

Windows® Phone 7 Development Internals

Covers Windows Phone 7 and Windows Phone 7.5



Andrew Whitechapel

www.allitebooks.com

Windows® Phone 7 Development Internals

Build Windows Phone applications optimized for performance and security

Drill into Windows Phone 7 design and architecture—and learn best practices for building a variety of applications. Each chapter focuses on a single Windows Phone building block or feature area, and shows you how to apply it in your applications. If you're an experienced .NET developer familiar with Microsoft® Silverlight®, you'll gain deep insights into the Windows Phone platform design and API surface.

Discover how to:

- Master the Windows Phone application model, including its lifecycle and events
- Use Silverlight UI controls to create engaging applications
- Manage databinding and decouple architectural layers with the Model View ViewModel pattern
- Employ built-in sensors such as Assisted GPS, the accelerometer, and camera
- Use media services APIs for video streaming, as well as audio input and playback
- Consume web services and connect to the cloud through Windows Azure™
- Apply Windows Phone 7.5 enhancements, such as multitasking and Fast Application Switching

Get code samples on the web

Ready to download at

<http://go.microsoft.com/fwlink/?Linkid=248889>

For **system requirements**, see the Introduction.

microsoft.com/mspress

ISBN: 978-0-7356-6325-1



U.S.A. \$59.99

Canada \$62.99

[Recommended]

Programming/Windows Phone

www.allitebooks.com



About the Author

Andrew Whitechapel is a senior program manager for the Windows Phone Application Platform team, responsible for core functionality such as the critical execution manager and resource manager components. He is also author of the Microsoft Press® book *Microsoft .NET Development for Microsoft Office*.

DEVELOPER ROADMAP

Start Here!

- Beginner-level instruction
- Easy to follow explanations and examples
- Exercises to build your first projects



Step by Step

- For experienced developers learning a new topic
- Focus on fundamental techniques and tools
- Hands-on tutorial with practice files plus eBook



Developer Reference

- Professional developers; intermediate to advanced
- Expertly covers essential topics and techniques
- Features extensive, adaptable code examples



Focused Topics

- For programmers who develop complex or advanced solutions
- Specialized topics; narrow focus; deep coverage
- Features extensive, adaptable code examples



Microsoft®

Windows® Phone 7 Development Internals

Andrew Whitechapel

Published with the authorization of Microsoft Corporation by:
O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, California 95472

Copyright © 2012 by Andrew Whitechapel.
All rights reserved. No part of the contents of this book may be reproduced or transmitted in any form or by any means without the written permission of the publisher.

ISBN: 978-0-7356-6325-1

1 2 3 4 5 6 7 8 9 LSI 6 5 4 3 2 1

Printed and bound in the United States of America.

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

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.

The example companies, organizations, products, domain names, email addresses, logos, people, places, and events depicted herein are fictitious. No association with any real company, organization, product, domain name, email address, logo, person, place, or event is intended or should be inferred.

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, O'Reilly Media, Inc., 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 Editor: Russell Jones

Developmental Editor: Russell Jones

Production Editor: Melanie Yarbrough

Editorial Production: Octal Publishing, Inc.

Technical Reviewer: Peter Torr

Copyeditor: Bob Russell

Indexer: WordCo Indexing Services

Cover Design: Twist Creative • Seattle

Cover Composition: Karen Montgomery

Illustrator: Robert Romano

Contents at a Glance

	<i>Foreword</i>	<i>xvii</i>
	<i>Introduction</i>	<i>xix</i>
PART I	BUILDING BLOCKS	
CHAPTER 1	Vision and Architecture	3
CHAPTER 2	UI Core	29
CHAPTER 3	Controls	61
CHAPTER 4	Data Binding and Layer Decoupling	91
CHAPTER 5	Touch UI	135
PART II	APPLICATION MODEL	
CHAPTER 6	Application Model	175
CHAPTER 7	Navigation State and Storage	199
CHAPTER 8	Diagnostics and Debugging	243
PART III	EXTENDED SERVICES	
CHAPTER 9	Phone Services	291
CHAPTER 10	Media Services	319
CHAPTER 11	Web and Cloud	349
CHAPTER 12	Push Notifications	409
CHAPTER 13	Security	445
CHAPTER 14	Go to Market	499
PART IV	VERSION 7.5 ENHANCEMENTS	
CHAPTER 15	Multi-Tasking and Fast App Switching	553
CHAPTER 16	Enhanced Phone Services	589
CHAPTER 17	Enhanced Connectivity Features	627
CHAPTER 18	Data Support	667
CHAPTER 19	Framework Enhancements	711
CHAPTER 20	Tooling Enhancements	745
	<i>Index</i>	<i>773</i>

Contents

Foreword xvii

Introduction xix

PART I BUILDING BLOCKS

Chapter 1	Vision and Architecture	3
Windows Phone Vision		3
Metro		4
Developer Guidelines		7
Windows Phone Architecture		8
Comparison of Silverlight and XNA		9
Developer Tools.		10
Development Cycle		11
The Anatomy of a Basic Windows Phone Application		13
XAP Contents		14
Standard Project Types		15
Themes and Accent Colors.		17
Standard Application Images.		22
Version 7 vs. Version 7.1.		24
Summary.		27

Chapter 2	UI Core	29
Phone UI Elements		29
Standard UI Elements		29
Visual Tree		32
Screen Layout		38
<i>UserControl</i> vs. Custom Control		41
Routed Events		44
Resources		47
Content vs. Resource		48
Resource Dictionaries		49

Dependency and Attached Properties	54
Dependency Properties	54
Attached Properties.	56
Summary.	59
Chapter 3 Controls	61
Standard Controls	61
Platform, SDK, Toolkit	61
SDK Controls: <i>Pivot</i>	63
SDK Controls: <i>Panorama</i>	69
Toolkit Controls	78
Transient Panels.	81
Summary.	89
Chapter 4 Data Binding and Layer Decoupling	91
Life without Data Binding	91
Simple Data Binding and <i>INotifyPropertyChanged</i>	94
Data Binding Collections	98
Data Templates.	100
Dynamic Data-Bound Collections.	103
Template Resources.	105
Type/Value Converters.	106
Element Binding	108
Data Validation	110
Separating Concerns	113
Design-Time Data	115
The Model-View ViewModel Pattern	117
The Visual Studio Databound Application Project	124
Summary.	133

Chapter 5 Touch UI 135

Logical Touch Gestures	135
Manipulation Events: Single Touch (Tap)	138
Manipulation Events: Single Touch (Flick)	140
Manipulation Events: Multi-Touch	142
Mouse Events	144
<i>FrameReported</i> Events	147
Combining Manipulation and Mouse Events	148
Click vs. Mouse/Manipulation Events	150
The Silverlight Toolkit <i>GestureService</i>	150
Pinch and Drag	152
Flick and Tap	153
Problems with the <i>GestureService</i>	154
Behaviors	155
Keyboard Input	159
Orientation	162
The Application Bar	167
Summary	172

PART II APPLICATION MODEL

Chapter 6 Application Model 175

Lifetime Events and Tombstoning	175
Application Closing	180
Application Deactivated	181
Application Deactivated (the Non-Tombstone Case)	183
Unhandled Exceptions	185
Why Is There No App.Exit?	186
Obscured and Unobscured	186
Launchers/Choosers and Tombstoning	189
User Expectations	189

Page Model	190
Page Creation Order	195
Summary.	198
Chapter 7 Navigation State and Storage	199
Navigation and State	199
Application and Page State	200
Detecting Resurrection	207
Navigation Options.	209
Using <i>NavigateUri</i>	209
Pages in Separate Assemblies	210
<i>Fragment</i> and <i>QueryString</i>	211
The <i>NavigationMode</i> Property	214
Rerouting Navigation and URI Mappers	215
Nonlinear Navigation Service	217
Isolated Storage	220
Simple Persistence	221
Persisting the ViewModel.	225
Serialization Options	229
Isolated Storage Helpers.	237
Summary.	241
Chapter 8 Diagnostics and Debugging	243
Visual Studio Debugging.	243
Simple Diagnostics	244
Setting Up a Diagnostics Pop-Up Window	244
Fixed Diagnostics Control.	249
Post-Release Diagnostics	251
Persisting Logs	253
Configurable Diagnostics	255
Screen Capture	259
Emulator Console Output.	261
Debugging Tombstoning and Lock-Screen.	263
Debugging MediaPlayer.	264

Device and User Information	267
Windows Phone Performance Counters.....	271
Memory Diagnostics	273
The Device Emulator.....	278
Emulator vs. Device	279
XDE Automation.....	280
Using the Microsoft Network Monitor	283
Fiddler.....	285
Silverlight Spy	287
Summary.....	288

PART III EXTENDED SERVICES

Chapter 9 Phone Services 291

Phone Hardware	291
Launchers and Choosers	293
Photo Extras	297
Accelerometer	301
Reactive Extensions for .NET	304
Level Starter Kit	307
Shake	311
Geo-Location.....	314
Summary	318

Chapter 10 Media Services 319

Audio and Video Hardware.....	319
Audio and Video APIs.....	320
Media Playback	320
The <i>MediaPlayerLauncher</i> Class.....	321
The <i>MediaElement</i> Class	321
The <i>MediaStreamSource</i> and <i>ManagedMediaHelpers</i> Classes....	323
<i>MediaElement</i> Controls.....	325

Audio Input and Manipulation	328
The <i>SoundEffect</i> and <i>SoundEffectInstance</i> Classes	329
Audio Input and the Microphone	331
The <i>DynamicSoundEffectInstance</i> Class	339
Music and Videos Hub	343
The FM Tuner	345
Summary	348

Chapter 11 Web and Cloud 349

The <i>WebClient</i> Class	349
<i>WebClient</i> : The <i>DownloadStringAsync</i> Method	349
<i>WebClient</i> : The <i>OpenReadAsync</i> Method	351
The <i>HttpWebRequest</i> Class	353
Web Browser Control	353
Silverlight and Javascript	355
Web Services	358
WCF Data Services	361
The OData Client and XML Data	361
JSON-Formatted Data	369
Bing Maps and Geolocation	372
Using the Map Control	372
Geolocation	374
Bing Maps Web Services	375
Deep Zoom (<i>MultiScaleImage</i>)	378
Windows Azure	383
Windows Azure Web Services	385
Windows Azure Toolkit for Windows Phone	390
bitly	394
Facebook	396
Windows Live	400
SkyDrive	405
Summary	407

Chapter 12 Push Notifications 409

Architecture	409
Push Notification Server	412
Push Notification Client	418
Additional Server Features	423
Batching Intervals	423
XML Payload	424
Response Information	426
Additional Client Features	427
Persistent Client Settings	427
The <i>ErrorOccurred</i> Event	428
User Opt-In/Out	429
Implementing a Push ViewModel	431
The Push Notification Server-Side Helper Library	437
Common Push Notification Service	439
Summary	443

Chapter 13 Security 445

Device Security	445
Application Safeguards	446
Application Deployment	447
Managed Code Constraints	449
Chambers and Capabilities	451
Missing Security Features	454
Data Encryption	455
SDL Tools	460
Threat Modeling	461
Static Code Analysis/FxCop	462
Web Service Security	467
Authentication	468
Forms Authentication	468
Basic Authentication	479
SSL	484

Push Notification Security	490
OAuth 1.0	491
OAuth 2.0	493
Securing Web Service IDs	494
Implementing Security for the <i>WebBrowser</i> Control	496
Summary.	497

Chapter 14 Go to Market 499

Threading	499
Performance	505
UI vs. Render Thread, and <i>BitmapCache</i> Mode	505
UI Layout and <i>ListBoxes</i>	512
More UI Performance Tips	513
Non-UI Performance Tips.	515
Silverlight Unit Testing Framework	517
Certification and Publication	523
Updates	530
Marketplace Reports	533
Beta Testing.	534
Versions.	534
Light-Up Features.	535
Obfuscation	537
Ads	540
Trial Mode	544
Silverlight Analytics Framework	546
Summary.	549

Fast Application Switching	553
Multi-Tasking	557
Alarms and Reminders	558
Alarms	558
Reminders	561
Background Transfer Service	564
Generic Background Agents	568
Background Audio	578
Background Audio: The Main Application	583
Background Audio: The Background Agent	585
Summary	587

Chapter 16 Enhanced Phone Services 589

Sensor APIs.....	589
Accelerometer.....	590
Compass.....	593
Gyroscope.....	598
Motion APIs.....	601
Camera Pipeline.....	606
Augmented Reality.....	610
The Geo Augmented Reality Toolkit.....	614
New Photo Extensibility.....	616
Launcher and Chooser Enhancements.....	619
The <i>DeviceStatus</i> and <i>DeviceNetworkInformation</i> classes.....	621
Version 7.1.1.....	623
Summary.....	626

Chapter 17 Enhanced Connectivity Features 627

Push, Tile, and Toast Enhancements	627
Local Tiles	628
Pinning Tiles	632
Push Enhancements.	638
Sockets	642
TCP Sockets	643
OData Client.	650
Search Extensibility.	657
App Connect	657
App Instant Answer	664
Summary.	665

Chapter 18 Data Support 667

Local Database and LINQ-to-SQL	667
Create and Read.	669
Update and Delete.	675
Schema Updates.	677
Associations.	681
Isolated Storage Explorer Tool.	684
Performance Considerations	692
Database Encryption	695
Encrypting Data and Credentials	697
Contacts and Calendar.	699
Sync Framework	703
Service Configuration	705
Database Provisioning.	707
Code Generation	707
Summary.	710

Chapter 19 Framework Enhancements **711**

Navigation Enhancements.	711
Frame and Page Navigation.	711
Backstack Management	714
UI Enhancements	717
Enhanced Controls.	718
The <i>AppBar</i> and <i>SystemTray</i> Classes, and the <i>ProgressIndicator</i> Property	723
The Clipboard API	727
32 Bits per Pixel	728
Background Image Decoding	729
Touch Thread	729
Silverlight 4.0	730
Implicit Styles	730
Command Binding.	732
Data-Binding Enhancements	736
Summary.	744

Chapter 20 Tooling Enhancements **745**

Emulator Improvements	745
Debugger Experience.	747
Marketplace Test Kit.	749
The Profiler.	754
UserVoice Forums.	764
Portable Library Tools	765
Async Framework	769
Summary.	772

<i>Index</i>	773
------------------------	-----

Foreword

So, you're curious about Windows Phone development? Welcome aboard! Whether you're an existing Microsoft Silverlight developer wanting to branch out into the mobile space, an existing mobile developer looking to extend your reach across a second or third ecosystem, a rising star who's ready to create the Next Big Thing and take the world by storm, or maybe just a curious phone user who wants to know what all the "app" fuss is about, Windows Phone is the platform for you.

