

Programming Windows Store Apps with HTML, CSS, and JavaScript

Second Edition



Kraig Brockschmidt

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

Copyright © 2014 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.

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

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

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 and Joel Panchot

Table of Contents

Introduction	21
Who This Book Is For	23
What You'll Need (Can You Say "Samples"?)	24
A Formatting Note.....	25
Acknowledgements	26
Free Ebooks from Microsoft Press	28
The "Microsoft Press Guided Tours" App	28
Errata & Book Support	28
We Want to Hear from You.....	29
Stay in Touch	29
Chapter 1 The Life Story of a Windows Store App: Characteristics of the Windows Platform	30
Leaving Home: Onboarding to the Windows Store	32
Discovery, Acquisition, and Installation.....	35
Playing in Your Own Room: The App Container	39
Different Views of Life: Views and Resolution Scaling	42
Those Capabilities Again: Getting to Data and Devices	46
Taking a Break, Getting Some Rest: Process Lifecycle Management	49
Remembering Yourself: App State and Roaming.....	51
Coming Back Home: Updates and New Opportunities	56
And, Oh Yes, Then There's Design	58
Feature Roadmap and Cross-Reference	59
Chapter 2 Quickstart	65
A Really Quick Quickstart: The Blank App Template	65
Blank App Project Structure	68

QuickStart #1: Here My Am! and an Introduction to Blend for Visual Studio	72
Design Wireframes	73
Create the Markup	76
Styling in Blend.....	78
Adding the Code.....	83
Extra Credit: Improving the App	97
Receiving Messages from the iframe	98
Improving the Placeholder Image with a Canvas Element.....	99
Handling Variable Image Sizes	100
Moving the Captured Image to AppData (or the Pictures Library)	103
Using a Thumbnail Instead of the Full Image	105
The Other Templates: Projects and Items	107
Navigation App Template.....	107
Grid App Template	107
Hub App Template.....	108
Split Template	108
Item Templates	108
What We've Just Learned	109
Chapter 3 App Anatomy and Performance Fundamentals	111
App Activation	112
Branding Your App 101: The Splash Screen and Other Visuals	113
Activation Event Sequence	117
Activation Code Paths.....	119
WinJS.Application Events	121
Optimizing Startup Time	124
WinRT Events and removeEventListener.....	126
App Lifecycle Transition Events and Session State	128
Suspend, Resume, and Terminate.....	129

Basic Session State in Here My Am!	133
Page Controls and Navigation	136
WinJS Tools for Pages and Page Navigation	136
The Navigation App Template, PageControl Structure, and PageControlNavigator	139
The Navigation Process and Navigation Styles	146
Optimizing Page Switching: Show-and-Hide	148
Page-Specific Styling	149
Async Operations: Be True to Your Promises	151
Using Promises	151
Joining Parallel Promises	153
Sequential Promises: Nesting and Chaining	153
Managing the UI Thread with the WinJS Scheduler	156
Scheduler Priorities	157
Scheduling and Managing Tasks	158
Setting Priority in Promise Chains	160
Long-Running Tasks	162
Debugging and Profiling.....	165
Debug Output and Logging.....	165
Error Reports and the Event Viewer	166
Async Debugging	169
Performance and Memory Analysis	170
The Windows App Certification Toolkit	175
What We've Just Learned	176
Chapter 4 Web Content and Services	177
Network Information and Connectivity.....	179
Network Types in the Manifest.....	180
Network Information (the Network Object Roster)	181
The ConnectionProfile Object.....	183

Connectivity Events	184
Cost Awareness	185
Running Offline	189
Hosting Content: the WebView and iframe Elements	191
Local and Web Contexts (and iframe Elements)	192
Dynamic Content.....	195
App Content URIs	197
The <x-ms-webview> Element.....	198
HTTP Requests	209
Using WinJS.xhr.....	210
Using Windows.Web.Http.HttpClient.....	211
Suspend and Resume with Online Content.....	216
Prefetching Content	218
Background Transfer	219
Basic Downloads	221
Basic Uploads.....	225
Completion and Error Notifications	226
Providing Headers and Credentials	227
Setting Cost Policy	227
Grouping Transfers.....	228
Suspend, Resume, and Restart with Background Transfers.....	228
Authentication, the Microsoft Account, and the User Profile	230
The Credential Locker	231
The Web Authentication Broker	233
Single Sign-On	237
Using the Microsoft Account	238
The User Profile (and the Lock Screen Image).....	244
What We've Just Learned	246

Chapter 5 Controls and Control Styling	248
The Control Model for HTML, CSS, and JavaScript	249
HTML Controls	251
Extensions to HTML Elements	254
WinJS Controls	255
Syntax for data-win-options	259
WinJS Control Instantiation	261
Strict Processing and processAll Functions	262
Example: WinJS.UI.HtmlControl.....	263
Example: WinJS.UI.Rating (and Other Simple Controls)	264
Example: WinJS.UI.Tooltip	265
Example: WinJS.UI.ItemContainer	266
Working with Controls in Blend	269
Control Styling	272
Styling Gallery: HTML Controls	274
Styling Gallery: WinJS Controls	276
Some Tips and Tricks	284
Custom Controls	285
Implementing the Dispose Pattern	288
Custom Control Examples	289
Custom Controls in Blend	293
What We've Just Learned	297
Chapter 6 Data Binding, Templates, and Collections	298
Data Binding	299
Data Binding Basics.....	299
Data Binding in WinJS	301
Under the Covers: Binding mixins	311
Programmatic Binding and WinJS.Binding.bind.....	313

Binding Initializers	315
Binding Templates.....	319
Template Options, Properties, and Compilation	322
Collection Data Types	324
Windows.Foundation.Collection Types	325
WinJS Binding Lists	331
What We've Just Learned	342
Chapter 7 Collection Controls	344
Collection Control Basics.....	345
Quickstart #1: The WinJS Repeater Control with HTML controls	345
Quickstart #2: The FlipView Control Sample	349
Quickstart #3: The ListView Essentials Sample	351
Quickstart #4: The ListView Grouping Sample	353
ListView in the Grid App Project Template	357
The Semantic Zoom Control.....	361
How Templates Work with Collection Controls.....	364
Referring to Templates	364
Template Functions (Part 1): The Basics	365
Creating Templates from Data Sources in Blend	368
Repeater Features and Styling.....	372
FlipView Features and Styling.....	377
Collection Control Data Sources	380
The Structure of Data Sources (Interfaces Aplenty!).....	381
A FlipView Using the Pictures Library	384
Custom Data Sources and WinJS.UI.VirtualizedDataSource	386
ListView Features and Styling.....	393
When Is ListView the Right Choice?	393
Options, Selections, and Item Methods.....	395

Styling	399
Loading State Transitions	401
Drag and Drop	402
Layouts	405
Template Functions (Part 2): Optimizing Item Rendering	414
What We've Just Learned	419
Chapter 8 Layout and Views	421
Principles of Page Layout	423
Sizing, Scaling, and Views: The Many Faces of Your App	426
Variable View Sizing and Orientations	426
Screen Resolution, Pixel Density, and Scaling	437
Multiple Views	442
Pannable Sections and Styles	446
Laying Out the Hub	447
Laying Out the Sections	448
Panning Styles and Railing	449
Panning Snap Points and Limits	451
Zooming Snap Points and Limits	452
The Hub Control and Hub App Template	453
Hub Control Styling	460
Using the CSS Grid	461
Overflowing a Grid Cell	463
Centering Content Vertically	463
Scaling Font Size	464
Item Layout	465
CSS 2D and 3D Transforms	466
Flexbox	466
Nested and Inline Grids	467

Fonts and Text Overflow	468
Multicolumn Elements and Regions	470
What We've Just Learned	472
Chapter 9 Commanding UI.....	474
Where to Place Commands	475
The App Bar and Nav Bar	480
App Bar Basics and Standard Commands	481
App Bar Styling	490
Command Menus	494
Custom App Bars	495
Nav Bar Features	497
Nav Bar Styling	505
Flyouts and Menu	507
WinJS.UI.Flyout Properties, Methods, and Events	509
Flyout Examples	510
Menus and Menu Commands	513
Message Dialogs	518
Improving Error Handling in Here My Am!	519
What We've Just Learned	525
Chapter 10 The Story of State, Part 1: App Data and Settings.....	527
The Story of State.....	529
App Data Locations.....	532
App Data APIs (WinRT and WinJS)	533
Settings Containers.....	534
State Versioning	536
Folders, Files, and Streams.....	537
FileIO, PathIO, and WinJS Helpers (plus FileReader)	543
Encryption and Compression.....	544

Q&A on Files, Streams, Buffers, and Blobs.....	544
Using App Data APIs for State Management.....	552
Transient Session State	552
Local and Temporary State.....	553
IndexedDB, SQLite, and Other Database Options	555
Roaming State	556
Settings Pane and UI.....	559
Design Guidelines for Settings	561
Populating Commands.....	563
Implementing Commands: Links and Settings Flyouts	566
Programmatically Invoking Settings Flyouts.....	568
Here My Am! Update	570
What We've Just Learned	571
Chapter 11 The Story of State, Part 2: User Data, Files, and OneDrive	573
The Big Picture of User Data	574
Using the File Picker and Access Cache	579
The File Picker UI	580
The File Picker API	585
Access Cache.....	589
StorageFile Properties and Metadata	592
Availability	593
Thumbnails	594
File Properties	598
Media-Specific Properties	601
Folders and Folder Queries	607
KnownFolders and the StorageLibrary Object.....	609
Removable Storage	612
Simple Enumeration and Common Queries.....	613

Custom Queries.....	618
Metadata Prefetching with Queries	623
Creating Gallery Experiences.....	625
File Activation and Association.....	627
What We've Just Learned	632
Chapter 12 Input and Sensors	634
Touch, Mouse, and Stylus Input	635
The Touch Language and Mouse/Keyboard Equivalents	636
What Input Capabilities Are Present?	643
Unified Pointer Events	645
Gesture Events.....	649
The Gesture Recognizer	658
Keyboard Input and the Soft Keyboard	659
Soft Keyboard Appearance and Configuration	660
Adjusting Layout for the Soft Keyboard.....	663
Standard Keystrokes	666
Inking	667
Geolocation	669
Geofencing	673
Sensors.....	676
What We've Just Learned	680
Chapter 13 Media.....	681
Creating Media Elements.....	682
Graphics Elements: Img, Svg, and Canvas (and a Little CSS)	684
Additional Characteristics of Graphics Elements	688
Some Tips and Tricks	689
Rendering PDFs	694
Video Playback and Deferred Loading.....	699

Disabling Screen Savers and the Lock Screen During Playback	703
Video Element Extension APIs	703
Applying a Video Effect	705
Browsing Media Servers.....	706
Audio Playback and Mixing	706
Audio Element Extension APIs	708
Playback Manager and Background Audio	708
The Media Transport Control UI	714
Playing Sequential Audio	717
Playlists	719
Text to Speech	723
Loading and Manipulating Media	725
Image Manipulation and Encoding.....	726
Manipulating Audio and Video	732
Handling Custom Audio and Video Formats	735
Media Capture	742
Flexible Capture with the MediaCapture Object	744
Selecting a Media Capture Device.....	748
Streaming Media and Play To.....	751
Streaming from a Server and Digital Rights Management.....	751
Streaming from App to Network.....	753
Play To.....	754
What We Have Learned.....	757
Chapter 14 Purposeful Animations.....	759
Systemwide Enabling and Disabling of Animations	761
The WinJS Animations Library	762
Animations in Action	765
CSS Animations and Transitions	769

Designing Animations in Blend for Visual Studio	775
The HTML Independent Animations Sample	777
Rolling Your Own: Tips and Tricks	779
What We've Just Learned	785
Chapter 15 Contracts.....	786
Share	788
Share Source Apps.....	793
Share Target Apps	805
The Clipboard.....	816
Launching Apps with URI Scheme Associations	818
Search	823
The Search Charm UI	825
The WinJS.UI.SearchBox Control.....	829
Providing Query Suggestions	831
Providing Result Suggestions	835
SearchBox Styling	837
Indexing and Searching Content	840
The Search Contract	849
Contacts	850
Contact Cards.....	850
Using the Contact Picker	856
Appointments	860
What We've Just Learned	864
Chapter 16 Alive with Activity: Tiles, Notifications, the Lock Screen, and Background Tasks.....	865
Alive with Activity: A Visual Tour	866
The Four Sources of Updates and Notifications	875
Tiles, Secondary Tiles, and Badges	878

Secondary Tiles	880
Basic Tile Updates	887
Cycling, Scheduled, and Expiring Updates	900
Badge Updates	902
Periodic Updates	904
Creating an Update Service	907
Debugging a Service Using the Localhost.....	911
Windows Azure and Azure Mobile Services	912
Toast Notifications	917
Creating Basic Toasts	919
Butter and Jam: Options for Your Toast	921
Tea Time: Scheduled Toasts and Alarms.....	923
Toast Events and Activation	926
Push Notifications and the Windows Push Notification Service	927
Requesting and Caching a Channel URI (App)	929
Managing Channel URIs (Service)	931
Sending Updates and Notifications (Service)	932
Raw Notifications (Service).....	933
Receiving Notifications (App)	934
Debugging Tips	935
Tools and Providers for Push Notifications	935
Background Tasks and Lock Screen Apps.....	937
Background Tasks in the Manifest	938
Building and Registering Background Tasks	939
Conditions	941
Tasks for Maintenance Triggers	942
Tasks for System Triggers (Non-Lock Screen)	944
Lock Screen–Dependent Tasks and Triggers	945

