations behind the Store (and the associated packaging technology) is to eliminate nearly all of the past risk of software acquisition: unknown or untrusted sources, potential malware, inconsistent install/uninstall procedures, and so forth. Microsoft wants Windows users to feel confident that they can experiment and try out new apps—*your apps!*—without corrupting their system, compromising their data, or in other ways being exposed to those sorts of problems.

Sidebar: Piracy Protection

The existence of the Windows Store and the fact that users cannot install an app except in the context of the Store provides a certain inherent level of piracy protection. Users are blocked from accessing the folders that contain installed appx packages, and even if they managed to extract and install one elsewhere, the Store would report that the app is unlicensed for that user and would thus refuse to run it.

Beyond that, any additional levels of protection are up to the app. It's perfectly allowable for an app to ask the user to register with the publisher (because customer information isn't shared from the Store) and to obtain a secondary license key. Windows does not block such procedures but doesn't provide any such services itself. Do consider, however, that customers might be annoyed by such additional requirements. It's best to exercise caution in such a decision.

In-App Purchases

In-app purchases are a primary means to monetize an application over time by selling incremental add-ons, options, periodicals, time-limited subscriptions/rentals, and so forth. By definition, the lack of any such options cannot interfere with the core operation of the app. In-app purchases cannot also be interdependent—that is, users cannot be required to purchase other options to use one they're already bought. Know too that an app is limited to 100 such in-app purchases when they are managed through the Windows Store; if you use your own commerce engine, as described in the next section, there is no such limit.

Whether in-app purchases are the right choice for your app involves a number of considerations:

- Implementing them well can be difficult because they introduce complexities into an app's architecture. (Note that Windows does maintain information for in-app purchases that can expire.)
- The app has full responsibility for correct delivery of the purchased item or feature, as opposed to the Store handling all the details.
- In-app purchases effectively create multiple variations of an app, which can increase user support and interaction.
- Overuse or inappropriate use of in-app purchases can generate the perception that you're trying to get money from users at every possible opportunity. Users who don't or won't pay for in-app purchases can still leave bad reviews about their experience.
- At present, the Windows Store supports only "durable" products (that can be purchased only once until they expire); there is no support for "consumable" products that can be repeatedly purchased. Consumables are under consideration for future versions; at present they can be implemented by using a custom commerce engine.

- In-app purchases through the Windows Store do not trigger download of additional content; they only change the user's license for that product. If needed, an app can initiate its own downloads once the product license has been acquired.

On the flip side, offering a new full version of an app with new features might generate better sales than offering the same features as in-app purchases. An app update is a real event in itself and can generate renewed interest in and energy around your product like the release of a new movie. In-app purchases, on the other hand, are by nature more prosaic, like the popcorn and drinks you buy in the theater—always there, and often considered essential for the whole experience, but not all that exciting outside that context. The best approach is probably to follow Hollywood's example and do both!

So it's worthwhile at even the earliest stages of design to think about what kinds of in-app purchases make sense for your product. You may not even at this time have anything you plan to offer, but you may want to add them later on. In short, keep the door open for expansion and creativity without necessarily having to revise the app. It's equally important to also think about what makes sense for *your customers*. We emphasize this point because there have been stories of outright abuse in this area. Apps aimed at young children, for example, have been known to dangle lots of in-app purchases like candy, enticing those children to press a "buy" button when they have no sense of the transaction. For this reason, the Windows Store will prompt the user for authentication with each purchase. Parents, protect your passwords!

The key thing is that if you try to be sleazy, you probably won't get far with your app. If you try to trick users out of their money, your app will certainly decline in ratings and reviews over time. And if you're found to be truly abusive, Microsoft does reserve the right to kicked your app out of the Store altogether, if it even passes certification at the outset.

All in all, these are all just considerations that will eventually affect how you set prices in the Store. You'll need to consider the tradeoffs involved between setting a higher price point with an initial-app purchase versus monetizing through multiple in-app purchases, and you'll need to be sensitive to how willing your target customers might be to making one purchase versus making multiple purchases. Apps that constantly nag their users to make additional purchases will be on par with pushy street vendors who just won't leave you alone.

Revenue Sharing and Custom Commerce for In-App Purchases

The subject of monetization is not complete without answering one of the most important questions: how much of the Store-related revenue stream do you, as the publisher of the app, get to keep? The basic answer is simple: 70% comes to you, 30% goes to the Store (you have to pay your rent). However, once an app achieves US\$25,000 in sales (from both the app and in-app purchases), your share increases to 80%.

Revenue sharing is always in effect for paid apps. For in-app purchases, however, you have the option to bypass the Windows Store altogether and use a commerce platform of your own, which potentially allows you to realize a much higher percentage of the revenue. This is an especially great option if you already have arrangements with a transaction provider through your existing websites.

Be aware that Sections 4.8 and 4.9 of the Store certification requirements apply here, where you need to ensure that the user enters credentials for each purchase and that each transaction meets the [PCI Data Security Standards](#).

With this custom commerce option, you're pretty much on your own where all the details are concerned, including UI—the Windows Store API itself doesn't provide for extensibility of its own mechanisms. You might draw from [The in-app purchase user experience for a customer](#) topic in the documentation to understand the flow, and you may also be able to find a third-party solution that provides an app control along with the backend commerce services.

Note that although the Windows Store does not presently support consumable in-app purchases, you can certainly implement this with your own commerce scheme. Doing so will also avoid the 100-item limit imposed by the Windows Store mechanisms.

The Windows Store APIs

Now that you've likely decided on a course for your app, let's see how you use the Windows Store APIs to accomplish those ends. These are found in the [Windows.ApplicationModel.Store](#) namespace; all objects referred to in this section are contained in this namespace unless noted.⁸⁰

First, know that basic licensing and trial enforcement comes for free: the app doesn't actually need to do anything at all! A user cannot acquire your app without going through the Store, and even if he did manage to, he'd have to have a developer license to install and run it. Furthermore, because the Store automatically tracks trial periods for apps, Windows will simply not launch an app once the trial is expired. Instead, Windows will redirect the user to the product's page in the Store where the user can purchase a full license.

An app can also set the expiration time of a license—not just for trials but the full app. This could be useful for apps that aren't valid or useful after a given date, such as event registration (conferences, meetings, etc.) or time-limited demos. Think about it, though: if the user has gone to the trouble of acquiring the app in the first place, do you really want to go and disable it? Far better, I imagine, is to maintain the usefulness of the app in some way. With event registration, for example, there are probably more events in the future that you could provide information about and perhaps open up registration at the appropriate time. Again, the user has the app already and must have had some intent in launching it even after it's expired—so can you leverage that intent in some way? It's a good question to ask.

As noted before, apps can enforce a secondary licensing scheme if desired. Here it would ask the user for a separate registration or a separately acquired license key of some sort. Again, Windows does not offer an API for this but will not block schemes of your own.

⁸⁰ To make a note, the [Windows.ApplicationModel.Package](#) class also provides a few details about the installed app package. Usage is simple, and you can refer to the [App package information sample](#) for more.

That said, WinRT provides for the following features:

- Retrieving app and product (in-app purchase) information from the Store, including price values formatted for the user's current locale.
- Retrieving license information for the app, indicating trials, expirations, etc. The app can make any decisions it wants with these details.
- Prompting the user to purchase a full license during or after a trial period; this is especially useful when the app is running and the trial period expires.
- Handling in-app (product) purchases.
- Generating receipts.
- Testing all the app's Store interactions prior to uploading to the store.

When an app runs for real—that is, after it has been uploaded to the Store and has made its way into the hands of customers—interaction with the API happens through the static [CurrentApp](#) object:

```
var currentApp = Windows.ApplicationModel.Store.CurrentApp;
```

whose methods and properties are as follows:

- [appId](#) The GUID that uniquely identifies the app in the Store.
- [linkUri](#) The URI ([Windows.Foundation.Uri](#)) to the app's listing page in the Store. (If you recall from Chapter 12, "Contracts," this is the value you want to store in the [application-ListingUri](#) property of a [DataPackage](#) used in the Share contract; doing so lets a user who receives the shared data easily find your app.)
- [licenseInformation](#) A [LicenseInformation](#) object.
- [loadListingInformationAsync](#) Retrieves the [ListingInformation](#) object for the app; through this you can retrieve information about the in-app products.
- [requestAppPurchaseAsync](#) Invokes the Store UI to invite the user to purchase the app. This is used when the app is running and detects that a license has expired.
- [requestProductPurchaseAsync](#) Invokes the Store UI to invite the user to do an in-app purchase.
- [getAppReceiptAsync](#) Requests an XML string that contains receipts for the app and any in-app purchases.⁸¹

A [ListingInformation](#) object contains a number of properties that come pre-localized as appropriate: [ageRating](#) (a number, currently one of 3, 7, 12, and 16), [currentMarket](#) (a BCP-47 string indicating the user's market that is used for transaction), [description](#) (a string containing the app's description from its Store page localized for the user's current market), [formattedPrice](#) (a string

⁸¹ There is also a [getProductReceiptAsync](#), but this is for future use and is currently not implemented.

containing the app's purchase price formatted for the user's current market and currency), `name` (a string with the app's name in the current market), and `productListings`. The latter is an array of `ProductListing` objects, each of which contains just three properties: `productId` (a string containing the app-defined product identifier), `formattedPrice` (a localized string containing the product price), and a localized `name` (a string). You can see that this collection is exactly what you'll use to present the user with a localized list of options they can purchase, where the `productId` could be used to retrieve additional content like images from your package or a web service.

The `LicenseInformation` object for its part contains simple properties of `expirationDate` (a `Date`), `isActive` (a Boolean), and `isTrial` (a Boolean). It has one event, `licenseChanged`, which you can use to detect any changes to these properties, such as the expiration of a license while the app is running, in which case you want to prompt for purchase. The remaining property, `productLicenses`, is a collection of `ProductLicense` objects. Each of these contains the appropriate `productId`, `expirationDate`, and `isActive` properties.

Tip For globalization purposes, never compare two dates with simple arithmetic operators like `<`, `>`, and `=`. Instead. Use the `Windows.Globalization.Calendar.compareDateTime` method, which will account for the specific needs of different calendar systems that might be in effect.

That's really the extent of the Store APIs in a nutshell. You may notice, by the way, that the APIs don't concern themselves with ad-supported apps, since ads don't involve the Store itself.

But you might be asking yourself some very significant questions: how on earth can this API return any meaningful information while the app is under development and has yet to be uploaded to the Store in the first place? How can you get product information and test all your purchase features when there's nothing yet available to purchase?

These are great questions, and the answer lies in the one other object in the `Windows.ApplicationModel.Store` namespace that is our next topic: the Windows Store app simulator.

The CurrentAppSimulator Object

To make it possible to test an app's interactions with the Store before the app is actually onboarded, WinRT provides the static `CurrentAppSimulator` object that is identical to `CurrentApp` with two exceptions: the simulator object works against data from a local XML file rather than live data from the Store, and the object has an extra method, `reloadSimulatorAsync`, to reinitialize the simulator with such XML. During development, you'll want to use this line of code to start your work with the API:

```
var currentApp = Windows.ApplicationModel.Store.CurrentAppSimulator;
```

and then delete the *Simulator* suffix when you're ready to send the app to the Store. (And if you forget, you'll fail Store certification.)

When your app accesses [CurrentAppSimulator](#), WinRT looks for a file called WindowsStore-Proxy.xml in your app data, specifically under %userprofile%\AppData\local\packages\<package name>\Microsoft\Windows Store\ApiData. If it exists, the simulator is initialized from that data; otherwise the file is created with the following defaults (slightly formatted to fit the page):

```
<?xml version="1.0" encoding="utf-16" ?>
<CurrentApp>
  <ListingInformation>
    <App>
      <AppId>00000000-0000-0000-0000-000000000000</AppId>
      <LinkUri>
        http://apps.microsoft.com/webdp/app/00000000-0000-0000-0000-000000000000
      </LinkUri>
      <CurrentMarket>en-US</CurrentMarket>
      <AgeRating>3</AgeRating>
      <MarketData xml:lang="en-us">
        <Name>AppName</Name>
        <Description>AppDescription</Description>
        <Price>1.00</Price>
        <CurrencySymbol>$</CurrencySymbol>
        <CurrencyCode>USD</CurrencyCode>
      </MarketData>
    </App>
    <Product ProductId="1" LicenseDuration="0">
      <MarketData xml:lang="en-us">
        <Name>Product1Name</Name>
        <Price>1.00</Price>
        <CurrencySymbol>$</CurrencySymbol>
        <CurrencyCode>USD</CurrencyCode>
      </MarketData>
    </Product>
  </ListingInformation>
  <LicenseInformation>
    <App>
      <IsActive>true</IsActive>
      <IsTrial>true</IsTrial>
    </App>
    <Product ProductId="1">
      <IsActive>true</IsActive>
    </Product>
  </LicenseInformation>
</CurrentApp>
```

The full XML schema for this can be found on the [CurrentAppSimulator](#) page, but it's straightforward to see exactly where you'd modify the XML to test different scenarios:

- Create additional [MarketData](#) elements to specify app details for other locales. The [CurrentMarket](#) element indicates the default.
- Create additional [Product](#) elements (including their [MarketData](#) children) for each in-app purchase.

- In the `App` element under `LicenseInformation`, change the values of `IsActive` (that is, not expired) and `IsTrial` between `true` and `false` to test the variations: active/non-trial, active/trial, expired/non-trial, and expired/trial. You can also add an `ExpirationDate` element to indicate when the app expires (in UTC time), using the form of `yyyy-mm-ddThh:mm:ss.ssZ` (replacing `yyyy:mm:dd` with the date and `mm:ss.ss` with the time). For automated testing, additional elements allow you to hard-code result codes; see the [CurrentAppSimulator](#) page for details.
- For each product, add a `Product` element under `LicenseInformation` with the appropriate `ProductId` attribute. Supported child elements are `IsActive` and `ExpirationDate`, with the same meaning as the app license.

It's important to note that using the methods in the simulator object that change license status, such as converting a trial app to a purchased app or acquiring in-app purchases, will not alter the contents of the `WindowsStoreProxy.xml` file. This means you can just restart the app to reset the state of the simulator object. But it also means you'd need to edit the XML and launch the app again to test how different variations are handled on startup. (Note also that the store simulator state is not persisted when the app is suspended and terminated.)

For this purpose, the simulator object's `reloadSimulatorAsync` method takes a `StorageFile` containing the XML initialization data. This can very much simplify your testing procedures, and often you'll have such files directly in your project folder such that you can refer to them with `ms-appx:///` URIs. However, make sure that these files don't end up in your app package when you upload to the Store. In Visual Studio, right-click the file in the Solution Explorer pane and select Properties. In the Property Pages dialog that appears, as shown in Figure 17-3, set Package Action to None.

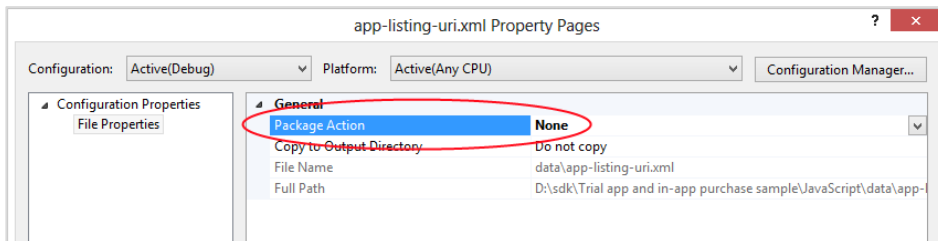


FIGURE 17-3 Make sure that XML configuration files for the simulator object don't end up in your Store packages.

The [Trial app and in-app purchase sample](#), which we'll be drawing from in the sections ahead, use `reloadSimulatorAsync` to load a specific XML file for each of its scenarios. In Scenario 4, for example (`js/api-listing-uris`), it loads `data/app-listing-uri.xml` as follows:

```
var currentApp = Windows.ApplicationModel.Store.CurrentAppSimulator;

var page = WinJS.UI.Pages.define("/html/app-listing-uri.html", {
    ready: function (element, options) {
        // ...
        loadAppListingUriProxyFile(); // Initialize the license proxy file
    },
    unload: function () {
```



```

        currentApp.licenseInformation.removeEventListener("licensechanged",
            appListingUriRefreshScenario);
    }
});

function loadAppListingUriProxyFile() {
    // We could also use folder.getFileFromPathAsync("ms-appx:///data/app-listing-ur.xml")
    // instead of the two-step process with getFileAsync as shown here.
    Windows.ApplicationModel.Package.current.installedLocation.getFolderAsync("data").done(
        function (folder) {
            folder.getFileAsync("app-listing-uri.xml").done(
                function (file) {
                    currentApp.licenseInformation.addEventListener("licensechanged",
                        appListingUriRefreshScenario);
                    Windows.ApplicationModel.Store.CurrentAppSimulator
                        .reloadSimulatorAsync(file).done();
                });
        });
}

```

Notice how this sample listens for the `licensechanged` event and makes sure to call `removeEventListener` when the page is unloaded. (See the “WinRT Events and `removeEventListener`” section in Chapter 3.)

This same Scenario 4 shows the basic retrieval of app information from the Store. When you click the Show Uri button on that page, it goes to the handler below that simply outputs the app’s `linkUri` property:

```

function displayLink() {
    WinJS.log && WinJS.log(currentApp.linkUri.absoluteUri, "sample", "status");
}

```

Getting at the app’s other properties would look the same, just using `currentApp.loadListingInformationAsync` first to obtain that data. This is shown in Scenario 1 (`js/trial-mode.js`):

```

function trialModeRefreshScenario() {
    currentApp.loadListingInformationAsync().done(
        function (listing) {
            document.getElementById("purchasePrice").innerText =
                "You can buy the full app for: " + listing.formattedPrice + ".";
        });

    displayCurrentLicenseMode();
}

```

And on that note, let’s look at the rest of the sample more fully because it shows the other use scenarios of the Store API as a whole.

Trial Versions and App Purchase

Implementing a trial version that hopefully leads to an app purchase is demonstrated in Scenario 1 of the [Trial app and in-app purchase sample](#). When you run this sample and select Scenario 1, as shown in Figure 17-4, the simulator object is initialized using data/trial-mode.xml where the app's `IsActive` and `IsTrial` elements are both set to true, meaning that we have a valid trial license. The `ExpirationDate` for this license is set to January 1, 2014, but we'll play around with that in a moment.

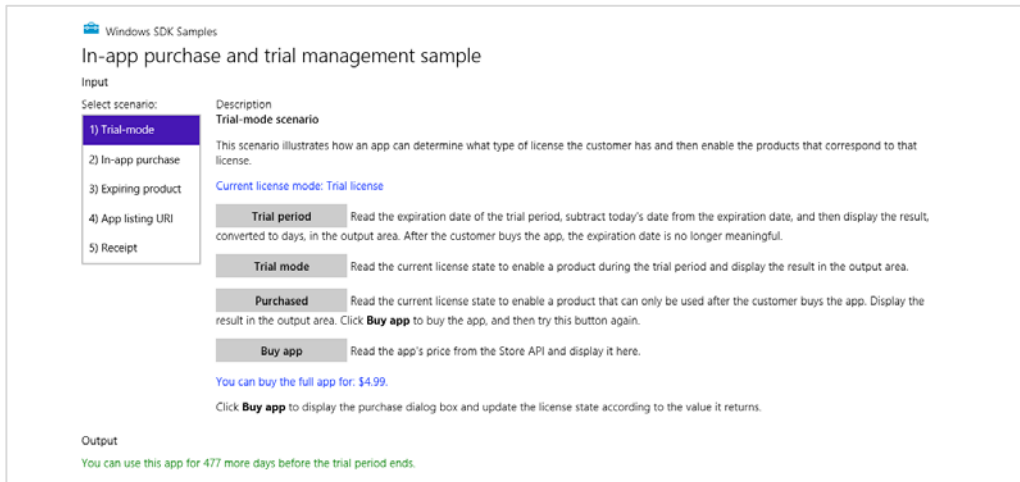


FIGURE 17-4 Scenario 1 of the Trial apps and in-app purchases sample (cropped slightly).