Getting started with Windows Phone development is free and easy; everything you need to write apps is just a couple of clicks away. You can have your first app up and running in a matter of minutes, even if you know next to nothing about Windows Phone development or don't even own a device. As your apps become more ambitious and you encounter more complex development issues, a vibrant developer community on the web is ready and willing to help you out along the way. Mastery of this platform, with its rich feature set, unique application model, integrated end-to-end experiences, and burgeoning international marketplace, takes time and effort—and an expertly written guide. Luckily for you, this book is just such a guide.

The Windows Phone platform stands on the shoulders of giants—giants such as Silverlight, XNA, Microsoft Visual Studio, and Microsoft Expression Blend—and as we built the platform, we embraced the power and familiarity that these existing technologies afforded us; our goal was to introduce new concepts only when strictly necessary to enable new scenarios, and to re-use existing concepts everywhere else. We spent less time re-solving old problems (such as navigation and microphone capture) and more time tackling new ones so that we could ship a vast array of new phone-specific APIs for developers to wrap their heads around—cameras, gyroscopes, multi-tasking, phone integration, user data access, and live tile updates, just to name a few—and Andrew covers all of them (and more!) in this book.

As you've probably heard (and seen), Windows Phone ushered in a new design language for Microsoft, code-named "Metro." Adhering closely to this design is critical when building user experiences that will delight and engage your customers. As you would expect, the Windows Phone developer tools give you a big helping hand in this department, providing user interface elements and application templates that "just work" by default. Nevertheless, as the owner of your application's overall experience you are ultimately responsible for ensuring it performs optimally, adhering not just to the graphic design rules, such as "content over chrome," but also the fast and fluid interaction model that your customers will come to expect. Throughout this book you'll find practical examples and guidance that show how to embody the Metro design language in your applications, along with examples of common pitfalls and how to avoid them—particularly with respect to application performance and responsiveness, which are key factors in user satisfaction (and hence, app ratings and profitability).

On a more personal note, I was thrilled when Andrew asked me to tech-review this book (although writing this foreword was more than a little daunting!). As an infrastructure guy at heart, I love building platforms and enabling developers to be successful on top of them, but there are only so many people you can reach via blogs or conference speaking sessions. I've been asked to author books before, but I've never had the time or inclination to do so. By piggybacking on Andrew's hard work with this book, I feel like I've made a difference—if only a small one—and that makes me grateful for the opportunity. I also learned a lot while reviewing this book, and I know that you will, too.

Peter Torr

*Program Manager in the
Windows Phone Application
Platform team*

Introduction

The smart phone is increasingly important in people's daily lives. It is used for a wide variety of tasks, both work-related and non-work related. People use smart phones to keep up to date with friends and family, for relaxation, and for entertainment, as well as for viewing documents and spreadsheets, surfing the Internet, and enriching their lives. There is therefore considerable scope for building smart phone applications. Windows Phone is not just another smart phone; rather, it is positioned as an opportunity for developers to build applications that can make a real difference to people's lives. The platform has been designed from the ground up to support an all-encompassing, integrated, and attractive user experience.

Windows Phone 7 Development Internals covers the breadth of application development for the Windows Phone platform, both the major 7 and 7.1/7.5 versions and the minor 7.1.1 version, and shows how you can build such compelling and useful applications. You can build applications for Windows Phone 7.x by using either the Microsoft Silverlight runtime or the XNA runtime. This book focuses on Silverlight applications. The primary development and design tools are Microsoft Visual Studio and Microsoft Expression Blend. Here again, this book focuses on Visual Studio.

Each chapter covers a handful of related features. For each feature, the book provides one or more sample applications and walks you through the significant code (C# and XAML). This will help you to understand the techniques used and also the design and implementation choices that you have in each case. Potential pitfalls are called out, as are scenarios in which you can typically make performance or user experience improvements. An underlying theme is to conform not only to the user interface design guidelines, but also to the notion of a balanced, healthy phone ecosystem.

Who Should Read This Book

This book is intended to help existing developers understand the core concepts, the significant programmable feature areas, and the major techniques in Windows Phone development. The book is tailored for existing Silverlight developers that want to jump into the exciting world of mobile application developer with the Windows Phone platform. Developers experienced with other mobile platforms will find this book invaluable in learning the ins and outs of Microsoft's operating system, but will likely need additional resources to pick up C# and XAML languages.

The Windows Phone 7 release only supports C#, and although support for Visual Basic was introduced with the 7.1 SDK, this book focuses purely on C# and XAML. The basic architecture of the platform is covered in Chapter 1, "Vision and Architecture," and most chapters go deeply into the internal behavior of the system. This is knowledge that helps to round out your understanding of the platform, and inform your design decisions, even though, in some cases, the internal details have no immediate impact on the exposed API.

Assumptions

The book assumes that you have a reasonable level of experience of developing in managed code, specifically in C#. Basic language constructs are not discussed, nor is basic use of Visual Studio, the project system or the debugger, although more advanced techniques, and phone-specific features are, of course, explained in detail. You should also have some knowledge of XAML development, preferably in Silverlight, although Windows Presentation Foundation experience would also be useful background.

Although many component-level diagrams are presented as high-level abstractions, there are also many sections that describe the behavior of the feature in question through the use of UML sequence diagrams. It helps to have an understanding of sequence diagrams, but it is not essential, as they are fairly self-explanatory.

Who Should Not Read This Book

This book is not intended for use by application designers—if designers are defined as developers who use Expression Blend—although designers might find it useful to understand some of the issues facing developers in the Windows Phone application space. The book is also not suitable for XNA developers because it does not cover game development at all.

Organization of This Book

Windows Phone 7 was first released in October 2010. The first major update, code-named “Mango,” was released in September 2011. The Mango release includes a wide range of enhancements and additional features. Note that the user-focused version number for the Mango release (that is, the product version) is version 7.5; however, the developer-focused number is 7.1 (for both the OS version and the SDK version). The reason for this slightly confusing numbering situation is that the Mango release includes improvements across the board—in the operating system, the developer tooling, in the emulator, in the application platform, in Silverlight itself, and also in the server-side experience of marketplace, and in ingestion. All of this is Windows Phone, or Windows Phone 7.5. A developer is normally focused more on the pure technical aspects: the operating system, tooling, and application platform subset of the overall release, and that is technically the 7.1 release (both SDK and OS).

This book covers all 7.x versions: the original Windows Phone 7 release, the later Windows Phone 7.1 release, and the minor 7.1.1 release. Applications built for version 7 also work without change on 7.1 devices. Note that, while there are still about a million version 7 phones in use, it is safe to assume that most of these will be upgraded to 7.1 at some point. However, to keep things simple, the first 14 chapters focus on the basic infrastructure, programming model, and the core features that are common to both versions. Where there are material differences, these are called out, with references to the later chapter where the 7.1 behavior is explained in detail. Chapter 15, “Multi-Tasking and Fast App Switching,” onward focuses on the features and platform enhancements that are specific to version 7.1.

The 7.1.1 version is a narrowly scoped release intended to support phones with low memory capabilities (256 MB) for specific target markets. Most developers—and most applications—will not be affected by this. For the small number that might be affected, the 7.1.1 release provides additional support for performance tuning and an additional marketplace submission option, as discussed in Chapter 16, “Enhanced Phone Services.”

It’s also worth reading Chapter 14, “Go To Market” ahead of time. This chapter focuses on the end-game of bringing your application to market, including tuning the design for performance and robustness, and marketplace certification. Even before you have a thorough understanding of the architecture and fundamentals, it is instructive to see what you’ll be aiming for.

Conventions and Features in This Book

This book presents information by using conventions designed to make the information readable and easy to follow.

- In some cases, especially in the early chapters, application code is listed in its entirety. More often, only the significant code is listed. Wherever code has been omitted for the sake of brevity, this is called out in the listing. In all cases, you can refer to the sample code that accompanies this book for complete listings.
- In the XAML listings, attributes that are not relevant to the topic under discussion, and that have already been explained in previous sections, are omitted. This applies, for example, to *Grid.Row*, *Grid.Column*, *Margin*, *FontSize*, and similarly trivial attributes. In this way, you can focus on the elements and attributes that do actually contribute to the feature at hand, without irrelevant distractions.
- Code identifiers (the names for classes, methods, properties, events, enum values, and so on) are all italicized in the text.
- In the few cases where two or more listings are given with the explicit aim of comparing alternative techniques (or “before” and “after” scenarios), the differences appear in bold.
- Boxed elements with labels such as “Note” provide additional information or alternative methods for completing a step successfully.
- Text that you type (apart from code blocks) appears in bold.
- A plus sign (+) between two key names means that you must press those keys at the same time. For example, “Press Alt+Tab” means that you hold down the Alt key while you press the Tab key.
- A vertical bar between two or more menu items (for example, File | Close), means that you should select the first menu or menu item, then the next, and so on.

System Requirements

You can build and run the accompanying sample code, or you can create your own solutions from scratch, following the instructions in the text. In either case, you will need the following hardware and software to create the sample applications in this book:

- Either Windows Vista (x86 and x64) with Service Pack 2, all editions except the Starter Edition, or Windows 7 (x86 and x64), all editions except the Starter Edition. If you install the Windows Phone SDK 7.1.1 Update, this also works with the Windows 8 Consumer Preview, although this is not a supported configuration.
- The Windows Phone SDK version 7.0 or 7.1. These are both free downloads that include Visual Studio 2010 Express Edition and all other standard tools, as listed in Chapter 1. If you install the SDK version 7.1, you can then also upgrade this with the SDK version 7.1.1. This is an update to 7.1, not a stand-alone install.
- Some of the server-side sample projects require Visual Studio Professional, but all of the Windows Phone samples work with Visual Studio Express.
- Installing the SDK requires 4 GB of free disk space on the system drive. If you use the profiler (described in Chapter 20, “Tooling Enhancements”) for an extended period, you will need considerably more disk space.
- 4 GB RAM (8 GB recommended).
- Windows Phone Emulator requires a DirectX 10 or above capable graphics card with a WDDM 1.1 driver.
- 2.6 GHz or faster processor (4GHz or 2.6GHz dual-core, recommended).
- Internet connection to download additional software or chapter examples, and for testing web-related applications.

Depending on your Windows configuration, you might require Local Administrator rights to install or configure Visual Studio 2010, and to install or configure features such as Internet Information Services, if not already installed.

For the latest requirements, visit the Windows Phone SDK download page at <http://www.microsoft.com/download/en/details.aspx?id=27570>.

Code Samples

All of the chapters in this book include multiple sample solutions with which you can interactively try out new material learned in the main text. All sample projects can be downloaded from the following page:

<http://go.microsoft.com/fwlink/?Linkid=248889>

Follow the instructions to download the WP7xDevInternals.zip file.

Installing the Code Samples

Follow these steps to install the code samples on your computer so that you can refer to them while learning about the techniques that they demonstrate.

1. Unzip the WP7xDevInternals.zip file that you downloaded from the book's website to any suitable folder on your local hard disk. The sample code expands out to nearly 200 MB, and you will need even more space for the binaries if you choose to build any of the samples.
2. If prompted, review the displayed end-user license agreement. If you accept the terms, select the accept option, and then click Next.



Note If the license agreement doesn't appear, you can access it from the same webpage from which you downloaded the WP7xDevInternals.zip file.

Using the Code Samples

When you unzip the sample code, this creates a number of subfolders, one for each chapter. Within each chapter's subfolder there are further subfolders. In most cases, there is one subfolder per application (or per version of an application), but in some cases, multiple applications are grouped together; for example, where there is a server-side application as well as a client-side application in the solution. In keeping with the book's structure, the samples for the first 14 chapters were built as version 7 projects, and the remaining samples were built as version 7.1 projects. However, you can use the version 7.1 (or 7.1.1) SDK for all the sample projects, for all chapters.

All of the samples are complete, fully functioning applications. Note, however, that in some cases, you might need to update assembly references, depending on where you install the SDK as well as where you install supplementary libraries and frameworks that don't ship with the main SDK (for instance the Silverlight toolkit, Live SDK, Azure toolkit, and so on).