Debugging Background Tasks	949
What We've Just Learned (Whew!)	950
Chapter 17 Devices and Printing	952
Declaring Device Access.....	956
Enumerating and Watching Devices	957
Scenario API Devices	962
Image Scanners	962
Barcode and Magnetic Stripe Readers (Point-of-Service Devices)	967
Smartcards.....	970
Fingerprint (Biometric) Readers.....	971
Bluetooth Call Control	972
Printing Made Easy.....	973
The Printing User Experience	974
Print Document Sources	977
Providing Print Content and Configuring Options.....	979
Protocol APIs: HID, USB, Bluetooth, and Wi-Fi Direct	981
Human Interface Devices (HID).....	983
Custom USB Devices	990
Bluetooth (RFCOMM)	992
Bluetooth Smart (LE/GATT)	996
Wi-Fi Direct.....	999
Near Field Communication and the Proximity API.....	1000
Finding Your Peers (No Pressure!)	1002
Sending One-Shot Payloads: Tap to Share	1007
What We've Just Learned	1009
Chapter 18 WinRT Components: An Introduction	1010
Choosing a Mixed Language Approach (and Web Workers).....	1012
Quickstarts: Creating and Debugging Components	1014

Quickstart #1: Creating a Component in C#	1015
Simultaneously Debugging Script and Managed/Native Code	1020
Quickstart #2: Creating a Component in C++	1021
Comparing the Results	1023
Key Concepts for WinRT Components	1026
Implementing Asynchronous Methods	1028
Projections into JavaScript	1042
Scenarios for WinRT Components	1044
Higher Performance (Perhaps)	1044
Access to Additional APIs	1047
Obfuscating Code and Protecting Intellectual Property	1051
Concurrency	1052
Library Components	1053
What We've Just Learned	1056
Chapter 19 Apps for Everyone, Part 1: Accessibility and World-Readiness	1058
Accessibility	1059
Screen Readers and Aria Attributes	1063
Handling Contrast Variations	1068
World Readiness and Localization	1075
Globalization	1077
Preparing for Localization	1087
Creating Localized Resources: The Multilingual App Toolkit	1101
Localization Wrap-Up	1108
What We've Just Learned	1109
Chapter 20 Apps for Everyone, Part 2: The Windows Store	1110
Your App, Your Business	1111
Planning: Can the App Be a Windows Store App?	1113
Planning for Monetization (or Not)	1114

Growing Your Customer Base and Other Value Exchanges	1125
Measuring and Experimenting with Revenue Performance	1126
The Windows Store APIs	1127
The CurrentAppSimulator Object.....	1130
Trial Versions and App Purchase.....	1133
Listing and Purchasing In-App Products.....	1137
Handling Large Catalogs	1145
Receipts	1146
Instrumenting Your App for Telemetry and Analytics	1148
Releasing Your App to the World.....	1155
Promotional Screenshots, Store Graphics, and Text Copy	1156
Testing and Pre-Certification Tools	1158
Creating the App Package	1159
Onboarding and Working through Rejection	1163
App Updates	1166
Getting Known: Marketing, Discoverability, and the Web	1168
Connecting Your Website and Web-Mapped Search Results	1170
Face It: You're Running a Business!	1171
Look for Opportunities.....	1172
Invest in Your Business	1172
Fear Not the Marketing	1172
Support Your Customers	1173
Plan for the Future	1173
Selling Your App When It's Not Running.....	1174
You're Not Alone	1175
Final Thoughts: Qualities of a Rock Star App	1175
What We've Just Learned	1176
Appendix A Demystifying Promises.....	1178

What Is a Promise, Exactly? The Promise Relationships	1178
The Promise Construct (Core Relationship)	1181
Example #1: An Empty Promise!	1183
Example #2: An Empty Async Promise.....	1185
Example #3: Retrieving Data from a URI.....	1186
Benefits of Promises	1187
The Full Promise Construct	1188
Nesting Promises.....	1192
Chaining Promises	1195
Promises in WinJS (Thank You, Microsoft!)	1200
The WinJS.Promise Class.....	1201
Originating Errors with WinJS.Promise.WrapError	1203
Some Interesting Promise Code	1204
Delivering a Value in the Future: WinJS.Promise.timeout	1204
Internals of WinJS.Promise.timeout	1205
Parallel Requests to a List of URIs.....	1205
Parallel Promises with Sequential Results	1206
Constructing a Sequential Promise Chain from an Array.....	1208
PageControlNavigator._navigating (Page Control Rendering)	1208
Bonus: Deconstructing the ListView Batching Renderer	1210
Appendix B WinJS Extras	1214
Exploring WinJS.Class Patterns	1214
WinJS.Class.define	1214
WinJS.Class.derive	1217
Mixins.....	1218
Obscure WinJS Features	1219
Wrappers for Common DOM Operations	1219
WinJS.Utilities.data, convertToPixels, and Other Positional Methods	1221

WinJS.Utilities.empty, eventWithinElement, and getMember	1222
WinJS.UI.scopedSelect and getItemsFromRanges	1222
Extended Splash Screens	1223
Adjustments for View Sizes	1229
Custom Layouts for the ListView Control	1231
Minimal Vertical Layout	1233
Minimal Horizontal Layout	1235
Two-Dimensional and Nonlinear Layouts	1239
Virtualization	1241
Grouping.....	1243
The Other Stuff	1244
Appendix C Additional Networking Topics.....	1249
XMLHttpRequest and WinJS.xhr	1249
Tips and Tricks for WinJS.xhr	1250
Breaking Up Large Files (Background Transfer API)	1251
Multipart Uploads (Background Transfer API)	1252
Notes on Encryption, Decryption, Data Protection, and Certificates	1255
Syndication: RSS, AtomPub, and XML APIs in WinRT	1255
Reading RSS Feeds.....	1256
Using AtomPub	1259
Sockets.....	1260
Datagram Sockets.....	1261
Stream Sockets.....	1265
Web Sockets: MessageWebSocket and StreamWebSocket.....	1268
The ControlChannelTrigger Background Task	1273
The Credential Picker UI	1273
Other Networking SDK Samples	1277
Appendix D Provider-Side Contracts	1279

File Picker Providers	1279
Manifest Declarations.....	1280
Activation of a File Picker Provider.....	1281
Cached File Updater	1288
Updating a Local File: UI.....	1291
Updating a Remote File: UI	1292
Update Events.....	1294
Contact Cards Action Providers	1297
Contact Picker Providers.....	1300
Appointment Providers.....	1303
About the Author	1309

Introduction

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

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

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

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

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

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

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

rather than hundreds or thousands. It would also enable direct native app development in multiple programming languages. This meant that new operating system capabilities would surface to those developers without having to wait for an update to some intermediate framework. It also meant that developers who had experience in any one of those language choices would find a natural home when writing apps for Windows 8 and Windows 8.1.

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

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

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

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

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

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

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

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

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

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

Who This Book Is For

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

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

endeavored to make this book comprehensive, helping you at least be aware of what's possible and where optimizations can be made.

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

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

What You'll Need (Can You Say “Samples”?)

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

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

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

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

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

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

<http://aka.ms/BrockschmidtBook2/CompContent>

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

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

<http://aka.ms/BrockschmidtBook2/Videos>

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

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

A Formatting Note

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

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

Acknowledgements

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

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

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

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

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

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

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

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

Finally, special hugs to my wife Kristi and our son Liam (now seven and a half), who have lovingly

been there the whole time and who don't mind my traipsing through the house to my office either late at night or early in the morning.

Free Ebooks from Microsoft Press

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

<http://aka.ms/mspressfree>

The “Microsoft Press Guided Tours” App

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

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

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

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

Errata & Book Support

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

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

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

We Want to Hear from You

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

<http://aka.ms/tellpress>

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

Stay in Touch

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

Chapter 3

App Anatomy and Performance Fundamentals

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

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

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

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

This chapter, then, is about those fundamentals: the core foundational structure of an app upon which you can build something that looks beautiful *and* really works well. This takes us first into the subject of app activation (how apps get running and get running quickly) and then app lifecycle transitions (how they are suspended, resumed, and terminated). We'll then look at page navigation

within an app, working with promises, async debugging, and making use of various profiling tools. One subject that we won't talk about here are background tasks; we'll see those in Chapter 16, "Alive with Activity," because there are limits to their use and they are best discussed in the context of the lock screen.

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

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

App Activation

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

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

Branding Your App 101: The Splash Screen and Other Visuals

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

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

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

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

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

Manifest Editor Tab	Text Item or Setting	Use
---------------------	----------------------	-----

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

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

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

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

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

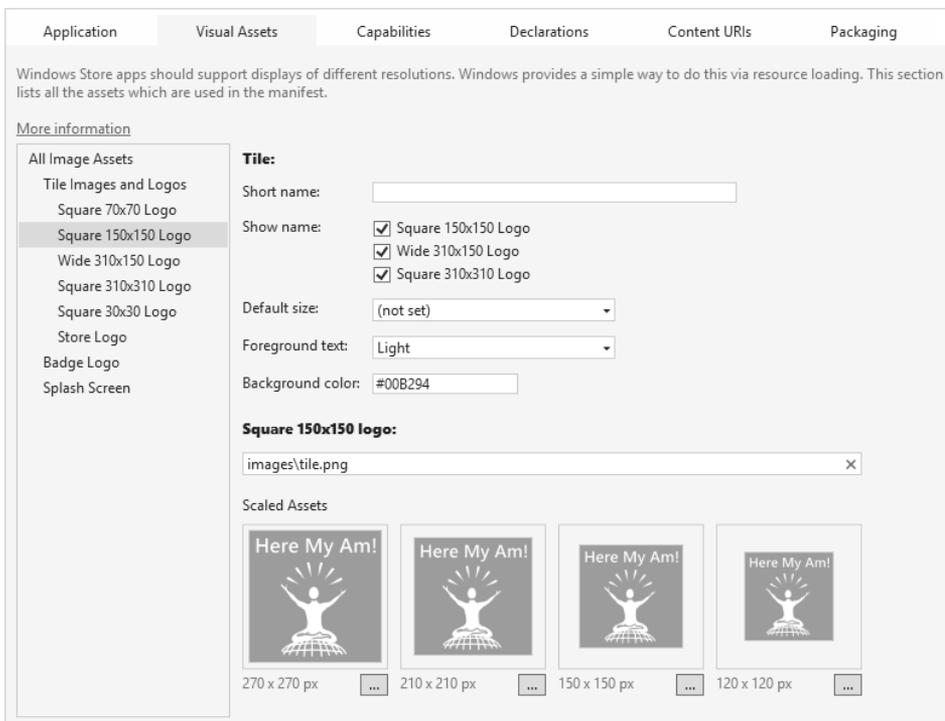


FIGURE 3-1 Visual Studio’s Visual Assets tab of the manifest editor. It automatically detects whether a scaled asset exists for the base filename (such as images\tile.png).

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

details.

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



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

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

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

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

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

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

Activation Event Sequence

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

1. Windows displays the default splash screen using information from the app manifest (except for pre-launching).
2. Windows launches the app host, identifying the app's installation folder and the name of the app's Start Page (an HTML file) as indicated in the Application tab of the manifest editor.¹⁷
3. The app host loads that page's HTML, which in turn loads referenced stylesheets and script (deferring script loading if indicated in the markup with the `defer` attribute). Here it's important that all files are properly encoded for best startup performance. (See the sidebar on the next page.)
4. `document.DOMContentLoaded` fires. You can use this to do early initialization specifically related to the DOM, if desired. This is also the place to perform one-time initialization work that should not be done if the app is activated on multiple occasions during its lifetime.
5. `window.onload` fires. This generally means that document layout is complete and elements will reflect their actual dimensions. (Note: In Windows 8 this event occurs at the end of this

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

list instead.)

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

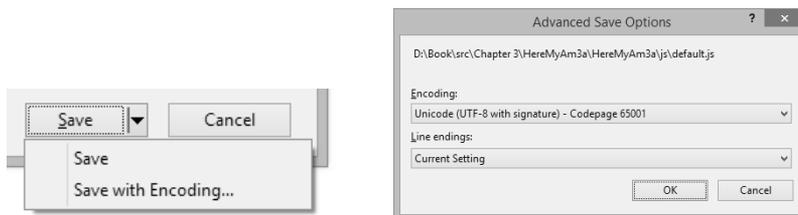
Once the `activated` event handler returns, the default splash screen is dismissed unless the app has requested a deferral, as discussed later in the "Activation Deferrals and `setPromise`" section. With the latter four events, your app's handling of these very much determines how quickly it comes up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

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

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

Sidebar: File Encoding for Best Startup Performance

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



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

Activation Code Paths

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

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

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

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

    app.oncheckpoint = function (args) {
    };

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

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

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

- `args.detail.previousExecutionState` identifies the state of the app prior to this activation, which determines whether to reload session state.
- `WinJS.UI.processAll` instantiates WinJS controls—that is, elements that contain a `data-win-control` attribute, as we’ll cover in Chapter 5, “Controls and Control Styling.”
- `args.setPromise` instructs Windows to wait until `WinJS.UI.processAll` is complete before removing the splash screen. (See “Activation Deferrals and `setPromise`” later in this chapter.)
- `app.oncheckpoint`, which is assigned an empty function, is something we’ll cover in the “App Lifecycle Transition Events” section later in this chapter.
- `app.start()` (`WinJS.Application.start()`) initiates processing of events that WinJS queues during startup.

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

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

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

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

<code>previousExecutionState</code>	<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>	<code>SplashScreen</code>	Contains an <code>onDismissed</code> event for when the system splash screen is dismissed along with an <code>imageLocation</code> property (<code>Windows.Foundation.Rect</code>) with coordinates where the splash screen image was displayed. For use of this, see "Extended Splash Screens" in Appendix B, "WinJS Extras."