The Trial Period button in this scenario just calculates the number of days remaining in the trial period, using basic arithmetic and the `licenseInformation.expirationDate` property. Again, let me point out that the proper way to do this is with the `Windows.Globalization.Calendar` class that we'll see later in the "World Readiness and Localization" section and demonstrated in the [Calendar details and math sample](#). Using the APIs designed for this purpose will insulate your app from regional variations.

The Trial Mode and Purchased buttons just output different messages based on the state of the `isActive` and `isTrial` properties. Both button click handlers start like this:

```
var licenseInformation = currentApp.licenseInformation;
if (licenseInformation.isActive) {
    if (licenseInformation.isTrial) {
```

What can make the output from these buttons more interesting is modifying the data/trail-mode.xml file with different initial values for `IsActive` and `IsTrial`. Also try setting the `ExpirationDate` to a time in the past (remembering that it's UTC time, not local time), and you'll see that `IsActive` automatically gets set to `false`. You can also try setting `ExpirationDate` about a minute in the future, set a breakpoint on the `trailModeRefreshScenario` function inside `js/trial-mode.js`, then run the sample again.

You won't hit your breakpoint immediately after `ExpirationDate` has passed, however. For performance reasons, the `licensechanged` event is not triggered immediately—there could be hundreds of expiration dates to track throughout the system. The event will instead fire reasonably soon, within about 20 minutes, so you might start such a test before going out for lunch.

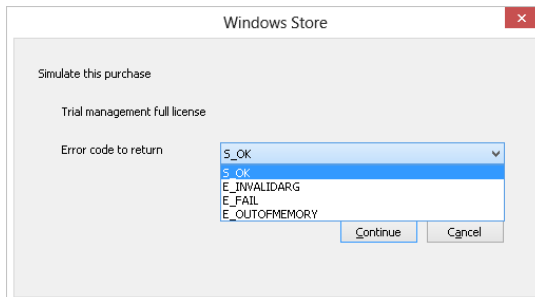
This sample, of course, merely changes some output messages according to the validity of the license. In a real app you would either disable certain features for an active trial license or let the user do nothing except purchase the app if the trial has expired. You'd want to make such checks both when the app is run and in the `resuming` event.

The latter case is handled by the Buy App button in this scenario, an option that you would almost always present to users of your trial version at appropriate times, regardless of expiration status. This button calls a function called `doTrialConversion` that makes use of the `CurrentApp.requestAppPurchaseAsync` method (sample output code has been replaced here with comments identifying the specific results):

```
var licenseInformation = currentApp.licenseInformation;
if (!licenseInformation.isActive || licenseInformation.isTrial) {
    currentApp.requestAppPurchaseAsync(false).done(
        function () {
            if (licenseInformation.isActive && !licenseInformation.isTrial) {
                // Purchase was fulfilled
            } else {
                // Purchase UI was shown, but the user canceled.
            }
        },
        function () {
            // There was an error in the transaction; purchase did not occur
        });
}
```

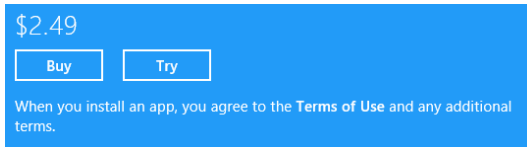
The one argument to `requestAppPurchaseAsync` indicates whether a receipt string is sent to your completed handler; see “Receipts” below. In any case, if the user makes a purchase, the `license-changed` event will fire as it does for trial expiration, so you can always consolidate your license handling there.

When running in the simulator and you invoke `requestAppPurchaseAsync`, you won't see the actual Store UI. Instead you'll get an ultra-prosaic dialog from the simulator object in which you can specify the exact return value (an HRESULT):



Sending back `S_OK` indicates that the purchase was made. The `isTrial` flag should change to `false` and `isActive` set to `true`. Returning any of the other errors will invoke the error handler for `requestAppPurchaseAsync`. Pressing Cancel, on the other hand, will call your completed handler but the values of `isTrial` and `isActive` will remain unchanged.

In the real world, of course, consumers will not be fiddling around with simulated Store conditions. Instead, if your app is marked to offer a trial version (something you set while uploading to the Store), they'll see a Try button on the app's listing page like this:



Tapping Try will install the app and set both `isActive` and `isTrial` to `true`. At the point when the app calls `requestAppPurchaseAsync`, Windows will launch the Store and take the user to the app's listing page where they can tap the Buy button if they choose.

Tip When writing this book, I looked at a number of apps that were available in the Windows Store and found that while many offered trials, few of them gave me any indication about why and how to purchase a full version. I was presented with such an option only when the (unknown) trial period had passed. If you want to convert trials into paid licenses, it's better, even as the sample demonstrates, to inform the user that she's running a trial and give her reminders and opportunities to convert!

Sidebar: Expiring Apps?

Although the app's expiration date is often used in conjunction with a trial license, there's no limitation that it must be so: a free or paid app can also expire. If you don't indicate a trial version when uploading the app to the Store yet indicate an expiration, the `isActive` flag will change to false at that date and time. A `licensechanged` event will also fire, allowing a running app to take appropriate action. The app can also check the active status and/or expiration date on startup and in the `resuming` event and display an appropriate message. Such an option is possibly useful for apps that truly have a limited lifespan, say an app that provides news and other information about a certain candidate's political campaign. It might not expire immediately after election day, of course—maybe the app is still good for another few months, at which point it could be taken down from the Store altogether. Yet, as noted before, if the user has gone to all the trouble of acquiring an app in the first place, why not make it useful even after its primary purpose has been fulfilled? Updates to the app can also add fresh content and capabilities later.

Listing and Purchasing In-App Products

There are two aspects of working with your in-app purchases, or *products* as the API calls them. The first is letting the user appropriately know that those products are available through your own UI. The second is then completing the purchase and activating the product's license.

If you're using a custom commerce engine, the app will use its own services to retrieve the necessary product information and handle all the UI and license management for a transaction—the Store API will not play any role here. If the products are handled through the Store, on the other hand, the [ListingInformation.productListings](#) collection supplies localized product details and the [CurrentApp.requestProductPurchaseAsync](#) method handles the transaction, including the UI. This section focuses on these APIs.

As noted before, in-app purchases can have expiration dates, after which time the user needs to repurchase the product to continue its use. This is appropriate for subscriptions or rentals where the user was fully aware at the time of purchase that the product would eventually expire. In general, you should always make it clear if a product purchase will expire. Don't surprise the user or the user will likely surprise you with a less than favorable review in the Store!

In-app purchases are demonstrated in Scenario 2 of the sample we've been using. In this case, the [CurrentAppSimulator](#) is initialized with data/in-app-purchase.xml, which defines two products (prosaically named Product 1 and Product 2), the first of which has an active license and the second of which is inactive. When you switch to this scenario (js/in-app-purchase.js), the sample loads the app's [ListingInformation](#), retrieves the product details from the [productListings](#) collection, and then displays those options (as shown in Figure 17-5):

```
currentApp.loadListingInformationAsync().done(
    function (listing) {
        var product1 = listing.productListings.lookup("product1");
        var product2 = listing.productListings.lookup("product2");
        document.getElementById("product1SellMessage").innerText =
            "You can buy " + product1.name + " for: " + product1.formattedPrice + ".";
        document.getElementById("product2SellMessage").innerText =
            "You can buy " + product2.name + " for: " + product2.formattedPrice + ".";
    });
```

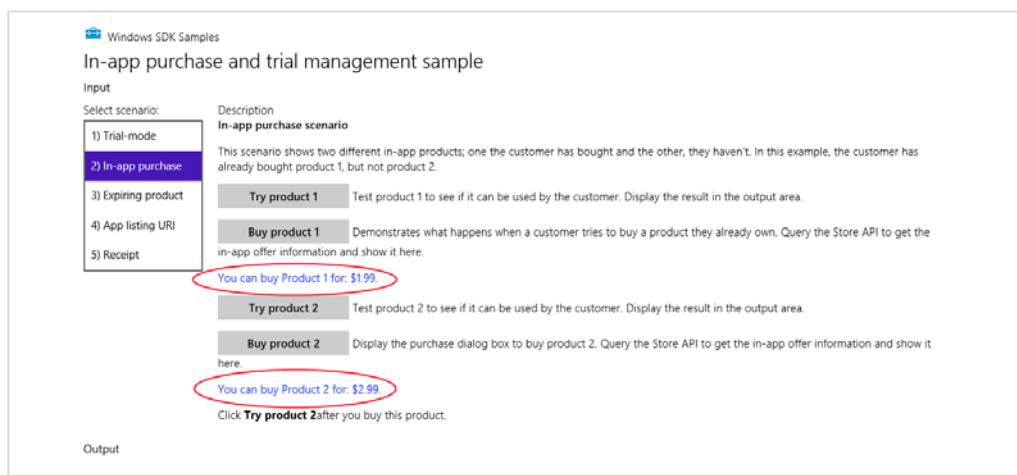


FIGURE 17-5 Scenario 2 of the Trial apps and in-app purchases sample (cropped); product information obtained from the Store is displayed in blue text (circled here in red).

Note that the `productListings` collection is an `IMapView` object and not an array; you can retrieve a specific item in the collection by using its `lookup` method, as in the code above, or by using a key-based array lookup:

```
var product1 = listing.productListings["product1"];
```

Iterating through the `IMapView` takes a little more work; it does not support index-based lookup nor the `foreach` method. You instead use an `IIterator` obtained through the `first` method, as shown here:

```
var iterator = listing.productListings.first()
var product;

while (iterator.hasCurrent) {
    product = iterator.current.value;
    document.getElementById(product.productId + "SellMessage").innerText =
        "You can buy " + product.name + " for: " + product.formattedPrice + ".";
    iterator.moveToNext();
};
```

This code is completely equivalent to the previous snippet and relies on the fact that the product ID just so happens to match the first part of the appropriate element ID in the HTML. In any case, this is the sort of code you would use to present a variable list of options to the user.

In doing so, you'll likely want to filter out those products that have already been purchased. In that case, you'd look up the product license within the `CurrentApp.licenseInformation.productLicenses` collection, which is another `IMapView` of `ProductLicense` objects, using the product's ID as the key. Here's how we'd modify the code above to perform this additional step:

```
var iterator = listing.productListings.first()
var licenses = currentApp.licenseInformation.productLicenses;
var product, message;

while (iterator.hasCurrent) {
    product = iterator.current.value;

    if (licenses[product.productId].isActive) {
        message = "You own " + product.name + ".";
    } else {
        message = "You can buy " + product.name + " for: " + product.formattedPrice + ".";
    }

    document.getElementById(product.productId + "SellMessage").innerText = message;
    iterator.moveToNext();
};
```

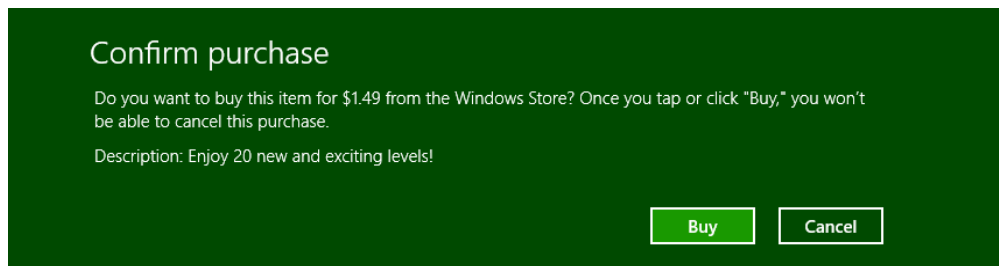
If you use this code in Scenario 2 of the sample (which is provided in the modified sample in this chapter's companion content), you'll see the message "You own Product 1" when you first switch to that scenario. You could also add a further refinement to check the `expirationDate` property of each product license and display its remaining time.

You might have noticed that a `ProductListing` (from the app's `productListings` collection) only contains a `name` and not a description. This really means that the name *is* the description and you should use it as such, rather than using it as another type of identifier, for which you already have `productId`. In other words, a product that provides 20 extra levels for a game should be named something like "Twenty extra levels with new challenges" rather than "extra_levels" because that name will appear in UI.

As you dangle all your product options in front of the user, the user will (I'm being very affirmative here!) at some point want to purchase one or more of them. When the user taps the appropriate button, like the Buy Product buttons in the sample (see Figure 17-5 again), the app just needs to call `requestProductPurchaseAsync` with the product ID and a Boolean indicating whether the method should provide a receipt:

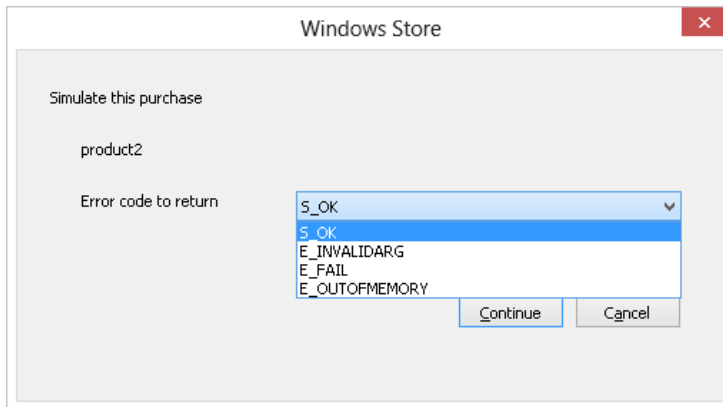
```
currentApp.requestProductPurchaseAsync("product1", false).done(
    function () {
        if (licenseInformation.productLicenses.lookup("product1").isActive) {
            // Purchase was fulfilled; UI is not shown if the user already owns the product.
        } else {
            // Purchase UI was shown, but the user canceled.
        }
    },
    function () {
        // There was an error in the transaction; purchase did not occur
    });
```

If the product already has an active license, `requestProductPurchaseAsync` will simply call your completed handler without showing any UI, as none is needed. Otherwise the user will see a series of prompts to confirm the purchase, including confirmation of their credentials. For the whole flow, see [The in-app purchase user experience for a customer](#). A typical confirmation message is shown below:



Note that the warning here, which exists to meet regulations in some countries, is not entirely true: you can also cancel entering your credentials in the next dialog.

When using the `CurrentAppSimulator`, of course, you won't see the Store prompts but only another simple dialog to control the result:



As with the app purchase, any changes in product license status will trigger the `licensechanged` event; your handler for that event makes a great place to update your app status and initiate any downloads related to the purchase. The same event will be fired if a time-limited product license expires, which is your signal to make the purchase available again. It's likely that you might also want to alert the user to that status, perhaps with a toast notification or with inline messages when the user tries to access that feature.

Receipts

Both the `requestAppPurchaseAsync` and `requestProductPurchaseAsync` methods of `CurrentApp` have an option, as we saw earlier, to provide a receipt string to the completed handler of the async operation. Its `getAppReceiptAsync` method also provides an all-up receipt (app and products) at any time. Generally speaking, receipts are most useful when a service needs to validate that an app is authorized to use certain functionality. The app acquires the receipt and sends it to the service, which can then do whatever validation it requires.

In all cases, the receipt is an XML string that contains information such as the app or product id, the dates when the purchase was made and when the receipt was issued, and a digital signature. The details of the XML schema can be found on the reference page linked above.

As an example, here's the receipt string provided from `requestProductPurchaseAsync` when purchasing Product 2 in Scenario 2 of the sample:

```
<?xml version="1.0" encoding="utf-8"?><Receipt Version="1.0" ReceiptDate="2012-09-11T17:35:55Z" CertificateId="" ReceiptDeviceId="7a61447d-c8f4-457a-8310-363cbdfdd21c"><ProductReceipt Id="e729be49-8299-4122-b6fb-a95bcfac6a7c" AppId="Microsoft.SDKSamples.Store.JS_8wekyb3d8bbwe" ProductId="product2" PurchaseDate="2012-09-11T17:35:55Z" ProductType="Durable" /></Receipt>
```

Here's what Scenario 5 of the same sample receives from `getAppReceiptAsync`:

```
<?xml version="1.0" encoding="utf-8"?><Receipt Version="1.0" ReceiptDate="2012-09-11T17:39:39Z" CertificateId="" ReceiptDeviceId="50b4267d-437d-429e-a4b8-88da96da9e52"><AppReceipt Id="1550eb89-31ae-4559-b516-267afe47ae19" AppId="Microsoft.SDKSamples.Store.JS_8wekyb3d8bbwe" PurchaseDate="2012-09-11T17:39:12Z" LicenseType="Full" /><ProductReceipt
```



```
Id=\e2a62d42-dbca-43d2-b779-66eb916d9df4\" AppId=\"Microsoft.SDKSamples.Store.JS_8wekyb3d8bbwe\"  
ProductId=\"product2\" PurchaseDate=\"2012-09-11T17:39:12Z\" ProductType=\"Durable\"  
ExpirationDate=\"2014-01-01T00:00:00Z\" /><ProductReceipt Id=\"21967f50-ac55-4b41-acd9-f1e86ad6c7b9\"  
AppId=\"Microsoft.SDKSamples.Store.JS_8wekyb3d8bbwe\" ProductId=\"product1\"  
PurchaseDate=\"2012-09-11T17:39:12Z\" ProductType=\"Durable\" /></Receipt>
```

If you want to consume a receipt as an XML document instead of a string (for display or print), it's a simple matter to create such an object like we did with tile and notification XML in Chapter 13:

```
var receiptDOM = new Windows.Data.Xml.Dom.XmlDocument();  
receiptDOM.loadXml(receipt);
```

Accessibility

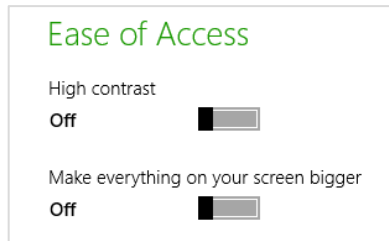
As I mentioned in this chapter's introduction, nearly 60% of users employ accessibility features in some capacity—sometimes because of a real disability, sometimes due to personal preference, and sometimes just to make the device easier to use within certain environments. In many countries, accessibility is actually a legal requirement, so it will be necessary if you plan to make an app available in those regions. In short, supporting accessibility is something that every app should do and do well, and fortunately this isn't the onerous task you might think it to be. (For reference, see [Making your app accessible](#) and [Introduction to Web Accessibility](#); the latter is written for web apps but is very applicable to Windows Store apps. Also see [Guidelines and checklist for Accessibility](#), [Practices to avoid for accessible apps](#), and [Implementing accessibility for particular content types](#).)

Accessibility might feel like a lot of work because developers are relatively unfamiliar with what it means. To remedy this, take a few minutes to give yourself some direct experience. But before you do anything else:

*Go to PC Settings > Sync Your Settings and **turn off** the options for Desktop Personalization and Ease of Access. Otherwise the effects of your tinkering will roam to other devices you might have. I learned this the hard way when I was playing around with contrast settings on my main laptop after which a game my son wanted to play on a tablet came up mostly black! Clearly, that app didn't handle high contrast well, but it also took me a while to figure out what was going on!*

Now that we've taken care of that detail, try the following:

- Press Left Shift+Alt+Print Screen or go to PC Settings > Ease of Access and toggle *high contrast* mode (see image below). How does the app respond? Are all of the critical elements visible? A mode like this is important for users who have difficulty distinguishing subtle colors.



- You can also select a particular high contrast theme through Control Panel > Appearance and Personalization > Personalization, where you have a choice between three black background themes and one white-background theme; the latter is the same one that's activated through PC Settings or Left Shift + Alt + Print Screen:



- In PC Settings > Ease of Access, tap Make Everything on the Screen Bigger. If your display is large enough, this will effectively scale everything by 140%. How does the app respond to the new screen dimensions? As discussed in Chapter 6, turning this on will activate the 140% resolution scaling, even though you're not using a high pixel density device.
- Press Win+Ctrl+U to start (and stop) the built-in *screen reader* called Narrator. Win+Enter also starts it, and you can press Win+U to go to Control Panel > Ease of Access Center and tap Start Narrator. (Note that Narrator is a desktop application that starts minimized; you need to close that application to stop Narrator.) Now turn off the monitor. Can you still use the app? What happens when you navigate around with the keyboard? Do you hear an audible indication of where the focus is? This is clearly important for users who are blind or visually impaired.
- If you have a mouse, disconnect it and try *keyboard-only navigation* (you can open your eyes now). This is important for users with mobility issues and those who rely on speech recognition.
- With your mouse connected, go to Control Panel > Ease of Access Center and tap Start On-Screen Keyboard to try *mouse/touch-only navigation*. This special on-screen keyboard is different from the one activated for touch with input fields (as we saw in Chapter 9, "Input and Sensors") because it always remains visible.