For samples that demonstrate the use of some supplementary framework, you will need to download and install that framework so that you can reference its assemblies. Also note that, in some cases, this requires a user ID, such as for Bing maps, FaceBook, or Google Analytics, as described in the relevant sections. In all cases, you can sign up for the ID without charge as of the time of this writing.

Acknowledgments

The Windows Phone development space is truly inspiring, and the Windows Phone teams at Microsoft are chock-full of smart, helpful people. The list of folks who helped me prepare this book is very long. I'd particularly like to thank Peter Torr for doing all the heavy lifting in the technical review. It's mainly thanks to Peter that this book isn't riddled with schoolboy errors. In addition, I'd like to thank all the other people who answered my dumb questions, and corrected my various misinterpretations of the internal workings of the platform, especially Tim Kurtzman, Wei Zhang, Jason Fuller, Vij Vasu, Abolade Gbadegesin, Andrew Clinick, Darin Miller, Mark Paley, Jeff Wilcox, Thomas Fennel, Matt Klupchak, Alper Selcuk, Gary Lin, Conrad Chang, Justin Horst, Sai Prasad Patro, Yasser Shaaban, Mike Battista, Jorge Raastroem, and Joao Guberman Raza. I'd also like to thank the folks at O'Reilly Media and Microsoft Press who helped to turn my scattered thoughts into polished prose, especially Russell Jones, Devon Musgrave, Melanie Yarbrough, and Bob Russell (at Octal Publishing, Inc.). Finally, none of this would have been possible without the patience and support of Narins Bergstrom.

Errata & Book Support

We've made every effort to ensure the accuracy of this book and its companion content. Any errors that have been reported since this book was published are listed on our Microsoft Press site at oreilly.com:

<http://go.microsoft.com/fwlink/?Linkid=248888>

If you find an error that is not already listed, you can report it to us through the same page.

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

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

We Want to Hear from You

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

<http://www.microsoft.com/learning/booksurvey>

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

Stay in Touch

Let's keep the conversation going! We're on Twitter: <http://twitter.com/MicrosoftPress>

PART I

Building Blocks

CHAPTER 1	Vision and Architecture	3
CHAPTER 2	UI Core	29
CHAPTER 3	Controls	61
CHAPTER 4	Data Binding and Layer Decoupling	91
CHAPTER 5	Touch UI	135

This part describes the overall vision and architecture of Windows Phone applications, and the basic building blocks that every application requires in order to support minimum functionality. This includes the core user interface features, the various standard controls, touch manipulation, and data-binding.

Vision and Architecture

There are two dimensions to this chapter: an introduction to the vision of the Windows Phone platform and the revolutionary Metro design philosophy, and an introduction to the basic architecture and building blocks of a Windows Phone application. The vision is extremely important; it should inform the way you think about applications, from first concept through design to implementation and final publication. The application platform provides support and guidance, such that it's quite difficult to build a truly bad application, but it is definitely possible to build an application that is in conflict with the system and fails to provide a good user experience (UX). An understanding of the vision should steer you toward good design choices and help you to identify potential pitfalls, thereby accelerating your development cycle.

Windows Phone Vision

"Windows Phone is a software platform that can improve a person's life." This is a quote from the *Microsoft User Experience Design Guidelines* (kept up-to-date online at [http://msdn.microsoft.com/en-us/library/hh202915\(VS.92\).aspx](http://msdn.microsoft.com/en-us/library/hh202915(VS.92).aspx)). It is a philosophy on which the teams that are responsible for building the Windows Phone operating system, application platform, and SDK are highly focused. It's not just marketing-speak, and it's not just an inspirational motto—it really does speak to the core vision of the platform.

Traditional desktop operating systems such as Windows 7 and Mac OS are used for all sorts of reasons and purposes, but predominantly for work-related activities. So applications built for the desktop (including browser clients for web applications) are typically business-focused (although buttoned-down PCs are also increasingly being used for social networking applications like Facebook). Applications for Microsoft Xbox and other game consoles are predominantly geared toward entertainment. The idea behind the Windows Phone UX model is that this is a device that users turn to for everything—for work and play, for keeping in touch with friends and family, for relaxation, games, music and videos, for research and fact-finding—in short, for enhancing their day-to-day lives. As developers, you have the weighty responsibility of living up to this vision. You are encouraged (and extensively supported through the application platform) to build applications that *do* improve the user's life.

A good Windows Phone application is engaging, compelling, and attractive to the user. A *great* Windows Phone application can make a meaningful contribution to the user's life. The user interaction is well thought out, predictable, and familiar. It gives him enjoyment every time he uses it, or it fulfills some pragmatic need in a simple, unobtrusive manner. The application integrates seamlessly

with standard features on the phone, such as email, contacts, photos, and so on. It's a "team player," cooperating with the system. It's conservative in its use of CPU, memory, and sensor resources. It doesn't consume battery power in an unbounded way, and it is thoughtful about its use of the user's data plan. The application should have a way to keep itself up-to-date so that its data is always fresh. It can also be location-aware; this way, it can ensure that it is always relevant to the user's context, and functions in the smart way that user's expect from a "smart" phone.

A good application should also look and feel like it's a part of the overall phone ecosystem, using the Metro design guidelines in an appropriate way, so that it doesn't jar the senses or look out of place. The same applies to navigation and the way the application behaves when the user switches away from it, and perhaps comes back to it later. It should be configurable, so that the user can personalize it, just as he can personalize other parts of the phone experience, such as the Start page, the theme, and accent colors. There should be a way for the user to provide feedback to the developer, to report bugs, or to request new features. The user should be able to consider the phone as an extension of his personality, configured the way he likes it, and tailoring the set of installed applications to suit his own personal needs and preferences.

Metro

Metro is the code name for the design language used in establishing the UX for Windows Phone. Metro is modern and clean, light, open, and fast. Metro is alive and in motion—hence, the extensive use of page transition animations and the considerable thought that has gone into perfecting the standard animations to provide a visually pleasing experience. Consider just one feature: when the user swipes to scroll through a list, the scrolling speed closely matches the force of the swipe gesture. It responds immediately when the user holds a finger on the screen to stop the scrolling. If the user drags a list down (or up) to its limits, there's a visible compression effect, which provides clear feedback that she's reached the end of the list. When she releases her finger, the compression bounces back in a pleasing and intrinsically recognizable way that makes the phone seem almost organically alive. The standard animations have all been designed to provide just the right level of feedback—again, in a pleasing way, without becoming intrusive or distracting.

You are encouraged to use the Metro design style for your application. Metro covers all aspects of the user interface (UI), including the use of space, control styles, and typography. Content is seen as an important part of any application; this is in deliberate and emphatic contrast to chrome (that is, artifacts of the UI that allow the user to manipulate the UI, such as buttons and grab-handles that are presented in addition to the actual content). Other phone platforms focus a lot on chrome—shiny buttons, gradient fills, 3D images, arbitrary custom animations, and so on. All these chrome effects might seem entertaining for a brief period, but they soon become a distraction, and they also lead to a cluttered, inconsistent, confusing UI. By contrast, Windows Phone and Metro considers content to be king, whereas chrome should always be unobtrusive: it should serve its purpose, and then get out of the user's way. Think about the implementation of the system tray; most of the time, this is invisible. If the user wants to see it, she taps the top of the screen, the tray drops down with a slight bounce, stays around long enough for the user to see the information she needs, and then disappears, leaving

the screen free for the application. It also has no shiny, colorful, arbitrarily animated distractions. The same applies for the application bar.

The aim is to achieve a UI that looks and feels clean, light, and open. Figure 1-1 shows an example of a panorama-based application in the spirit of Metro: clean lines, no gradient fills, content-driven.

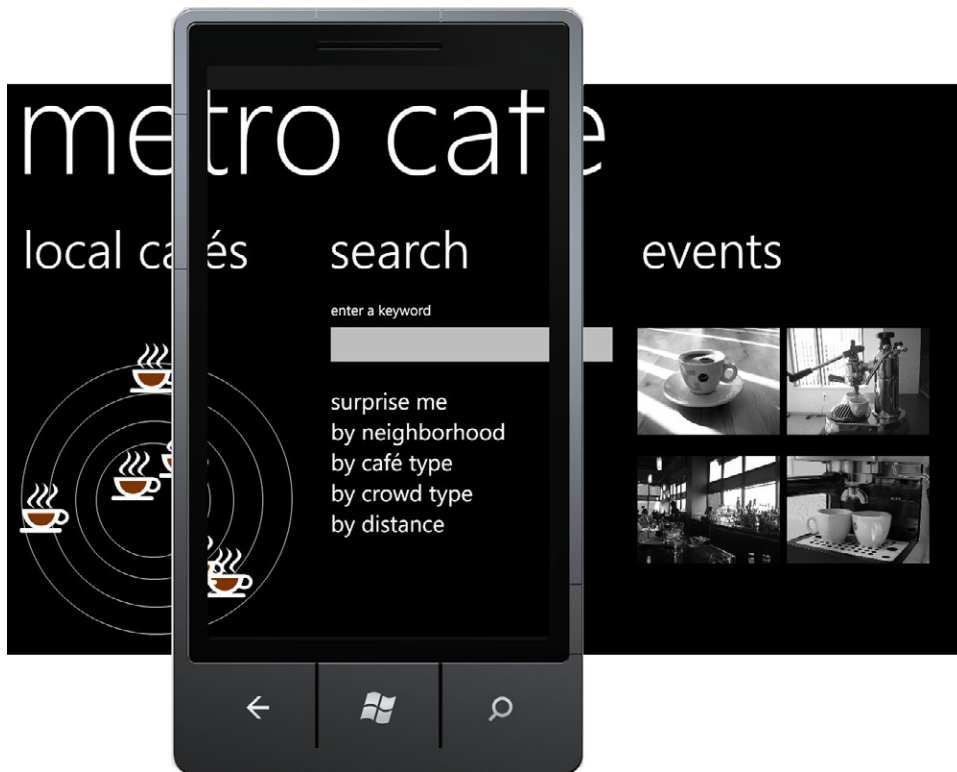


FIGURE 1-1 Metro applications should be clean and open.

This really represents a major shift in the way you design applications. What you might call the “90s aesthetic” was all about iconography, whereby the chrome took center stage, and the model was hyper-realistic. Metro brings a content-focused aesthetic, through which content is represented in its cleanest form, without embellishment or adornment, and without distractions. The premise is that the user can get to the information directly, without having to navigate some developer’s view of what is suitable UI decoration. Consider, for example, the scrollbar. In Metro, this is mostly invisible; the user scrolls the *content*, not some shiny scrollbar chrome thumb or widget. This is especially important on a mobile device, given the smaller screen real estate and the use of a touch-based input model.

Metro celebrates typography: after all, this is a large part of what the user will see in all applications, so the typography must be true to the Metro principles. To embody this principle, the font selected as the standard system font on Windows Phone is Segoe WP (in regular, bold, semi-bold, semi-light and black variants), as shown in Figure 1-2.

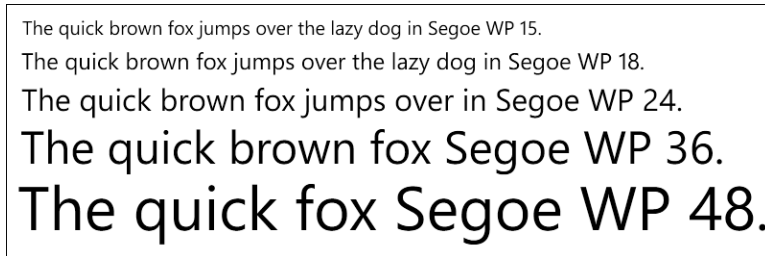


FIGURE 1-2 Segoe WP is the standard font on Windows Phone.

The *FontSize* values for the standard theme fonts are listed in Table 1-1.

TABLE 1-1 Standard Theme Font Sizes

Resource Name	Size in Points	Size in Pixels
<i>PhoneFontSizeSmall</i>	14	18.667
<i>PhoneFontSizeNormal</i>	15	20
<i>PhoneFontSizeMedium</i>	17	22.667
<i>PhoneFontSizeMediumLarge</i>	19	25.334
<i>PhoneFontSizeLarge</i>	24	32
<i>PhoneFontSizeExtraLarge</i>	32	42.667
<i>PhoneFontSizeExtraExtraLarge</i>	54	72
<i>PhoneFontSizeHuge</i>	140	186.667

In addition to the design guidelines, two further UI/UX guidance resources are available from MSDN at [http://msdn.microsoft.com/en-us/library/ff637515\(VS.92\).aspx](http://msdn.microsoft.com/en-us/library/ff637515(VS.92).aspx).

- **The Windows Phone Design System—Codename Metro** A visual explanation of the inspiration behind the Windows Phone design system, including the Red Threads principles (Personal, Relevant, Connected).
- **Design Templates for Windows Phone 7** A collection of 28 layered Photoshop templates that help designers to maintain a consistent look and feel across applications.

The focus is on building applications that can take part in a holistic UX. This extends beyond the visuals to include all aspects of the application. Navigation, for example, is deliberately enforced to be very simple, very linear, and predictable. The user can quickly become familiar with the standard navigation model, and all applications follow this model. Thus, the user can quickly become familiar with any new application. There are constraints on what your application can do to avoid interfering with this predictability. The user must be able to trust the hardware, so all the hardware buttons always behave the same way, and you cannot override this in a damaging way. The Start button always goes to the Start page. The Back button always goes back in navigation (both within an application's pages, and between applications in the backstack). When the user backs up to the beginning of the backstack, he always ends up at the Start page.