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

WinJS.Application Events

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

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

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

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

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

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

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

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

Activation Deferrals and `setPromise`

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

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

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

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

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

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

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

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

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

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

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

```
var app = WinJS.Application;

app.onactivated = function (args) {
    if (args.detail.kind === activation.ActivationKind.launch) {
```

```

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

```

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

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

```

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

```

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

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

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

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

Optimizing Startup Time

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

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

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

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

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

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

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

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

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

Review all the resources that your markup references as well, and place any critical ones directly into the app package where you can reference them with `ms-appx:///` URIs. Any remote resources will, of

course, require a round trip to the network with possible connectivity failures. Where making HTTP requests is unavoidable, suggest your most critical URIs to the `Windows.Networking.BackgroundTransfer.ContentPrefetcher` object (see “Prefetching Content” in Chapter 4). If the prefetcher determines that those URIs are among the top requests, it will actively cache requests to those URIs such that requests from your code will draw directly from that cache. This won’t help the app the first time it’s run, but it can help with subsequent activations.

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

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

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

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

WinRT Events and `removeEventListener`

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

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

```
var myNumber = 1;
```

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

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

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

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

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

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

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

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

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

App Lifecycle Transition Events and Session State

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

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

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

For *view state changes*, an app can detect these in several ways. As shown in the Here My Am! example, an app typically uses media queries (in declarative CSS or in code through media query listeners) to reconfigure layout and visibility of elements, which is really all that view states should affect. (Again, view state changes never change the mode of the app.) At any time, an app can also retrieve its view state through `Windows.UI.ViewManagement.ApplicationView.orientation`

(returning an `ApplicationViewOrientation` value of either `portrait` or `landscape`), the size of the app window, and other details from `ApplicationView` like `isFullScreen`; details in Chapter 8, “Layout and Views.”¹⁹

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

Suspend, Resume, and Terminate

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

- **Suspending** When an app is not visible in any view state, it will be suspended after five seconds (according to the wall clock) to conserve battery power. This means it remains wholly in memory but won’t be scheduled for CPU time and thus won’t have network or disk activity (except when using specifically allowed background tasks, discussed in Chapter 16). When this happens, the app receives the `Windows.UI.WebUI.WebUIApplication.onsuspending` event, which is also exposed through `WinJS.Application.oncheckpoint`. Apps must return from this event within the five-second period, or Windows will assume the app is hung and terminate it (period!). During this time, apps save transient session state and should also release any exclusive resources acquired as well, like file streams or device access. (See [How to suspend an app](#).) If you need to do async work in the `suspending` handler, WinRT provides a deferral object as with activation and WinJS provides the `setPromise` equivalent. Using the deferral will not, however, extend the suspension deadline.
- **Resuming** If the user switches back to a suspended app, it receives the `Windows.UI.WebUI.WebUIApplication.onresuming` event. This is *not* surfaced through `WinJS.Application`, mind you, because WinJS has no value to add, but it’s easy enough to use `WinJS.Application.queueEvent` for this purpose. We’ll talk more about this event in coming chapters, as it’s used to refresh any data that might have changed while the app was suspended. For example, if the app is connected to an online service, it would refresh that content if enough time has passed while the app was suspended, as well as check connectivity status (Chapter 4). In addition, if you’re tracking sensor input of any kind (like compass, geolocation, or orientation, see Chapter 12, “Input and Sensors”), resuming is a good time to get a fresh reading. You’ll also want to check license status for your app and in-app purchases if you’re using trials and/or expirations (see Chapter 20). There are also times when you might want to refresh your layout (as we’ll see in Chapter 8), because it’s possible for your app to resume

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

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

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

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

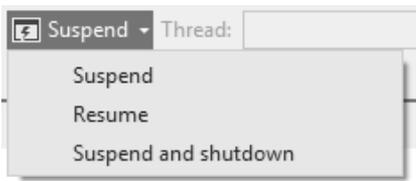


FIGURE 3-2 The Visual Studio toolbar drop-down to simulate suspend, resume, and terminate.

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

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

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

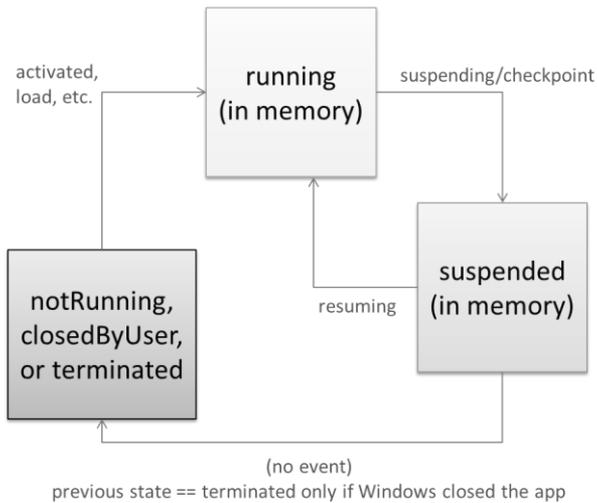


FIGURE 3-3 Process lifecycle events and `previousExecutionState` values.

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

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

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

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

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

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

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

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

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

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

Finally, if you don't want to use the `sessionState` object or you have state that won't work with it,

the `WinJS.Application` object makes it easy to write your own files without having to use async WinRT APIs. Specifically, it provides (as shown in the [documentation](#)) `local`, `temp`, and `roaming` objects that each have methods called `readText`, `writeText`, `exists`, and `remove`. These objects each work within their respective appdata folders and provide a simplified API for file I/O, as shown in scenario 1 of the [App model sample](#).

Suspending Deferrals and Deadlines

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

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

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

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

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

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

Basic Session State in Here My Am!

To demonstrate some basic handling of session state, I've made a few changes to Here My Am! as given in the HereMyAm3b example in the companion content. Here we have two pieces of information

we care about: the variables `lastCapture` (a `StorageFile` with the image) and `lastPosition` (a set of coordinates). We want to make sure we save these when we get suspended so that we can properly apply those values when the app gets launched with the previous state of `terminated`.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

updatePosition();
```

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

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

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

Page Controls and Navigation

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

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

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

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

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

WinJS Tools for Pages and Page Navigation

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

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

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

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

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

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

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

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

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

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

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

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

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

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

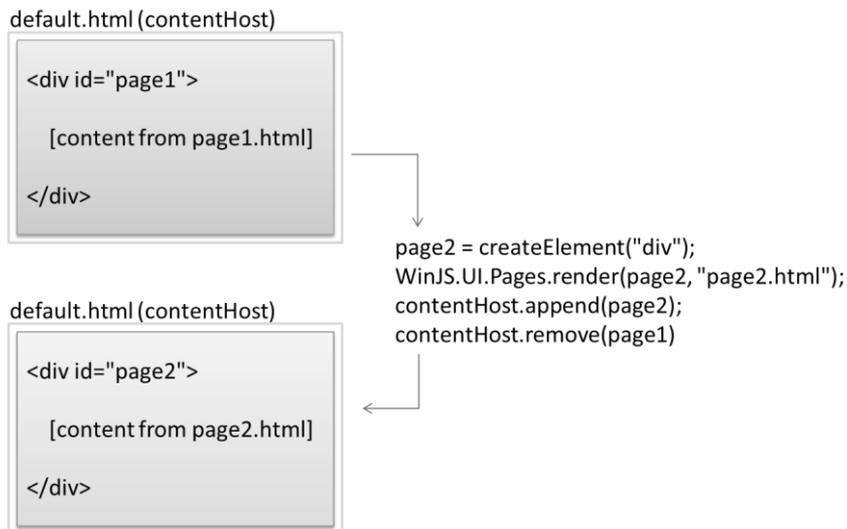


FIGURE 3-4 Performing page navigation in the context of a single host (typically `default.html`) by replacing appending the content from `page2.html` and removing that from `page1.html`. Typically, each page occupies the whole display area, but page controls can just as easily be used for smaller areas.

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

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

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

The Navigation App Template, PageControl Structure, and PageControlNavigator

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

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

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

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

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

```

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

```

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

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

```

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

```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

    WinJS.Namespace.define("Application", {
        PageControlNavigator: WinJS.Class.define(
```

```

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

  this.home = options.home;

  // ...

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

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

```

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

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

```

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

```

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

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

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

elements to the navigator control:

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

    this._lastNavigationPromise.cancel();

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

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

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

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

And that, my friends, is how it works! In addition to the [HTML Page controls sample](#), and to show a concrete example of doing this in a real app, the code in the `HereMyAm3c` sample has been converted to use this model for its single home page. To make this conversion, I started with a new project by using the `NavigationApp` template to get the page navigation structures set up. Then I copied or imported the relevant code and resources from `HereMyAm3b`, primarily into `pages/home/home.html`,

home.js, and home.css. And remember how I said that you could open a page control directly in Blend (which is why pages have WinJS references)? As an exercise, open the HereMyAm3c project in Blend. You'll first see that everything shows up in default.html, but you can also open home.html by itself and edit just that page.

Note To give an example of calling `removeEventListener` for the WinRT `datarequested` event, I make this call in the `unload` method of `pages/home/home.js`.

Be aware that WinJS calls `WinJS.UI.processAll` in the process of loading a page control (before calling the `processed` method), so we don't need to concern ourselves with that detail when using WinJS controls in a page. On the other hand, reloading state when `previousExecutionState == terminated` needs some attention. Because this is picked up in the `WinJS.Application.onactivated` event *before* any page controls are loaded and before the `PageControlNavigator` is even instantiated, we need to remember that condition so that the home page's `ready` method can later initialize itself accordingly from `app.sessionState` values. For this I simply write another flag into `app.sessionState` called `initFromState` (true if `previousExecutionState` is terminated, false otherwise.) The page initialization code, now in the page's `ready` method, checks this flag to determine whether to reload session state.

The other small change I made to HereMyAm3c is to use the `updateLayout` method in the page control rather than attaching my own handler to `window.onresize`. With this I also needed to add a `height: 100%;` style to the `#mainContent` rule in `home.css`. In previous iterations of this example, the `mainContent` element was a direct child of the `body` element and it inherited the full screen height automatically. Now, however, it's a child of the `contentHost`, so the height doesn't automatically pass through and we need to set it to 100% explicitly.

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

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

Here's the syntax: `var ns = WinJS.Namespace.define(<name>, <members>)` where `<name>` is a string (dots are OK) and `<members>` is any object contained in `{}`'s. Also, `WinJS.Namespace.defineWithParent(<parent>, <name>, <members>)` defines one within the `<parent>` namespace.

If you call `WinJS.Namespace.define` for the same `<name>` multiple times, the `<members>` are

combined. Where collisions are concerned, the most recently added members win. For example:

```
WinJS.Namespace.define("MyNamespace", { x: 10, y: 10 });
WinJS.Namespace.define("MyNamespace", { x: 20, z: 10 });
//MyNamespace == { x: 20, y: 10, z: 10}
```

`WinJS.Class.define` is, for its part, a shortcut for the object pattern, defining a constructor so that objects can be instantiated with `new`.

Syntax: `var className = WinJS.Class.define(<constructor>, <instanceMembers>, <staticMembers>)` where `<constructor>` is a function, `<instanceMembers>` is an object with the class's properties and methods, and `<staticMembers>` is an object with properties and methods that can be directly accessed via `<className>.<member>` (without using `new`).

Variants: `WinJS.Class.derive(<baseClass>, ...)` creates a subclass (... is the same arg list as with `define`) using prototypal inheritance, and `WinJS.Class.mix(<constructor>, [<classes>])` defines a class that combines the instance (but not static) members of one or more other `<classes>` and initializes the object with `<constructor>`.

Finally, note that because class definitions just generate an object, `WinJS.Class.define` is typically used inside a module with the resulting object exported to the rest of the app as a namespace member. Then you can use `new <namespace>.<class>` anywhere in the app.

For more details on classes in WinJS, see Appendix B.

Sidebar: Helping Out IntelliSense

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

The Navigation Process and Navigation Styles

Having seen how page controls, `WinJS.UI.Pages`, `WinJS.Navigation`, and the `PageControlNavigator` all relate, it's straightforward to see how to navigate between multiple pages within the context of a single HTML container (e.g., `default.html`). With the `PageControlNavigator` instantiated and a page control defined via `WinJS.UI.Pages`, simply call `WinJS.Navigation.navigate` with the URI of that page control (its identifier). This loads that page's contents into a child element inside the

`PageControlNavigator`, unloading any previous page. That becomes page visible, thereby “navigating” to it so far as the user is concerned. You can also use (like the WinJS `BackButton` does) the other methods of `WinJS.Navigation` to move forward and back in the nav stack, which results in page contents being added and removed. The `WinJS.Navigation.canGoBack` and `canGoForward` properties allow you to enable/disable navigation controls as needed. Just remember that all the while, you’ll still be in the overall context of your host page where you created the `PageControlNavigator` control.

As an example, create a new project using the Grid App template and look at these particular areas:

- **pages/groupedItems/groupedItems** is the home or “hub” page. It contains a `ListView` control (see Chapter 6, “Data Binding, Templates, and Collections”) with a bunch of default items.
- Tapping a group header in the list navigates to section page (**pages/groupDetail**). This is done in `pages/groupedItems/groupedItems.html`, where an inline `onClick` handler event navigates to `pages/groupDetail/groupDetail.html` with an argument identifying the specific group to display. That argument comes into the `ready` function of `pages/groupDetail/groupDetail.js`.
- Tapping an item on the hub page goes to detail page (**pages/itemDetail**). The `itemInvoked` handler for the items, the `_itemInvoked` function in `pages/groupedItems/groupedItem.js`, calls `WinJS.Navigation.navigate("/pages/itemDetail/itemDetail.html")` with an argument identifying the specific item to display. As with groups, that argument comes into the `ready` function of `pages/itemDetail/itemDetail.js`.
- Tapping an item in the section page also goes to the details page through the same mechanism—see the `_itemInvoked` function in `pages/groupDetail/groupDetail.js`.
- The back buttons on all pages wire themselves into `WinJS.Navigation.back` for keyboard, mouse, and touch events.