Through this experience I hope you've gained some understanding of what accessibility means. Simply said, there are four key scenarios for accessibility support: screen readers, keyboard-only or mouse-only input, high contrast, and resolution scaling.

The latter two we've already covered. In Chapter 6, "Layout," we saw how to work with different resolution scales, how to handle varying screen sizes (which can occur as a result of scaling), and how to provide raster graphics for different scales so that they always look their best. For a quick review, you might want to revisit the [Scaling according to DPI sample](#).

Input considerations were also covered in Chapter 9, and I'll remind you again that the Store certification policy (section 3.5) requires that apps support all forms of input. Typically, this isn't an issue for mouse and touch; the real work to be done is making sure that your app can be used with nothing but a keyboard. As noted in Chapter 9, see [Implementing keyboard accessibility](#) for full details. Testing your app with Narrator turned on will also reveal whether you've paid any attention to keyboard navigation, because no matter how well your elements are labeled for that purpose, those labels don't do any good if the user can never set the focus to them!

It's also worth mentioning that including closed captions in video will assist users who are hearing impaired. Doing so, however, is a detail for the video data itself or can be implemented via text overlays on a `video` element. See the [HTML5 and Accessibility](#) in MSDN Magazine for more on video accessibility.

Let's now look at how we support screen readers and contrast variations.

Sidebar: Accessibility Test Tools

The Windows SDK includes two tools to help you verify your implementation of accessibility. The first is called Inspect, a UI automation tool that checks through the accessibility information you've made available to screen readers and lets you know what you've missed. The second is called AccChecker, which runs a series of verifications on the rest of the app. You'll find these tools in the Windows SDK install folder, typically `c:\Program Files (x86)\Windows Kits\8.0\bin\x86`. You might also be interested in the Accessible Event Watcher and the UI Automation Verify tools. For usage details on all of these, see [Automation Testing Tools](#) in the documentation as well as [Testing your app for accessibility](#). Of course, for the most complete kind of testing, find yourself a few users who regularly work with assistive technologies, set them up with a developer license (so that you can share your app package), and let them put your app through its paces!

Sidebar: Narrator and `tabindex` Attributes

When making your app navigable by keyboard, be careful not to overuse `tabindex` attributes on elements that don't need them, thinking that this will help Narrator for noninteractive elements. It actually doesn't. Narrator has its own keyboard commands (like CapsLock+arrows) and its own navigation modes that skilled users employ to read anything on a page, irrespective of `tabindex`. For this reason, setting `tabindex` properties on static elements *decreases* Narrator's usability; you should set the property only on interactive elements. In other words, understand that using an app through keyboard and using it through Narrator are different processes, and think only of `tabindex` in the context of keyboard navigation.

Screen Readers and Aria Attributes

Screen readers like the built-in Narrator can work only if the app provides some kind of information that tells the screen reader about the elements in the UI. For Windows Store apps written in HTML, CSS, and JavaScript, this is achieved through `aria-*` attributes on your UI elements.

Tip If you need separate text to speech capabilities, the [Bing Translator](#) API, for example includes a `Speak` method available through AJAX, SOAP, and HTTP interfaces that generates a WAV or MP3 stream for text in a given language. Other web services also exist for this purpose, and third-party libraries might be available.

ARIA stands for Accessible Rich Internet Applications, a standard that's spelled out in the [WAI-ARIA specifications](#). WIA itself stands for the [W3C Web Accessibility Initiative](#). Two other W3C documents of interest are the [WAI-ARIA Primer](#) and [WAI-ARIA Authoring Practices](#).

What it really boils down to is that assistive technologies like Narrator are first able to automatically derive what they need from certain elements, like header element, paragraphs, the `title` attribute, `label` elements associated with focusable elements (using the label's `for` attribute), button text, input elements, the `caption` attribute of a table, and the `alt` attribute of `img` elements. The `role` attribute also comes into play here.

For everything else, as for `div` elements (including custom controls) whose role cannot be inferred, the specs define a number of attributes, starting with `aria-`, to indicate the role that a particular element plays in the app. Full details can of course be found in the specifications linked above, along with the [ARIA reference](#) on the Windows Developer Center. Another good reference topic is also [Exposing basic information about UI elements](#) in the documentation, that shows examples of a number of the core `aria-*` attributes. That page makes a special note about the `canvas` element. Because a `canvas` is just a pixel bucket, it generally doesn't have content that is accessible to screen readers, even though it might appear as text. Make a special effort, then, to give a canvas appropriate attributes as you would with other custom elements (again see the [HTML5 and Accessibility](#) article for more on `canvas`).

All of the controls in WinJS are fully stocked with ARIA attributes and other bits that work with assistive technologies, so by using them you get lots of accessibility for free. (An exception is the SemanticZoom control that specifically does not have an `aria-label` itself because it's a container for other controls that should have such labels themselves.) That said, it's still necessary for you to properly adorn other elements, including HTML controls like `progress`. Here's a summary of the core `aria-*` attributes:

- `aria-label` Directly provides text for screen readers.
- `aria-labelledby` (Note the spelling with two l's.) Specifies the identifier of another element that contains the appropriate label for an element. The specifications state that `aria-labelledby` should be used instead of `aria-label` if that text is already on the screen.
- `aria-describedby` Similar to `aria-labelledby`, identifies an element that contains fuller description instead of just label text. Narrator reads that text when the user presses the Win+Alt+F key on the element with this attribute. This is a good option to use with a `canvas` that contains drawn text: if you also store that text in another element linked with this attribute, even a hidden element, then the user has a way to hear that content.
- `aria-valuemin`, `aria-valuemax`, and `aria-valuenow` For `div` elements whose `role` is set to *slider*, *progressbar*, or *spinbutton*, these indicate the values within the control. `aria-valuetext` can also provide text that corresponds to the value of `aria-valuenow`.
- `aria-selected`, `aria-checked`, `aria-disabled` and `aria-hidden` Indicate the state of an element.
- `aria-live` Needed for content that changes dynamically, such as master-detail views, chat, RSS feeds, fragment loading, and so forth.

Back in Chapter 4, “Controls, Control Styling, and Data Binding,” we saw that a special syntax was necessary to do data-binding on attributes of target elements where there are no associated JavaScript properties. The `aria-*` attributes are the primary example of this, because of their hyphenated names, for which we use `this[]` along with special WinJS initializers in `data-win-bind`:

```
<div data-win-bind="this['aria-label']: title WinJS.Binding.setAttribute"></div>
<div data-win-bind="this['aria-label']: title WinJS.Binding.setAttributeOneTime"></div>
```

It’s probably more typical, though, that you’ll provide localized ARIA labels in your app’s resources, and for this there is a different declarative syntax that we’ll see later on in “World Readiness and Localization.”

The ARIA Sample

To see the various `aria-*` attributes and Narrator in action, the best place to turn is a unique sample in the Windows SDK, the [ARIA sample](#). One of its unique characteristics is that it doesn’t at all *look* like an SDK sample, as you can see in Figure 17-6. This was done to intentionally represent the content of a typical app, without all the other chrome that normally decorates the samples. In this case the sample emulates a simple chat app with a kind of master-detail view on the left side.

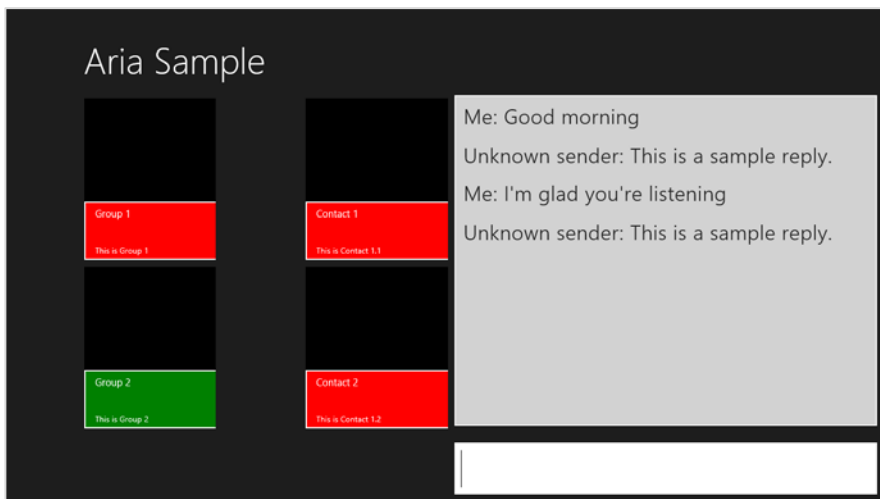


FIGURE 17-6 The ARIA sample's main page.

When you run this sample, be sure to turn on Narrator to hear what it has to say. (I highly recommend doing this in the Visual Studio simulator because then Narrator is only running in that session and not for your entire machine!) You'll find that it's accurately reflecting what's happening on the screen, especially as you Tab or Shift+Tab between controls, press Enter or the spacebar to select items, and enter chat text. Again, turn off your monitor, close your eyes, get a blindfold, or have your five-year-old come up behind you and cover your eyes to get the full experience.

Pressing Enter on an item in the left-hand list will update the contacts shown in the middle. Selecting one of those contacts and pressing enter will then open another page containing a table—a contrived table, certainly, but one that shows the `aria-*` attributes that apply there. The page, shown in Figure 17-7, also provides an opportunity to experience page navigation through the keyboard and Narrator.

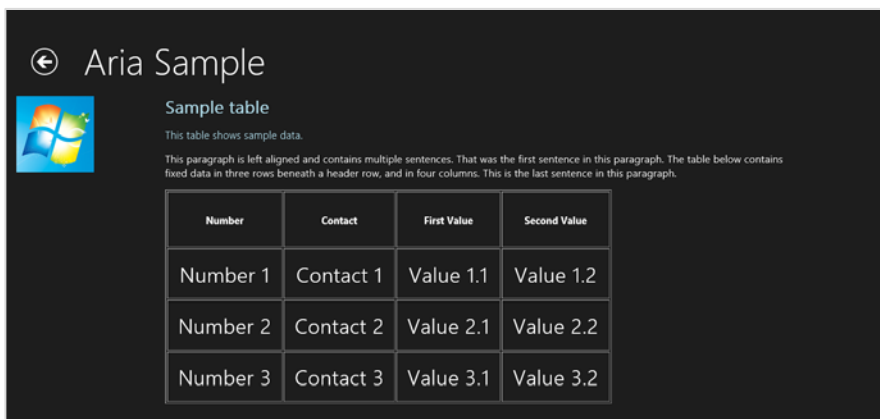


FIGURE 17-7 The ARIA sample's secondary page (cropped a bit).

Apart from the small bits of code in `pages/chat/chat.js` to work the chat window, the really interesting parts of this sample are all contained in the markup, specifically `pages/chat/chat.html` (the main page) and `pages/table/table.html` (the secondary page). In the first we can see `aria-label` on most of the controls (with the text you hear as you tab around) and much more extensive roster of attributes for the chat output `div` near the bottom:

```
<div class="chatpage fragment">
  <header aria-label="Header content" role="banner">
    <button class="win-backbutton" aria-label="Back" disabled></button>
    <h1 class="titlearea win-type-ellipsis">
      <span class="pagetitle">Aria Sample</span>
    </h1>
  </header>
  <section aria-label="Main content" role="main">
    <div class="chat">
      <div class="groupslist" aria-label="List of groups"
        data-win-control="WinJS.UI.ListView"
        data-win-options="{ selectionMode: 'none' }"></div>
      <div class="contactslist" aria-label="List of contacts"
        data-win-control="WinJS.UI.ListView"
        data-win-options="{ selectionMode: 'none' }"></div>
      <div class="ChatTextContainer">
        <div class="chatTextEchoContainer" aria-label="Chat text area"
          aria-live="assertive" aria-multiline="true" aria-readonly="true"
          aria-relevant="additions" role="log"
          tabindex="0"></div>
        <input class="chatTextInput" accesskey="i"
          aria-label="Chat input area" type="text"
          value="Type here...">
      </div>
    </div>
  </section>
</div>
```

That `div`, with the `chatTextEchoContainer` class, is updated at run time to contain child `div` elements for each text entry. For this reason it has the `aria-live` attribute, whose values are described as a “politeness level” in the W3C spec. The value of `assertive` says “communicate the change right away,” which is appropriate for chat but should be used carefully. The other value, `polite` (the default for `role="log"` elements), indicates a lower priority such that Narrator won’t interrupt the current task. The `aria-relevant="additions"` attribute is related to this, indicating what kind of changes are relevant to the live area. Its values are `additions`, `removals`, `text`, and `all`. With `additions`, if we happened to add an image to the chat window with an `alt` attribute, that would be communicated; if we set this to `text`, only text elements would be read.

The `aria-multiline` attribute indicates that the chat window is a multiline textbox such that the Enter key is taken as text input rather than as a button press that would submit a form (as with the single-line textbox). The `aria-readonly` attribute then indicates that this control cannot be edited, to distinguish it from those marked with `aria-disabled`.

If you play with the sample, you'll notice that when you tab to the chat window, Narrator reads the entire contents. When you enter a line of in the single line control, on the other hand, Narrator only reads the new element that's been added. This is due to a default value of `false` for the `aria-atomic` attribute (not present in the markup). When used on an `aria-live` element, this tells the screen reader to read only the changed node in that element. If you set `aria-atomic` to `true`, a change to any child element is considered a change to the whole element such that all the contents will be read. This can apply on multiple levels, mind you, so that if you add a child element that is atomic and add grandchild elements within it, only that atomic child element would be read if the parent element is not atomic.

As for the markup in `pages/table/table.html`, this gives us an example of `aria-describedby`. Here's the relevant section, omitting the table contents:

```
<div class="detail">
  <h2 id="title" role="heading" aria-level="2">Sample table</h2>
  <p id="subtitle" role="note">This table shows sample data.</p>
  <p class="generaltext">...</p>
  <table class="tabledetail" aria-describedby="subtitle"
    aria-labelledby="title" border="1">
    <!-- Contents omitted -->
  </table>
</div>
```

When you set the focus to the table in the running sample (you have to use the mouse for this unless you add a `tabindex` to the table), you'll initially hear "Sample table" according to the `aria-labelledby` attribute. Then press Win+Alt+F, and you'll hear "Item described by..." followed by the `aria-describedby` text. (And yes, go ahead and change it so that Narrator says some silly things. You know you want to!)

Note, finally, that it's essential that the *title* and *subtitle* elements also have some aria-related attributes, such as `role`. Otherwise `aria-labelledby` and `aria-describedby` won't work.

Handling Contrast Variations

Working with high contrast modes is primarily one of accommodating changes to the Windows color theme and making sure that you apply graphics that meet high contrast requirements. Technically speaking, high contrast is defined by the W3C as a minimum luminosity ratio of 4.5 to 1. A full explanation including how to measure this ratio can be found on <http://www.w3.org/TR/WCAG20-TECHS/G18.html>. A [Contrast Analyzer](#) (from the Paciello Group) is also available to check your images (some of mine in *Here My Am!* failed the test). Do note, however, that creating high contrast graphics isn't required for non-informational content such as logos and decorative graphics. At the same time, full-color graphics might look out of place in a high contrast mode, so be sure to evaluate your entire end-to-end user experience under such conditions.

An app handles high contrast through four means. The first is to use built-in controls (both HTML and WinJS) and let the system do the work! To see what happens, run a few of the controls samples, such as the HTML essential controls sample and [HTML essential controls sample](#) and the [HTML Rating](#)

[control sample](#), and switch between the different high contrast themes in the Personalization section of Control Panel.

Of course, an app will almost always have some layout of its own, such as `div` elements with custom color schemes and such defined in CSS. You'll want to make sure you have appropriate style rules for high contrast settings, for which we have the `-ms-high-contrast` media feature for media queries, similar to `-ms-view-state` as we saw in Chapter 6. This feature can have the values of `active` (to apply its rules to all high contrast themes), `black-on-white` (the white background theme), `white-on-black` (a black background theme), and `none`. Clearly, `none` is implied when you don't use `-ms-high-contrast` to group any rules; `active` is also implied if you use `-ms-high-contrast` without a value. We'll take a closer look at all this in the next section.

As with view states, you can use media query listeners and `matchMedia` to pick up contrast themes in code. This is useful for updating `canvas` elements, as we'll see shortly. There is also the `-ms-high-contrast-adjust` CSS style that indicates whether to allow the element's normal CSS properties to be overridden for high contrast. The default value, `auto`, allows this; the value of `none` will prevent this behavior. Again, we'll see more shortly.

Next, WinRT surfaces the current contrast settings through the `Windows.UI.ViewManagement.AccessibilitySettings` class. This has two properties: `highContrast`, a Boolean indicating if high contrast is on), and `highContrastTheme`, a string with the name of the high contrast color scheme. For the black on white theme this will be "High Contrast White"; for the other three themes in Control Panel > Personalization the strings will be "High Contrast #1", "High Contrast #2", and "High Contrast Black" (going from left to right). You can see these results through Scenario 2 of the [UI contrast and settings sample](#), where the code is very simple:

```
var accessibilitySettings = new Windows.UI.ViewManagement.AccessibilitySettings();
id("highContrast").innerHTML = accessibilitySettings.highContrast;
id("highContrastScheme").innerHTML = accessibilitySettings.highContrast ?
    accessibilitySettings.highContrastScheme : "undefined";
```

WinRT also provides detailed color information through the `Windows.UI.ViewManagement.UISettings.UIElementColor` method. (Note the odd casing on `UIElementColor`, an artifact of WinRT names projecting into JavaScript.) This returns a `Windows.UI.Color` object for an element identified with a `UIElementType`. Scenario 1 of the UI contrast and settings sample shows all these possibilities with a piece of instructive but otherwise uninspiring code that I won't duplicate here!

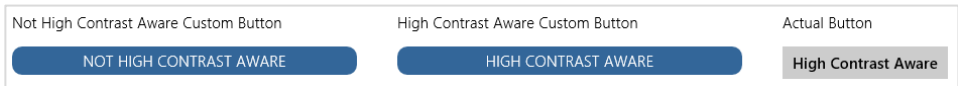
The `AccessibilitySettings` object also supports one event, `highcontrastchanged`, that lets you know when high contrast is turned on or off; its `EventArgs.target` is the updated `AccessibilitySettings` object. You can use this event to trigger any programmatic updates you need to make in your UI, such as redrawing a `canvas` with high contrast colors if you're not using a media query listener for that purpose.

Finally, with both raster and vector images, there are file naming conventions that you use in conjunction with the `.scale-100`, `.scale-140`, and `.scale-180` suffixes for pixel density. For contrast, the appropriate suffixes are `.contrast-standard`, `.contrast-high`, `.contrast-black` (black-on-white), and `.contrast-white` (white on black). We'll see this in action in the second section below, "High Contrast Resources," and see how to combine both the scaling and contrast suffixes in the third section, "Scale + Contrast = Resource Qualifiers."

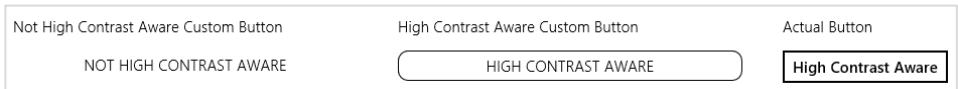
CSS Styling for High Contrast

The [CSS styling for high contrast mode sample](#) provides a valuable look at dealing with high contrast modes where media queries and image files are concerned. As you might expect, most of these features are demonstrated declaratively in CSS and the app's resources; only one scenario actually has any JavaScript code at all!