Adhering to Metro design principles is not a burden: it actually frees you to concentrate on making your application the best it can be, because you don't have to spend time on inventing a new design paradigm. Instead, you can focus on making your application truly useful, compelling, and delightful, such that it offers a benefit to the user's daily life. Note that Metro is also being applied within Microsoft for Windows 8 and for Xbox. It will be the dominant design aesthetic for the next several years.

Developer Guidelines

If you want to build an ecosystem in which the inhabitants co-exist in harmony to their mutual benefit, then there must be rules and guidelines. Rules in Windows Phone development are enforced by the operating system, the application platform, and the platform API. There are certain programming techniques that might be appropriate in a desktop development environment, but you simply cannot do them in Windows Phone. You cannot currently build a Windows Phone application using native code. You also cannot use COM or RPC, or indeed, any kind of inter-process communication. Every application is strictly "sandboxed" and has no access to any other application. You cannot consume CPU cycles (and battery power) when you're not running in the foreground. However careless you might be in handling exceptions, you cannot bring down the entire system. Clearly, these rules have been put in place to optimize the stability of the phone and the overall UX.

Then there are guidelines. These are recommendations for how your application should look and behave, but they are not explicitly enforced by the phone itself. Rather, they are enforced by the marketplace: if you build a bad application, users will quickly uninstall it and give it a bad review. You're not forced to use Metro in your design, but if you don't, you might end up with an application that clashes with the ecosystem. You're not constrained in how much disk space you use up with application data, but if you use too much, the user will soon realize your application is making it difficult to run or install other applications. You can use more than 90 MB of memory at runtime, but sooner or later the system will run up against a memory cap, and then your application will crash. The Windows Phone marketplace publishes a comprehensive set of certification requirements, and the developer has a lot of support (especially in Windows Phone SDK 7.1) in preparing applications for publication. However, the marketplace performs only a small set of automated and manual tests; consequently, it is entirely possible to publish a bad application which then fails at runtime. Users typically have very low tolerance for things that crash, so this is very much a self-policing system.

In the market today, there is a spectrum of application models from iOS at one end (for which there's really only one logical device, and applications are severely constrained as to what they can and cannot do), to Android at the other (where there's a very wide and heterogeneous range of devices, and applications have very wide latitude to do all kinds of things, which might or might not function well, or at all, on every device). Windows Phone sits somewhere in the middle, offering the best of both worlds: there is a small set of supported devices, with a tightly controlled hardware requirements specification to which they must minimally adhere, and an application platform that offers a wide range of features, yet enforces some reasonable constraints in the interests of maintaining a consistent UX as well as overall device health.

Windows Phone Architecture

Apart from a few applications provided by Microsoft, the device manufacturer, or the service provider, Windows Phone 7 applications are developed in managed code, using either the Windows Phone version of the Microsoft Silverlight runtime or the Windows Phone version of the XNA runtime (or some combination). These are slightly modified, phone-specific versions of the standard Silverlight/XNA libraries. The phone application platform includes a set of standard controls and wrapper classes to the phone services. A high-level view of the architecture is shown in Figure 1-3. The underlying operating system is Windows CE, with the Microsoft .NET Compact Framework and the Silverlight or XNA runtime layered on top.

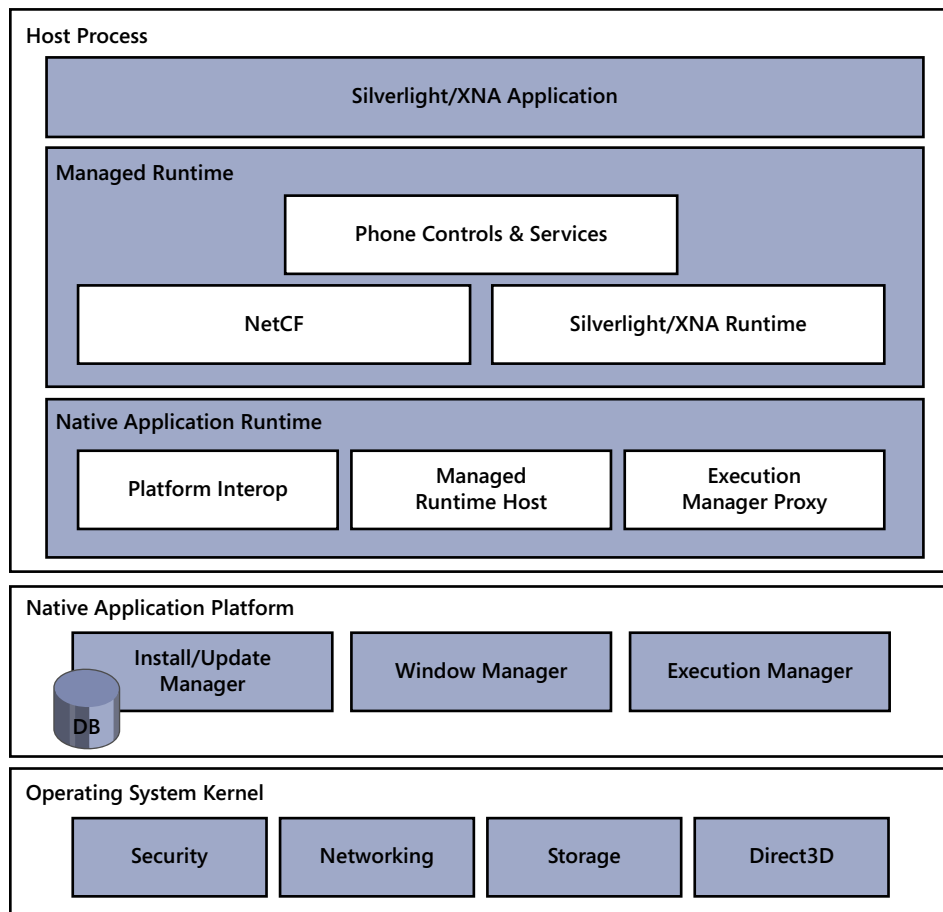


FIGURE 1-3 The Windows Phone application architecture has managed and native layers.

The application model exposed to marketplace developers is very robust. An important principle is that the core services of the phone must continue to function at all times, regardless of what custom applications might be installed or running.

From a UX perspective, this means that the user can rely on the following behavior:

- The user is always able to get back to the Start menu, lock the phone, and turn the phone off. She is always able to make and receive calls, navigate to hubs, and so on. The phone protects itself mainly by its application isolation/security model.
- A new application installs cleanly and cannot break any existing core feature of the phone or any other installed application. The same applies to updates. The architecture ensures that there can be no versioning conflicts with shared components, because applications cannot install shared components.
- There is no mechanism for inter-application communication or custom shared libraries. This removes the opportunity for a new application or update to destabilize any other application on the system.
- Uninstalling an application is always reliable and complete; nothing remains behind (with the exception of pictures that the application might have added to the picture gallery).
- The application platform manages application install/uninstall in a very controlled way, so it always knows what applications are installed and displays the list to the user. It's also not possible to have "hidden" or "uninstallable" applications on the phone—even applications from the carrier or mobile operator are visible to the user and can always be uninstalled. While you can deploy applications to a developer-unlocked phone directly, you cannot deploy to a retail phone without publishing to the marketplace.

Comparison of Silverlight and XNA

You can build applications for Windows Phone by using either Silverlight or XNA or a constrained combination of both, as described in Table 1.2.

TABLE 1-2 Comparison Between Silverlight and XNA Applications

Feature	Silverlight	XNA
Primary target app	N-tier business applications, tools.	Games.
Developer experience	XAML-driven, using Microsoft Expression and Microsoft Visual Studio, very similar to Windows Presentation Foundation (WPF) and desktop Silverlight development.	Visual Studio—code-driven. Also similar to Xbox console and Windows games development.
Typical artifacts	Built-in support for controls, data binding, web services, and text.	Models, meshes, sprites, textures.
Execution model	Event-driven (via UI controls or external events), and application-focused.	Gaming loop, display-focused.
Graphics	Retained mode graphics (application code updates an in-memory model of the graphics, which the OS renders later on), with 2D and limited 3D (PlaneProjection). Cannot use XNA graphics in 7.	Immediate mode graphics (application code causes direct rendering of graphics), with full 3D support.
Audio	Focused on simple media playback (music and video); can use XNA sounds and recording.	Multichannel audio, including recording.
Screen	Standard Silverlight visual tree model, with a hierarchy of controls.	The game always uses the full screen.

Feature	Silverlight	XNA
Controls	Rich set of standard controls.	None.
Data binding	Built in to the standard controls.	None.

Regardless of whether you choose Silverlight or XNA as the primary framework, you can also use some features from the other framework. In Windows Phone 7, a Silverlight application can use any of the XNA classes, except those in *Microsoft.Xna.Framework.Games* and *Microsoft.Xna.Framework.Graphics*. A common scenario is for a Silverlight application to use the advanced audio playback and recording support in XNA. Conversely, an XNA application can use some of the Silverlight features, but is more restricted; it cannot use any of the controls, data, messaging, input, media, browser, navigation, or threading features. Windows Phone 7.1 introduced significant support for combining the two runtimes in a seamless manner.

Developer Tools

The primary reference for Windows Phone platform APIs is MSDN, which is kept up to date online at [http://msdn.microsoft.com/en-us/library/ff402535\(VS.92\).aspx](http://msdn.microsoft.com/en-us/library/ff402535(VS.92).aspx). Another great resource is the AppHub, which you can access at <http://create.msdn.com>. This is a development portal for Windows Phone and Xbox 360 development, including links to download the core WP7 tools. The primary tools are listed in Table 1-3. Further ancillary tools and early-release frameworks are also mentioned in each chapter of this book, where appropriate.

TABLE 1-3 The Primary Windows Phone Development Tools are available from the AppHub.

Tool	Description
Windows Phone Developer Tools	This is the 7 version of the tools: you only need these if you want to use the 7 emulator. Otherwise, you can use the 7.1 or 7.1.1 SDK for developing both version 7 applications and version 7.1 applications. Available at http://www.microsoft.com/download/en/details.aspx?id=13890 .
Windows Phone Developer Tools January 2011 Update	An update to the version 7 tools. Available at http://www.microsoft.com/download/en/details.aspx?id=23854 .
Windows Phone SDK 7.1	<p>This is the 7.1 version of the tools. You can use this to target both versions 7 and 7.1. Available as a free download from http://www.microsoft.com/download/en/details.aspx?displaylang=en&id=27570.</p> <p>The SDK includes the following features:</p> <ul style="list-style-type: none"> ■ Microsoft Visual Studio 2010 Express for Windows Phone ■ Windows Phone Emulator ■ Windows Phone SDK 7.1 Assemblies ■ Silverlight 4 SDK and DRT ■ Windows Phone SDK 7.1 Extensions for XNA Game Studio 4.0 ■ Microsoft Expression Blend SDK for Windows Phone 7 ■ Microsoft Expression Blend SDK for Windows Phone OS 7.1 ■ WCF Data Services Client for Window Phone ■ Microsoft Advertising SDK for Windows Phone

Tool	Description
Windows Phone SDK 7.1.1 Update	An incremental update to the existing Windows Phone SDK 7.1. This update adds support for developing applications that are optimized to run on 256 MB devices. It includes an updated version of the standard (512 MB) emulator, plus a new 256 MB emulator. For details, see Chapter 16, “Enhanced Phone Services.” With version 7.1.1 installed, you can target all versions: 7, 7.1, or 7.1.1. Available for download at http://www.microsoft.com/download/en/details.aspx?id=29233 .
Windows Phone 7 Training Kit for Developers	A set of hands-on labs for all the core features of Windows Phone 7. Available at http://www.microsoft.com/download/en/details.aspx?displaylang=en&id=1678 .
Windows Phone 7.5 Training Kit	Hands-on labs for the Windows Phone 7.5 application platform. Available at http://www.microsoft.com/download/en/details.aspx?id=28564 .
Windows Azure Toolkit for WP7	Visual Studio templates for Windows Phone projects that connect with cloud services running in Windows Azure. Available at http://watwp.codeplex.com .
Windows Phone Developer Guide	Patterns & Practices guide to Windows Phone development. Available at http://wp7guide.codeplex.com .
Silverlight Media Framework	Includes Windows Phone 7 support for video playback, live/smooth streaming. Available at http://smf.codeplex.com .



Note The online MSDN documentation covers version 7.1. If you need to see documentation that’s specific to version 7, you can download the 7 offline documentation from <http://www.microsoft.com/download/en/details.aspx?id=20558>.

The Windows Phone 7.1 SDK is a complete replacement for the version 7 SDK. You cannot run these side by side on the same computer, but you can use the version 7.1 SDK to target both 7 applications and 7.1 applications. The one and only reason why you might want to keep a computer with only the version 7 tools on it is because the version 7.1 SDK uses the 7.1 emulator; the version 7.1 emulator will run version 7 applications in backward-compatibility mode on top of the version 7.1 platform. Although this has a very high degree of compatibility, you should always run on a real version 7 device (or emulator) to ensure that your application will work on the version 7 OS. Therefore, if you want to be able to test your application on a version 7 emulator, you need to have an installation of the version 7 tools. In all other respects, the version 7.1 tools are preferred.