The Split App template works similarly, where each list item on `pages/items` is wired to navigate to `pages/split` when invoked. Same with the Hub App template that has a hub page using the `WinJS.UI.Hub` control that we’ll meet in Chapter 8.

The Grid App and Hub App templates also serve as examples of what’s called the *Hub-Section-Item* navigation style (it’s most explicitly so in the Hub App). Here the app’s home page is the hub where the user can explore the full extent of the app. Tapping a group header navigates to a section, the second level of organization where only items from that group are displayed. Tapping an item (in the hub or in the section) navigates to a details page for that item. You can, of course, implement this navigation style however you like; the Grid App template uses page controls, `WinJS.Navigation`, and the `PageControlNavigator`. (Semantic zoom, as we’ll see in Chapter 7, “Collection Controls,” is also supported as a navigation tool to switch between hubs and sections.)

An alternate navigation choice is the *Flat* style, which simply has one level of hierarchy. Here, navigation happens to any given page at any time through a *navigation bar* (swiped in along with the app bar, as we’ll see in Chapter 9). When using page controls and `PageControlNavigator`, navigation

commands or buttons can just invoke `WinJS.Navigation.navigate` for this purpose. Note that in this style, there typically is no back button: users are expected to always swipe in the navigation bar from the top and go directly to the desired page.

These styles, along with many other UI aspects of navigation, can be found on [Navigation design for Windows Store apps](#). This is an essential topic for designers.

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

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

Typically, such pages appear only the first time the app is run. If the user provides a valid login, or if you obtain an access token through the Web Authentication Broker (see Chapter 4), those credentials/token can be saved for later use via the [Windows.Security.Credentials.PasswordVault](#) API. If the user accepts a EULA, that fact should be saved in appdata and reloaded anytime the app needs to check. These settings (login and acceptance of a license) should then always be accessible through the app's Settings charm. Legal notices, by the way, as well as license agreements, should always be accessible through Settings as well. See [Guidelines and checklist for login controls](#).

Optimizing Page Switching: Show-and-Hide

Even with page controls, there is still a lot going on when navigating from page to page: one set of elements is removed from the DOM, and another is added in. Depending on the pages involved, this can be an expensive operation. For example, if you have a page that displays a list of hundreds or thousands of items, where tapping any item goes to a details page (as with the Grid App template), hitting the back button from a detail page will require complete reconstruction of the list (or at least its visible parts if the list is virtualized, which could still take a long time).

Showing progress indicators can help alleviate the user's anxiety, of course, but users are notoriously impatient and will likely want to quickly switch between a list of items and item details. (You've probably already encountered apps that seem to show progress indicators all the time for just about everything—how do they make you feel?) Indeed, the recommendation is that switching between fully interactive pages takes a quarter second or less, if possible, and no more than half a second. In some cases, completely swapping out chunks of the DOM with page controls will just become too time-consuming. (You could use a split master-detail view, of course, but that means splitting the available screen real estate.)

A good alternative is to actually keep the list/master page fully loaded the whole time. Instead of navigating to the item details page in the way we've seen, simply render that details page (using `WinJS.UI.Pages.render` directly) into another `div` that occupies the whole screen and overlays the list (similar to what we do with an extended splash screen), and then make that `div` visible *without* removing the list page from the DOM. When you dismiss the details page, just hide its `div`. This way you get the same effect as navigating between pages but the whole process is much quicker. You can also apply WinJS animations like [enterContent](#) and [exitContent](#) to make the transition more fluid.

If necessary, you can clear out the details `div` by just setting its `innerHTML` to `""`. However, if each details page has the same structure for every item, you can leave it entirely intact. When you "navigate" to the next details page, you would go through and refresh each element's data and properties for the new item before making that page visible. This could be significantly faster than rebuilding the details page all over again.

Note that because the `PageControlNavigator` implementation in `navigator.js` is provided by the templates and becomes part of your app, you can modify it however you like to handle these kinds of optimizations in a more structured manner that's transparent to the rest of your code.

Page-Specific Styling

When creating an app that uses page controls, you'll end up with each page having its own `.css` file in which you place page-specific styles. What's very important to understand here, though, is that while each page's HTML elements are dynamically added to and removed from the DOM, *any and all CSS that is loaded for page controls is cumulative to the app as a whole*. That is, styles behave like script and are preserved across page "navigations." This can be a source of confusion and frustration, so it's essential to understand what's happening here and how to work with it.

Let's say the app's root page is `default.html` and its global styles are in `css/default.css`. It then has several page controls defined in `pages/page1` (`page1.html`, `page1.js`, `page1.css`), `pages/page2` (`page2.html`, `page2.js`, `page2.css`), and `pages/page3` (`page3.html`, `page3.js`, `page3.css`). Let's also say that `page1` is the "home" page that's loaded at startup. This means that the styles in `default.css` and `page1.css` have been loaded when the app first appears.

Now the user navigates to `page2`. This causes the contents of `page1.html` to be dumped from the DOM, *but its styles remain in the stylesheet*. So when `page2` is loaded, `page2.css` gets added to the overall stylesheet as well, and any styles in `page2.css` that have identical selectors to `page1.css` will overwrite those in `page1.css`. And when the user navigates to `page3` the same thing happens again: the styles in `page3.css` are added in and overwrite any that already exist. But so far we haven't seen any unexpected effect of this.

Now, say the user navigates back to `page1`. Because the apphost's rendering engine has already loaded `page1.css` into the stylesheet, `page1.css` won't be loaded again. This means that any styles that were overwritten by other pages' stylesheets will not be reset to those in `page1.css`—basically you get whichever ones were loaded most recently. As a result, you can see some mix of the styles in `page2.css`

and page3.css being applied to elements in page1.²⁴

There are two ways to handle CSS files to avoid these problems. The first way is to take steps to avoid colliding selectors: use unique selectors for each page or can scope your styles to each page specifically. For the latter, wrap each page's contents in a top-level `div` with a unique class (as in `<div class="page1">`) so that you can scope every rule in page1.css with the page name. For example:

```
.page1 p {
  font-weight: bold;
}
```

Such a strategy can also be used to define stylesheets that are shared between pages, as with implementing style themes. If you scope the theme styles with a theme class, you can include that class in the top-level `div` to apply the theme.

A similar case arises if you want to use the `ui-light.css` and `ui-dark.css` WinJS stylesheets in different pages of the same app. Here, whichever one is loaded second will define the global styles such that subsequent pages that refer to `ui-light.css` might appear with the dark styles.

Fortunately, WinJS already scopes those styles that differ between the two files: those in `ui-light.css` are scoped with a CSS class `win-ui-light` and those in `ui-dark.css` are scoped with `win-ui-dark`. This means you can just refer to whichever stylesheet you use most often in your `.html` files and then add either `win-ui-light` or `win-ui-dark` to those elements that you need to style differently. When you add either class, note that the style will apply to that element and all its children. For a simple demonstration of an app with one dark page (as the default) and one light page, see the [PageStyling example](#) in the companion content.

The other way of avoiding collisions is to specifically unload and reload CSS files by modifying `<link>` tags in the page header. You can either remove one `<link>` tag and add a different one, toggle the `disabled` attribute for a tag between `true` and `false`, or change the `href` attribute of an existing link. These methods are demonstrated for styling an `iframe` in the [CSS styling and branding your app sample](#), which swaps out and enables/disables both WinJS and app-specific stylesheets. Another demonstration for switching between the WinJS stylesheets is in scenario 1 of the [HTML NavBar control sample](#) that we'll see more of in Chapter 9 (`js/1-CreateNavBar.js`):

```
function switchStyle() {
  var linkEl = document.querySelector('link');
  if (linkEl.getAttribute('href') === "//Microsoft.WinJS.2.0 /css/ui-light.css") {
    linkEl.setAttribute('href', "//Microsoft.WinJS.2.0 /css/ui-dark.css");
  } else {
    linkEl.setAttribute('href', "//Microsoft.WinJS.2.0 /css/ui-light.css");
  }
}
```

²⁴ The same thing happens with `.js` files, by the way, which are not reloaded if they've been loaded already. To avoid collisions in JavaScript, you either have to be careful to not duplicate variable names or to use namespaces to isolate them from one another.

The downside of this approach is that every switch means reloading and reparsing the CSS files and a corresponding re-rendering of the page. This isn't much of an issue during page navigation, but given the size of the WinJS files I recommend using it only for your own page-specific stylesheets and using the `win-ui-light` and `win-ui-dark` classes to toggle the WinJS styles.

Async Operations: Be True to Your Promises

Even though we've just got our first apps going, we've already seen a lot to do with async operations and promises. We've seen their basic usage, and in the "Moving the Captured Image to AppData (or the Pictures Library)" section of Chapter 2, we saw how to combine multiple async operations into a sequential chain. At other times you might want to combine multiple parallel async operations into a single promise. Indeed, as you progress through this book you'll find that async APIs, and thus promises, seem to pop up as often as dandelions in a lawn (without being a noxious weed, of course)! Indeed, the implementation of the `PageControlNavigator._navigating` method that we saw earlier has a few characteristics that are worth exploring.

To reiterate a very important point, promises are simply how async operations in WinRT are projected into JavaScript, which matches how WinJS and other JavaScript libraries typically handle asynchronous work. And because you'll be using all sorts of async APIs in your development work, you're going to be using promises quite frequently and will want to understand them deeply.

Note There are a number of different specifications for promises. The one presently used in WinJS and the WinRT API is known as [Common JS/Promises A](#). Promises in jQuery also follow this convention and are thus interoperable with WinJS promises.

The subject of promises gets rather involved, however, so instead of burdening you with the details in the main flow of this chapter, you'll find a full treatment of promises in Appendix A, "Demystifying Promises." Here I want to focus on the most essential aspects of promises and async operations that we'll encounter throughout the rest of this book, and we'll take a quick look at the features of the `WinJS.Promise` class. Examples of the concepts can be found in the [WinJS Promise sample](#).

Using Promises

The first thing to understand about a promise is that it's really nothing more than a code construct or a calling convention. As such, promises have no inherent relationship to async operations—they just so happen to be very *useful* in that regard! A promise is simply an object that represents a value that might be available at some point in the future (or might be available already). It's just like we use the term in human relationships. If I say to you, "I promise to deliver a dozen donuts," it doesn't matter when and how I get them (or even whether I have them already in hand), it only matters that I deliver them at some point in the future.

A promise, then, implies a relationship between two people or, to be more generic, two *agents*, as I

call them. There is the *originator* who makes the promise—that is, the one who has some goods to deliver—and the *consumer* or recipient of that promise, who will also be the later recipient of the goods. In this relationship, the originator creates a promise in response to some request from the consumer (typically an API call). The consumer can then do whatever it wants with both the promise itself and whatever goods the promise delivers. This includes sharing the promise with other interested consumers—the promise will deliver its goods to each of them.

The way a consumer listens for delivery is by subscribing a *completed handler* through the promise's `then` or `done` methods. (We'll discuss the differences later.) The promise invokes this handler when it has obtained its results. In the meantime, the consumer can do other work, which is exactly why promises are used with async operations. It's like the difference between waiting in line at a restaurant's drive-through for a potentially very long time (the synchronous model) and calling out for pizza delivery (the asynchronous model): the latter gives you the freedom to do other things.

Of course, if the promised value is already available, there's no need to wait: it will be delivered synchronously to the completed handler as soon as `then/done` is called.

Similarly, problems can arise that make it impossible to fulfill the promise. In this case the promise will invoke any *error handlers* given to `then/done` as the second argument. Those handlers receive an error object containing `name` and `message` properties with more details, and after this point the promise is in what's called the *error state*. This means that any subsequent calls to `then/done` will immediately (and synchronously) invoke any given error handlers.

A consumer can also cancel a promise if it decides it no longer needs the results. A promise has a `cancel` method for this purpose, and calling it both halts any underlying async operation represented by the promise (however complex it might be) and puts the promise into the error state.

Some promises—which is to say, some async operations—also support the ability to report intermediate results to any *progress handlers* given to `then/done` as the third argument. Check the documentation for the particular API in question.²⁵

Finally, two static methods on the `WinJS.Promise` object might come in handy when using promises:

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

²⁵ If you want to impress your friends while reading the WinRT API documentation, know that if an async function shows it returns `IAsync[Action | Operation]WithProgress` (for whatever result type), it will invoke progress handlers. If it lists only `IAsync[Action | Operation]`, progress is not supported.

Tip If you're new to the concept of *static methods*, these refer to functions that exist on an object class that you call directly through the fully-qualified name, such as `WinJS.Promise.thenEach`. These are distinct from *instance methods*, which must be called through a specific instance of the class. For example, if you have a `WinJS.Promise` object in the variable `p`, you cancel that particular instance with `p.cancel()`.