Scenario 1 shows the difference between elements that are and are not aware of contrast. With a normal color scheme in effect, its three buttons appear as follows, where the first two are `div` elements and the third a true `button`:



When high contrast is turned on (Left Shift + Alt + Print Screen is very handy to toggle the setting for this sample), they appear like so:



The first control, lacking contrast awareness, is still using white for its border, which of course disappears against a white background. The second button, on the other hand, has styles that use system-defined colors associated with a high contrast media query, so the button works well with any theme (`css/scenario1.css`):

```
@media (-ms-high-contrast) {
    .s1-hc {
        background-color: ButtonFace;
        color: ButtonText;
        border: 1px solid ButtonText;
    }
    /* ... */
}
```

Tip If you just stick with system colors entirely, both in CSS and in SVGs, then you won't need to use media queries or different SVG files at all, because those colors will be adjusted for high contrast modes automatically. See [User-defined system colors](#) for a reference. You can also use the `current-Color` value in SVGs for `fill`, `stroke`, `stop-color`, `flood-color`, and `lighting-color` properties to reflect contrast settings.

Scenario 2 shows similar effects with button elements that use SVGs for their background images. With normal settings, those buttons appear as follows:



With high contrast turned on, they appear like this:



All that's happening here is that we're using a media query to use a high contrast background image for the button when necessary:

```
.s2-button-hc-bg-svg {
  background-image: url(../button-not-aware.svg);
  background-size: 100% 100%;
  width: 200px;
  height: 200px;
}

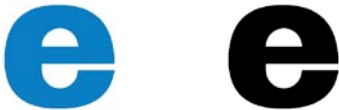
@media (-ms-high-contrast) {
  .s2-button-hc-bg-svg {
    background-image: url(../button.contrast-high.svg);
    background-repeat: no-repeat;
    background-size: cover;
  }
}
```

If you look in `button-not-aware.svg`, you'll see that its gradient colors have many different values; in `button.contrast-high.svg`, on the other hand, those colors are generally set to *black* or *ButtonFace*, the latter reflecting the system color setting as is appropriate. (It would probably be better, in fact, to replace *black* with *ButtonText*, to use a [system color](#) that will automatically adjust to contrast settings.)

What's going on with the first and second buttons? If you look in the CSS (`css/scenario2.css`), you'll see that the only difference is that the style class for the first button, `.s2-button`, lacks a rule within the high contrast media query, whereas the second, `.s2-button-hc`, has a rule there that just specifies the exact same background image. So what's the deal? What's happening is that because the first button lacks any applicable style rule within the media query, its styles are automatically overridden with high contrast values. As described in [Introduction to Web Accessibility](#), turning on high contrast overrides most color styles as well as `background-image`, in the latter case simply removing those images. This is why the first button shows up blank. The second button has a rule to define a `background-image` within the media query, so that image appears.

This brings us to the purpose of the `-ms-high-contrast-adjust` style. By default this is set to `auto`, allowing CSS properties to be overridden. If we set this to `none` within a style rule, we prevent those styles from being overridden or adjusted. Thus, if you add `-ms-high-contrast-adjust: none;` to the `.s2-button` rule in `css/scenario2.css`, you'll see that the first and second buttons behave exactly the same. You can see this change in the copy of the sample included with this chapter's companion content.

Moving now to Scenario 3, it normally draws the Internet Explorer logo on a `canvas` in color (below left), whereas in high contrast mode it draws the logo in black and white (below right):



In this case, high contrast is picked up in JavaScript (`js/scenario3.js`) using a media query listener; no CSS is involved (this code is simplified for clarity; the actual sample also detects high contrast on startup):

```
var fillStyleOuterColor = "rgb(9, 126, 196)";
var fillStyleInnerColor = "rgb(255, 255, 255)";

var mq1 = matchMedia("(-ms-high-contrast)");
mq1.addListener(updateColorValues);

function updateColorValues(listener) {
    if (listener.matches) {
        fillStyleOuterColor = "ButtonText";
        fillStyleInnerColor = "ButtonFace";
        draw();
    }
    else {
        fillStyleOuterColor = "rgb(9, 126, 196)";
        fillStyleInnerColor = "rgb(255, 255, 255)";
        draw();
    }
}
```

Note that the `AccessibilitySettings.onhighcontrastchanged` event could be used here instead of the media query listener.

Canvas a better choice? To this point in the Here My Am! app, I've been using images to provide messages in `img` elements when a photograph or the map isn't available. When considering the needs for contrast and localized variations of those images, it's easier to take localized string resources, as we'll work with later on, and just generate the images on the fly with a `canvas`. This eliminates the need for many different image files and make the app package smaller, while still fully addressing both accessibility and localization needs. The version of Here My Am! in this chapter now works this way.

High Contrast Resources

In the previous section with the [CSS styling for high contrast mode sample](#) we saw a bit of the filename conventions that the Windows resource loader uses for high contrast: `button.contrast-high.svg`, for example. Scenario 4 of that sample shows how this lookup can happen automatically. In the project there is a file named `button.svg` alongside one named `button.contrast-high.svg`, with an `img` element declared in `html/scenario4.html` as follows:

```

```

If the system is running with normal contrast, the resource loader resolves the URI here to `button.svg`. (The `../../` is because the scenario page is one level down in the HTML folder.) When high contrast is in effect, the resource loader instead looks for that same filename but with `.contrast-high` inserted before the extension.

Note If you're using custom app bar icons, as discussed in the "Custom Icons" section of Chapter 7, "Commanding UI," remember to include high contrast variants of your source images using this naming scheme.

If you like having more parallel filenames, you can also name the normal contrast file with `.contrast-standard`, as in `button.contrast-standard.svg`. If you do this in the sample project, leaving the HTML as is, you'll see no difference in the output. At the same time, because of behavior nuances with contrast handling, it's only recommended to use `.contrast-standard` if you also supply `.contrast-white` and `.contrast-black` variants.

As noted before, these variants are applied automatically for black-on-white (white background) and white-on-black (black background) themes, respectively. To see this, make a copy of `button.contrast-high.svg` and name it `button.contrast-white.svg`, and then make a second copy names `button.contrast-black.svg`. In that second copy, modify the gradient colors in the CDATA block by exchanging `black` with `ButtonFace`. When you then switch on a black background theme, you'll see a button that's white on black, as it should be.

All these changes can be found in the copy of the sample included with this chapter's companion content.

The one caveat with the `img` element in Scenario 4 is that it won't be updated when contrast is changed while the app is running, as happens with media queries in Scenarios 1–3. That is, the app host will not re-render the `img` element in response to a contrast switch. To change this behavior, we basically

have to trick the app host into thinking that the source URI has changed by appending some dummy URI parameters. We can do this inside [AccessibilitySettings.onhighcontrastchanged](#) with `eventArgs.target.highContrastScheme` providing a decent variable for the URI (see `js/scenario4.js` in the modified sample):

```
var page = WinJS.UI.Pages.define("/html/scenario4.html", {
  ready: function (element, options) {
    var accSet = new Windows.UI.ViewManagement.AccessibilitySettings();

    accSet.addEventListener("highcontrastchanged", function (e) {
      var image = document.getElementById("buttonImage");

      //Use the scheme name (sans whitespace) as the dummy URI parameter
      var params = e.target.highContrast ?
        "?" + e.target.highContrastScheme.replace(/\s*/g, "") : "";
      image.src = "../button.svg" + params;
    });
  }
});
```

One significant advantage to `highcontrastchanged` over media query listeners is that the latter will be fired very soon after the change happens, at which point the resource loader might not have picked up the change by the time you set the `img.src` attribute. This results in the wrong image being displayed. `highcontrastchanged` is fired much later, so the code above generally works. That said, my experiments along these lines (with the sample running in snap view and the desktop control panel in filled view) show that it's still not 100% reliable: changing contrasts is an expensive operation that triggers many events throughout the system, and there's no guarantee when the resource loader will get reset. For this reason you can consider just bypassing the whole matter and explicitly setting the `src` attribute to a known file with a specific name. The modified sample actually runs with code like this (commenting out the code above). Or you can just use media queries!

Scale + Contrast = Resource Qualifiers

Because the graphics we worked with in the previous section are SVGs, there is no need to supply separate files for different pixel densities. But what if we have raster graphics? How do we combine scaling and contrast? This will also come up when we look at localization in the next section, because we might also need to include language variants.

This brings us to the matter of resource *qualifiers*, a topic that's discussed in its fullest extent on [How to name resources using qualifiers](#). Qualifiers include scale and contrast as we've seen, along with language, layout direction, home region, and a few other obscure variants.

To combine qualifiers within a single filename, append them together with underscores. The general form is `filename.qualifiername-value_qualifiername-value.ext`. So, a graphic named `logo.png` can have variants like `logo.contrast-high_scale-180.png` and `logo.scale-100_contrast-white.png` (the order of qualifiers doesn't matter). Clearly, with the full set of three or four scales (accounting for the few scale-80 cases) and four possible contrasts, you might have as many as 16 distinct graphics files for that

one resource. For a few examples of this, load the [Application resources and localization sample](#) into Visual Studio and look in the *images* folder. (Although the sample shows only contrast+100% scale examples, be sure to provide at 140% and 180% scales as well in your own app; Here My Am! for this chapter does so with its splash screen, tile, and other logo graphics.)

As we get into the topic of world readiness, we'll find that localized image resources will require a set of scale and contrast variants for each language. As you can guess, the file naming conventions here could get really messy as the file count increases! Fortunately, the resource loader also allows qualifiers in folder names, so localized resources are typically placed within language-specific folders. We'll see more of this later on in the section entitled "Part 2: Structuring Resources for the Default Language." We'll also avoid this complexity entirely in Here My Am! by using a [canvas](#) instead of discrete images for those graphics that contain text messages (the logos aren't localized).

High Contrast Tile and Toast Images

Like any other images in your app, tile images in your manifest, images sent to the tile through updates, and images used in toast notifications all respect contrast settings. (Badges are not an issue as they are already monochromatic and adapt automatically.) In the manifest, naming images with resource qualifiers work for both scale and contrast, as well as language as we'll see later.

XML payloads for tiles and toasts can refer to local images using [ms-appx:///](#) URIs, and the resource loader will look for the appropriately qualified file. This does not apply to [ms-appdata:///](#) URIs, however, so if you're working with downloaded or dynamically generated images, you'll need to identify a specific file yourself.

For XML payloads that refer to remote images, setting the [addImageQuery](#) option in the payload to [true](#), as discussed in the "Using Local and Web Images" section of Chapter 13, will append query strings to the remote URIs that indicate scale, contrast, and language:

```
?ms-scale=<scale>&ms-contrast=<contrast>&ms-lang=<language>
```

These details are described on [Globalization and accessibility for tile and toast notifications](#), along with how to localize strings in the XML payload. We'll see these details for ourselves later on.

World Readiness and Localization

Over the years I've heard a number of words used to describe the process of making an app ready for different regional markets, and I imagine you have too: localization, localizability, internationalization, globalization, and world readiness. To be honest, the differences between these terms have confused me for some time, but I finally found a good explanation in an older book for desktop apps called *Developing International Software* by Dr. International (Microsoft Press, 2003). The same ideas are also expressed on [Globalization Step-by-Step](#). Let me begin this section then by offering a simple summary of that view.

The goal with Windows Store apps is to make them available in many markets around the world, as provided for so conveniently by the Store itself. To do this, an app needs to be written such that it can adapt itself to just about any language and culture it might encounter. In some situations you may need to produce specific versions of the app, but hopefully you can have one app with localized resources that works for most markets. Truly, the days of monolingual apps are over.

To reach this goal you must first make your app *world-ready*. World readiness means that even though the app initially supports only one language and culture (most likely your own), it doesn't actually make any assumptions about those specifics anywhere within its HTML, CSS, and JavaScript (and any WinRT components). That is, the core app is language-, culture-, and market-neutral, taking all these factors into account:

- Each and every string that might be shown in the app's user interface, including element attributes like `aria-label` and `img.alt`, has been separated out into a resource file such that different resources can be loaded for different languages. (Using Unicode text is pretty much a given nowadays, so displaying text in many languages isn't an issue, but be sure to keep this in mind if you're migrating older software or using web services that might work otherwise).
- Each and every localized image (those that contain text or culture-specific content) has been organized into language-specific folders, appropriately named so that the resource loader can find them automatically.
- Any formatting and manipulation of dates, times, and currencies use APIs that automatically apply regional settings.
- Any sorting or collation of data takes the user's language into account, using APIs for this purpose.
- No assumptions are made about how strings are concatenated; format strings with appropriate placeholders are instead included in language-specific resource strings so that they can be localized.
- The web services an app uses might vary from location to location, because of the need for local information or regional legal requirements.
- Text might be laid out left to right or right to left. Vertical is also possible but might be implemented in a separate version of an app because of its unique layout needs.
- Text input just works for all languages, whether from a keyboard or an Input Method Editor (IME), which implies that you should avoid hard-coding font names that don't have full Unicode support. It's good to stick with the typography in the WinJS stylesheets—they have built-in support for at least 109 languages.
- The user might switch languages at run time, and the app responds accordingly.

A world-ready app, in short, is both *globalized*—using APIs that isolate regional specifics—and is readily *localizable* such that adding support for another language requires no code changes, just the

addition of new string and image resources. This is mostly a matter of how you structure those resources and how you reference them within the app's markup and source code.

The process of *localization*, then, is one of generating or acquiring those language- and culture-specific resources, for which some very helpful tools are available to streamline translation work.

In the following sections we'll look first at matters of globalization, explore how to structure resources to be localizable, and then see how to go about obtaining localized resources. After that, we'll be ready to look at the last step in the long journey of an app's creation: uploading to the Store.

Globalization

Besides language, the things that vary around the world are the representation of dates and times (including calendars); the representation of numbers, measures (units), phone numbers, and addresses; currencies; paper sizes (already discussed in Chapter 15, "Devices and Printing"); how text is sorted (collation); the direction of text; and the fonts used for text along with the input method.

To globalize an app means to make no assumptions about how any of this is accomplished, instead using the WinRT APIs that will do the right thing according to the current user's settings. Working with those APIs is what globalization is mostly about.

Beyond using the APIs, look at the content of the app itself, checking for words, phrases, or expressions that might be very hard to translate (or potentially politically offensive), especially colloquialisms, vernacular, slang, metaphors, jargon, and the like. Use images that travel well, and aren't likely to be misinterpreted elsewhere in the world (imagine wearing a T-shirt with such imagery in a country where you intend to market the app!). And exercise caution with maps because there is disagreement among different nations about where, exactly, their borders should be drawn. Be sure also to refer to "country/region" rather than just "country," because disputed territories might not be recognized specifically as a country.

Also be aware of your regional export laws regarding encryption algorithms, because you might not be allowed to make the app available in certain markets. See [Staying within export restrictions on cryptography](#). In addition, if you're writing a game, be mindful of regional game rating requirements that might create more work for you than it's worth. See [Windows game publishing requirements](#).

If you use web services, make sure you also use services that are appropriate to the user's locale. This might be required by law in some parts of the world (especially for financial transactions and maps) and often ensures that the user gets regionally relevant information from that service, unless they've specifically configured the app otherwise. You also want to be able to communicate the user's locale and language to those services so that they can return content that's already localized. It's also helpful for the app's overall performance to use servers that are relatively close to the user rather than on the other side of the world!

The first step in any of this, however, is to know where your app is actually running and the user's language and cultural preferences, so let's see how that is accomplished.

User Language and Other Settings

When a user first acquires a Windows 8 device or installs Windows 8 on a machine, it will likely be configured for their country of residence. However, many users speak multiple languages irrespective of where they live and might want to work with Windows in a particular language that has nothing to do with their location. For this reason, you always want to think about the user's preferences separately from the actual location of the device, applying the user's preferences to how your app displays information but using the physical location to control the services you use and other more functional aspects.

Languages and other preferences are configured through Control Panel > Clock, Language, and Region. Here you can add languages and select your primary one (see Figure 17-8), change input methods, specifically set your location (a country or territory), and set date, time, number, and currency formats (see Figure 17-9).

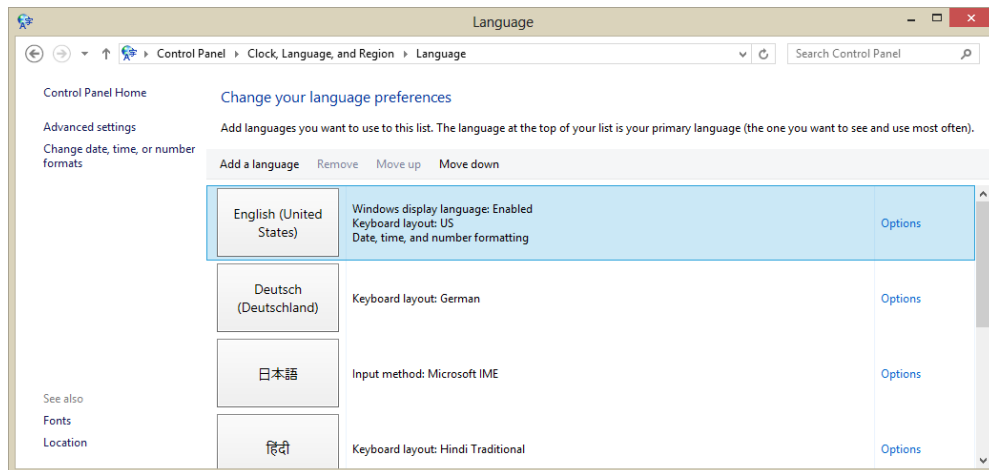


FIGURE 17-8 Managing and selecting a language in Control Panel.

It's a good thing there are globalization APIs, because dealing with all the variations here would be quite a chore otherwise! (Note that changes to the formats in Figure 17-9 will affect only those Windows Store apps that are running in the language you're configuring; each set of custom formats is particular to a language.)

The basic details of the user's settings are available through the [Windows.System.UserProfile.-GlobalizationPreferences](#) object and the classes in the [Windows.Globalization](#) namespace. [GlobalizationPreferences](#) just provides a handful of properties. Four of these, [calendars](#), [clocks](#), [currencies](#), and [languages](#) are each an array of strings (an [IVectorView](#) to be precise) with the user's preferred settings in order of preference. In the case of [languages](#), it contains a list of [BCP-47 language tags](#).

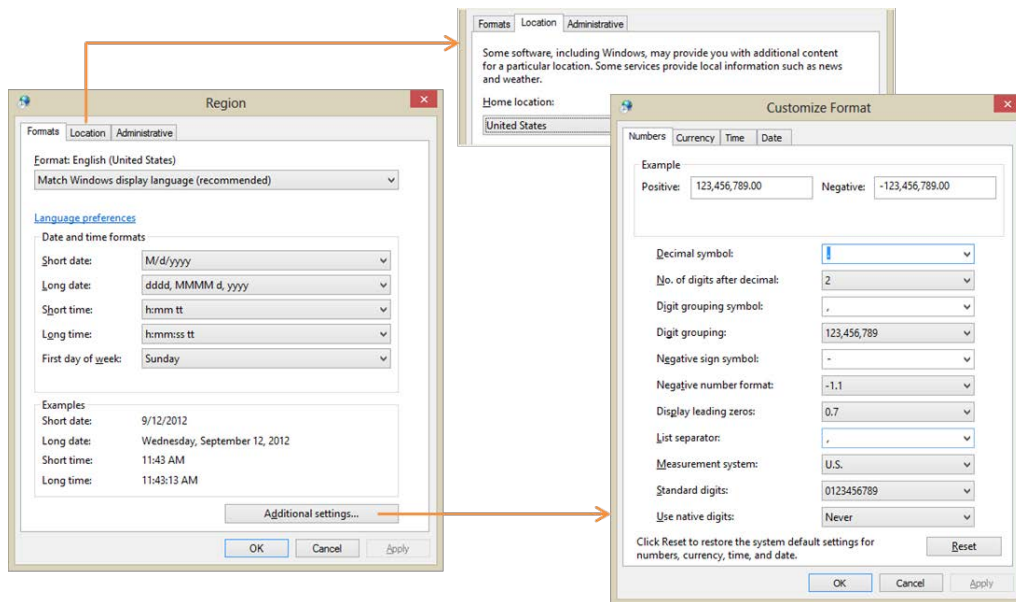


FIGURE 17-9 Control panel dialogs for formatting and region.