Development Cycle

Developing and publishing applications for Windows Phone is very straightforward, as shown in Figure 1-4.

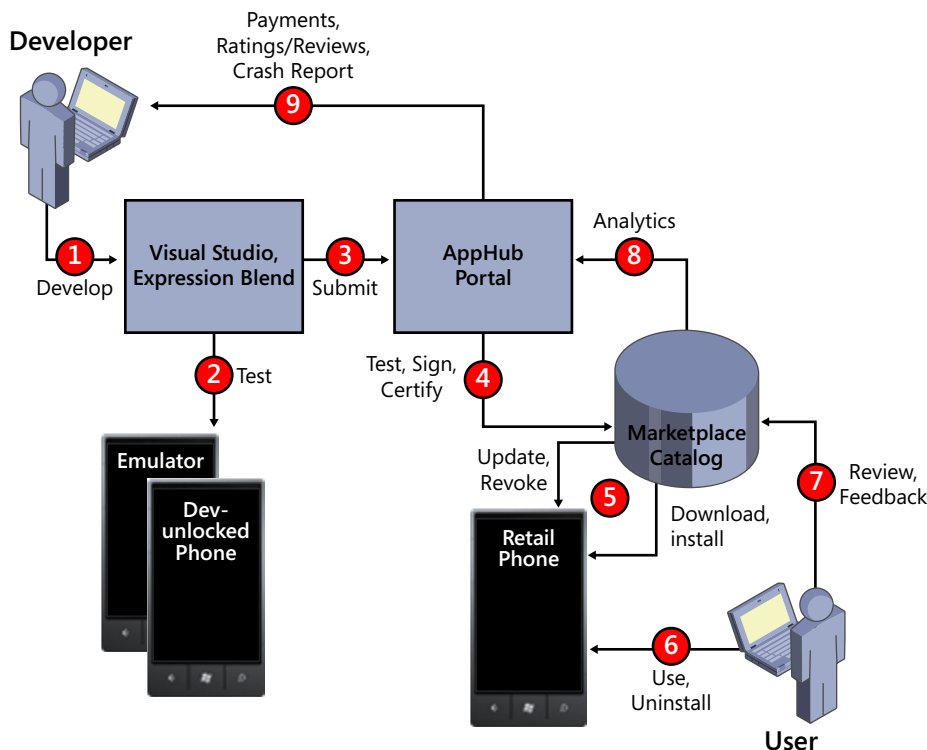


FIGURE 1-4 The develop-publish-feedback cycle is very straightforward.

You can download the free tools and get started building applications straight away. As part of the development phase, you will typically test first on the emulator that is part of the SDK and then also on a physical device. To deploy unpublished applications to a phone, that phone must be developer-unlocked. To do this, you must first register with the AppHub as a developer. Each developer account allows you to unlock up to three physical devices. These devices will be registered with your AppHub account.

When you're ready to submit your application for publication, there is a simple set of forms to fill out on the AppHub. This is a front-end to the Windows Phone marketplace. The marketplace ingestion process will test your application against the certification requirements and then rebuild and sign the final XAP, making it available publicly in the marketplace catalog.

Customers can then download and install your application onto their retail phones. They can also subsequently update the application (if you publish an update to marketplace), provide a review and feedback, and uninstall/reinstall whenever they choose. You can track payments, feedback, and purchase analytics in the AppHub portal.

The Anatomy of a Basic Windows Phone Application

When you build a Windows Phone application, the output of the build process is a XAP file. XAP (pronounced “zap”) is the file extension for a Silverlight-based application package (.xap). This is a zip-format compressed file that contains all your assemblies, your manifest, and any loose image files or other data files that you chose not to embed in your assemblies. When the user installs your application on his phone, the XAP is deployed to an install folder on the filesystem, as shown in Figure 1-5. Each application gets its own install folder. The exact location is opaque—there is no supported way to examine the phone filesystem and no good reason to do so.

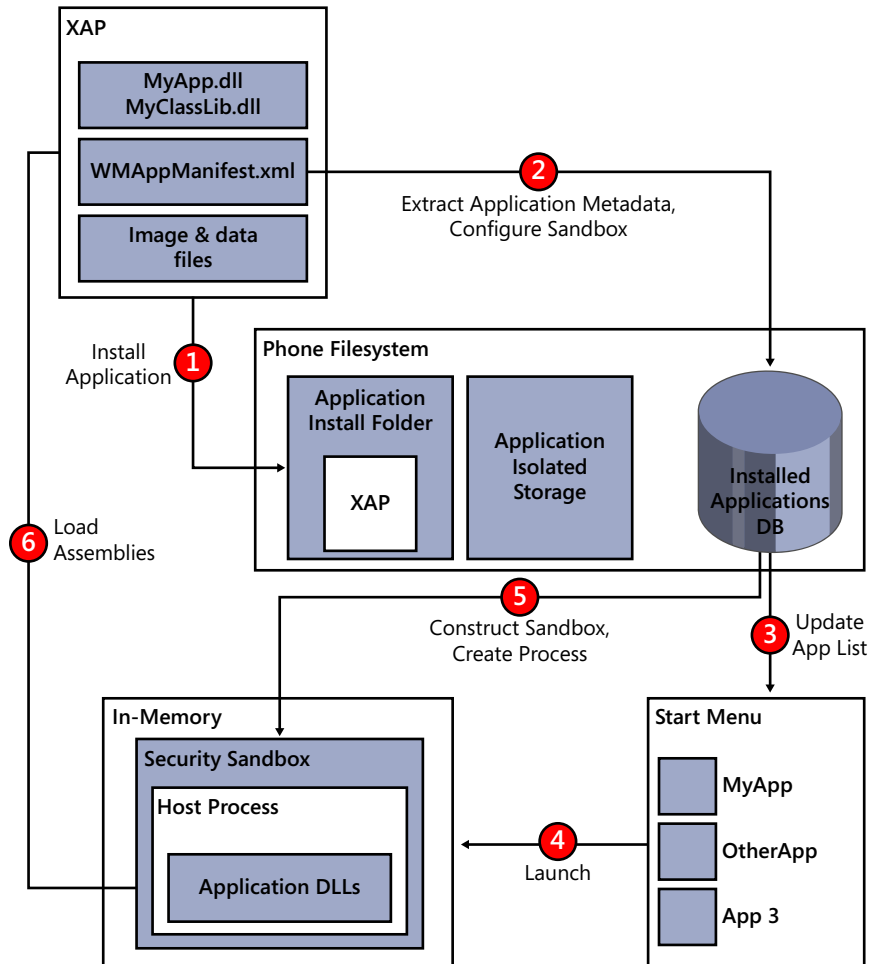


FIGURE 1-5 Components from your XAP are loaded into memory at runtime.

At install time, the installer service extracts critical metadata from your `WAppManifest.xml` and stores it in the phone's database. Information about all installed applications is entered in this database. Standard parts of the phone UI such as the Start page and the installed application list gather the information they need from this database, including application name, icon, and background tile image. Your application manifest includes details of the capabilities that your application requires, and these are used to configure the correct security sandbox for your application. This configuration is also stored in the application database.

When the user chooses to launch your application, the system reads the database to construct the security sandbox within which your application will run. Then, the platform loads your application assemblies into memory, performs security checks, and instantiates the class that you've marked as the entry point to the application. Once your application is running, it has implicit access to any of the loose data files that were in the original XAP and deployed at install time to the application's install folder. Any reference in your code to such files is assumed to be relative to the root of the application's install folder. The application also has exclusive read/write access to its own private isolated storage area on the filesystem. This is in a different location than the install folder, both for security reasons and to aid in the application update process.

XAP Contents

A Windows Phone XAP is essentially the same as a regular desktop Silverlight XAP—a zip file that contains all the application's local DLLs, resources, and the `AppManifest.xml`, plus an additional file named `WAppManifest.xml`. The `AppManifest.xml` file is a regular Silverlight artifact: it contains a list of "deployment parts" (typically, assemblies local to the application), and the name of the class (and its containing assembly) to be instantiated as the entry point for the application. It also contains information about localized resources (if any), and thus the languages that the application supports.

```
<Deployment xmlns="http://schemas.microsoft.com/client/2007/deployment" xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml" EntryPointAssembly="WindowsPhoneApplication1" EntryPointType="WindowsPhoneApplication1.App" RuntimeVersion="3.0.40624.0">
  <Deployment.Parts>
    <AssemblyPart x:Name="WindowsPhoneApplication1"
      Source="WindowsPhoneApplication1.dll" />
    <AssemblyPart x:Name="WindowsPhoneClassLibrary1"
      Source="WindowsPhoneClassLibrary1.dll" />
  </Deployment.Parts>
</Deployment>
```

The `WAppManifest.xml` includes a list of phone capabilities that the application needs to use, the default main page, the application's icon and background images, and the title to use in the application listings on the device. Capabilities are used to configure the security context under which the application will run, and the identifiers listed in the `WAppManifest` correspond to defined sets of platform APIs. So, for example, if you use any network capabilities (including web service calls, web browser, and so on), then you will need the `ID_CAP_NETWORKING` capability.

The capability names listed in the following code are reasonably self-explanatory:

```
<Deployment xmlns="http://schemas.microsoft.com/windowsphone/2009/
deployment" AppPlatformVersion="7.0">
  <App xmlns="" ProductID="{692d41b5-0d98-4dce-93c6-5485be20bf18}" Title="WindowsPhone
Application1" RuntimeType="Silverlight" Version="1.0.0.0" Genre="apps.normal" Author="Windows
PhoneApplication1 author" Description="Sample description" Publisher="WindowsPhoneApplication1">
    <IconPath IsRelative="true" IsResource="false">ApplicationIcon.png</IconPath>
    <Capabilities>
      <Capability Name="ID_CAP_GAMERSERVICES"/>
      <Capability Name="ID_CAP_IDENTITY_DEVICE"/>
      <Capability Name="ID_CAP_IDENTITY_USER"/>
      <Capability Name="ID_CAP_LOCATION"/>
      <Capability Name="ID_CAP_MEDIALIB"/>
      <Capability Name="ID_CAP_MICROPHONE"/>
      <Capability Name="ID_CAP_NETWORKING"/>
      <Capability Name="ID_CAP_PHONEDIALER"/>
      <Capability Name="ID_CAP_PUSH_NOTIFICATION"/>
      <Capability Name="ID_CAP_SENSORS"/>
      <Capability Name="ID_CAP_WEBBROWSERCOMPONENT"/>
    </Capabilities>
    <Tasks>
      <DefaultTask Name="_default" NavigationPage="MainPage.xaml"/>
    </Tasks>
    <Tokens>
      <PrimaryToken TokenID="WindowsPhoneApplication1Token" TaskName="_default">
        <TemplateType5>
          <BackgroundImageURI IsRelative="true" IsResource="false">
            Background.png</BackgroundImageURI>
          <Count>0</Count>
          <Title>WindowsPhoneApplication1</Title>
        </TemplateType5>
      </PrimaryToken>
    </Tokens>
  </App>
</Deployment>
```

In Windows Phone 7, there is only one possible task in your application: this will be named “_default”, and defines the page in your application that the system will create and to which it will navigate upon startup. There must also be an arbitrarily named token identifier that maps to your default task. The remaining elements of this manifest are discussed in the following sections.

Standard Project Types

The Windows Phone 7 SDK offers five Silverlight project templates in Visual Studio: a standard phone application, databound application, phone class library, panorama application, and a pivot application. There is also a set of XNA templates, but these are not covered in this book. The version 7.1 SDK adds four more project types: a combined Silverlight and XNA application, audio playback agent, audio streaming agent, and a scheduled task agent, as shown in Figure 1-6.

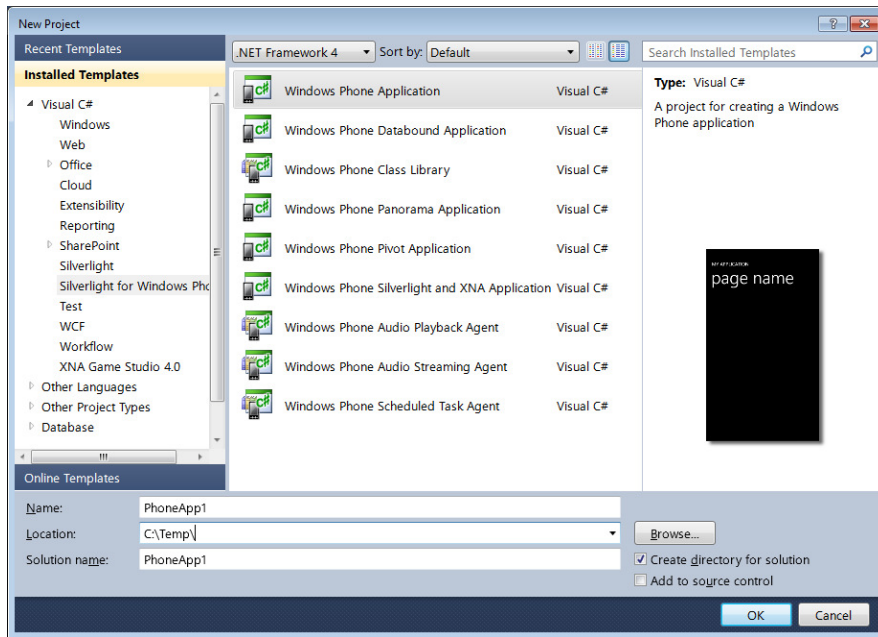


FIGURE 1-6 Visual Studio offers a range of phone-specific project types.