Joining Parallel Promises

Because promises are often used to wrap asynchronous operations, it's certainly possible that you can have multiple operations going on in parallel. In these cases you might want to know either when one promise in a group is fulfilled or when all the promises in the group are fulfilled. The static functions `WinJS.Promise.any` and `WinJS.Promise.join` provide for this. Here's how they compare:

Function	<code>any</code>	<code>join</code>
Arguments	An array of promises	An array of promises
Fulfilled when	One of the promises is fulfilled (a logical OR)	All of the promises are fulfilled (a logical AND)
Fulfilled result	This is a little odd. It's an object whose <code>key</code> property identifies the promise that was fulfilled and whose <code>value</code> property is an object containing that promise's state. Within that state is a <code>_value</code> property that contains the actual result of that promise.	This isn't clearly documented but can be understood from the source code or simple tests from the consumer side. If the promises in the <code>join</code> all complete, the completed handler receives an array of results from the individual promises (even if those results are <code>null</code> or <code>undefined</code>). If there's an error in the <code>join</code> , the error object passed to the error handler is an array that contains the individual errors.
Progress behavior	None	Reports progress to any subscribed handlers where the intermediate results are an array of results from those individual promises that have been fulfilled so far.
Behavior after fulfillment	All the operations for the remaining promises continue to run, calling whatever handlers might have been subscribed individually.	None—all promises have been fulfilled.
Behavior upon cancellation	Canceling the promise from <code>any</code> cancels all promises in the array, even if the first has already been fulfilled.	Cancels all other promises that are still pending.
Behavior upon errors	Invokes the subscribed error handler for every error in the individual promises. This one error handler, in other words, can monitor conditions of the underlying promises.	Invokes the subscribed error handler with an array of error objects from any failed promises, but the remainder continue to run. In other words, this reports cumulative errors in the way that progress reports cumulative completions.

Appendix A, by the way, has a small code snippet that shows how to use `join` and the array's `reduce` method to execute parallel operations but have their results delivered in a specific sequence.

Sequential Promises: Nesting and Chaining

In Chapter 2, when we added code to Here My Am! to copy the captured image to another folder, we got our first taste of using chained promises to run sequential async operations. To review, what makes this work is that any promise's `then` method returns another promise that's fulfilled when the given

completed handler returns. (That returned promise also enters the error state if the first promise has an error.) That completed handler, for its part, returns the promise from the next async operation in the chain, the results of which are delivered to the next completed handler down the line.

Though it may look odd at first, chaining is the most common pattern for dealing with sequential async operations because it works better than the more obvious approach of nesting. Nesting means to call the next async API within the completed handler of the previous one, fulfilling each with `done`. For example (extraneous code removed for simplicity):

```
//Nested async operations, using done with each promise
captureUI.captureFileAsync(Windows.Media.Capture.CameraCaptureUIMode.photo)
    .done(function (capturedFileTemp) {
        //...
        local.createFolderAsync("HereMyAm", ...)
            .done(function (myFolder) {
                //...
                capturedFile.copyAsync(myFolder, newName)
                    .done(function (newFile) {
                        })
                    })
            })
    });
```

The one advantage to this approach is that each completed handler will have access to all the variables declared before it. Yet the disadvantages begin to pile up. For one, there is usually enough intervening code between the async calls that the overall structure becomes visually messy. More significantly, error handling becomes much more difficult. When promises are nested, error handling must be done at each level with distinct handlers; if you throw an exception at the innermost level, for instance, it won't be picked up by any of the outer error handlers. Each promise thus needs its own error handler, making real spaghetti of the basic code structure:

```
captureUI.captureFileAsync(Windows.Media.Capture.CameraCaptureUIMode.photo)
    .done(function (capturedFileTemp) {
        //...
        local.createFolderAsync("HereMyAm", ...)
            .done(function (myFolder) {
                //...
                capturedFile.copyAsync(myFolder, newName)
                    .done(function (newFile) {
                        },
                        function (error) {
                            })
                    },
                    function (error) {
                        });
            },
            function (error) {
                });
    },
    function (error) {
        });
```

I don't know about you, but I really get lost in all the }'s and)'s (unless I try hard to remember my LISP class in college), and it's hard to see which error function applies to which async call. And just imagine throwing a few progress handlers in as well!

Chaining promises solves all of this with the small tradeoff of needing to declare a few extra temp variables outside the chain for any variables that need to be shared amongst the various completed handlers. Each completed handler in the chain again returns the promise for the next operation, and each link is a call to `then` except for a final call to `done` to terminate the chain. This allows you to indent all the `async` calls only once, and it has the effect of propagating errors down the chain, as any intermediate promise that's in the error state will be passed through to the end of the chain very quickly. This allows you to have only a single error handler at the end:

```
captureUI.captureFileAsync(Windows.Media.Capture.CameraCaptureUIMode.photo)
    .then(function (capturedFileTemp) {
        //...
        return local.createFolderAsync("HereMyAm", ...);
    })
    .then(function (myFolder) {
        //...
        return capturedFile.copyAsync(myFolder, newName);
    })
    .done(function (newFile) {
    },
    function (error) {
    })
})
```

To my eyes (and my aging brain), this is a much cleaner code structure—and it's therefore easier to debug and maintain. If you like, you can even end the chain with `done(null, errorHandler)`, as we did in Chapter 2:

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

Remember, though, that if you need to pass a promise for the whole chain elsewhere, as to a `setPromise` method, you'll use `then` throughout.

Error Handling in Promise Chains: `then` vs. `done`

This brings us to why we have both `then` and `done` and to why `done` is used at the end of a chain as well as for single `async` operations. To begin with, `then` returns another promise, thereby allowing chaining, whereas `done` returns `undefined`, so it always occurs at the end of a chain. Second, if an exception occurs within one `async` operation's `then` method and there's no error handler at that level, the error gets stored in the promise returned by `then` (that is, the returned promise is in the error state). In contrast, if `done` sees an exception and there's no error handler, it throws that exception to the app's event loop. This will bypass any local (synchronous) `try/catch` block, though you can pick them up in either in `WinJS.Application.onerror` or `window.onerror` handlers. (The latter will get the error if the former doesn't handle it.) If you don't have an app-level handler, the app will be terminated

and an error report sent to the Windows Store dashboard. For that reason we recommend that you implement an app-level error handler using one of the events above.

In practical terms, then, this means that if you end a chain of promises with a `then` and not `done`, all exceptions in that chain will get swallowed and you'll never know there was a problem! This can place an app in an indeterminate state and cause much larger problems later on. So, unless you're going to pass the last promise in a chain to another piece of code that will itself call `done` (as you do, for example, when using a `setPromise` deferral or if you're writing a library from which you return promises), always use `done` at the end of a chain even for a single async operation.²⁶

Promise error events If you look carefully at the `WinJS.Promise` documentation, you'll see that it has an `error` event along with `addEventListener`, `removeEventListener`, and `dispatchEvent` methods. This is primarily used within WinJS itself and is fired on exceptions (but *not* cancellation). Promises from async WinRT APIs, however, do not fire this event, so apps typically use error handlers passed to `then/done` for this purpose.

Managing the UI Thread with the WinJS Scheduler

JavaScript, as you are probably well aware, is a single-threaded execution environment, where any and all of your code apart from web workers and background tasks run on what we call the *UI thread*. The internal working of asynchronous APIs, like those of WinRT, happen on other threads as well, and the internal engines of the app host are also very much optimized for parallel processing.²⁷ But regardless of how much work you offload to other threads, there's one very important characteristic to always keep in mind:

The results from all non-UI threads eventually get passed back to the app on the main UI thread through callback functions such as the completed handler given to a promise.

Think about this very clearly: if you make a whole bunch of async WinRT calls within a short amount of time, such as to make HTTP requests or retrieve information from files, those tasks will execute on separate threads but each one will pass their results back to the UI thread when the task is complete. What this means is that the UI thread can become quite overloaded with such incoming traffic! Furthermore, what you do (or what WinJS does on your behalf) in response to the completion of each operation—such as adding elements to the DOM or innocently changing a simple layout-affecting style—can trigger more work on the UI thread, all of which competes for CPU time. As a result, your UI can become sluggish and unresponsive, the very opposite of “fast and fluid”!

²⁶ Some samples in the Windows SDK might still use `then` instead of `done`, especially for single async operations. This came from the fact that `done` didn't yet exist at one point and not all samples have been updated.

²⁷ In Windows 8 and Internet Explorer 10, most parsing, JavaScript execution, layout, and rendering on a single thread. Rewriting these processes to happen in parallel is one of the major performance improvements for Windows 8.1 and Internet Explorer 11, from which apps also benefit.

This is something we certainly saw with JavaScript apps on Windows 8, and developers created a number of strategies to cope with it such as starting async operations in timed batches to manage their rate of callbacks to the UI thread, and batching together work that triggers a layout pass so as to combine multiple changes in each pass.

Still, after plenty of performance analysis, the WinJS and app host teams at Microsoft found that what was really needed is a way to asynchronously prioritize different tasks *on the UI thread itself*. This meant creating some low-level scheduling APIs in the app host such as `MSApp.executeAtPriority`. But don't use such methods directly—use the `WinJS.Utilities.Scheduler` API instead. The reason for this is that WinJS very carefully manages its own tasks through the `Scheduler`, so by using it yourself you ensure that all the combined work is properly coordinated. This API also provides a simpler interface to the whole process, especially where promises are concerned.

Let's first understand what the different priorities are, then we'll see how to schedule and manage work at those priorities. Keep in mind, though, that using the scheduler is not at all required—it's there to help you tune the performance of your app, not to make your life difficult!

Scheduler Priorities

The relative priorities for the WinJS `Scheduler` are expressed in the `Scheduler.Priority` enumeration, which I list here in descending order: `max`, `high`, `aboveNormal`, `normal` (the default for app code), `belowNormal`, `idle`, and `min`. Here's the general guidance on how to use these:

Priority	Best Usage
<code>max</code> , <code>high</code>	Use sparingly for truly high priority work as these priorities take priority over layout passes in the rendering engine. If you overuse these priorities, the app can actually become <i>less</i> responsive!
<code>aboveNormal</code> , <code>normal</code> , <code>belowNormal</code>	Use these to indicate the relative importance between most of your tasks.
<code>idle</code> , <code>min</code>	Use for long-running and/or maintenance tasks where there isn't a UI dependency.

Although you need not use the scheduler in your own code, a little analysis of your use of async operations will likely reveal places where setting priorities might make a big difference. Earlier in “Optimizing Startup Time,” for example, we talked about how you want to prioritize non-UI work while your splash screen is visible, because the splash screen is noninteractive by definition. If you're doing some initial HTTP requests, for example, set the most critical ones for your home page to `max` or `high`, and set secondary requests to `belowNormal`. This will help those first requests get processed ahead of UI rendering, whereas your handling of the secondary requests will then happen after your home page has come up. This way you won't make the user wait for completion of those secondary tasks before the app becomes interactive. Other requests that you want to start, perhaps to cache data for a secondary leaderboard page, can be set to `belowNormal` or `idle`. Of course, if the user navigates to a secondary page, you'll want to change its task priorities to `aboveNormal` or `high`.

WinJS, for its part, makes extensive use of priorities. For example, it will batch edits to a data-binding source at `high` priority while scheduling cleanup tasks at `idle` priority. In a complex control like the `ListView`, fetching new items that are necessary to render the visible part of a `ListView` control is done at

max, rendering of the visible items is done at `aboveNormal`, pre-loading the next page of items forward is set to `normal` (anticipating that the user will pan ahead), and pre-loading of the previous page (to anticipate a reverse pan) is set to `belowNormal`.

Scheduling and Managing Tasks

Now that we know about scheduling priorities, the way to asynchronously execute code on the UI thread at a particular priority is by calling the `Scheduler.schedule` method (whose default priority is `normal`). This method allows you to provide an optional object to use as `this` inside the function along with a name to use for logging and diagnostics.²⁸

As a simple example, scenario 1 of the [HTML Scheduler sample](#) schedules a bunch of functions at different priorities in a somewhat random order (`js/schedulesjobsscenario.js`):

```
window.output("\nScheduling Jobs...");
var S = WinJS.Utilities.Scheduler;

S.schedule(function () { window.output("Running job at aboveNormal priority"); },
  S.Priority.aboveNormal);
window.output("Scheduled job at aboveNormal priority");

S.schedule(function () { window.output("Running job at idle priority"); },
  S.Priority.idle, this);
window.output("Scheduled job at idle priority");

S.schedule(function () { window.output("Running job at belowNormal priority"); },
  S.Priority.belowNormal);
window.output("Scheduled job at belowNormal priority");

S.schedule(function () { window.output("Running job at normal priority"); }, S.Priority.normal);
window.output("Scheduled job at normal priority");

S.schedule(function () { window.output("Running job at high priority"); }, S.Priority.high);
window.output("Scheduled job at high priority");

window.output("Finished Scheduling Jobs\n");
```

The output then shows that the "jobs," as they're called, which execute in the expected order:

```
Scheduling Jobs...
Scheduled job at aboveNormalPriority
Scheduled job at idlePriority
Scheduled job at belowNormalPriority
Scheduled job at normalPriority
Scheduled job at highPriority
Finished Scheduling Jobs
Running job at high priority
```

²⁸ The `Scheduler.execHigh` method is also a shortcut for directly calling `MSApp.execAtPriority` with `Priority.high`. This method does not accommodate any added arguments.

Running job at aboveNormal priority
Running job at normal priority
Running job at belowNormal priority
Running job at idle priority

No surprises here, I hope!

When you call `schedule`, what you get back is an object with the `Scheduler.IJob` interface, which defines the following methods and properties:

Properties	Description
<code>id</code>	(read-only) A unique id assigned by the scheduler.
<code>name</code>	(read-write) The app-provided name assigned to the job, if any. The name argument to <code>schedule</code> will be stored here.
<code>priority</code>	(read-write) The priority assigned through <code>schedule</code> ; setting this property will change the priority.
<code>completed</code>	(read-only) A Boolean indicating whether the job has completed (that is, the function given to <code>schedule</code> has returned and all its dependent async operations are complete).
<code>owner</code>	(read-write) An owner token that can be used to group jobs. This is undefined by default.
Methods	Description
<code>pause</code>	Halts further execution of the job.
<code>resume</code>	Resumes a previously paused job (no effect if the job isn't paused).
<code>cancel</code>	Removes the job from the scheduler.