It also contains a string property called `homeGeographicRegion`, which is the abbreviation for the selected value in Control Panel's Location tab of Figure 17-9, and a property called `weekStartsOn`, which is a `DayOfWeek` value. Scenario 1 of the [Globalization preferences sample](#) will retrieve and display these values, except that you'll want to add a line for `currencies`, which is missing from the sample. Having made that change and added a number of languages to my system, I see this output:

```
Languages: en-US,de-DE,ja,hi,ar-AE,ru
Home Region: US
Calendar System: GregorianCalendar
Clock: 12HourClock
Currency: USD
First Day of the Week: 0
```

Generally speaking, these values are exactly what you'll typically need to communicate to a web service if it will be providing localized data to the app. However, the user's language preference is best obtained in a slightly different manner, as we'll see shortly.

Oftentimes you'll need more detail for all of these settings, for which we can turn to the classes in `Windows.Globalization`. Some of these are static classes that are just there in the API to provide you with all the string identifiers that you would use to make comparisons in code without writing out the strings explicitly. `ClockIdentifiers`, for instance, just contains two string properties, `twelveHour` and `twentyFourHour`, whose values match those returned from `Globalization-Preferences.clocks`. Similarly, `CalendarIdentifiers` contains string properties for `gregorian`, `hebrew`, `hijri`, `japanese`, `julian`, `korean`, `taiwan`, `thai`, and `umAlQura`. So, if you wanted to compare the user's preferred calendar to a specific one, you'd write code like this:

```
var userCalendar = Windows.System.UserProfile.GlobalizationPreferences.calendars[0];
if (userCalendar == Windows.Globalization.CalendarIdentifiers.julian) {
    // ...
}
```

This way you're fully honoring the key principle of globalization by not making any assumptions about what those calendar strings are.

The other globalization classes are somewhat richer in scope and function. The [Language](#) class, which you typically instantiate with a specific BCP-47 tag, provides details like the [displayName](#), [nativeName](#), [languageTag](#), and [script](#). Scenario 2 of the aforementioned sample demonstrates this. The Language class also has two static members. One is the [isWellFormed](#) method that will tell you if a string contains a valid BCP-47 tag. The other is the [currentInputMethodLanguageTag](#) property that contains the BCP-47 tag for the user's preferred input, which can be customized in Control Panel to be something other than the language's default. (See the Options links on the right side of Figure 17-8; this is also demonstrated in Scenario 4 of the sample.)

Then we have the [GeographicRegion](#) class, which, if instantiated with no arguments, provides details on the user's home region. You can also instantiate it with a specific location string. Either way, it then provides you with a [displayName](#), a [nativeName](#), a variety of code formats for the region ([code](#), [codeThreeDigit](#), [codeThreeLetter](#), and [codeTwoLetter](#)), and [currenciesInUse](#) (an array of ISO 4217 three-letter codes). Scenario 3 of the sample shows these values for your configuration, such as:

```
User's Preferred Geographic Region
Display Name: United States
Native Name: United States
Currencies in use: USD
Codes: US, USA, 840
```

The [ApplicationLanguages](#) class, for its part, contains just a few things. [manifestLanguages](#) is an array of languages as defined by the app's manifest; you'll set these when you localize an app. [languages](#) contains a combination of the [GlobalizationPreferences.languages](#) array and those from [manifestLanguages](#). The first item in this list is the best value to use for the user's preferred language in your app, so this will be the one to send to web services for localization purposes.

Lastly, there's [primaryLanguageOverride](#), a property (BCP-47 tag) for apps that allow the user to select an app-specific language preference (and add it to the mix of [languages](#)). Setting this tells the system what language the app is using so that it can configure its own UI, and the setting is persistent across sessions. As it's a relatively expensive operation, avoid using [primaryLanguageOverride](#) for transient purposes, such as rendering a few elements in a different language. For that, create a new language context and use that explicitly; see [Windows.ApplicationModel.Resources.Code.-ResourceContext](#) and scenario 13 of the [Application resources and localization sample](#).

The last class, [Calendar](#), is quite extensive and contains too many members to list here, many of which work with formatting as well performing calendar math. Before stepping into that arena, however, let's look more broadly at the question of formatting data.

Formatting Culture-Specific Data and Calendar Math

If you clicked around within Control Panel's formatting and region dialogs (refer back to Figure 17-9), you'll find that the possible permutations for formatting something as simple as a number is quite mind boggling, let alone dates, times, and currencies!

Fortunately, "formatter" classes in WinRT take care of all the details such that you can take a value from `new Date()`, for example, and get back a string that completely reflects the user's preferences. The APIs also provide parsing services that work in the opposite direction.

In `Windows.Globalization.NumberFormatting` we have [CurrencyFormatter](#), [DecimalFormatter](#), [PercentFormatter](#), and [Per milleFormatter](#), which you should always use these when converting data values into UI display strings. All of these classes are demonstrated in the [Number formatting and parsing sample](#), where the basic process is to instantiate the formatter with or without specific codes or languages, set any necessary properties for the formatter (such as the number of digits and using separators), and then call its `format` method to obtain a string or, alternately, one of its `parse*` methods to turn a string into a number.

For example, to format a currency value, instantiate a `CurrencyFormatter` with a currency identifier (or a currency identifier plus a language list and a geographic region), set up any options, and then call `format` (`js/CurrencyFormatting.js`):

```
var userCurrency = Windows.System.UserProfile.GlobalizationPreferences.currencies;
var wholeNumber = 12345;
var fractionalNumber = 12345.67;

// Apply user defaults
var userCurrencyFormat =
    new Windows.Globalization.NumberFormatting.CurrencyFormatter(userCurrency);
var currencyDefault = userCurrencyFormat.format(fractionalNumber);

// Apply a specific currency
var currencyFormatUSD = new Windows.Globalization.NumberFormatting.CurrencyFormatter("USD");
var currencyUSD = currencyFormatUSD.format(fractionalNumber);

// Apply a specific currency, language, and region (France, then Ireland)
var currencyFormatEuroFR =
    new Windows.Globalization.NumberFormatting.CurrencyFormatter("EUR",
        ["fr-FR"], "ZZ");
var currencyEuroFR = currencyFormatEuroFR.format(fractionalNumber);

var currencyFormatEuroIE =
    new Windows.Globalization.NumberFormatting.CurrencyFormatter("EUR",
        ["gd-IE"], "IE");
var currencyEuroIE = currencyFormatEuroIE.format(fractionalNumber);

// Include fractions with a whole number
var currencyFormatUSD1 =
    new Windows.Globalization.NumberFormatting.CurrencyFormatter("USD");
currencyFormatUSD1.fractionDigits = 2;
var currencyUSD1 = currencyFormatUSD1.format(wholeNumber);
```

```
// Group integers
var currencyFormatUSD2 =
    new Windows.Globalization.NumberFormatting.CurrencyFormatter("USD");
currencyFormatUSD2.isGrouped = 1;
var currencyUSD2 = currencyFormatUSD2.format(fractionalNumber);
```

The output of this code is as follows:

```
Fixed number (12345.67)
With user's default currency: $12345.67
Formatted US Dollar: $12345.67
Formatted Euro (fr-FR defaults): 12345,67 €
Formatted Euro (gd-IE defaults): €12345.67
Formatted US Dollar (with fractional digits): $12345.00
Formatted US Dollar (with grouping separators): $12,345.67
```

The other number formatters all work like this, so I'll leave it to you to check out the details in the documentation and the sample.

To format dates and time, we can turn to the [Windows.Globalization.DateTimeFormatting](#) namespace where we find the [DateTimeFormatter](#) class along with many enumerations for the different ways to format seconds, minutes, hours, days, months, and years. To use the API, you instantiate a formatter object specifying the desired formats and applicable languages. (There are no less than eight separate constructors here!) You then set options like the [clock](#), [geographicRegion](#), and so forth and call its [format](#) method with the [Date](#) value you need to format. You can even apply custom formats if desired. Many such variations are demonstrated in the [Date and time formatting sample](#); I trust a simple snippet of its code will suffice here (from Scenario 2, `js/stringtemplate.js`);

```
var mydatefmt1 = new Windows.Globalization.DateTimeFormatting.DateTimeFormatter(
    "month day");
var mytimefmt1 = new Windows.Globalization.DateTimeFormatting.DateTimeFormatter(
    "hour minute ");
var dateToFormat = new Date();
var mydate1 = mydatefmt1.format(dateToFormat);
var mytime1 = mytimefmt1.format(dateToFormat);
```

The other bit of code from the SDK that's relevant here is the [Calendar details and math sample](#). As mentioned earlier in this chapter when I described expiration times for app trials and in-app purchases, a world-ready app must not make assumptions about how time periods are computed or compared because this can vary depending on regional calendars. This is why the extensive [Windows.-Globalization.Calendar](#) class contains ten distinct [add*](#) methods that range from [addNanoseconds](#) to [addEras](#), along with its [compare](#) and [compareDateTime](#) methods (and a bunch to get all the little bits of calendar-related text). In other words, drill it into your mind now to never, ever use arithmetic operators on date and time values because they won't work properly in every locale. Even in the United States you'll end up getting wrong answers at times because you won't be taking things like daylight savings time into account, where the number of hours in two days of every year will not actually be 24!

Sorting and Grouping

Just as a world-ready app cannot make assumptions about comparing date and time values, it cannot make assumptions about how strings are sorted. Simply said, every language has its own way of sorting that doesn't necessarily have anything to do with the values of the character codes involved. The bottom line here is that you should never sort by such values; always use a language-aware API instead.

For Windows Store apps written in HTML and JavaScript, you can use the [localeCompare](#) method that's already built into strings (even for individual characters). This performs the comparison based on the user's current language. You can also use a string's [toLocaleLowerCase](#) and [toLocaleUpperCase](#) methods. In Chapter 5, "Collections and Collection Controls," specifically in the section "Quickstart #2b: The ListView Grouping Sample," we also saw how to use the [Windows.Globalization.Collation.CharacterGroupings](#) API to create proper groupings by the first character of item titles. You can compare the original SDK sample and the modified sample in Chapter 5's companion content to see how such code was globalized.

Fonts and Text Layout

Thanks to Unicode and the ability of HTML directly handle text in different languages, there's little you need to do to make such text appear properly within your layout. For example, if you look at the [Language font mapping sample](#), pages like `html/scenario2.html` that contain this markup:

```
<div id="scenario2Document">
  <h2 lang="hi" id="scenario2Heading" contenteditable="true">
    है।अभी पत्रिका लाभान्वित</h2>
  <p lang="hi" contenteditable="true">
    है।अभी पत्रिका लाभान्वित सना समजते संभव ध्वनि विभाजन वैश्विक बनाति संभव विकसित
    विचरविमर्श प्रोत्साहित जिम्मे वर्णित प्रेरना सचना जाता भाषाओ लिये दिनांक भेदनक्षमता
    सचनाचलचित्र डाले। लिए। मशिकल विभाजनक्षमता मुक्त दस्तावेज विचारशिलता विचरविमर्श
    उसके नवंबर रचना उद्योग वातावरण पहोचाना समजते तकनिकल अंग्रेजी बनाए सभिसमज जानकारी
    संदेश अधिक दुनिया अनुवाद सकती मुख्य रचना समजते उपलब्ध सभीकुछ देखने</p>
</div>
```

just show up like you expect they would (and if you read Hindi, you'll see that this is just gibberish):

है।अभी पत्रिका लाभान्वित

है।अभी पत्रिका लाभान्वित सुना समजते संभव ध्वनि विभाजन वैश्विक बनाति संभव विकसित विचरविमर्श प्रोत्साहित जिम्मे वर्णित प्रेरना सचना जाता भाषाओ लिये दिनांक भेदनक्षमता सचनाचलचित्र डाले। लिए। मशिकल विभाजनक्षमता मुक्त दस्तावेज विचारशिलता विचरविमर्श उसके नवंबर रचना उद्योग वातावरण पहोचाना समजते तकनिकल अंग्रेजी बनाए सभिसमज जानकारी संदेश अधिक दुनिया अनुवाद सकती मुख्य रचना समजते उपलब्ध सभीकुछ देखने

What this particular sample is actually meant to demonstrate is the [Windows.Globalization.-Font.LanguageFontGroup](#) object, which provides specific font recommendations for different parts of the UI. Once created using a specific BCP-47 tag, the object contains a number of properties, each of type [LanguageFont](#), such as [uITextFont](#) and [uIHeadingFont](#) (notice the odd casing again). Each [LanguageFont](#) object then contains properties of [fontFamily](#), [fontStretch](#), [fontStyle](#), [fontWeight](#), and [scaleFactor](#). Through a couple of helper functions in `js/langfont.js`, which are deceptively added to the

WinJS.UI namespace without being part of WinJS itself, these recommendations are applied to elements in the DOM simply by setting the appropriate styles for those elements.

Be clear that these font recommendations are really refinements and not necessary for basic function. As Scenario 4 of the sample demonstrates, a basic English font (with Unicode characters, of course) applied to mixed English/Japanese text will still render the Japanese but perhaps not optimally. Applying the recommended font will make that refinement.

The other aspect to working with different fonts and languages is how these affect your overall layout, something we didn't go into in Chapter 6, "Layout." This is discussed in the documentation on [How to adjust layout for RTL languages and localize fonts](#), but let me summarize that material and add a bit more.

First of all, a world-ready app leaves extra space for various bits of content like headings and labels because the words and phrases will be longer in some languages and shorter in others. The general guidelines are to leave at least 30% more room over what's needed in English for typical strings and as much as 300% for really short strings or single words. As a simple example, the English word "wrench" translates into German as "Schraubenschlüssel"; the word "click" (if I'm to trust Bing Translator), translates into Greek as "Κάντε κλικ στο κουμπί." You may need to enable word wrapping in some cases.

For all such purposes you can and should use the `:lang()/:-ms-lang()` pseudo-class selector in CSS to adjust styles like `width` as needed for specific languages. Just be sure to test your app with those languages, or test thoroughly with the *pseudo-language* (see "Testing with the Pseudo Language" later on).

Secondly, different languages flow text in directions other than the left to right (then top to bottom), like English and many Indo-European languages. Arabic and Hebrew, for instance, read right to left (RTL) instead of left to right; a few will flow top to bottom first, then right to left.

When making your app world-ready for RTL languages (considering that such markets are significant), you'll want to support what is called *mirroring* in your layout. It really means reversing your layout, including images, the direction of the back button, the direction of animations, panning directions, and so forth.

Fortunately, HTML and CSS layout automatically accommodate this, and the WinJS stylesheets, `ui-light.css` and `ui-dark.css`, set the CSS `direction` style appropriately as follows (something you should use on the element level for RTL languages rather than `align`):

```
html:-ms-lang(ar, dv, fa, he, ku-Arab, pa-Arab, prs, ps, sd-Arab, syr, ug, ur, qps-plocm) {  
    direction: rtl;  
}
```

In fact, look around in the WinJS stylesheets and you'll find many adjustments made for RTL languages with `:-ms-lang`, specifically with margins and padding. So by using HTML, CSS, and WinJS—including built-in controls—much of the mirroring is taken care of automatically; Here My Am!, for instance, just works in Hebrew.

With images, you can reverse them when needed by applying a `transform: scaleX(-1)` style to the necessary elements. If, however, you have images that really need to be replaced (as when some parts would be mirrored by other parts would not), you can use `layoutdir-RTL` in the image filename in the same way we've seen for pixel densities and contrast. In fact, there are many qualifiers for use with resources that are described on [How to name resources using qualifiers](#), something we'll be looking at more closely in the next section.

Sometimes you'll need to reverse a certain portion of text, as when mixing languages in the same paragraph. For this you can apply the `unicode-bidi` style in conjunction with `direction`. (Do note that numbers are generally direction-neutral so that they take on the directionality of their containing element, so you might need to set direction separately.) Along similar lines, you can also use the `-ms-writing-mode` style to flow text in just about any other direction, something you might use for an app that presents classical Chinese, Japanese, or Korean poetry.

Preparing for Localization

Once your app has been made world-ready such that it can handle just about any language and regional settings you want to throw at it, the next step is to make sure that language-specific resources in the app are cleanly separated from your HTML, CSS, and JavaScript and placed in your resources where the Windows resource loader (also referred to as the Resource Management System) can find them.

Before going further, there's an excellent topic in the documentation on this subject, [How to prepare for localization](#), which provides suggestions for translation and other details. It's not productive to repeat all of that here, of course, so I want instead to break that guidance down into a couple of steps that you can apply to an app and its default language before adding support for additional languages.

Note The resource loader supports *sparse localization* for dealing with slight variations between similar. It means that with languages like American English (en-US) and British English (en-GB), most of the app's resources can be assigned to en-US with en-GB resources for only those bits that vary, like "color" vs. "colour" and "favorite" vs. "favourite," or vice-versa. Because each resource is resolved individually according to the user's preferences, an app running in an en-GB context will find those specific bits first, if they exist, otherwise the loader will look in the en-US resources. There is also support for dealing with specific language exceptions through the use of resources marked with the undertermined tag `und`. See [How to manage language and region](#), step 4 (toward the end) for details along with [Language Matching](#).

Part 1: Separating String Resources

The first step in preparing for localization is to move language- or region-specific strings from source files into a string resource file and inserting references to that file where necessary. In the next section (Part 2), we'll then set up the folder structure for this file and image resources that will then accommodate localized versions.

To create your first string resource file, right-click your project in Visual Studio's solution explorer, select Add > New Item, and then select Resources File (.resjson). Although you can change the filename, just leave it set to the default resources.resjson for now. Press Add, and the file will be created in your project root, where we'll also leave it until Part 2.

Omitting a comment at the top, the contents of this file appear as follows:

```
{
  "greeting"          : "Hello World!",
  "_greeting.comment" : "This is a comment to the localizer"
}
```

As you can see, the file is just plain JSON where each property has a string identifier and a string value; any resjson file can have as many properties as you want.

Clearly, too, there is a relationship between the two entries above. The first entry of the form `<identifier> : <value>`, is a real string resource that maps an valid JSON identifier (no whitespace) to a string value. This is what the resource loader will use to replace references to the identifier with the string value.

Any entry of that begins with an underscore, such as the conventional `<_identifier.comment> : <value>` is ignored by the resource loader. Such entries provide notes for a translator so that they can fully understand how the string is used and specific parts that shouldn't be translated. A second optional entry, `<_identifier.source> : <value>`, provides the original string in the default language, which is very helpful for reference.

If you want to see some more extensive resjson files, open the [Application resources and localization sample](#) and look in the *strings* folder under a particular language. In the `ja/resources.resjson` file, for example, you'll see the string resources along with both comment and source entries:

```
{
  "displayName"          : "アプリケーション リソース JS SDK サンプル",
  "_displayName.source"   : "Application Resources JS SDK Sample",
  "_displayName.comment" : "Don't change 'SDK'",

  "description"          : "アプリケーション リソース JS SDK サンプル",
  "_description.source"   : "Application Resources JS SDK Sample",
  "_description.comment" : "Don't change 'SDK'",
}
```

Turning back to your own app with the new resources.resjson file in hand, we're now ready to go on a search and replace mission throughout the app project, looking for localizable strings, extracting them into the resource file, and replacing them in the source files with an appropriate reference. The three primary places we need to look at are your HTML files, JavaScript files, and the app manifest. To demonstrate, I'll show what I did with the Here My Am! example that we've been working with in this book (to which I've added a resources.resjson file).

Note CSS files can contain string literals in the `content` and `quotes` styles; however, resource lookup from CSS is not supported for Windows Store apps in Windows 8. Localization must be done in CSS with the `:lang` and `:-ms-lang` pseudo-selectors.