The version 7.1 project types will be examined in later chapters. For now, the focus is on the core project types available in version 7. There is one class library project, and four application projects. The application projects are summarized as follows:

- **Application** A simple, one-page application that includes a title block and an empty grid.
- **Data-Bound Application** An application consisting of two pages. The “main” page offers a list of items. When the user selects an item, the application navigates to the “details” page, which displays more comprehensive information for that item. The application uses data binding (discussed in Chapter 4, “Data Binding”) to bind the source data to the UI.
- **Panorama Application** The main page uses a *Panorama* control (discussed in Chapter 3, “Controls”) to display a list of items. This application also uses data binding.
- **Pivot Application** Very similar to the Panorama Application, except that it uses a *Pivot* control instead of a *Panorama* control.

For version 7, the files generated by each project type are described in Table 1-4.

TABLE 1-4 The Application Files for Each Project Type

Project Type	Files	Description
Application	App.xaml, App.xaml.cs	The Silverlight application entrypoint class, which includes handlers for application lifecycle events.
	WMAppManifest.xml	The Windows Phone marketplace application manifest, which specifies the required capabilities, the starting page, and background image. This is built as a loose content file in the XAP.
	AppManifest.xml	A regular Silverlight application manifest that is used during build. The built version lists dependent assemblies, and is embedded in the XAP.
	MainPage.xaml, MainPage.xaml.cs	The start page for your application.
	ApplicationIcon.png, Background.png, SplashScreenImage.jpg	Default image files. These are built as a loose content file in the XAP.
Data-Bound Application	<i>Same as Application, plus:</i> DetailsPage.xaml, DetailsPage.xaml.cs	The <i>MainPage</i> definition includes a <i>ListBox</i> . When the user selects an item from the list, the application navigates to the <i>DetailsPage</i> , passing through the selected item.
	MainViewModel.cs	The <i>App</i> class has a property that is an instance of the <i>MainViewModel</i> class. This represents a simple object model for the data in the application. The <i>MainPage</i> is data-bound to this <i>MainViewModel</i> object.
	ItemViewModel.cs	The <i>DetailsPage</i> is data-bound to the <i>MainViewModel.Items</i> collection object. These items are represented by the <i>ItemViewModel</i> class.
	MainViewModelSampleData.xaml	Sample data used only at design time. This is set in XAML as the <i>DataContext</i> for both the <i>MainPage</i> and <i>DetailsPage</i> .
Panorama Application	<i>Same as Databound Application, except:</i> No DetailsPage	The <i>MainPage</i> has a <i>Panorama</i> control that contains a <i>ListBox</i> , which itself is data-bound to items in the <i>MainViewModel</i> class.
	PanoramaBackground.png	A 1024x768-pixel image used for the panorama background.
Pivot Application	<i>Same as Panorama Application, except</i> that it uses a <i>Pivot</i> control instead of a <i>Panorama</i> control	The <i>MainPage</i> has a <i>Pivot</i> control. This contains two <i>PivotItem</i> controls, each with a <i>ListBox</i> that is data-bound to items in the <i>MainViewModel</i> class.
Class Library	Class1.cs	A regular .NET class library project, with an empty class definition.

Themes and Accent Colors

A theme is a set of style resources that lend consistency to your UI elements by providing complementary default colors, fonts, sizes, thicknesses, and so on. The standard controls are all theme-aware, so they will match the user's selected theme. Windows Phone offers 2 standard themes (dark or light) and 10 standard accent colors (plus, possibly, an additional eleventh color supplied by the phone hardware manufacturer or mobile operator). The themes and accent colors are applied to standard UI elements by default, as shown in Figure 1-7.

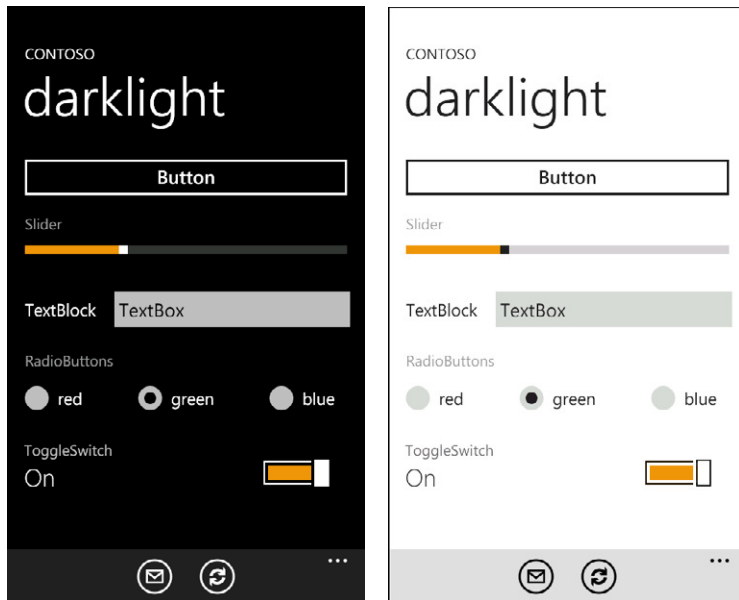


FIGURE 1-7 The dark and light themes and accent colors are applied to standard UI elements.

The values for each accent color are defined in Table 1-5. The most useful column is the Hex column; this is the format for the value that you would define in XAML if you needed to match one of the accent colors explicitly.



Note The definitions for the lime and magenta accent colors changed between Windows Phone 7 and 7.1. Also, teal is named viridian in the Microsoft SDKs\Windows Phone\v7.0\Design folder.

TABLE 1-5 The Ten Standard Accent Colors

Color	Name	ARGB	Hex	HSV
	magenta 7.0	255,255,0,151	#FFFF0097	324,100,100
	magenta 7.1	255,216,0,115	#FFD80073	328,100,84
	purple	255,162,0,255	#FFA200FF	278,100,100
	teal (viridian)	255,0,171,169	#FF00ABA9	179,100,67
	lime 7.0	255,140,191,38	#FF8CBF26	80,80,74
	lime 7.1	255,162,193,57	#FFA2C139	73,70,75
	brown	255,160,80,0	#FFA05000	30,100,62
	pink	255,230,113,184	#FFE671B8	323,50,90
	orange (mango)	255,240,150,9	#FFF09609	36,96,94
	blue	255,27,161,226	#FF1BA1E2	199,88,88
	red	255,229,20,0	#FFE51400	5,100,89
	green	255,51,153,51	#FF339933	120,66,60

You are encouraged to use the predefined theme resources. These themes use the naming structure, *Phone<Feature><Type>*; for example, *PhoneForegroundBrush* or *PhoneFontSizeLarge*. You can find documentation for this at [http://msdn.microsoft.com/en-us/library/ff769552\(VS.92\).aspx](http://msdn.microsoft.com/en-us/library/ff769552(VS.92).aspx). These theme resources are injected into your application at startup (before *Application.InitializeComponent* is called), and are available in XAML at compile-time at %ProgramFiles%\Microsoft SDKs\Windows Phone\v7.0\Design\ThemeResources.xaml. Under the covers, the XAML theme definitions live in a resource dictionary which is added to the *ApplicationResources.MergedDictionaries* as the first item in the collection so that it is searched last, and so that other items in the *Application* resources collection can refer to it.

The accent colors were introduced specifically for Windows Phone 7. They are the result of considerable design effort, usability studies, and so on. They are not the same as the standard Silverlight/.NET/HTML colors.

Theming is applied at startup: there is no event that informs you that the theme has changed—after all, your application will have been suspended when the user switched to the Settings application. If you want to determine the current theme, you can check to see if either the *PhoneDarkThemeVisibility* or *PhoneLightThemeVisibility* resource value is visible (that is, set to *Visibility.Visible*, as opposed to *Visibility.Collapsed*). Alternatively, you can check the value of *PhoneBackgroundColor*, but this is not recommended because the precise value of a given color might change in future releases. To determine the current accent color, you can check the *PhoneAccentColor* resource value (and if you want actually to use the accent color in code, you would use the *PhoneAccentBrush* resource, because most uses of the color would typically involve a brush). The *ThemeAccent* sample illustrates this, as shown in Figure 1-8.



FIGURE 1-8 The *ThemeAccent* sample determines the theme and lists the accent colors.

The accent colors are rendered in a *ListBox*, which uses data binding and a custom *DataTemplate*. Both of these will be discussed in Chapter 3. For now, the interesting code to consider is that which determines the current theme and accent color. First, take a look at the theme. The recommended way to determine this is to examine the *Visibility* of the *PhoneDarkThemeVisibility* or *PhoneLightThemeVisibility*; whichever of these is visible indicates the current theme.

```
Visibility v = (Visibility)Resources["PhoneDarkThemeVisibility"];
String theme = (v == System.Windows.Visibility.Visible) ? "dark" : "light";
```

Checking the current accent color is also very straightforward: this is provided in the *PhoneAccentColor* resource value. The default *ToString* method in the *Color* class returns a hex string that corresponds to the ARGB values. It's a simple matter to look these up with the known set of accent color values.

```
Color accent = (Color)Resources["PhoneAccentColor"];
String accentText = String.Empty;
switch (accent.ToString())
{
    case "#FFFF0097":
        accentText = "magenta 7.0";
        break;
    case "#FFD80073":
        accentText = "magenta 7.1";
        break;
    case "#FFA200FF":
        accentText = "purple";
        break;
    case "#FF00ABA9":
        accentText = "teal";
        break;
    case "#FF8CBF26":
        accentText = "lime 7.0";
        break;
    case "#FFA2C139":
        accentText = "lime 7.1";
        break;
    case "#FFA05000":
        accentText = "brown";
        break;
    case "#FFE671B8":
        accentText = "pink";
        break;
    case "#FFF09609":
        accentText = "orange (mango)";
        break;
    case "#FF1BA1E2":
        accentText = "blue";
        break;
```

```

        case "#FFE51400":
            accentText = "red";
            break;
        case "#FF339933":
            accentText = "green";
            break;
        default:
            accentText = "custom";
            break;
    }

```

A marginally more elegant approach—and the one used in this sample to populate the *ListBox*—is to set up a collection of *Color* objects. Using this approach, you create a *Dictionary* collection, wherein each key is a string representing the color name, and each value is a *Brush* initialized with a corresponding *Color* value.

```

private Dictionary<string, SolidColorBrush> colors =
    new Dictionary<string, SolidColorBrush>();
private void InitializeColorList()
{
    colors.Add("magenta 7.0",    new SolidColorBrush(Color.FromArgb(255, 255, 000, 151)) );
    colors.Add("magenta 7.1",    new SolidColorBrush(Color.FromArgb(255, 216, 000, 115)) );
    colors.Add("purple",         new SolidColorBrush(Color.FromArgb(255, 162, 000, 255)) );
    colors.Add("teal",           new SolidColorBrush(Color.FromArgb(255, 000, 171, 169)) );
    colors.Add("lime 7.0",       new SolidColorBrush(Color.FromArgb(255, 140, 191, 038)) );
    colors.Add("lime 7.1",       new SolidColorBrush(Color.FromArgb(255, 162, 193, 057)) );
    colors.Add("brown",          new SolidColorBrush(Color.FromArgb(255, 160, 080, 000)) );
    colors.Add("pink",           new SolidColorBrush(Color.FromArgb(255, 230, 113, 184)) );
    colors.Add("orange (mango)", new SolidColorBrush(Color.FromArgb(255, 240, 150, 009)) );
    colors.Add("blue",           new SolidColorBrush(Color.FromArgb(255, 027, 161, 226)) );
    colors.Add("red",            new SolidColorBrush(Color.FromArgb(255, 229, 020, 000)) );
    colors.Add("green",          new SolidColorBrush(Color.FromArgb(255, 051, 153, 051)) );
    ColorList.ItemsSource = colors;
}

```

With the collection set up, you can now query it to find the matching accent color:

```

Color accent = (Color)Resources["PhoneAccentColor"];
string accentText = "custom";
var pair = colors.FirstOrDefault(x => x.Value.Color == accent);
if (!string.IsNullOrEmpty(pair.Key))
{
    accentText = pair.Key;
}

```

As pointed out earlier, be aware that some hardware manufacturers and mobile operators install an eleventh accent color to their phones. You should therefore allow for the possibility that there are 11 colors, not just 10. The preceding code either uses a default case or sets the default value of the color name to allow for this.

Standard Application Images

A Windows Phone application uses at least three images, all with the build action set to *Content*, as summarized in Table 1-6.

TABLE 1-6 Requirements for the Three Standard Application Images.