In practice, if you've scheduled a job at a low priority but navigate to a page that really needs that job to complete before the page is rendered, you simply bump up its `priority` property (and then drain the scheduler as we'll see in a moment). Similarly, if you scheduled some work on a page that you don't need to continue when navigating away, then call the job's `cancel` method within the page's `unload` method. Or perhaps you have an index page from which you typically navigate into a details page, and then back again. In this case you can `pause` any jobs on the index page when navigating to the details, then `resume` them when you return to the index. See scenarios 2 and 3 of the sample for some demonstrations.

Scenario 2 also shows the utility of the `owner` property (the code is thoroughly mundane so I'll leave you to examine it). An owner token is something created through `Scheduler.createOwnerToken` and then assigned to a job's `owner` (which replaces any previous owner). An owner token is simply an object with a single method called `cancelAll` that calls the `cancel` method of whatever jobs are assigned to it, nothing more. It's a simple mechanism—the owner token really does nothing more than maintain an array of jobs—but clearly allows you to group related jobs together and cancel them with a single call. This way you don't need to maintain your own lists and iterate through them for this purpose. (To do the same for pause and resume you can, of course, just duplicate the pattern in your own code.)

The other important feature of the Scheduler is the `requestDrain` method. This ensures that all jobs scheduled at a given priority or higher are executed before the UI thread yields. You typically use this to guarantee that high priority jobs are completed before a layout pass. `requestDrain` returns a promise that is fulfilled when the jobs are drained, at which time you can drain lower priority tasks or schedule new ones.

A simple demonstration is shown in scenario 5 of the sample. It has two buttons that schedule the same set of varying jobs and then call `requestDrain` with either `high` or `belowNormal` priority. When the returned promise completes, it outputs a message to that effect (`js/drainingscenario.js`):

```
S.requestDrain(priority).done(function () {
    window.output("Done draining");
});
```

Comparing the output of these two side by side (`high` on the left, `belowNormal` on the right), as below, you can see that the promise is fulfilled at different points depending on the priority:

Draining scheduler to high priority Running job2 at high priority Done draining Running job1 at normal priority Running job5 at normal priority Running job4 at belowNormal priority Running job3 at idle priority	Draining scheduler to belowNormal priority Running job2 at high priority Running job1 at normal priority Running job5 at normal priority Running job4 at belowNormal priority Done draining Running job3 at idle priority
--	---

The other method that exists on the Scheduler is `retrieveState`, a diagnostic aid that returns a descriptive string for current jobs and drain requests. Adding a call to this in scenario 5 of the sample just after the call to `requestDrain` will return the following string:

```
Jobs:
  id: 28, priority: high
  id: 27, priority: normal
  id: 31, priority: normal
  id: 30, priority: belowNormal
  id: 29, priority: idle
Drain requests:
  *priority: high, name: Drain Request 0
```

Setting Priority in Promise Chains

Let's say you have a set of `async` data-retrieval methods that you want to execute in a sequence as follows, processing their results at each step:

```
getCriticalDataAsync().then(function (results1) {
    var secondaryPages = processCriticalData(results1);
    return getSecondaryDataAsync(secondaryPages);
}).then(function (results2) {
    var itemsToCache = processSecondaryData(results2);
    return getBackgroundCacheDataAsync(itemsToCache);
}).done(function (results3) {
    populateCache(results3);
});
```

By default, all of this would run at the current priority against everything else happening on the UI thread. But you probably want the call to `processCriticalData` to run at a `high` priority,

processSecondaryData to run at normal, and populateCache to run at idle. With schedule by itself, you'd have to do everything the hard way:

```
var S = WinJS.Utilities.Scheduler;

getCriticalDataAsync().done(function (results1) {
    S.schedule(function () {
        var secondaryPages = processCriticalData(results1);
        S.schedule(function () {
            getSecondaryDataAsync(secondaryPages).done(function (results2) {
                var itemsToCache = processSecondaryData(results2);
                S.schedule(function () {
                    getBackgroundCacheDataAsync(itemsToCache).done(function (results3) {
                        populateCache(results3);
                    });
                }, S.Priority.idle);
            });
        }, S.Priority.normal);
    }, S.Priority.high);
});
```

Urg. Blech. Ick. It's more fun going to the dentist than writing code like this! To simplify matters, you could encapsulate the process of setting a new priority within another promise that you can then insert into the chain. The best way to do this is to dynamically generate a completed handler that would take the results from the previous step in the chain, schedule a new priority, and return a promise that delivers those same results (see Appendix A for the use of new WinJS.Promise):

```
function schedulePromise(priority) {
    //This returned function is a completed handler.
    return function completedHandler (results) {
        //The completed handler returns another promise that's fulfilled
        //with the same results it received...
        return new WinJS.Promise(function initializer (c) {
            //But the delivery of those results are scheduled according to a priority.
            WinJS.Utilities.Scheduler.schedule(function () {
                c(results);
            }, priority);
        });
    }
}
```

Fortunately we don't have to write this code ourselves. The WinJS.Utilities.Scheduler already has five pre-made completed handlers like this that also automatically cancel a job if there is an error. These are called [schedulePromiseHigh](#), [schedulePromiseAboveNormal](#), [schedulePromiseNormal](#), [schedulePromiseBelowNormal](#), or [schedulePromiseIdle](#).

Because these APIs are pre-made completed handlers rather than methods you call directly, simply insert the appropriate name at those points in a promise chain where you want to change the priority, as highlighted below:

```
var S = WinJS.Utilities.Scheduler;
```

```

getCriticalDataAsync().then(S.schedulePromiseHigh).then(function (results1) {
    var secondaryPages = processCriticalData(results1);
    return getSecondaryDataAsync(secondaryPages);
}).then(S.schedulePromise.normal).then(function (results2) {
    var itemsToCache = processSecondaryData(results2);
    return getBackgroundCacheDataAsync(itemsToCache);
}).then(S.schedulePromiseIdle).done(function (results3) {
    populateCache(results3);
});

```

Long-Running Tasks

All the jobs that we've seen so far are short-running in that we schedule a worker function at a certain priority and it just completes its work when it's called. However, some tasks might take much longer to complete, in which case you don't want to block higher priority work on your UI thread. To help with this, the scheduler has a built-in interval timer of sorts for tasks that are scheduled at `aboveNormal` priority or lower, so a task can check whether it should cooperatively yield and have itself rescheduled for its next bit of work. Let me stress that word *cooperatively*: nothing forces a task to yield, but because all of this is affecting the UI performance of your app and your app alone, if you don't play nicely you'll just be hurting yourself!

The mechanism for this is provided through a *job info* object that's passed as an argument to the worker function itself. To make sure we're clear on how this fits in, let's first look at everything a worker has available within its scope, which is best explained with a few comments within the basic code structure:

```

var job = WinJS.Utilities.Scheduler.schedule(function worker(jobInfo) {
    //jobInfo.job is the same as the job returned from schedule.
    //Scheduler.currentPriority will match the second argument to schedule.
    //this will be the third argument passed to schedule.
}, S.Priority.idle, this);

```

The members of the *jobInfo* object are defined by `Scheduler.IJobInfo`:

Properties	Description
job	(read-only) The same job object as returned from <code>schedule</code> .
shouldYield	(read-only) A Boolean flag that is typically <code>false</code> when the worker is first called and then changes to <code>true</code> if the worker should yield the UI thread and reschedule its work.
Methods	
setWork	Provides the worker for the rescheduled task.
setPromise	Provides a promise that the scheduler will wait upon before rescheduling the task, where the worker to reschedule is the fulfillment value of the promise.

Scenario 4 of the [HTML Scheduler sample](#) shows how to work with these. When you press the Execute a Yielding Task button, it schedules a function called *worker* at `idle` priority that just spins within itself until you press the Complete Yielding Task button, which sets the `taskCompleted` flag below to `true` (js/yieldingscenario.js, with the 2s interval changed to 200ms):

```

S.schedule(function worker(jobInfo) {
  while (!taskCompleted) {
    if (jobInfo.shouldYield) {
      // not finished, run this function again
      window.output("Yielding and putting idle job back on scheduler.");
      jobInfo.setWork(worker);
      break;
    }
    else {
      window.output("Running idle yielding job...");
      var start = performance.now();
      while (performance.now() < (start + 200)) {
        // do nothing;
      }
    }
  }

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

```

Provided that the task is active, it does 200ms of work and then checks if `shouldYield` has changed to `true`. If so, the worker calls `setWork` to reschedule itself (or another function if it wants). You can trigger this while the idle worker is running by pressing the Add Higher Priority Tasks to Queue button in the sample. You'll then see how those tasks are run before the next call to the worker. In addition, you can poke around elsewhere in the UI to observe that the idle task is not blocking the UI thread.

Note here that the worker function checks `shouldYield` first thing to immediately yield if necessary. However, it's perfectly fine to do a little work first and then check. Again, this is all about cooperating within your own app code, so such self-throttling is your choice.

As for `setPromise`, this is slightly tricky. Calling `setPromise` tells the scheduler to wait until that promise is fulfilled before rescheduling the task, where the next worker function for the task is provided directly through the promise's fulfillment value. (As such, `IJobInfo.setPromise` doesn't pertain to handling async operations like other `setPromise` methods in WinJS that are tied in with WinRT deferrals. If you called `IJobInfo.setPromise` with a promise from some random async API, the scheduler would attempt to use the fulfillment value of that operation—which could be anything—as a function and thus likely throw an exception.)

In short, whereas `setWork` says "go ahead and reschedule with this worker," `setPromise` says "hold off rescheduling until I deliver the worker sometime later." This is primarily useful to create a work queue composed of multiple jobs with an ongoing task to process that queue. To illustrate, consider the following code for such an arrangement:

```

var workQueue = [];

function addToQueue(worker) {
  workQueue.push(worker);
}

```

```

}

S.schedule(function processQueue(jobInfo) {
  while (work.length) {
    if (jobInfo.shouldYield) {
      jobInfo.setWork(processQueue);
      return;
    }
    work.shift(); //Pull the first from the FIFO queue and call it.
  }
}}, S.Priority.belowNormal);

```

Assuming that there are some jobs in the queue when you first call `schedule`, the *processQueue* task will cooperatively empty that queue. And if new jobs are added to the queue in the meantime, *processQueue* will continue to be rescheduled.

The problem, however, is that the *processQueue* worker will finish and exit as soon as the queue is empty, meaning that any jobs you add to the queue later on won't be processed. To fix this you could just have *processQueue* repeatedly call `setWork` on itself again and again even when the queue is empty, but that would be wasteful. Instead, you can use `setPromise` to have the scheduler wait until there is more work in the queue. Here's how that would work:

```

var workQueue = [];
var haveWork = function () {}; //This function is just a placeholder

function addToQueue(worker) {
  workQueue.push(worker);
  haveWork();
}

S.schedule(function processQueue(jobInfo) {
  while (work.length) {
    if (jobInfo.shouldYield) {
      jobInfo.setWork(processQueue);
      return;
    }
    work.shift(); //Pull the first from the FIFO queue and call it.
  }

  //If we reach here the queue is empty, but we don't want to exit the worker.
  //Instead of calling setWork without work to do, create a promise that's fulfilled
  //when addToQueue is called again, which we do by replacing the haveWork function
  //with one that calls the promise's completed handler.
  jobInfo.setPromise(new WinJS.Promise(function (completeDispatcher) {
    haveWork = function () { completeDispatcher(processQueue) };
  }));
});

```

With this code, say we populate `workQueue` with a number of jobs and then make the call to `schedule`. Up to this point and so long as the queue doesn't become empty, we stay inside the `while` loop of *processQueue*. Any call to the empty `haveWork` function so far is just a no-op.

If the queue becomes empty, however, we'll exit the `while` loop but we don't want `processQueue` to exit. Instead, we want to tell the scheduler to wait until more work is added to the queue. This is why we have that placeholder function for `haveWork`, because we can now replace it with a function that will complete the promise with `processQueue`, thereby triggering a rescheduling of that worker function.

Note that an alternate way to accomplish the same goal is to use this assignment for `haveWork`:

```
haveWork = completeDispatcher.bind(null, processQueue);
```

This accomplishes the same result as an anonymous function and avoids creating a closure.

Debugging and Profiling

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

Tip Debug logging, which is local to and only relevant on your development machine, is a very different concern from telemetry logging, with which you monitor and record user activity. See "Instrumenting Your App for Telemetry and Analytics" in Chapter 20.

Debug or release? Because JavaScript is not a compiled language, it lacks conditional compilation directives like `#ifdef` in C#/C++. There are, however, a few ways to more or less make this determination at run time (with some caveats). See "Sidebar: Debug or Release?" in Chapter 2.

Debug Output and Logging

It's sometimes heartbreaking to developers that `window.prompt` and `window.alert` are not available to Windows Store apps as quickie debugging aids. Fortunately, you have two other good options for that purpose. One is `Windows.UI.Popups.MessageDialog`, which is actually what you use for real user prompts in general (see Chapter 9). The other is `console.log`, as we've used in our code already, which sends text to Visual Studio's output pane. These messages can also be logged as Windows events, as we'll see shortly.

For readers who are seriously into logging, beyond the kind you do with chainsaws, there are two other options: a more flexible method in WinJS called `WinJS.Log`, and the logging APIs in `Windows.Foundation.Diagnostics`.

`WinJS.Log` is a curious beast because although it's ostensibly part of the WinJS namespace, it's actually not implemented within WinJS itself! At the same time, it's used all over the place in the library for errors and other reporting. For instance:

```
WinJS.Log && WinJS.Log(safeSerialize(e), "winjs", "error");
```

This kind of JavaScript syntax, by the way, means “check whether `WinJS.log` exists and, if so, call it.” The `&&` is a shortcut for an `if` statement: the JavaScript engine will not execute the part after the `&&` if the first part is `null`, `undefined`, or `false`. It’s a very convenient bit of concise syntax.