JavaScript: Let's start with JavaScript, where you need to scrub your code for string literals, including any you are drawing to a `canvas`. In Here My Am! I found only a couple localizable strings, namely a folder name used in the Pictures library and the title and description used when formatting text for the Share contract and our live tiles (`pages/home/home.js`):

```
var folderName = "HereMyAm";
data.properties.title = "Here My Am!";
return "At latitude " + lat + ", longitude " + long;
return "At (" + lat + ", " + long + ")";
```

and the Settings commands in `js/default.js`:

```
app.onsettings = function (e) {
    e.detail.applicationcommands =
    {
        "about": { title: "About", href: "/html/about.html" },
        "help": { title: "Help", href: "/html/help.html" },
        "privacy": { title: "Privacy Statement", href: "/html/privacy.html" }
    };
    WinJS.UI.SettingsFlyout.populateSettings(e);
};
```

Note that in `home.js` English strings are used for exceptions, but because they're only for debugging purposes they don't need to be localized.

Extracting the other strings into `resources.resjson`, then, that file appears as follows where I'm using regular comments to identify where the strings are used. Notice that I'm now using a format string to create a descriptive location for Share and tiles instead of hard-coding its construction (see the `formatLocation` function in `js/home.js` for how these are used):

```
{
    // pages/home/home.js
    "foldername" : "HereMyAm",
    "share_title" : "Here My Am!",
    "location_formatShort" : "At (%s, %s)",
    "_location_formatShort_comment" : "Used to format a short location as in 'At (120, 45)',",
    "location_formatLong" : "At latitude %s, longitude %s",
    "_location_formatLong_comment" :
        "Used to format a long location, 'At latitude 120, longitude 45'",

    // default.js
    // Settings panel commands
    "about_command" : "About",
    "help_command" : "Help",
    "privacy_command" : "Privacy Statement",
}
```

I highly recommend that you organize your entries by source file like this, and you can also use multiple resource files if you like, as explained in the next section. Also be careful about how you reuse the same string that occurs in multiple places. If it's for the same kind of UI with the same intent, that's fine, but if the usage context is different it's better to duplicate the string as they might translate differently in other languages. In the resources above, notice how I also included a comment for `location_formatShort` because the word "At" by itself probably needs more context to translate properly.

With the strings separated as resources, we can now use the resource loader to obtain those strings at run time. This can be done in two ways. First is with the WinRT APIs directly, namely

[Windows.ApplicationModel.Resources.ResourceLoader.getString](#):

```
var loader = new Windows.ApplicationModel.Resources.ResourceLoader();
var text = loader.getString('location_formatShort');
```

or, more simply, with the [WinJS.Resources.getString](#) wrapper:

```
var text = WinJS.Resources.getString('location_formatShort').value;
```

that also happens to work in the web context where WinRT isn't defined (see Scenario 12 of the [Application resources and localization sample](#).) Note that `getString` returns an object that includes a `value` property with the string along with `lang` and an `empty` flag indicating if the resource wasn't found.

The WinJS method, being one line, is clearly helpful in cases like our settings commands because we can call it inline. Thus, in our code we just replace the string literals with the WinJS call, such as the following in `pages/home/home.js`:

```
data.properties.title = WinJS.Resources.getString('about_command').value;
```

and the following inside the object for the Settings commands:

```
"about": { title: WinJS.Resources.getString('about_command').value,
  href: "/html/about.html" },
```

Note that WinJS, being optimized for common scenarios, supports loading strings in the user's default language only. The WinRT `ResourceLoader` class, on the other hand, is much more flexible and can load a string for any specific language. You'll need to use that API when your requirements exceed what WinJS provides.

And that's really it for JavaScript. If you've made these changes to your app, now is a good time to use the Build > Build Solution command in Visual Studio. This will compile the `resources.resjson` file into a more efficient binary format called `resources.pri`, ignoring entries that begin with an underscore. Doing an occasional build (without string the app) is a good practice when working with resources so that you can clean up any problems in your files, such as duplicate entries or syntax errors. Then you can run the app to see the resource loader in action—mostly by seeing no difference from the app as it was before! Be sure to test all the code paths that were affected, however, to ensure that all the strings are being loaded properly.

Manifest: Let's turn now to the manifest, where the story is even simpler because there's no code involved. The textual pieces here that might need localization are

- The display name, description, and short name (for tiles) on the Application UI tab.
- Any text descriptions for specific entries on the Declarations tab.
- The package display name on the Packaging tab. (You would change the package name only if you were going to submit the app to the Store under a different name in certain markets, in which case it's a different app package entirely.)
- Possibly some URIs on the Content URIs tab.

While we're looking at the manifest, note the Default Language setting on the Application UI tab. This is what defines the app's default or *fallback* language if the user runs the app with a language that isn't provided for in your resources. We'll also come back to images in the manifest in the next section.

Looking at all the strings in the manifest, extract these to your resources.resjson file, giving them appropriate identifiers. Again, if you have some strings in the manifest that match those elsewhere in the app, carefully evaluate them to determine if they can all use the same resource. When in doubt, keep them separate as the overhead is quite small. In the case of Here My Am!, the app's display name and the string used for the header on the main page are the same and have the same usage, so they can refer to the same resource.

To refer to those resources in the manifest, then, use *ms-resource:<identifier>*. For example, I moved the Application UI > Display Name value into the resource file and called it *app_title*, so in that field of the manifest editor I simply write *ms-resource:app_title*. I did the same for the description and the package display name.

Once you've made these changes, run the app and make sure text on your tile, if you're using it, shows up properly. You might temporarily set the Application UI > Show Name to "All Logos" as a check, but be sure to change it back before you forget!

Sidebar: Localized Strings in Tile and Toast XML Payloads

As described on [Globalization and accessibility for tile and toast notifications](#), XML payloads for tile and toast notifications use the *ms-resource:* syntax to identify string resources in *text* elements. This will trigger the resource loader's lookup mechanism when the tile is rendered, and this works regardless of whether the notification was issued locally, obtained from a web service, or received as a push notification. The web services just need to be sure they use the app's particular resource identifiers.

A web service can also issue localized tile updates directly. In this case, an app will generally append query strings to the service URI to communicate the desired language, updating those parameters as necessary when the language changes (see "Localization Wrap-Up" for details). An app can also combine this with using regional web services that help localize the update content.

HTML: The final place we need to look for strings is our HTML, which I've saved for last because it's the most involved. In HTML, really scrub, scrub, scrub your markup for any hard-coded text that will become visible in the UI. Check the body content of elements like `p`, `h1`, `span`, `div`, `button`, `option`, and so on, as well as the value of attributes like `title`, `alt`, `aria-label`, etc. Also look inside WinJS controls like `AppBar` and `Flyout`, and look for any embedded URLs that you'll want to localize, including those of services you employ and content you show in an `iframe`. Note, though, that `title` elements in a page `head` are not shown and do not need localization.

In Here My Am! I found lots of strings in `pages/html/home.html`, which I've highlighted below:

```
<header id="header" aria-label="Header content" role="banner">
<section id="section" aria-label="Main content" role="main">
<div id="photoSection" class="subsection" aria-label="Photo section">
<h1 class="titlearea win-type-ellipsis"><span class="pagetitle">Here My Am! (8)</span></h1>
<h2 class="group-title" role="heading">Photo</h2>

<div id="locationSection" class="subsection" aria-label="Location section">
<h2 class="group-title" role="heading">Location</h2>
<div id="floatingError" class="win-type-x-large">Unable to obtain geolocation; check
<br />permissions and use the app bar to try again.</div>
<div id="retryFlyout" data-win-control="WinJS.UI.Flyout" aria-label="Trying geolocation"
  data-win-options="{anchor: 'mapDiv', placement: 'bottom', alignment: 'center'}">
  <div class="win-type-large">Attempting to obtain geolocation...</div>
</div>
```

where I also made a note that I'll need to localize the `taphere.png` image, as it contains text, but this is for the next section. In `default.html`, I also found labels and tooltips in the `data-win-options` attributes of the `appbar` commands (and I'm omitting some of the other markup for brevity):

```
<div id="appbar" data-win-control="WinJS.UI.AppBar" data-win-options="">
  <button data-win-options="{id:'cmdPickFile', label:'Load picture', icon:'browsephotos',
    section:'global', tooltip:'Load a picture through the file picker'}">
  </button>
  <button data-win-options="{id:'cmdRecentPictures', label:'Recent pictures',
    icon:'pictures',
    section:'global', tooltip:'Browse recent pictures taken in the app'}">
  </button>
  <button data-win-options="{id:'cmdRefreshLocation', label:'Refresh location',
    icon:'globe',
    section:'global', tooltip:'Refresh your location'}">
  </button>
</div>
```

Tip When preparing for localization, consider whether your `appbar` icons have a universal meaning. If not, they'll also need to be localized. Fortunately, the icon values are strings and can be localized as such, in which case you can treat them like the labels and tooltips.

Other affected files in this app include all of those in the HTML folder that are used for Settings commands. With such commands, be especially careful to note the short header labels that might be in a `div` like the one below amongst a bunch of other markup. Leave nothing behind!

Once you've located your strings, copy them as before to `resources.resjson`. This will likely be an extensive workout with copy and paste, so grab some refreshments and go for it. It's also fine to have HTML in these strings—they'll just be inserted into the markup and will render as such if you attach them to a property like `innerHTML`, but don't include the surrounding tag (we'll need it shortly). For example, in `html/about.html` I have a number of `p` elements with text, such as:

```
<p>Here My Am!<br />Version 1.0.0.0<br /></p>
```

for which I make the following string in the resources (no `p` tag):

```
"about1" : "Here My Am!<br />Version 1.0.0.0<br />",
```

Now for the fun part: how to we reference the string resources in markup? If you think about it a little bit, we have to run some piece of code to go through and replace whatever references we make with the appropriate string from the resource file. Hmmm. Haven't we seen something like this before? Indeed we have. With controls, we added `data-win-control` attributes to the markup and used `WinJS.UI.processAll` or `WinJS.UI.process` to run the code to instantiate the control. We have a similar setup for resources: a `data-win-res` attribute and `WinJS.Resources.processAll`, the latter of which should be called in each page's `ready` method or wherever else HTML content like the appbar is loaded, such as in the app's `activated` handler after `WinJS.UI.processAll` (so the controls are instantiated).⁸² Here's what you do in markup:

- Replace attributes and their string values with `data-win-res="{<attribute> : '<identifier>'}"` where `<attribute>` is the original attribute name and `<identifier>` matches the desired string in the resource file, contained in single quotes.
- Where there are multiple attributes in the same element, you can separate each `<attribute> : '<identifier>'` pair with a comma.
- When the string is directly inside a tag, we use `data-win-res` with the equivalent attribute name, such as `textContent` for a `div`, `p`, or `span`. If the string contains markup, use `innerHTML` instead, but only when necessary because `textContent` is much faster.
- For hyphenated attributes like `aria-label`, use the syntax `{<attribute> : '<identifier>'}'` in the `data-win-res` value, using single quotes around `<attribute>`. This is how you combine localization and accessibility together.
- For properties of WinJS controls that would normally appear within `data-win-options`, place those in `data-win-res` with the syntax `{<property> : '<identifier>'}'`. Multiple properties are again separated with a comma within the inner `{ }`'s.

⁸² If you see a `null` reference exception within `WinJS.Resources.processAll`, it's probably because you're trying to map resources for a WinJS control that hasn't been instantiated.

Here are some examples as modified from the earlier markup:

Original Markup	Modified Markup
<code></code>	<code></code>
<code>Here My Am! (8)</code>	<code></code>
<code><div id="locationSection" class="subsection" aria-label="Location section"></code>	<code><div id="locationSection" class="subsection" data-win-res="{attributes: {'aria-label' : 'aria_location'}}" ></code>
<code><div id="floatingError" class="win-type-x-large">Unable to obtain geolocation; check
permissions and use the app bar to try again.</code>	<code><div id="floatingError" class="win-type-x-large" data-win-res="{innerHTML : 'error_obtaingeoloc'}"></code>
<code><button data-win-control="WinJS.UI.AppBarCommand" data-win-options="{id:'cmdPickFile', label:'Load picture', icon:'browsephotos', section:'global', tooltip:'Load a picture through the file picker'}"></code>	<code><button data-win-control="WinJS.UI.AppBarCommand" data-win-options="{id:'cmdPickFile', icon:'browsephotos', section:'global'}" data-win-res="{winControl: {label : 'appbar_label1', tooltip : 'appbar_tooltip1'}}"></code>

When `WinJS.Resources.processAll` goes through the DOM, it actually doesn't remove any of the `data-win-res` attributes; it just processes those values and adds other attributes to the element that contain the string resource. The advantage of this is that a later call to `processAll` will go through the DOM and refresh all those strings. That means that if you handle the `WinJS.Resources.oncontextchanged` event, which tells you when the language changes, you can call `processAll` again and your UI appears in that new language! We'll add this little piece of code later on once we've added a few more languages to Here My Am!.

It also means that if you want to perform WinJS data binding in conjunction with a string that originates in your resources, simply include `data-win-bind` attributes inside those strings, assign that string to an element's `innerHTML` property within `data-win-res`, and then be sure to call `WinJS.Resources.processAll` *before* calling `WinJS.Binding.processAll`. This is demonstrated in Scenario 8 of the [Application resources and localization sample](http://html/scenario8.html) (html/scenario8.html and js/scenario8.js):

```
HTML: <p id="messageCount" data-win-res="{innerHTML: 'scenario8MessageCount'}">
Resources: "scenario8MessageCount" : "You have <span
           data-win-bind=\"innerHTML:count\"></span> message(s)",
```

And with that (except for the following sidebar that I've cleverly inserted), we're ready for the next step that will take care of our image resources and set us up to localize all this content we've extracted.

Sidebar: String Resources in Settings Flyouts

In all of this, one of the more challenging pieces of markup to work with were the HTML pages for settings flyouts, namely the `about.html`, `help.html`, and `privacy.html` in the project's HTML folder. These pages are not loaded until the settings command is invoked, and because that's happening all within WinJS we have to use the flyout's `beforeShow` event to call `WinJS.Resources.processAll` for the flyout's markup. To catch that event, I added `onbeforeShow : beforeShow` in each flyout's `data-win-options` string and then this piece of code within a `script` tag at the end of the `body` element:

```
function beforeShow() {  
    WinJS.Resources.processAll();  
}  
beforeShow.supportedForProcessing = true;
```

where the last line is necessary because WinJS will be calling `WinJS.UI.processAll` when the page is loaded. In any case, this works just great for patching up the string resources, with one exception. In `privacy.html`, if you remember from Chapter 8, "State, Settings, Files, and Documents," I'm using an `iframe` to load a remote page with a privacy statement. Because this should be localized, I placed the URI itself into the string resources and attempted to load it up like other strings:

```
<!-- This won't work -->  
<iframe data-win-res="{src : 'privacy_URI'}" height="600"></iframe>
```

However, this caused an exception within `WinJS.Resources.processAll` that complained about something not being marked with `supportedForProcessing`. Say what? The only such function I clearly had marked, and I couldn't think of what that has to do with an `iframe`. It turns out that `iframe` elements are specifically blocked from the WinJS `processAll` methods just like unmarked functions. As a result, you simply can't use `data-win-res` with an `iframe`!

Fortunately, the simple solution was to give the `iframe` an id (`privacyFrame`), load the string manually in the `beforeShow` handler, and then set the `iframe.src` attribute:

```
document.getElementById("privacyFrame").src = WinJS.Resources.getString('privacy_URI').value;
```

Now you'll have to excuse me for a moment while I file a bug to make sure this fact gets documented!

Part 2: Structuring Resources for the Default Language

In the previous section we created only a single `resources.resjson` file in the root folder of the project and we deferred any work on images. The next step is to introduce a little more structure into the project that will allow us to add localized resources for additional languages, including our images.

Starting with strings, do the following steps:

1. Create a *strings* folder in the root of your app project.
2. Within that folder, create a subfolder that matches the BCP-47 language tag specified as the default language in the manifest (for example, *en-US*, *fr-FR*, *ja-JP*, or just the base language like *en* or *ru*).
3. Move your *resources.resjson* file into that folder.

If you run your app again at this point, you should see that everything still works. If you go back to the [How to name resources using qualifiers](#) topic we mentioned earlier, you'll see that the resource loader is perfectly happy when you use qualifiers like a BCP-47 name as a folder name. It basically parses entire folder names looking for qualifiers, so you can create deep hierarchies to sort your resources however you like. That is, you can sort by contrast or scale first, if desired, and include language suffixes in the filename (where the format is *lang-<BCP-47 tag>*). What's more, you can create secondary *.resjson* files in these folders as well and play some other tricks. See "Sidebar: Secondary String Resource Files" for details.

Anyway, what you've just done by moving your resources into a folder for your default language is set your *fallback* language resources—this is what the resource loader will turn to if it cannot find a more specific match for the user's current language. Finding a match is actually a sophisticated process wherein the resource loader measures a kind of "distance" between the user's preferences and the available resources and chooses whichever is closest. This makes it possible to select *en-GB* as a closer match to *en-AU* than *en-US*, for example. Generally, though, it means that the resource loaded will search for a specific match like *de-DE* (German) first, then the next closest language using the base qualifier *de*, and then eventually fall back to your default language (if there are no resources for the user's other languages). The short of it is that you should always make sure the language identified in your manifest is fully populated with your full set of resources! Then even if you don't happen to localize some of those resources (say, for exact cultural alignment with images), one will still be found. For the complete story on this subject, refer to [Language Matching](#) in the documentation.

Sidebar: Secondary String Resource Files

Both WinRT and WinJS are able to work with secondary string resource (*.resjson*) files, allowing you to organize your strings in multiple files, if desired. For example, it's common to separate error strings into a file called *errors.resjson*. When referencing a string identifier in one of these secondary files, all that's needed is that you use the syntax */<file>/<identifier>* instead of just *<identifier>*. This syntax works in HTML, JavaScript, and the app manifest. See Scenario 5 of the [Application resources and localization sample](#) for an example.

Something else you can do with *.resjson* files is name them with other qualifiers for contrast, scale, home region, and so forth and even organize those files under any old folder. This is demonstrated in Scenario 13 of the same sample, where it has many different *.resjson* files underneath *strings/scenario13*, each of which is named as *scenario13.<qualifiers>.resjson*. Because

the folder name itself doesn't use a standard qualifier, you have to do a little more work to get at everything, using the [Windows.ApplicationModel.Resources.Core.ResourceManager](#) API, but it can be done if you're a serious resource junkie!

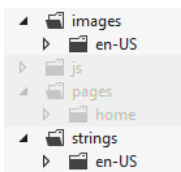
With images, we've already seen that if you have something like an *images* folder and place files like *logo.contrast-high_scale-140.png* therein, you can just refer to that file with a nonqualified relative URI like */images/logo.png* and the resource loader will find them.

Tip The potential multiplicity of images with (scale variants * contrast variants * language variants) and potentially others (like direction) is an important consideration for your app package: more images will make the package size larger. A large package in the Store might deter some users from downloading your app, especially those running on metered networks. So it's worth carefully evaluating exactly which images really need to vary with these different factors, especially the larger ones, and to optimize the degree of compression for all of your images to minimize the package size. Ask especially whether your splash screen image—typically one of the largest, especially at the 180% scale—needs to be localized at all, and even test whether the 180% image will look good when scaled down to 140%, 100%. If your splash screen, in other words, is just imagery with a sufficient contrast ratio and you have used a universally acceptable app name, one file might work everywhere.

Note also that you don't want to provide an unqualified resource if you provide any other specific variants, because a scale variant will always be matched before the unqualified one. As a result, the unqualified resource will just take up space in the app package but will never be used.

To prepare for localization, we need only move those images into a folder for our default language, as we did with strings. Because you're already using relative URIs to refer to your images (with or without `ms-appx:///`), you can use whatever folder path you want as your image root. There, create a folder with the appropriate BCP-47 tag and move all your default language images into that one. In *Here My Am!*, for example, images live in the *images* folder, so all I need to do is create an *en-US* folder under there, move all the images, and all my references such as */images/tile.png* will still work. And because they now live in a folder that corresponds to the app's default language, they become the fallback images.