Default Image File Name	Size in Pixels	Description	Acceptable File Name and Location	Format	Required or Optional
ApplicationIcon.png	62x62	Used to identify your application in the installed applications list on the phone.	Any name, must be at the root of the project. Must be set in the application properties.	JPG or PNG	Required
Background.png	173x173	Used as your application's tile background, if the user pins your application to his home screen.	Any name, any location in the project. Must be set in the application properties.	JPG or PNG	Required
SplashScreenImage.jpg	480x800	The application's splash screen. This is optional if your application loads its first page within 5 seconds of launch.	Must be named SplashScreenImage.jpg, and must be at the root of the project.	JPG only	Optional

To be technically accurate, you could omit the application icon image altogether by selecting “(default)” in the properties for the app. This will select the default star icon and set the following value into your WManifest.xml; however, note that this is not recommended behavior:

```
<IconPath IsRelative="true" IsResource="true">
    res://StartMenu!AppIconGeneric.png
</IconPath>
```

The same is not true for the background image. The relevant entry in the WManifest.xml is shown in the code that follows. You cannot omit the *<BackgroundImageURI>* element or the file will fail validation and the application will fail to build. If you omit the value (for example, Background.png), the application will build, but there will be no background image. So, if the user pins your application to the Start page, there will be a tile with no image. This will fail marketplace certification.

```
<TemplateType5>
    <BackgroundImageURI IsRelative="true" IsResource="false">
        Background.png
    </BackgroundImageURI>
    <Count>0</Count>
    <Title>NoImages</Title>
</TemplateType5>
```

Notice how there are two *Title* entries in the manifest: one is an attribute of the application element, the other is a subelement of the *Tokens* element. The application element *Title* attribute is used for the application’s listing in the list of installed applications. The *Tokens* element *Title* is used for the tile when the application is pinned to the start page.


```
<App xmlns="" ProductID="{692d41b5-0d98-4dce-93c6-5485be20bf18}"
  Title="WindowsPhoneApplication1" RuntimeType="Silverlight" Version="1.0.0.0"
  Genre="apps.normal" Author="WindowsPhoneApplication1 author" Description="Sample
  description" Publisher="WindowsPhoneApplication1">
```

If you specify a background (tile) image that is larger or smaller than 173x173, it will be cropped or scaled up; in this scenario, the upper-left corner is referenced as the origin.

The image `SplashScreenImage.jpg` can be omitted. This is the splash screen that is displayed briefly while the application is starting up. The splash screen remains visible until your application receives the `PhoneApplicationFrame.Navigated` event. If you don't provide a splash screen image, the system simply displays a blank screen during application startup. If you do provide one, it must be named `SplashScreenImage.jpg`.

As always, you should follow the Metro guidelines in composing application images. In this example, the same raw image (with a transparent background) was used to create all three image files. Using transparency requires saving the files in PNG format, and it has the advantage that transparent areas of the image will show the phone's current accent color (for the all-applications list on version 7 devices, it will be gray). This technique helps to make your application integrated as a seamless part of the overall phone ecosystem. Figure 1-9 shows an application that uses Metro-compliant images for the application icon, the tile, and the splash screen.

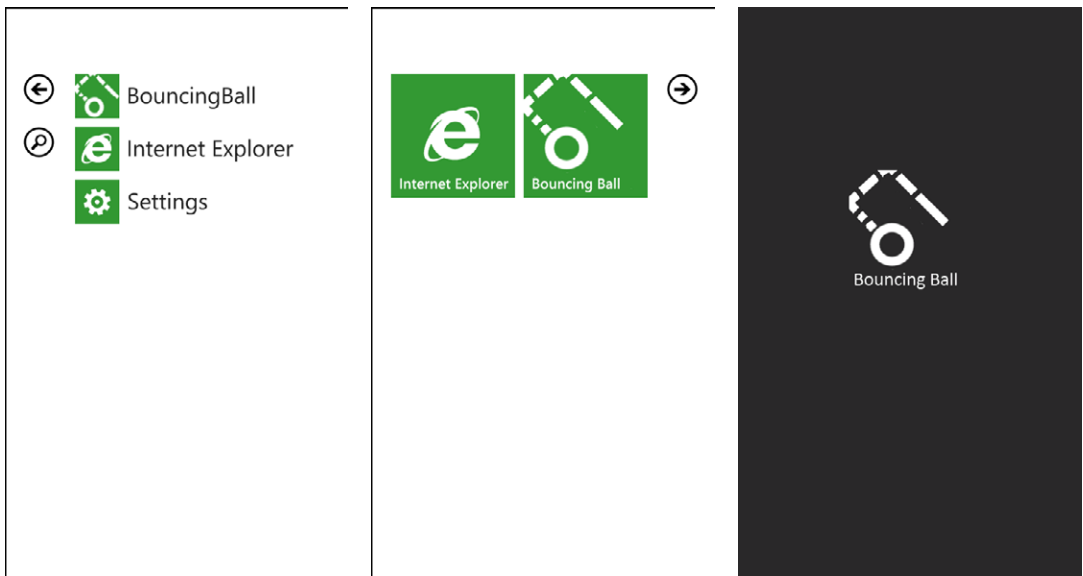


FIGURE 1-9 You should use Metro-compliant application images.

The application icon and background image can be set in the properties window in Visual Studio, as shown in Figure 1-10. You could set these by editing the `WMAppManifest.xml`, but this is not recommended, because Visual Studio caches this information and will overwrite your manual edits without warning whenever you read or edit the properties in the properties dialog.

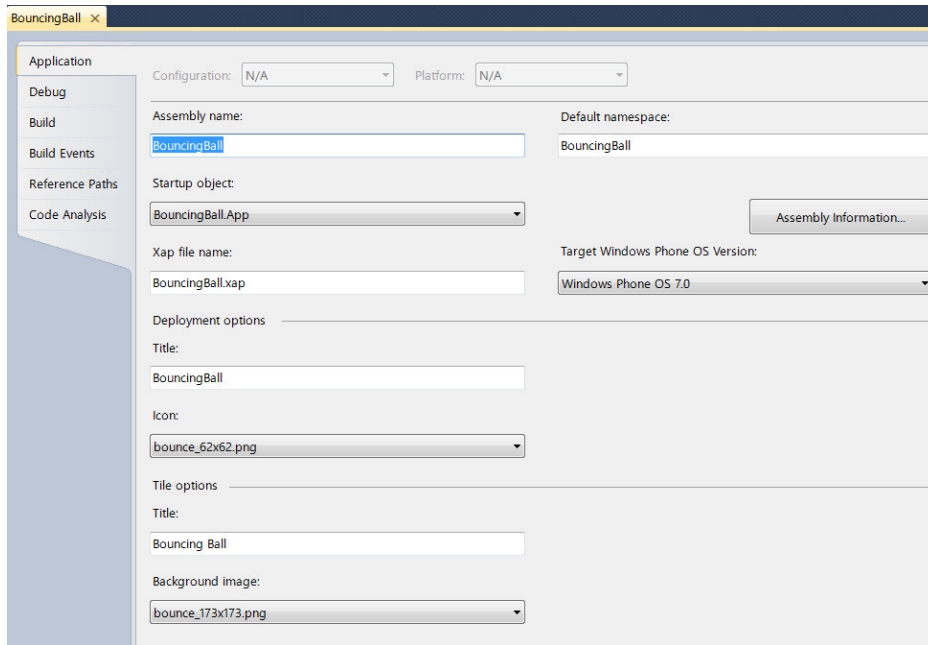


FIGURE 1-10 Use the Application properties to set the default images.

Although some of the applications that ship with the phone itself use the current theme for the application icon, this is not true of third-party (that is, marketplace) applications. For these, only the tile can use the current theme. If you want to use the theme for your tile, you simply set the tile background to transparent. Note that this behavior changes in version 7.1, for which marketplace developers can now use transparency in their application icon in addition to the tile.

Version 7 vs. Version 7.1

There is a high degree of backward compatibility built into the platform, so Windows Phone 7.1 is effectively a complete superset of Windows Phone 7. All version 7 applications should work without change in version 7.1, with the single exception of photo extensibility (covered in detail in Chapter 9, “Phone Services,” and Chapter 16, “Enhanced Phone Services”). Of course, version 7.1 introduces some new features that will only work on 7.1 phones. You can use the 7.1 SDK to build applications for either version 7 or 7.1, and if you want to target the largest possible consumer base, you should build applications for version 7. Table 1-7 summarizes the differences between the two versions.

TABLE 1-7 Additional Features Provided by Windows Phone 7.1

Feature	7 Behavior	7.1 Behavior
Advertising	You can use the Microsoft Advertising SDK, which is a separate download.	The Advertising SDK is now fully integrated in Visual Studio.
Alarms and reminders	Not available.	Your application can create alarms and reminders that are displayed at some later time, whether your application is running or not.
Application bar	You can set the visibility and/or opacity programmatically.	You can additionally switch between normal and a new minimized size.
Background agents	Not available.	Your application can create an agent that can run either on a periodic schedule or when certain conditions are met, regardless of whether your application is running at the time.
Background audio playback	Not available.	Your application can provide audio playback features that allow audio to continue playing in the background, even when the user navigates to another application.
Background file transfers	Not available.	Your application can initiate file uploads or downloads that continue after the user has navigated away from your application.
Backstack	You can navigate forward to an arbitrary page URL or backward to the immediately preceding page in the backstack.	You can remove items from the backstack so that you can effectively navigate backward to any previously visited page that's still in the backstack.
Camera	You can programmatically invoke the camera launcher to allow the user to take photos.	You can work with the camera data pipeline directly, which allows you to build augmented reality applications.
Clipboard	Not available programmatically.	You can set the clipboard contents programmatically.
Contacts and calendar	Not available programmatically.	Read-only programmatic access is introduced.
Cryptography	API support for encrypting data locally on the phone, but no support for encrypting passwords or secure credential storage.	Data Protection API (DPAPI) support for encryption and secure storage of passwords and other credentials on the phone.
Data binding enhancements	n/a	You can use the <i>StringFormat</i> attribute to format data-bound strings, group and sort data-bound collections, and use asynchronous validation.
Device information	You can use the <i>DeviceExtendedProperties</i> API to report information about the device and current memory usage.	<i>DeviceExtendedProperties</i> has been superseded by the <i>DeviceStatus</i> API, which should be used instead.
Emulator	Supports most application scenarios, but not sensors.	Support for simulating accelerometer and location events, map support, and screenshot capability.
Fast Application Switching	When the user navigates away from an application, that application is generally tombstoned, meaning it is terminated and removed from memory. If the user subsequently navigates back to the application, it is restarted and initialized with the previous context.	When the user navigates away from an application, it typically remains in memory but cannot execute any code. If the user subsequently navigates back to the application, it is simply reactivated, not restarted from scratch.

Feature	7 Behavior	7.1 Behavior
Image rendering	16 bits per pixel (bpp) only.	16 bpp or 32 bpp, on a per-application basis.
JPEG decoding	Executes on the UI thread, which can slow down user responsiveness.	Can be set to execute on a background thread.
Launchers and Choosers	The platform provides 10 Launchers and 6 Choosers for invoking standard features from your application.	The platform adds a further five Launchers and four Choosers.
Local database	Not available.	You can create a database in isolated storage, and perform standard create/read/update/delete (CRUD) operations on it via LINQ-to-SQL.
Marketplace preparation	Use the Capability Detection tool to identify the capabilities required by your application.	Use the Marketplace Test Kit to identify capabilities, validate marketplace images, automatically test critical publication requirements, and manually track certification test cases.
Multi-tasking	An application can only execute actions while it is active in the foreground.	An application can deploy agents which can run in the background even when the user is doing something else in the foreground. These include background agents, and background audio playback. An application can also set up alarms and reminders as well as background file transfers to run in the background.
OData client	Your application can invoke remote data web services that return data in OData format, and you can generate client-side proxies by using the DataSvcUtil command-line tool.	OData client proxy generation is now built in to Visual Studio, and generates code that is more robust and flexible.
Photo extensibility	You can add your application to the extras menu in the standard photo/picture library on the phone.	You can add your application to the apps menu in the standard photo/picture library on the phone, to the new apps pivot on the pictures hub, and to the share link in the individual picture viewer. Note that this is the one breaking change from version 7 to 7.1.
Profiler	Not available.	Analyze your application's performance, use of CPU and memory, use of UI and background threads, and frame rendering behavior—so that you can identify memory leaks and performance issues, and take corrective action.
Push notifications	You can use the Microsoft Push Notification Service to send raw, toast, or tile notifications to your phone application.	Toast notifications can now deep-link to specific pages in your application and supply custom parameters. Tile notifications now support both a back and a front.
Search extensibility	Not available.	You can extend the Bing search experience with custom behavior, integrating your application with the search results.
Sensors	The platform provides API support for the accelerometer sensor only.	The platform adds API support for the compass and gyroscope as well as an aggregated virtual sensor called motion.
Silverlight and XNA integration	Not available.	You can build an application that uses both Silverlight and XNA together.
Silverlight runtime	Uses a modified version of Silverlight 3.0.	Uses a modified version of Silverlight 4.0.
Sockets	Not available.	Your application can use TCP and UDP sockets for two-way communication with remote services.
Standard controls	Support for 21 platform controls, plus 10 more in the Silverlight toolkit.	Three additional platform controls, and five additional Silverlight toolkit controls.

Feature	7 Behavior	7.1 Behavior
System tray	You can set the visibility, background and foreground color.	You can additionally set the opacity and a built-in progress bar.
Tiles	The user can pin your application tile to the Start menu.	Your application can programmatically pin one or more secondary tiles to the Start menu, in addition to the user-pinned main application tile. Tiles also now have a back side, with customizable background image and text.
UI commands	You can hook up handler methods for UI command events such as button clicks.	You can data-bind to commands, which increases the decoupling between view and viewmodel.
<i>WebBrowser</i> control	Supports Internet Explorer 8 and HTML 4.	Supports Internet Explorer 9 and HTML5.
XAML styles	You can create named styles, which you then apply explicitly to selected elements.	You can also create unnamed styles, which are automatically applied to all elements of the target type implicitly.