Anyway, the purpose of `WinJS.log` is to allow you to implement your own logging function and have it pick up `WinJS`’s logging as well as any you add to your own code. What’s more, you can turn the logging on and off at any time, something that’s not possible with `console.log` unless, well, you write a wrapper like `WinJS.log`!

Your `WinJS.log` function, as described in the documentation, should accept three parameters:

1. The message to log (a string).
2. A string with a tag or tags to categorize the message. `WinJS` always uses “winjs” and sometimes adds an additional tag like “binding”, in which case the second parameter is “winjs binding”. I typically use “app” in my own code.
3. A string describing the type of the message. `WinJS` will use “error”, “info”, “warn”, and “perf”.

Conveniently, `WinJS` offers a basic implementation of this which you set up by calling [`WinJS.Utilities.startLog\(\)`](#). This assigns a function to `WinJS.log` that uses [`WinJS.Utilities._formatLog`](#) to produce decent-looking output to the console. What’s very useful is that you can pass a list of tags (in a single string) to `startLog` and only those messages with those tags will show up. Multiple calls to `startLog` will aggregate those tags. Then you can call [`WinJS.Utilities.stopLog`](#) to turn everything off and start again if desired (`stopLog` is not made to remove individual tags). As a simple example, see the [HereMyAm3d](#) example in the companion content.

Tip Although logging will be ignored for released apps that customers will acquire from the Store, it’s a good idea to comment out your one call to `startLog` before submitting a package to the Store and thus avoid making any unnecessary calls at run time.

`WinJS.log` is highly useful for generating textual logs, but if you want to go much deeper you’ll want to use the WinRT APIs in [`Windows.Foundation.Diagnostics`](#), namely the `LoggingSession` and `FileLoggingSession` classes. These work with in-memory and continuous file-based logging, respectively, and generate binary “Event Trace Log” (ETL) data that can be further analyzed with the Windows Performance Analyzer (`wpa.exe`) and the Trace Reporter (`tracertp.exe`) tools in the Windows SDK. This is a subject well beyond the scope of this book (and this author’s experience), so refer to the [Windows Performance Analyzer documentation](#) for more, along with the [LoggingSession sample](#) and [FileLoggingSession sample](#).

Error Reports and the Event Viewer

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

dashboard for your app, with a message telling you to not use it! Generally, Store apps should not provide their own close affordances.

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

In addition to the Store dashboard, you should make fast friends with the Windows Event Viewer.²⁹ This is where error reports, console logging, and unhandled exceptions (which again terminate the app without warning) can be recorded. To enable this, start Event Viewer, navigate to Application And Services Logs on the left side (after waiting for a minute while the tool initializes itself), and then expand Microsoft > Windows > AppHost. Then left-click to *select* Admin (this is important), right-click Admin, and select View > Show Analytic And Debug Logs. This turns on full output, including tracing for errors and exceptions, as shown in Figure 3-5. Then right-click AppTracing (also under AppHost) and select Enable Log. This will trace any calls to `console.log` as well as other diagnostic information coming from the app host.

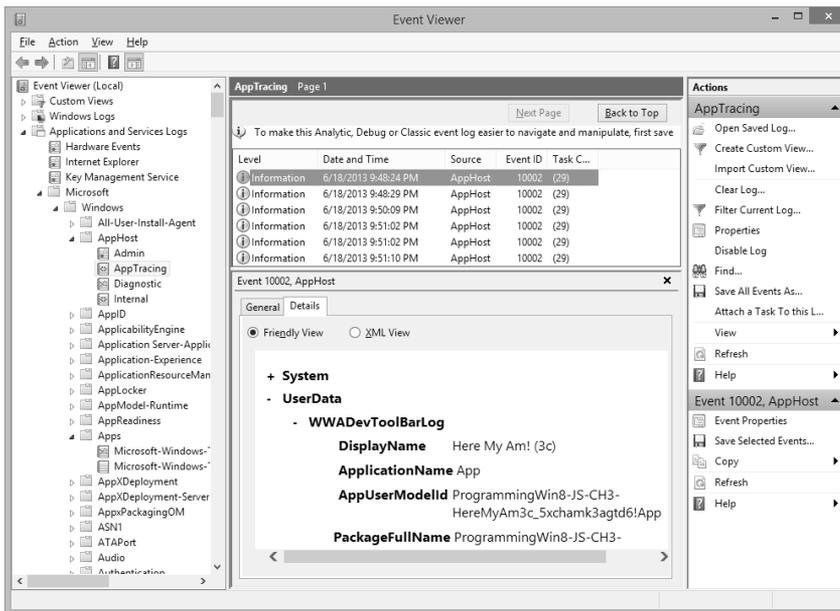


FIGURE 3-5 App host events, such as unhandled exceptions, load errors, and logging can be found in Event Viewer.

We already introduced Visual Studio's Exceptions dialog in Chapter 2; refer back to Figure 2-16. For each type of JavaScript exception, this dialog supplies two checkboxes labeled Thrown and User-

²⁹ If you can't find Event Viewer, press the Windows key to go to the Start screen and then invoke the Settings charm. Select Tiles, and turn on Show Administrative Tools. You'll then see a tile for Event Viewer on your Start screen.

unhandled. Checking Thrown will display a dialog box in the debugger (see Figure 3-6) whenever an exception is thrown, regardless of whether it's handled and before reaching any of your error handlers.



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

If you have error handlers in place, you can safely click the Continue button in the dialog of Figure 3-6 and you'll eventually see the exception surface in those error handlers. (Otherwise the app will terminate; see below.) If you click Break instead, you can find the exception details in the debugger's Locals pane, as shown in Figure 3-7.

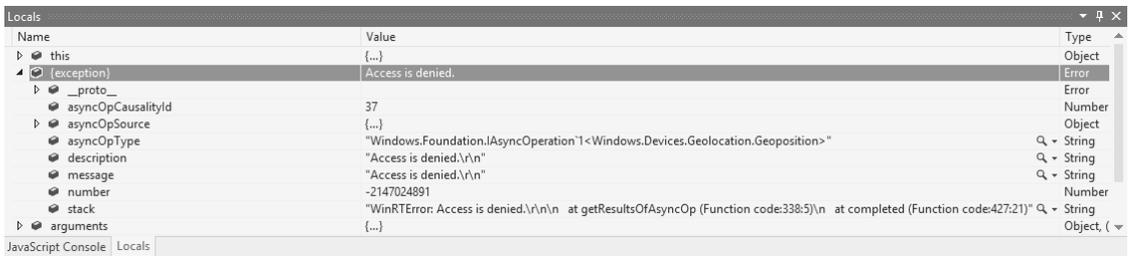


FIGURE 3-7 Information in Visual Studio's Locals pane when you break on an exception.

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

You typically turn on Thrown for only those exceptions you care about; turning them all on can make it very difficult to step through your app! But it's especially helpful if you're debugging an app and end up at the debugger line in the following bit of WinJS code, just before the app is terminated:

```
var terminateAppHandler = function (data, e) {  
  debugger;  
  MSApp.terminateApp(data);  
};
```

If you turn on Thrown for all JavaScript exceptions, you'll then see exactly where the exception occurred. You can also just check Thrown for only those exceptions you expect to catch.

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

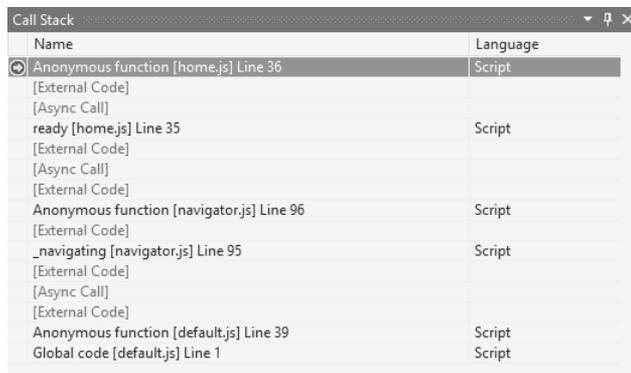
WinJS.validation Speaking of exceptions, if you set `WinJS.validation` to `true` in your app, you'll instruct WinJS to perform a few extra checks on arguments and internal state, and throw exceptions if something is amiss. Just search on "validation" in the WinJS source files for where it's used.

Async Debugging

Working with asynchronous APIs presents a challenge where debugging is concerned. Although we have a means to sequence async operations with promise chains (or nested calls, for that matter), each step in the sequence involves an async call, so you can't just step through as you would with synchronous code. If you try this, you'll step through lots of promise code (in WinJS or the JavaScript projection layer for WinRT) rather than your completed handlers, which isn't particularly helpful.

What you'll need to do instead is set a breakpoint on the first line of each completed handler and on the first line of each error function. As each breakpoint is hit, you can step through that handler. When you reach the next async call in a completed handler, click the Continue button in Visual Studio so that the async operation can run. After that you'll hit the breakpoint in the next completed handler or the breakpoint in the error handler.

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



The Call Stack shows you the sequence of functions that lead up to the point where the debugger stopped, at which point you can double-click any of the lines and examine that function's context. With async calls, this can get really messy with all the generic handlers and other chaining that happens within WinJS and the JavaScript projection layer. Fortunately—very fortunately!—Visual Studio spares

you from all that. It condenses such code into the gray [Async Call] and [External Code] markers, leaving only a clear call chain for your app’s code. In this example I set a breakpoint in the completed handler for geolocation in HereMyAm3d. That completed handler is an anonymous function, as the first line of the Call Stack indicates, but the next reference to the app code clearly shows that the real context is the `ready` method within `home.js`, which itself is part of a longer chain that originated in `default.js`. Double-clicking any one of the app code references will open that code in Visual Studio and update the Locals pane to that context.

The real utility of this comes when an exception occurs somewhere other than within you own handlers, because you can then easily trace the causality chain that led to that point.

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

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

Performance and Memory Analysis

Alongside its excellent debugging tools, Visual Studio also offers additional aids to help evaluate the performance of an app, analyze its memory usage, and otherwise discover and diagnose problems that affect the user experience and the app’s effect on the system. To close this chapter, I wanted to give you a brief overview of what’s available along with pointers to where you can learn more—because this subject could fill a book in itself! (In lieu of that, a general pointer is to [filter the //build 2013 videos by the “performance” tag](#), which turns up a healthy set.)

For starters, the [Writing efficient JavaScript](#) topic is well worth a read (as are its siblings under [Best practices using JavaScript](#)), because it explains various things you should and should not do in your code to help the JavaScript engine run best. One thing you *shouldn’t* worry about is the performance of `querySelector` and `getElementById`, both of which are highly optimized because they’re used so often. Keep this in mind, because I know for myself that any function that starts with “query” just sounds like it’s going to do a lot of work, but that’s not true here.

Next, when thinking about performance, start by setting specific goals for your user experience, such as “the app should become interactive within 1.5 seconds” and “navigating between the gallery and details pages happens in 0.5 seconds or less.” In fact, such goals should really be part of the app’s design that you discuss with your designers, because they’re just as essential to the overall user

experience as static considerations like layout. In the end, performance is not about numbers but about creating a great user experience.

Establishing goals also helps you stay focused on what matters. You can measure all kinds of different performance metrics for an app, but if they aren't serving your real goals, you end up with a classic case of what Tom DeMarco, in his book *Why Does Software Cost So Much?* (Dorset House, 1995), calls "measurement dysfunction": lots of data with meaningless results or results that lead to undesired action.³⁰

Along the same lines, when running analysis tools, it's important that you *exercise the app like a user would*. That way you get results that are meaningful to the real user experience—that is, the human experience!—rather than results that would be meaningful to a robot. In the end, all the performance analysis in the world won't be worth anything unless it translates into two things: better ratings and reviews in the Windows Store, and greater app revenue.

With your goals in mind, run analysis tools on a regular basis and evaluate the results against your goals. Then adjust your code, run the tools again, and evaluate. In other words, running performance tools to evaluate your performance goals is just another part of making sure you're creating the app according to its design—the static and dynamic parts alike.

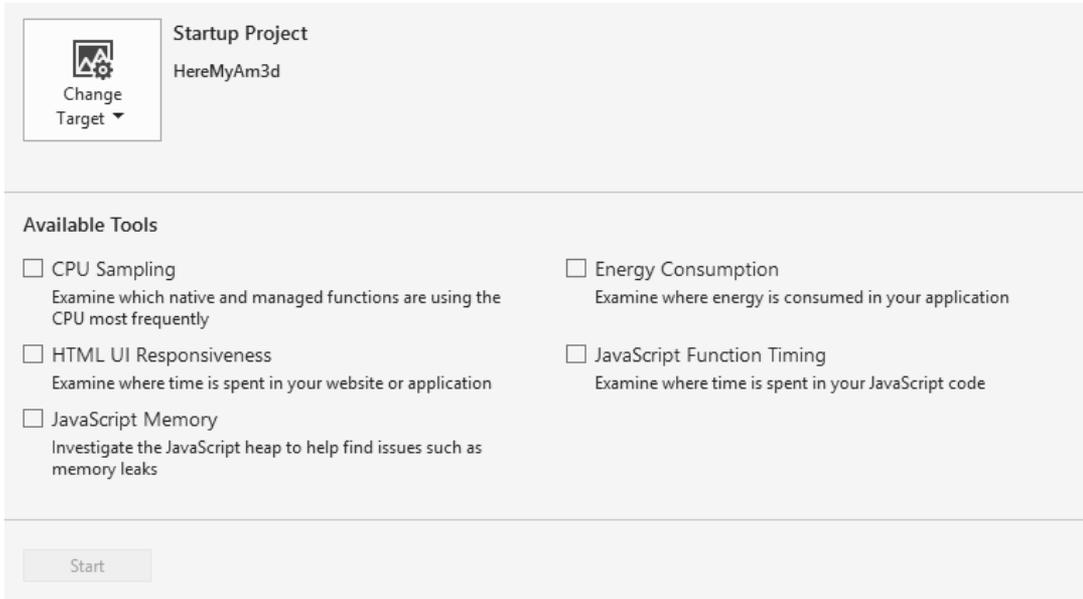
Remember also to *run performance analysis on a variety of hardware*, especially lower-end devices such as ARM tablets that are much more sensitive to performance issues than is your souped-up dev machine. In fact, slower devices are the ones you should be most concerned about, because their users will probably be the first to notice any issues and ding your app ratings accordingly. And yes, you can run the performance tools on a remote machine in the same way you can do remote debugging (but not in the simulator). Also be aware that analysis tools always run *outside* of the debugger for obvious reasons, because stopping at breakpoints and so forth would produce bad performance data!