I will mention that I did have one image, *maperror.png*, that was located in *pages/home* alongside the *home.html* file that referenced it. I moved this to *images/en-US* and updated URI references accordingly (but later eliminated it and *taphere.png* in favor of drawing a canvas dynamically). You can, of course, place images in as many folders as you like, provided that they each have language-specific folders therein. It's probably most convenient, however, to use a single root folder, or just a few; with *Here My Am!*, at the end of this step I had just two language folders in the project:



In your own app, then, look for app image references throughout your project. In HTML, look especially for `img` elements. In CSS, look especially for `background-image` styles. In the manifest, look at the Application UI tab (logos and badges), the Declarations tab (more logos), and the Packaging tab (the store logo). In JavaScript, finally, check any URIs you might be assigning to element properties or CSS styles, as well as any you might be referring to in the XML for tiles, badges, and toasts.

After that, evaluate each graphic to determine whether it will require localization, including those that need to be reversed generally for right to left languages (for these you can use single copies for all RTL languages, named with the *layoutdir* qualifier; refer back to [How to name resources using qualifiers](#)). For images that don't require localization (perhaps your tile and other logos, along with plain graphic elements you use in your layout), keep them in your fallback language folder. These will be used if no other match to the current set of qualifiers is found (language, scale, contrast, etc.) Depend on the fallbacks only if you have no other variants.

With that, we're now ready to localize!

Sidebar: The Application Resources and Localization Sample

The [Application resources and localization sample](#) shows many different scenarios for managing and referring to localized resources. It's worth spending some time with this sample because it will reinforce much of what we've discussed here: image resources (Scenario 1); string resources in HTML, JavaScript, and the manifest (Scenarios 2–4); using secondary resource files (Scenario 5); sending language info to web services (Scenario 7); combining resources and data binding (Scenario 8); using resources with hyphenated attributes (Scenario 9); triggering and detecting language changes (Scenarios 10 and 6); overriding the default language context (Scenario 11); using WinJS for resource lookup in the web context (Scenario 12); and multidimensional fallback (Scenario 13).

Creating Localized Resources: The Multilingual App Toolkit

Congratulations! With all the work you've done in the previous sections, you should have an app that's completely ready to be localized. The process here is really just one of acquiring translated versions of your .resjson files (for strings, taking sparse localization into account) and translated copies of any necessary images.

Tip If you have images that contain text, make sure that you have strings in your resources that match the image content, as you'd typically use for `img alt` attributes. In doing so, you'll obtain the necessary translations for the graphics in the process of localizing the strings.

If you like, you can just send your .resjson files, along with the text inside images, to an appropriate translator or translation agency and have them do the work. When you get them back, simply create additional BCP-47 folders in your *strings* and *images*, drop in those files, and away you go. You'll see such structures in the Application resources and localization sample, as we've referred to before.

Such manual translation can take a long time, however, and can become expensive. This is partly because professional translators don't necessarily have tools to work with resjson files other than a text editor. What they do have—sophisticated tools that help them track the status of translation jobs and much more—work with an industry-standard XML format known as XLIFF (XML Language Files). So it behooves us (and our checkbooks!) to make life as easy as we can for translators, even reducing their job to reviewing suggested translations rather than generating everything from scratch.

To assist in this, Microsoft offers the free [Multilingual App Toolkit for Visual Studio 2012](#). Once you've downloaded and installed the Toolkit, load your project into Visual Studio and select the Tools > Enable Multilingual App Toolkit menu item. You have to do this for each project separately, because what happens from here on is that the Toolkit will be generating multilingual resources for your app—in the resources.pri file—without you having to actually add any more .resjson files.

Once you've enabled the Toolkit, a command appears on the Project menu called Add Translation Languages. This brings up the Translation Languages dialog, as in Figure 17-10, in which you select desired target languages. At the very top of the list, the Pseudo Language option (qps-ploc) will be automatically checked; we'll be using it in the next section to test localization. This is something you typically want to do before doing specific localizations. Also note that many languages sport a "Microsoft Translator" logo, which means they can be mostly translated automatically, saving paid translators much time and you, much money.

Videos! For a video series on the Toolkit from the team who created it, see the following:

[Introduction to the Multilingual App Toolkit](#) (3m 50s)

[Build Multi-language apps using the Multilingual App Toolkit](#) (9m 01s), covers creating string resources as we've already discussed.

[Test Multi-language apps using the Multilingual App Toolkit](#) (5m 36s), covers what we'll talk about in "Testing with the Pseudo Language" later on.

[Localize Multi-language apps using the Multilingual App Toolkit](#) (6m 40s) demonstrates the Multilingual App Toolkit Editor as we'll see shortly.

[Submitting your localized app to the Store](#) (9m, 05s) highlights considerations for getting your app to the right markets.

Once you've made your selections (you can add more later), press OK and the Toolkit will create a folder in your project called MultilingualResources stocked with a bunch of XLF files (the ones the translators like). At first these will be mostly empty, but now here's why it was worth the effort to build up your default resources.resjson file. Right-click your project in Solution Explorer and select Build or Rebuild, or use the Build > [Re]build Solution menu item. This will go through your string resources (including any localized variants you might have created already) and populate all the XLF files with all your strings. The process will also draw in references to nonlogo images (that is, tiles and splash screens are omitted) that might also need translation.

Now for the real fun: double-click an XLF file to launch the Multilingual App Toolkit Editor shown in Figure 17-11. Here you can manage which resources can or should be translated along with the state of the translation. If the language is also supported by Microsoft Translator, the Translate button at the top will be enabled to translate a single entry as well as to Translate All. Select the latter and sit back to enjoy the show. In a few moments you'll see that the tool has translated all your strings, marking each with the state of Review, as shown in Figure 17-12.

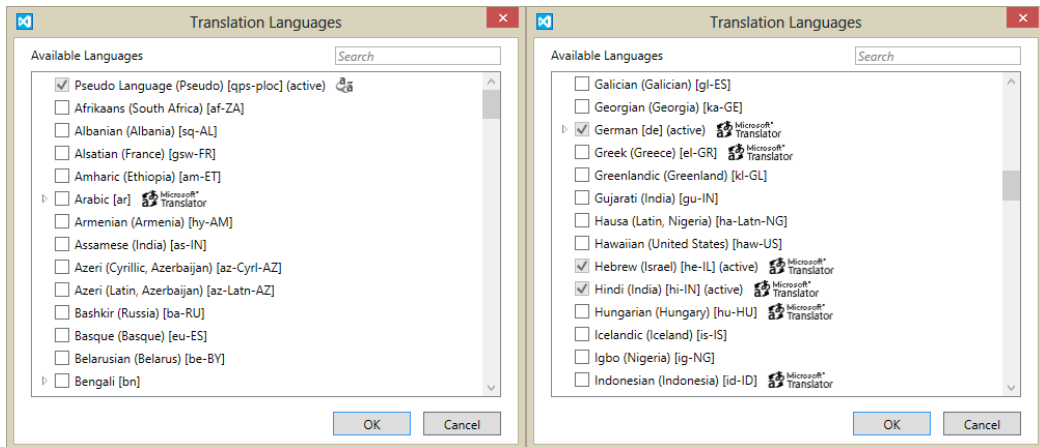


FIGURE 17-10 The first dialog of the Multilingual App Toolkit's language selection feature. At left we see the option for Pseudo Language, an artificial language with lots of funky characters that represents the needs of most other languages.

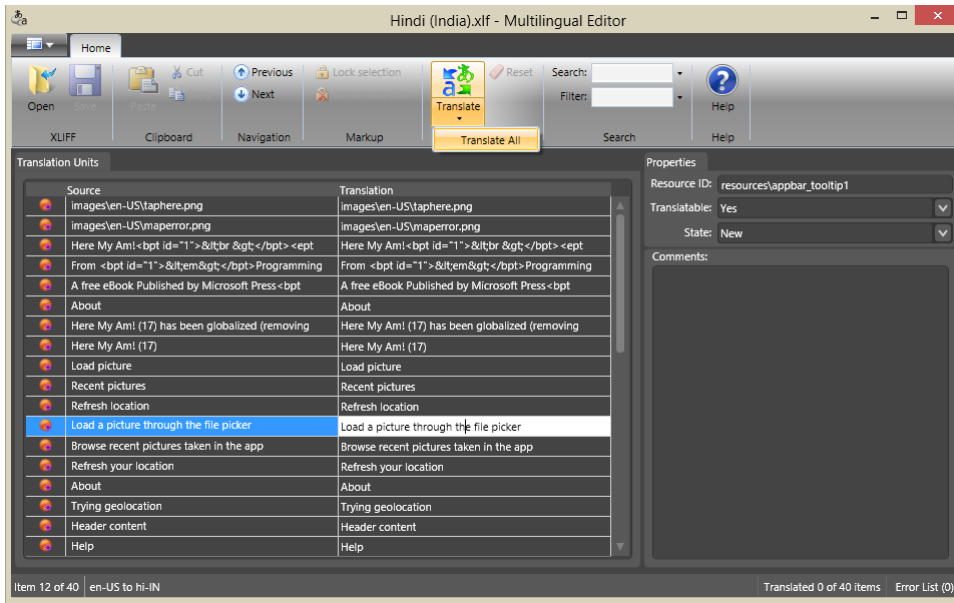


FIGURE 17-11 The Multilingual App Toolkit Editor, an XLF editor with machine translation built in.

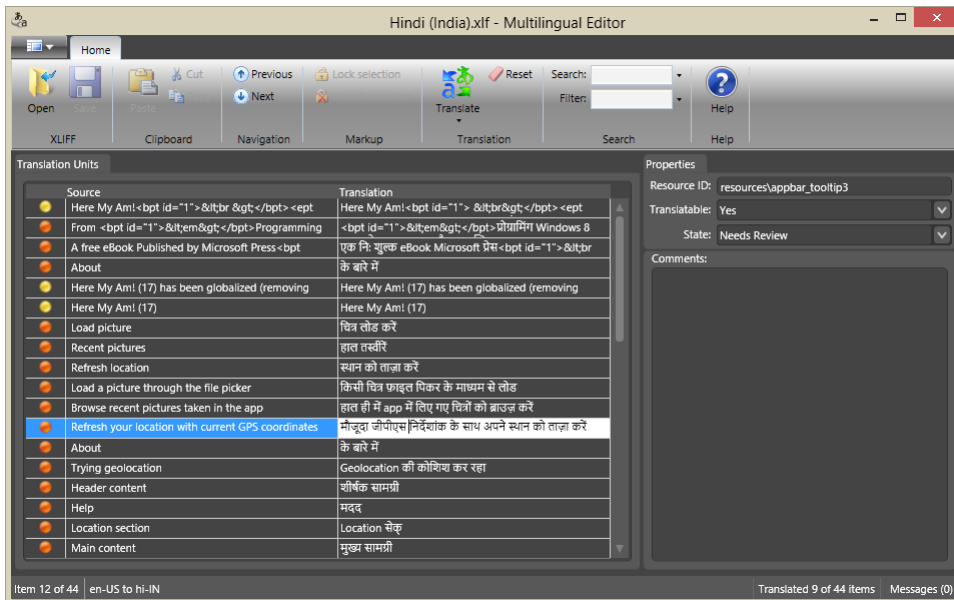


FIGURE 17-12 The string resources of Here My Am! after being machine-translated into Hindi.

Once the translation is complete, save the file and close the editor if you want. Go to Control Panel > Control Clock, Language, and Region > Language and make sure the target language is added and is placed at the top of the list. Then return to Visual Studio and launch the app—and there is your first-pass localization, as shown in Figure 17-13 for Here My Am! (Notice that I elected to not translate the title, something that my translators suggested I keep in English because of its unique grammar.)

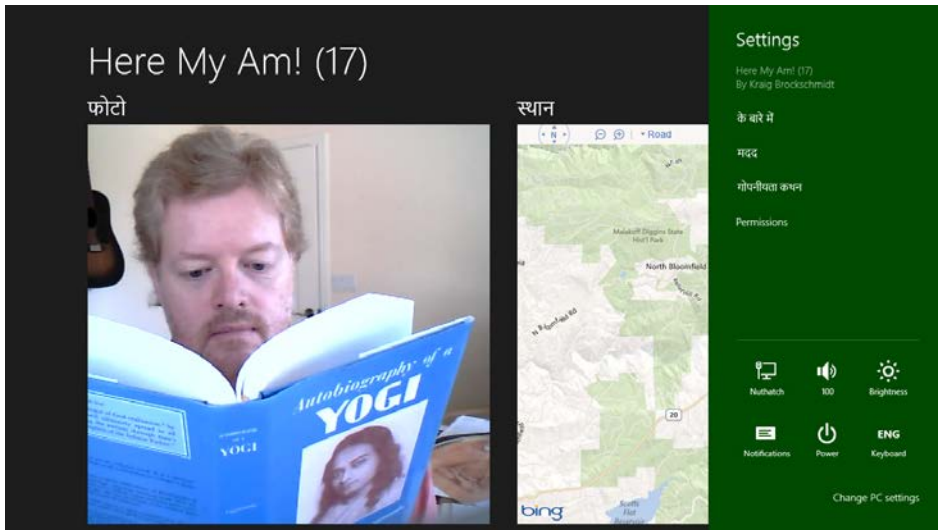


FIGURE 17-13 Here My Am! running in Hindi using machine translation output, which should still be checked by a native speaker, of course. As you can see, I’m checking if Yogananda has any advice apropos to the language.

If you like living on the edge and don’t mind shipping an app that people in other markets might laugh at or otherwise criticize for your carelessness, there’s nothing stopping you from making your app available to those markets in the Windows Store with such machine translations. If you like good positive ratings and reviews, on the other hand, it’s a good idea to at least find some native speaker who can validate and correct what the automatic translation process suggested. You can have this helpful person use the Toolkit editor to review your XLF files, in fact. When those files are reviewed and returned to you, import them back into your project by right-clicking the existing XLF file in Visual Studio and selecting Import Translation. The new translations will then be included in your next build.

When working with professional translators, you can also select specific XLIFF Translation file formats by right-clicking the XLF file in Visual Studio and selecting Send for Translation.

Three other notes about this process. First, there may be some strings or parts of strings that don’t require translation. In the Toolkit editor you can set the Translatable option to No for whole strings to prevent the machine translation from changing that string. For parts of a string, those will be translated but you can edit them back to their original and make a note in the Comments area for your translators.

Second, the Toolkit will detect if you've already made translations in an XLF file such that running a Build/Rebuild will not overwrite those strings. At the same time it will import any new strings you've added to your resource file in the meantime and remove any that have been deleted. A change in a resource identifier, however, is treated as a delete+add, meaning that the translation will be lost.

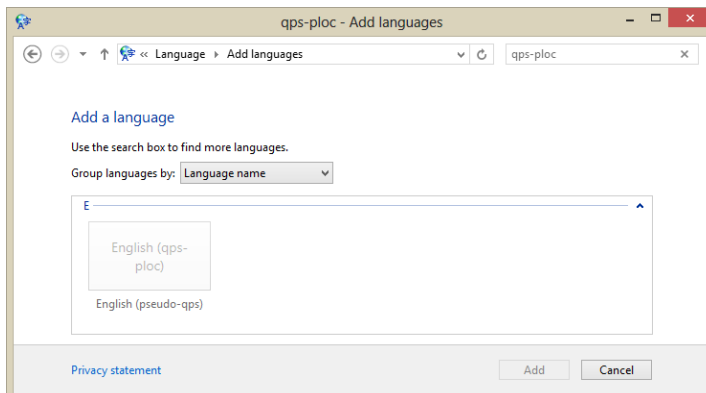
Lastly, if you want to remove a language just right-click the XLF file and select Exclude From Project. This will keep the language out of the build while preserving the file (and its translations) in your project folder.

Testing with the Pseudo Language

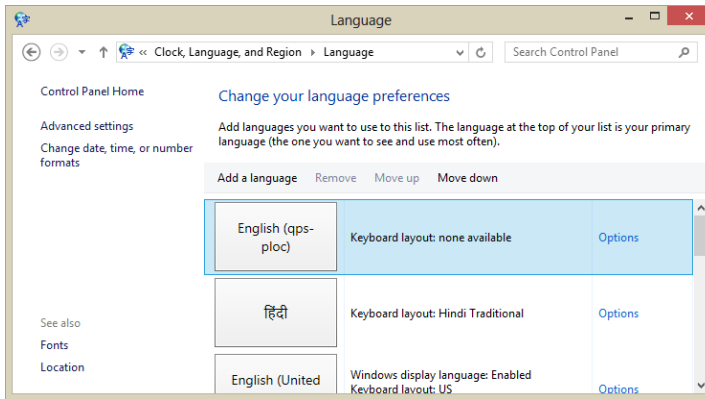
As much fun as it is to produce many translations for your app, there is still the matter of testing it well, a task that is clearly overwhelming if you're targeting many languages! To reduce this burden, the best approach is to test your app using the Pseudo Language, a step that's ideally done before incurring the cost of specific translations. It helps you validate that your app can handle a variety of languages, because the fictitious Psuedo Language contains some of the most problematic characteristics of localized text.

As noted in the previous section, this language is automatically added to your project through the Multilingual App Toolkit's language selection dialog. This creates a Pseudo Language (pseudo).xlf file in your MultilingualResources folder, alongside the real translations. Next, right-click that file and select the Generate Pseudo Translations command. This will populate the XLF file with translations of your default resources where basic characters are often converted to extended characters and strings are generally expanded with extra !!!'s tacked on. So, a string like "Recent pictures" gets translated to "[62BD8][!!_ŘęćęŃť þíćťµřěš_!!!!]" where the hexadecimal stuff in the first []'s is a resource identifier that helps testers identify the exact resource that's being used. (Note that this process will "translate" every string whether you will ultimately translate those strings for real, because it's helpful for testing.)

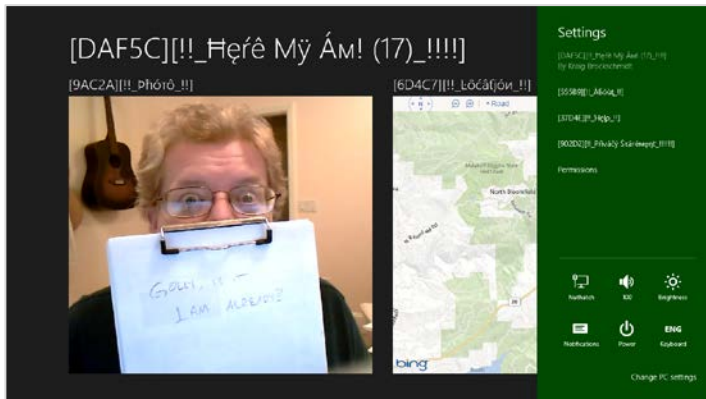
To run the app with this translation, you need to make Pseudo Language the system default. In Control Panel > Clock, Language, and Region > Language, click Add a Language, and then enter *qps-ploc* in the search box. This is the only way to make the Pseudo Language option appear:



Select that language, click Add, and then move it to the top of the list:



When you run your app now, you should see it appear in Pseudo Language:



With your app running in Pseudo Language, be sure to exercise every feature and option. Check every page in each view state; check all your app bar commands; check all of your settings; check any error messages, flyouts, and message dialogs that might only appear under specific circumstances like changes in network connectivity; and test your app with all its activation paths according to the contracts you support. As you do so, look for any strings that don't appear in Pseudo Language, a clear indication that you missed pulling that string from your markup or code. Also check for truncated text, unintended word wrapping, and so forth, which reveals where your layout isn't accommodating longer translated strings.

This is the time to be as thorough as possible, because once you upload to the Store, issuing another update will take at least one or two weeks, during which time your customers might find those problems and ding your ratings accordingly. It's always something to keep in mind, especially if you've been accustomed to instantly fixing bugs on websites: apps simply take longer, so you want to invest in testing ahead of time.

Localization Wrap-Up

Well, we're almost done with the app and ready to go to the Store! There are just a few more things to mention about localization:

- Testing with the Pseudo Language does not cover RTL language considerations; you'll need to run with those languages separately. I'm happy to say that when I ran Here My Am! under such conditions (such as Hebrew), the layout automatically mirrored thanks to the `direction` style set in the WinJS stylesheets.
- Be sure to test any and all interactions with online services, including periodic tile/badge updates and those that arrive through push notifications.
- If you want to dynamically update your app when the user changes languages (instead of having to restart the app), listen for the `WinJS.Resources.oncontextchanged` event and call `WinJS.Resources.processAll`. This code is in Here My Am! as well as Scenario 6 of the [Application resources and localization sample](#):

```
WinJS.Resources.addEventListener("contextchanged", function () {  
    WinJS.Resources.processAll();  
});
```

- The above code will refresh string resources but neither image resources nor content obtained from online sources. You'll want to do those other updates with additional code, such as giving Windows a new URI for periodic tile updates or indicating that language to a service that issues push notifications. For the app's overall UI, try using `document.location = document.location + "?reload"`, picking up that URI parameter in your `activated` handler to take additional steps. This essentially mimics relaunching your app.
- If you like, you can allow the user to select the language for the app independent of the system settings. This is done by setting the `Windows.Globalization.ApplicationLanguages.primaryLanguageOverride` property, as demonstrated in Scenario 10 the [Application resources and localization sample](#). Scenario 11 also shows loading specific language resources rather than the default.
- In Visual Studio, open your manifest in the XML code editor (right-click and select View Code) and check if you see this line within the `Resources` element: `<Resource Language="x-generate" />`. If so, replace that line with individual entries like `<Resource Language="en-US" />`, where the first is your default language, and you must have localized resources for all the rest. In addition, you must have at least one "certification language," as described on [Building your app package](#), or the app will fail Store certification.
- For translation on the fly, you can use various web services such as [Bing Translator](#).

Releasing Your App to the World

We have arrived at the last section of this chapter and the last section of this book, coming full circle to the exact point where we started in Chapter 1: onboarding our world-ready app to the Windows Store and making it available to that world.

Because the onboarding process is well documented already in the [Selling apps](#) topic, I'm not going to spend our time here together giving you a bunch of screen shots from the [Store Dashboard](#), where all of this action takes place. I'll point you to specific pages in those docs when appropriate, but it's definitely a section of the documentation that you should review. After all, the Windows Store is *the* retail channel for your app, so you want to understand that channel as best you can. The Store dashboard is also designed to lead you through the process directly.

What we'll focus on here specifically are those aspects of the process that aren't always so obvious, based on the real-world experience that I and my teammates in the Windows Ecosystem Team have gained through working with the first partners to submit apps to the Store. Through this I hope to raise your awareness of issues that you'll likely face so that you're more prepared to address them. We'll then conclude with a look at app updates and increasing discoverability of your app through linkage with your website.

Sidebar: Setting Build Targets

When you create your app package to upload to the Store, be mindful that you set your project's configuration to Release instead of Debug, otherwise it will fail certification. When choosing the target platform, set this to "Any CPU" unless you specifically have WinRT components written in C++. That is, JavaScript and .NET languages (C#/VisualBasic) are architecture-neutral; anything written in C++, on the other hand, must target x86, x64, and ARM specifically. More often you'll need to create three builds for these architectures that you'll upload to the Store individually.

Promotional Screenshots, Store Graphics, and Text Copy

Before you do anything else with your app and the Store, review the topic [How customers see your app in the Windows Store](#) and its four subsidiary topics: [App listing info](#), [app listing overview](#), [app listing details](#), and [app images](#). Also review the [Pre-development checklist](#), which provides a blend of pre-development and pre-onboarding topics, especially [Naming your app](#) and [Describing your app](#).

The reason why I specifically call out these topics is because you've invested or you're going to invest a lot of time and energy developing your app (and testing it, as we'll discuss in the next section), and so you should make a comparable effort to make it look great in the Store. All of the content described by the links above—your app's name and description, its details, and its promotional images—constitute your customers' first experience of your app.

Let me say that again: all of this information is what potential customers will use to evaluate your app before they tap any button to acquire it. It is marketing material, plain and simple, so make it shine! Spend time writing really good copy for your app description—even to the point of having it professionally edited or hiring a professional writer. If you feel your app is fun and engaging, *communicate* that experience through your description and imagery. Truly, you want customers' first impression of your app—just from a quick glance at your app's page in the Store—to be WOW! And this content is all that determines that response.

The other reason I emphasize this so strongly is that you won't otherwise know you need any of this information until you begin the onboarding process, at which point the Store will ask you to paste in text and upload images. If you haven't prepared those materials already, then, and you're trying to get the app into the Store as quickly as possible, you'll end up cutting some serious corners. As a result, your app's first impression will be nowhere near as good as it could be.

Game Rating Certificate When uploading a game to the Windows Store, you'll also be required to provide a game rating certificate in the form of a GDF file. For details, see [Windows game publishing requirements](#).

Testing and Pre-Certification Tools

Unless you're a born tester, app testing is an activity that has little glory and thrill compared to development, yet it can make a huge difference in the success of your app.

Indeed, for many developers—especially those who have been primarily focused on the web, as I expect many readers are—rigorous testing is not one of their skill sets. I think this is because the nature of web development, where you can upload a fix to a site and have it take effect immediately, has not demanded much testing discipline. How often have you seen one of your favorite websites just blow up one day, hobble around for a few hours, and then come back to life? It's probably because some developer introduced a nasty bug which was discovered and purged during those hours of awkwardness. For some sites, that downtime can be disastrous, but for many others the impact is small to negligible.

Put another way, the costliness of bugs in web apps is generally quite small because the update time is also very small. But this is *not* a reality with apps. The time from when you submit an app to the Windows Store to when it's made available is at least a week, if not longer, depending on the Store's backlog. This means that each submission is far more significant.

Just look at it in terms of turnaround time. Let's say it takes five minutes to upload a fix to a web app. Compare that to the number of minutes in a week, which is 10080. The ratio? 1 to 2016. In other words, *it's at least 2000 times more expensive in terms of time and effort to update an app in the Store.* Practically speaking, this means that you might need to spend orders of magnitude more effort testing apps than testing websites. That's significant! (And don't make the argument that because you spend zero time testing web apps the multiple still comes out zero.)

If you don't have some testing methodology in place, then, start building one, even from the basics. For example, be sure to always test your app on a clean install of Windows 8 on a machine without a developer license, as well as on low-end machines whose performance is similar to many ARM devices. One developer I worked with had an app rejected by the Store because it came up blank on first run—he never saw this happen because of all the cached data on his development machine!

You also want to develop a solid checklist of how to poke and prod your app to exercise all its code paths. This should include subjecting it to all the conditions that come from outside your app: changing view states and device orientations; invocation of the different charms; changes in network connectivity; running on slow networks; varying screen sizes and pixel densities; input from different sources; having your temp files cleaned out with the Disk Cleanup tool; signing on with different credentials; suspending, resuming, and restarting after termination; running with high contrast modes and other accessibility features; and running under different languages. The better your app behaves under all these circumstances, the more solid it will look and feel to the customers who will be writing ratings and reviews. I cover these topics in a two-part video called “Beyond Just Beautiful” that you can find on the [Concepts and architecture](#) page of the Developer Center.

Beyond that there are some great topics in the documentation to help you take the next steps:

- [Debugging and Testing Windows Store apps](#)
- [Analyzing the code quality of Windows Store apps with Visual Studio code analysis](#)
- [Creating and running unit tests on a Windows Store app](#)
- [Analyzing the performance of Windows Store apps](#)

The other very important part of testing is running your app through the Windows App Certification Kit, otherwise known as the WACK. This tool subjects your app to all the automated tests that will happen when you onboard to the Store, thereby letting you correct any problems it finds beforehand. Passing the tests in the WACK is no guarantee that your app will be accepted, but it will certainly save you a great deal of time waiting for onboarding results and having to resubmit over and over. You should, in fact, run the WACK just about every day during development. You won't necessarily fix everything it brings up right away, but the ongoing data will be very valuable.

For complete details on the tool and what it does, see [Testing your app with the Windows App Certification Kit](#) and [Windows App Certification Kit tests](#).

Tip If you find the WACK coming up blank (showing no apps to test), try uninstalling SDK samples that you might have run from Visual Studio. It seems the tool can get overloaded sometimes.

Onboarding and Working through Rejection

When you're really ready to upload your app to the Store, you can use the Visual Studio's Store menu options. Create App Package will let you create a Store-ready package (and run the WACK); the Upload App Package command will take you to the Store dashboard to complete the process. And just in case you're interested, the maximum size of an app package for upload is 2GB (see [Building the app package](#)).

When onboarding your app, you'll be asked for all the promotional details discussed earlier, as well as URLs where support and privacy information can be found. You'll also select target markets, set pricing, enter details for in-app purchases, set trials and expiration dates, set a release date, and provide notes to the Store testers. (For an overview, see the [App submission checklist](#).) The last item is essential if there are any details that will be necessary for a real person to test your app as part of the process, such as credentials for a test account. And yes, a real human being will look at your app (and read your notes, so be courteous)! Automated tests can only accomplish so much—in the end, someone needs to run the app and make sure it does what it says.

If you've done the work to make your app accessible, by the way, there's a special place to say so. See [Declaring your app as accessible](#).

Once your app has completed the testing process, it will either be accepted or rejected. Acceptance is really a nonissue—that's what you're looking for! If your app is rejected, on the other hand, the Store will tell you why, specifically citing violations of the [Windows 8 app certification requirements](#). Indeed, these policies contain the *only* reasons that an app can be rejected, so any rejection must necessarily indicate the particular requirement that isn't being met. The Store also provides some information about failures, such as where an app crashed (a violation of requirement 1.2).

By and large, most of the policies are straightforward such that if you fail on them, it's pretty clear why. A few, however, seem to be more confusing or subjective, and in the early days of Windows 8 previews they were downright mysterious. Now, fortunately, there is an extensive list of reasons why an app might fail a number of requirements on [Resolving certification errors](#), many of which come from our experience with real apps submitted to the Store. The Store testers can also provide direct feedback regarding specific failures, like where and when an app might have crashed which certainly caused them to reject it.

App Updates

One thing that apps and books share in common is that the moment you release them, you'll find errors, bugs, typos, and a hundred other things you wish you could change. Fortunately, updating apps is easier than updating books (even though fixing app bugs is often far more difficult than correcting a typo).

You might want to issue an update for many reasons besides the obvious problems you see yourself and the features that didn't make it into your current version. You might want to respond to user reviews and requests, add more in-app purchase options, or add features to pursue new opportunities. For nearly all of these purposes, the [Reports and data](#) that come from the Windows Store will be highly valuable. You'll want to review this as soon as your app is in the Store and make a plan for monitoring

those reports that are most interesting to you. Some suggestions for this are found on [Updating your Windows Store app](#). In fact, schedule some time in the future right now to check in with these reports, lest you forget they exist. Truly, these reports are your best link to real customers!⁸³

All such data will surely feed back into your planning and development processes, ultimately bringing you to the point of one again uploading your app with new promotional materials and more features. When you're ready to upload a new package, be sure to increment the version number in the manifest so that you can keep track of things. (This is available at run time from [Windows.ApplicationModel.Package.current.id.version](#).) Onboarding is then the same as for any other app—no matter how little you might have changed, the app goes through the whole certification process again. For this reason, don't think to make whimsical updates—make each one count (fixing a single critical bug certainly counts!). Also, be aware that the certification requirements can change over time, so just because an app was accepted once doesn't mean it will be accepted again. Make it a point to periodically review the requirements.

In your updated app, be prepared to migrate any state that might already exist on the machine, if there have been changes. We talked about this in Chapter 8 in "Versioning App State," where we distinguished between the version of an app and the version of its state; many app versions can use the same state version. However, if the app now uses a new state version, the old state must be migrated. Remember too that you can use the [servicingComplete](#) background task for this purpose, as mentioned in Chapter 13 in "Tasks for System Triggers (Non-Lock Screen)." Finally, once you introduce new versions of your state, roaming data will roam between apps of the same version only—you'll be able to migrate old state when the new app is run, but once the version is increased, that data will no longer roam to devices with older versions of the app.

The last point to mention about updates is that although your new app package might be fairly large, existing customers will *not* have to download the whole thing again. If you go way, way back in this book to Figure 1-1 in Chapter 1, we talked about the package's *blockmap*. To summarize, an app package is segmented into 64K blocks, and *only those blocks that have actually changed between versions are necessary to download for the update*. In practical terms, this means that you shouldn't worry about making a critical update to your app: if it affects only a small part of the code, your existing customers might end up downloading only one or two 64K blocks total! To help this along, try to have more small files in your project than a few large ones, and it's better to make changes at the ends of files than at the beginning or the middle.

Getting Known: Marketing, Discoverability, and the Web

As Gandalf the White says to Frodo, Sam, Merry, and Pippin at the conclusion of the *Lord of the Rings* movies, "Here at least, on the shores of the sea, comes the end of our fellowship." And, my friends, it has been a delight to share the journey with you! In this last section, then, I wanted to leave you with one technical matter—that of linking your app to your website—before sharing a few final thoughts.

⁸³ These reports will not give you any personal information about your customers, of course. If you want to collect that, you'll need to implement an opt-in registration system in the app that complies with requirement 4.1.2.

Connecting Your Website

If you have an app, you'll almost certainly have a site that provides additional information and support. (Requirement 6.3 deals with support specifically.) What, then, if potential customers come to your site first? Surely you'll want to provide an easy way for them to acquire your app if they're running on Windows 8.

For this you can simply link to a special URI for the Windows Store that starts with `ms-windows-store`, as described on [Creating links with the Windows Store protocol](#). You can also use the form `ms-windows-store:REVIEW?` to link directly to your app's ratings and reviews. And also remember that you can include a link to your app's page in the Store with data packages you provide to the Share contract, as covered in Chapter 12, "Contracts."

With Internet Explorer, a little bit of metadata in your web page's `<head>` will enable a feature that makes it simple for a customer to acquire your app and even run it if the app is installed. For example:

```
<meta name="msApplication-ID" content="ProgrammingWin8-JS-CH17-HereMyAm17"/>
<meta name="msApplication-PackageFamilyName"
      content="ProgrammingWin8-JS-CH17-HereMyAm17_5xchamk3agtd6"/>
```

where the two `content` values come from the Package Name and Package Family Name fields in your manifest's Packaging tab. Again, if the user doesn't have your app, this makes the acquisition process easy. If the user does have your app, he'll have the opportunity to launch it in which case the app will be activated with the launch kind of `protocol`.

For more, see [Linking to your apps on the web](#) on the Windows Store developer blog and [Connect your Web Site to Your Windows 8 App](#) on the Internet Explorer blog. And for a working example, visit the site of [Inrix Traffic](#), one of the earliest app partners who implemented these features.

One other possibility comes to mind here as you make the effort to promote your app: you might be approached by an OEM to include the app preinstalled on their devices. If this happens—it's quite a prize!—the OEM will share with you some special instructions about how to onboard and maintain an app specifically for their customers.

Final Thoughts: Qualities of a Rock Star App

It almost goes without saying that the Windows Store will at some point become crowded, so differentiating your app and yourself as a developer will become increasingly important. Again, there's plenty to do with marketing and gaining awareness for your app, as well as being responsive to customers. Beyond that, though, what does it really mean to make an app that's truly special?

Early on, long before Microsoft landed on the term "Windows Store apps," we referred to them as "tailored apps." To play with that older term, think of what tailoring means in the context of clothes: well-tailored clothes are very distinctive. They make you look really good. They make you feel *great*. That's how you want the users of your app to feel when they're immersed in your experience. Indeed, just as joy and happiness are the undercurrent behind your own app-building efforts, so also do they

live in the hearts of your customers. If you can deliver joy to them through your app, then I think you'll have a winner!

Another meaning of “tailored” implies that the kinds of apps we’ve been building in this book—apps that run full screen and deeply immerse a user in an experience—lend themselves well to being very specific to both the device and the user’s context. As we saw in Chapter 9, sensors give you the ability to know the device’s relationship to the physical world, which is an extension of the user who is holding that device. Ask then, “What can I do with that information? How can the app really light up when it has a deeper understanding of where the user is and how the user is moving about in this world of ours? Is there something more the app can do to say, ‘Aren’t you glad you brought me along?’”

To differentiate your app, think through how a consumer might use various form factors in different situations and have the app present itself differently in those contexts. This kind of tailoring means that the app surfaces the most relevant features or content for the most likely or appropriate use cases. As shown in the last figure of Chapter 1 within “Sidebar: The Opportunity of Per-User Licensing and Data Roaming,” I like to think of there being one app across many devices and that the user has a much stronger relationship to the app than to the devices it’s running on. The app and its underlying state becomes the consistent element across the whole experience, with the devices just being the vehicles. The more you can deliver an app that understands and support this (and obviously roaming data is important here!), the more I think the app will stand out from others that, sure, run in the new environment of Windows, but otherwise offer the same experience as we’ve had for many years.

So, what about being a rock star? Let’s be honest here. You’re in this game for name and fame, right? And for the big money that could come with it? What kind of app will get you there?

In what is now the very last paragraph of this book (apart from the end-of-chapter summary), I can’t really give you a bunch of specific ideas. (Otherwise I’d be writing those apps instead of writing books, but someone has to do this dirty work....) But ponder this: what makes a rock star in the music industry? Well, it’s not typically about the philosophical depth of the lyrics or the virtuosity of the musicians, it’s about performance, personality, and sheer entertainment value. It’s about delivering a joyful experience that turns everyday customers into raving lunatic fans who can’t wait to be your greatest champions. In a very real way, the experience is one that truly lets people escape their everyday realities and become part of something larger for a time, or even just part of a fantasy. And like great music or movies, the app experience is one that people want to repeat many times over and not just check the box as another “been there, done that.” Although there are certainly aspects of timing and sheer luck, all rock stars—along with great athletes, Oscar-winning movies like *Lord of the Rings*, and so on—strive for and achieve one thing above all: *excellence*. Commit yourself to that. Commit yourself to excellence in everything you do—not just in your apps, but in all parts of your life. Such striving, certainly, will eventually bring many rewards!

What We've Just Learned

- An app's relationship to the Windows Store is very closely related to your business as a developer, because it supports a range of options from free apps, ad-supported apps, limited-time trials, paid apps, and in-app purchases (using a custom commerce engine for the latter if desired).
- Side loading of app packages is supported for developers (on a machine with a developer license) and for enterprises. Otherwise all apps come from the Windows Store.
- The Windows Store APIs provide for managing app licenses, licenses for in-app purchases, and receipts. During development, the app uses a simulator object where data is obtained from a local XML file instead of the live Store, which allows for testing different types of transactions and license conditions.
- Accessibility features are a concern for the majority of users, even those without disabilities who find those features useful at different times. Apps support accessibility through ARIA attributes (for screen readers), implementing keyboard interaction, resolution scaling, and responding to high contrast modes.
- Globalization is the process of removing language and cultural assumptions from an app, using globalization APIs to properly handle user language, varying, calendars, formatting of numbers, dates, times, and currencies, sort orders, how strings are combined, varying text input methods, and which web services are used from which regions.
- To prepare for localization, an app needs to be scrubbed for text and image content that will be subject to translation, separating strings into resources files and inserting references in their place, and then structuring those resources in folders and files that employ resource qualifiers.
- For efficient localization, the Multilingual App Toolkit for Visual Studio generates and translates an app's default resources into any number of other languages, using the file formats employed by professional translators who can verify the results. It also produces a Pseudo Language translation for localization testing.
- Getting an app in the store starts with testing the app both manually and through the Windows App Certification Kit and being prepared for possible rejection during the onboarding process.
- App updates can be submitted to the Store at a later time, with improvements based on feedback and telemetry, and the updated code needs to be ready to migrate state.
- Being in the Windows Store does not reduce the need for marketing; getting found will become increasingly difficult as more apps appear in the Store. Cross-linking your app and website can thus very much help discoverability.



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

www.SoftGozar.com