I very much encourage you, then, to spend a few hours exercising the available tools and getting familiar with the information they provide. Make them a regular part of your coding/testing cycle so that you can catch performance and memory issues early on, when it's easier and less costly to fix them. Doing so will also catch what we call "regressions," where a later change to the code causes performance problems that you fixed a long time ago to rear their ugly heads once again. As the character Alistor Moody of the Harry Potter books says, "Constant vigilance!"

³⁰ DeMarco tells an amusing story of metrics at their worst: "Consider the case of the Soviet nail factory that was measured on the basis of the number of nails produced. The factory managers hit upon the idea of converting their entire factory to production of only the smallest nails, tiny brads. Some commissar, realizing this as a case of dysfunction, came up with a remedy. He instituted measurement of *tonnage* of nails produced, rather than numbers. The factory immediately switched over to producing only railroad spikes. The image I propose to mark the dysfunction end of the spectrum is a Soviet carpenter, looking perplexed, with a useless brad in one hand and an equally useless railroad spike in the other."

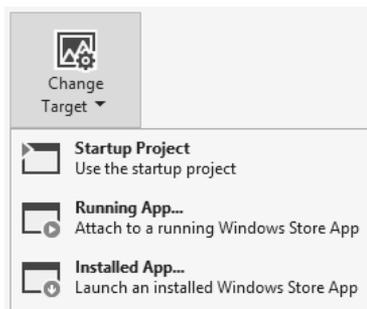
Tip Two topics in the documentation also contain loads of detailed information in these areas: [Performance best practices for Windows Store apps using JavaScript](#) and [General best practices for performance](#).

So, on to the tools. These are found on the Debug > Performance And Diagnostics... menu, which brings up the hub shown below with tools that are appropriate to your project's language:



Get Visual Studio updates New tools are often released with updates to Visual Studio, so be sure to install them and read the accompanying blogs or release notes to understand what's new.

By default, Visual Studio will set the target to be the currently loaded project. However, you can run the tools on any app by using the options on the Change Target drop-down:



As the drop-down indicates, the Installed App option will launch an app anew, whereas the Running App option attaches to one that's already been launched. Both are essential for profiling apps on

devices where your full project is not present; the latter is also useful if your app is already running and you want to analyze specific user interactions for a set of conditions that you've already set up. This way you won't collect a bunch of extra data that you don't need.

Note that you can run these tools on *any* installed app, not just your own, which means you can gather data from other apps that have the level of performance you'd like to achieve for yours.

The Performance and Diagnostic Hub as a whole is designed to be extensible with third-party tools, giving you a one-stop shop for enabling multiple tools simultaneously. The ones shown above are those built into Visual Studio, and be sure to install new Visual Studio updates because that's often how new tools are released.

Here's a quick overview of what the current tools accomplish:

Tool	Description
HTML UI Responsiveness	Provides a graph of Visual Throughput (frames per second) for the rendering engine over time, helping to identify places where your UI is not as responsive as you'd like. It also provides a millisecond breakdown of CPU utilization in various subsystems: loading, scripting, garbage collection, styling, rendering, and image decoding, with various important lifecycle events indicated along the way. This data is also shown on a time line where you can select any part to see the breakdown in more detail. All this is helpful for finding areas where the interactions between subsystems is adding lots of overhead, where there's excessive fragmentation, or where work being done in a particular subsystem is causing a drop in visual throughput. A walkthrough is on HTML UI Responsiveness tool in Visual Studio 2013 (MSDN blogs). Also see Analyze UI responsiveness .
Energy Consumption	Launches the app and collects data about power usage (in milliwatts) over time, split up by CPU, display, and network. This is very important to writing power-efficient apps for tablet devices. It can also help you determine whether it's more power efficient to use the local CPU or a network server for certain tasks, as network I/O can take as much and even more power than a burst of CPU activity. For more, see Energy Consumption tool in Visual Studio 2013 .
JavaScript Memory	Launches the app and provides a dynamic graph of memory usage over time as well as the ability to take heap snapshots, allowing you to see memory spikes that occur in response to user activity, and whether that memory is being properly freed. Refer to JavaScript memory analysis for Windows Store apps in Visual Studio 2012 (MSDN blogs) and Analyzing memory usage in Windows Store apps .
JavaScript Function Timing (also called the JavaScript Profiler)	Displays data on when and where function calls are being made in JavaScript and how much time is spent in what part of your code. A walkthrough can be found on How to profile a JavaScript App for performance problems (MSDN blogs). Also see Analyzing JavaScript Performance in Windows Store apps , which covers both local and remote machines.
CPU Sampling	Similar to the JavaScript Function Timing tool but works for managed (C#/Visual Basic) and native (C++) code. This is useful only if you're writing a multi-language app with both JavaScript and one of the other languages.

For a video demonstration of most of these, watch the [Visual Studio 2013 Performance and Diagnostics Hub](#) video on Channel 9 and [Diagnosing Issues in JavaScript Windows Store Apps with Visual Studio 2013](#) from the //build 2013 conference, both by Andrew Hall, the real expert on these matters. Note that everything you see in these video (with the exception of the console app profiler) is

available in the Visual Studio Express edition that we've been using, and if you want to skip the part about XAML UI responsiveness in the first video, you can jump ahead to about 13:30 where he talks about the JavaScript tools.

Tip In the first video, the responsiveness problems for the demo apps written both in XAML/C# and HTML/JavaScript primarily come from loading full image files just to generate thumbnails for gallery views. As the video mentions, you can avoid this entirely and achieve much better performance by using `Windows.Storage.StorageFile.getThumbnailAsync`. This API draws on thumbnail caches and other mechanisms to avoid the memory overhead and CPU cost of loading full image files. We'll see more of this in Chapter 11.

It's important, of course, with all these tools to clearly correlate certain events in the app with the various measurements. This is the purpose of the `performance.mark` function, which exists in the global JavaScript namespace.³¹ Events written with this function appear as User Marks in the timelines generated by the different tools, as shown in Figure 3-8. In looking at the figure, note that the resolution of marks on the Memory Analyzer timeline on the scale of *seconds*, so use marks to indicate only significant user interaction events rather than every function entry and exit. (With other tools, however, the resolution is much finer, so you can use `performance.mark` more frequently.)

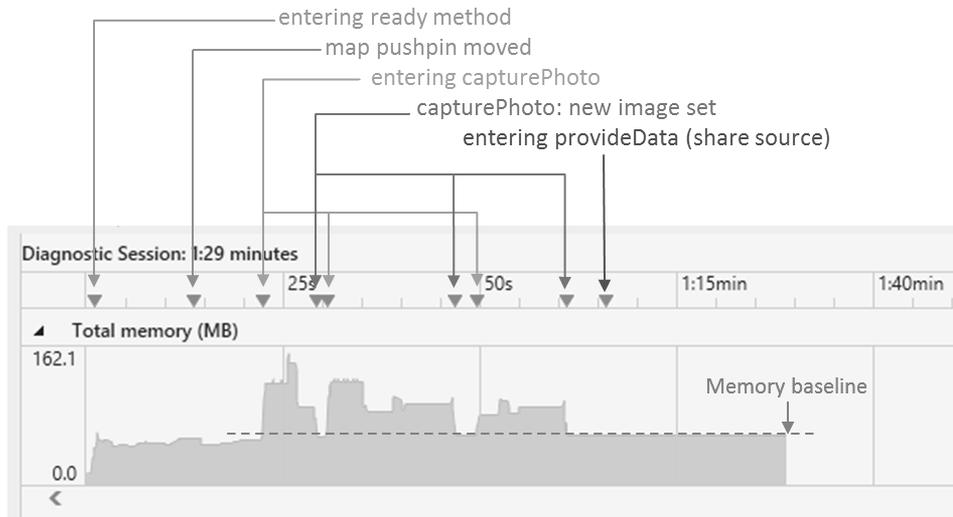


FIGURE 3-8 Output of the JavaScript Memory analyzer annotated with different marks. The red dashed line is also added in this figure to show the ongoing memory footprint; it is not part of the tool's output.

As one example of using these tools, let's run the Here My Am! app through the memory analyzer to see if we have any problems. We'll use the HereMyAm3d example in the companion code where I've

³¹ This function is part of a larger group of methods on the `performance` object that reflect developing standards. For more details, see [Timing and Performance APIs](#), `performance.mark` specifically replaces `msWriteProfilerMark`.

added some `performance.mark` calls for events like startup, capturing a new photo, rendering that photo, and exercising the Share charm. Figure 3-8 shows the results. For good measure—logging, actually!—I’ve also converted `console.log` calls to `WinJS.Log`, where I’ve used a tag of “app” in each call and in the call to `WinJS.Utilities.startLog` (see `default.js`).

Referring to Figure 3-8, here’s what I did after starting up the app in the memory analyzer. Once the home page was up (first mark), I repositioned the map and its pushpin (second mark), and you can see that this increased memory usage a little within the Bing maps control. Next I invoked the camera capture UI (third mark), which clearly increased memory use as expected. After taking a picture and displaying it in the app (fourth mark), you can see that the allocations from the camera capture UI have been released, and that we land at a baseline footprint that now includes a rendered image. I then do into the capture UI two more times, and in each case you can see the memory increase during the capture, but it comes back to our baseline each time we return to the main app. There might be some small differences in memory usage here depending on the size of the image, but clearly we’re cleaning up the image when it get replaced. Finally I invoked the Share charm (last mark), and we can see that this causes no additional memory usage in the source app, which is expected because all the work is being done in the target. As a result, I feel confident that the app is managing its memory well. If, on the other hand, that baseline kept increasing over time, then I’d know I have a leak somewhere.

Tip There’s no rule anywhere that says you have to profile your full app project. When you’re trying to compare different implementation strategies, it can be much easier to create a simple test project and run the profiling tools on it so that you can obtain very focused comparisons for different approaches. Doing so will speed up your investigations and avoid disturbing your main project in the process.

The Windows App Certification Toolkit

The other tool you should run on a regular basis is the Windows App Certification Toolkit (WACK), which is actually one of the first tools that’s automatically run on your app when you submit it to the Windows Store. If this toolkit reports failures on your local machine, you can be certain that you’ll fail certification very early in the process.

Running the toolkit can be done as part of building an app package for upload, but until then, launch it from your Start screen (it’s called Windows App Cert Kit). When it comes up, select Validate Windows Store App, which (after a disk-chewing delay) presents you with a list of installed apps, including those that you’ve been running from Visual Studio. It takes some time to generate that list if you have lots of apps installed, so you might use the opportunity to take a little stretching break. Then select the app you want to test, and take the opportunity to grab a snack, take a short walk, play a few songs on the guitar, or otherwise entertain yourself while the WACK gives your app a good whacking.

Eventually it’ll have an XML report ready for you. After saving it (you have to tell it where), you can view the results. Note that for developer projects it will almost always report a failure on bytecode generation, saying “This package was deployed for development or authoring mode. Uninstall the package and reinstall it normally.” To fix this, uninstall it from the Start menu, select a Release target in Visual Studio, and then use the Build > Deploy Solution menu command. But you can just ignore this

particular error for now. Any other failure will be more important to address early on—such as crashes, hangs, and launch/suspend problems—rather than waiting until you’re ready to submit to the Store.

Note Visual Studio also has a code analysis tool on the Build > Run Code Analysis On Solution menu, which examines source code for common defects and other violation of best practices. However, this tool does not presently work with JavaScript.

What We’ve Just Learned

- How apps are activated (brought into memory) and the events that occur along the way.
- The structure of app activation code, including activation kinds, previous execution states, and the `WinJS.UI.Application` object.
- Using deferrals when needing to perform async operations behind the splash screen, and optimizing startup time.
- How to handle important events that occur during an app’s lifetime, such as focus events, visibility changes, view state changes, and suspend/resume/terminate.
- The basics of saving and restoring state to restart after being terminated, and the WinJS utilities for implementing this.
- How to implement page-to-page navigation within a single page context by using page controls, `WinJS.Navigation`, and the `PageControlNavigator` from the Visual Studio/Blend templates, such as the Navigation App template.
- Details of promises that are commonly used with, but not limited to, async operations.
- How to join parallel promises as well as execute a sequential async operations with chained promises.
- How exceptions are handled within chained promises and the differences between `then` and `done`.
- How to create promises for different purposes.
- Using the APIs in `WinJS.Utilities.Scheduler` for prioritizing work on the UI thread, including the helpers for prioritizing different parts of a promise chain.
- Methods for getting debug output and error reports for an app, within the debugger and the Windows Event Viewer.
- How to debug asynchronous code and how Visual Studio makes it easy to see the causality chain.
- The different performance and memory analysis tools available in Visual Studio.