Summary

In this chapter, you learned how the overarching vision for Windows Phone is a significant departure from traditional mobile device development. The importance of a holistic UX and of applications that contribute in a meaningful way to the user's daily life should inform the way you make decisions about functionality and design. It should guide you to build applications that are compelling, beautiful, and truly useful to the user. From a more technical viewpoint, the Windows Phone architecture involves many moving parts, with a complex interaction between native and managed code, and between the runtime, the application platform, and your code. That having been said, the API surface exposed for your application to use has been very thoughtfully put together; it makes it easy to build sophisticated applications that integrate seamlessly with standard phone features.

UI Core

Windows Phone development with Microsoft Silverlight has a lot in common with Silverlight desktop development, notwithstanding some subtle differences and a few phone-specific quirks. The extreme physical constraints of a mobile device also impose some challenges for application developers. This chapter will examine the fundamental user interface (UI) infrastructure in the Windows Phone application platform, and how that platform exposes significant support for mobile developers. You'll also look at how Metro principles translate into standard UI layout, how to customize that layout, and the various options for incorporating graphical resources in an application.

Phone UI Elements

In the following sections, you will be introduced to the primary elements that make up the Windows Phone UI.

Standard UI Elements

The Windows Phone 7 chassis requirements are based on extensive market research and discussions with hardware manufacturers. The requirements specify that all Windows Phone 7 phones have WVGA screens at 800x480 pixel resolution. They must also support both portrait and landscape display orientations. All screens must support multi-touch, include a light sensor that improves power consumption (by adjusting screen brightness according to ambient conditions), and incorporate a set of proximity sensors that turn off the touch screen when the device is held close to the head during phone calls or when it is in a pocket or handbag. The screen physical size must be 3.5" to 4.4", and render 16 bits of color per pixel minimum. There will always be three hardware buttons on the front of the phone at the bottom of the screen for Back, Start, and Search. The graphics processing unit (GPU) supports Direct3D 10 Level 9, and includes driver-level support for GDI and DirectDraw.

The fact that there is just one chassis specification with relatively few variations permitted is a significant benefit for developers. Developers don't need to worry about different form factors and can write and test applications geared to a well-defined target.

The Windows Phone shell layers a couple of standard elements on top of the chassis: the System Tray and the Application Bar, both of which are optional for use in your applications. The standard application UI model for Silverlight applications is to use an outermost *Frame* which represents the entire screen. On top of this, the application itself defines one or more *Page* objects, each *Page*

typically occupies the whole *Frame* (allowing for System Tray and Application Bar). Only one *Page* is visible at a time.

The System Tray (or Status Bar) runs across the top of the screen (the top 32 pixels in portrait mode, 72 pixels in landscape) and provides indicators for cell/wireless/bluetooth strength, data connection, roaming, battery level, and the system clock. This is represented by the *SystemTray* object in code—it cannot be modified programmatically, although it can be hidden. In Windows Phone 7.1, you can also set its opacity.

The Application Bar is 72 pixels in height in portrait mode; it is also 72 pixels wide in landscape mode. It is always displayed on the same edge as the device hardware buttons—at the bottom in portrait, on the left or right in landscape. Your application can use this space to provide up to four icon buttons, plus a short menu. You can use standard icons or custom icons, but these are constrained to be white only in a 26x26 pixel area within the overall 48x48 icon area. The Application Bar is not a conventional Silverlight element—it is rendered by the phone shell—so it is not part of your application’s visual tree, and there are significant constraints on what you can do with it.

The *Frame* is the top-level UI container for a Windows Phone app, represented in code by the *PhoneApplicationFrame* type, which will always be 800x480.

Within that, your application can have one or more *Page* elements, represented by the *PhoneApplicationPage* type, whose size varies according to how you’re showing the System Tray and Application Bar. These sizes are summarized in Table 2-1.

TABLE 2-1 Page Sizes

Orientation	SystemTray	AppBar	Page	ContentPanel
Portrait	Visible and Opacity==1	Visible and Opacity==1	696x480	517x444
Portrait	Hidden or Opacity <1	Visible and Opacity==1	728x480	549x444
Portrait	Visible and Opacity==1	Hidden or Opacity <1	768x480	589x444
Portrait	Hidden or Opacity <1	Hidden or Opacity <1	800x480	621x444
Landscape	Visible and Opacity==1	Visible and Opacity==1	480x656	301x620
Landscape	Hidden or Opacity <1	Visible and Opacity==1	480x728	301x692
Landscape	Visible and Opacity==1	Hidden or Opacity <1	480x728	301x692
Landscape	Hidden or Opacity <1	Hidden or Opacity <1	480x800	301x764

Just to give an indication of how much real estate you have to play with in your app, the Microsoft Visual Studio templates generate code that provides for a standard *TitlePanel*, composed of a *TextBlock* for the application title, and another *TextBlock* for the page title, both using predefined *Style* resources. You don’t have to stick to this model, but it helps to maintain consistency across applications. If you adopt the Visual Studio–generated starter code (which is, of course, based on the Metro guidelines), with an *ApplicationBar.Opacity=1*, *ApplicationBar.IsVisible=true*, *SystemTray.IsVisible=true*, and *SystemTray.Opacity=1* (settable in 7.1 only), then the general shape of the real estate you’ll have at your disposal is as shown in Figure 2-1.

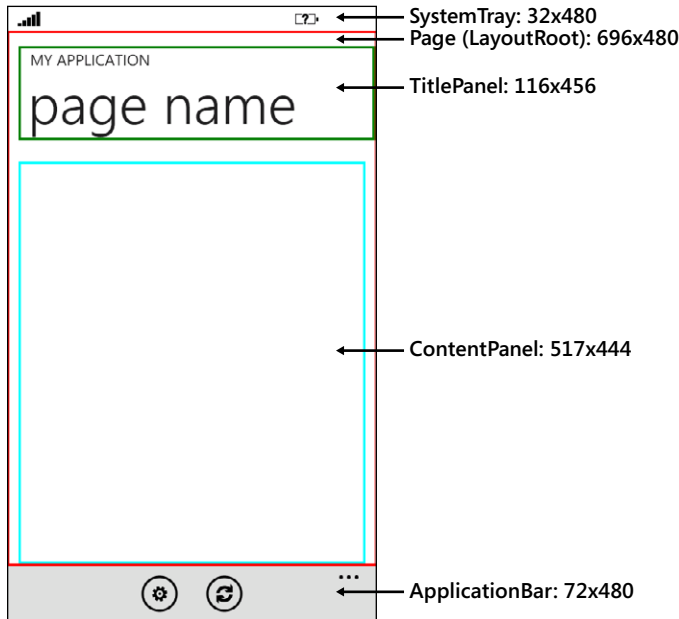


FIGURE 2-1 The Visual Studio template generates a standard real estate layout.

From the XAML that follows, you can see that the *TitlePanel* is offset 12 pixels from the left and 17 pixels from the top. The *PageTitle* is offset 9 pixels from the left and -7 pixels from the top. *PageTitle* is the second child of a *StackPanel*, which lays out its children one after the other, in the order declared. The next child is positioned by default immediately after the previous child. So, the *PageTitle* position depends on the position and size of the *ApplicationTitle*. The *ApplicationTitle* uses a default *PhoneText NormalStyle*, which includes a 20 pixel (15 pt) font size.

```
<phone:PhoneApplicationPage>
  <Grid x:Name="LayoutRoot" Background="Transparent">
    <Grid.RowDefinitions>
      <RowDefinition Height="Auto"/>
      <RowDefinition Height="*" />
    </Grid.RowDefinitions>

    <StackPanel x:Name="TitlePanel" Grid.Row="0" Margin="12,17,0,28">
      <TextBlock
        x:Name="ApplicationTitle" Text="MY APPLICATION"
        Style="{StaticResource PhoneTextNormalStyle}" />
      <TextBlock
        x:Name="PageTitle" Text="page name" Margin="9,-7,0,0"
        Style="{StaticResource PhoneTextTitle1Style}" />
    </StackPanel>

    <Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0"></Grid>
  </Grid>
</phone:PhoneApplicationPage>
```

You're also given an inner *Grid* named *ContentPanel* as a suggested starting point for your content. You're free to replace anything and everything on the page, according to the requirements of your application. However, the starter template layout is an excellent Metro-compliant guideline, so you are encouraged to adopt this whenever possible.

Visual Tree

The normal state of affairs is that most or all of the visual aspects of your UI are defined in XAML, with only UI logic in the code-behind. As with Silverlight generally, the XAML for a Windows Phone application is hierarchical in nature. At the root is a *PhoneApplicationFrame*, set as the *RootVisual* property in the *Application* base class. Each *PhoneApplicationPage* in the application is a child of the *RootVisual*—or rather, it becomes a child of the *RootVisual* when the user navigates to that page and the corresponding page class is instantiated. A page typically has children of its own. Figure 2-2 shows a simple page with three obvious controls: two *TextBlock* controls and a *Button*. This is the *Simple VisualTree* solution in the accompanying sample code. However, there are actually more visual elements, some are visible, some not. If you think for a moment, you'll realize there are at least two more: the *Page* and the *Frame*.



FIGURE 2-2 Some visible and non-visible visual elements.

The XAML definition of this UI is shown in the following code snippet. From this, you can see that, in addition to the three visible and two non-visible controls that we've identified thus far, there are at least three additional controls: two *Grid* controls and a *StackPanel*, which now brings the total to eight.

```

<phone:PhoneApplicationPage
... namespace and style declarations omitted for brevity
>

    <Grid x:Name="LayoutRoot" Background="Transparent">
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto"/>
            <RowDefinition Height="*/>
        </Grid.RowDefinitions>

        <StackPanel x:Name="TitlePanel" Grid.Row="0">
            <TextBlock Text="CONTOSO"/>
            <TextBlock Text="visual tree"/>
        </StackPanel>

        <Grid x:Name="ContentPanel" Grid.Row="1">
            <Button Content="press me"/>
        </Grid>
    </Grid>

</phone:PhoneApplicationPage>

```

At runtime, the layout engine parses the XAML and creates a set of objects to help correctly construct and maintain the UI. Some of these objects are visual, some are not—some are merely helper objects that have no visual representation. For example, a *ListBox* holds a collection of *ListBoxItems*, but the collection object itself has no visual representation. The visual hierarchy, or tree, is that subset of the object hierarchy that has a visual representation. This applies regardless of whether the object is actually visible. The *Grid* controls and *StackPanel* in this example are not visible, but they determine the visual layout of their respective child objects, so they do have an impact on the composition of the screen image. So, the *Grid* controls and the *StackPanel* are all part of the visual tree.

Quite often, visual objects that are not directly defined in the application's XAML are created as part of the tree. For example, a *Button* is actually made up of a number of other elements. You can see this if you examine the default template for the *Button* control. The default template is in the *System.Windows.xaml* file, which you can find in the install folder for the Windows Phone SDK (typically: %ProgramFiles%\Microsoft SDKs\Windows Phone\v7.x\Design\System.Windows.xaml). An abstract of this template is listed in the following code:

```

<ControlTemplate TargetType="ButtonBase">
    <Grid Background="Transparent">
        <Border x:Name="ButtonBackground">
            <ContentControl x:Name="ContentContainer"/>
        </Border>
    </Grid>
</ControlTemplate>

```

You can see that the *Button* is made up of a *Grid*, a *Border*, and a *ContentControl*. This brings our total number of controls so far to 10. Now, the derivation hierarchy for the *Button* control includes the *Control* class. The *Control* class has a *Content* field of type object, which means it can hold anything.

In the case of the *Button* control, the default *Content* type is a *TextBlock*. The total now is 11. In fact, in this simple page, there are 16 visual elements. To see them all, you can use the *VisualTreeHelper* class to walk the tree. What follows is some simple code to do just that. This example defines a custom method, *PrintVisualTree*, which is used recursively, starting at the *RootVisual*. For each visual element, it prints the type name to the debug console. It tracks the current tree level and indents three spaces for each level. So, starting at the top, it prints out the *RootVisual* data, and then calls *VisualTreeHelper.GetChildrenCount* to get a count of the *RootVisual* control's children. If there are any, it gets each child with *VisualTreeHelper.GetChild*, and then recurses to print the data and find any further child levels.

```
private int elementCount = 0;

private void MainPage_Loaded(object sender, RoutedEventArgs e)
{
    PrintVisualTree(App.Current.RootVisual, 0, 0);
}

private void PrintVisualTree(DependencyObject obj, int level, int indent)
{
    elementCount++;

    String indentString = String.Empty;
    for (int i = 0; i < level; i++)
    {
        indentString += "  ";
    }

    String name = String.Empty;
    if (obj is FrameworkElement)
    {
        name = (obj as FrameworkElement).Name;
        Debug.WriteLine(String.Format("{0} {1}{2} \"{3}\"",
            elementCount, indentString, obj, name));
    }

    int childCount = VisualTreeHelper.GetChildrenCount(obj);
    if (childCount > 0)
    {
        level++;
        int childIndent = 0;
        for (int i = 0; i < childCount; i++)
        {
            DependencyObject child = VisualTreeHelper.GetChild(obj, i);
            PrintVisualTree(child, level, childIndent);
        }
    }
}
```

The debug output for this simple page is listed below. You can map this to the application XAML and to the default *Button* template to more clearly see the 16 visual elements on this page.

```
1 Microsoft.Phone.Controls.PhoneApplicationFrame ""
2     System.Windows.Controls.Border "ClientArea"
3         System.Windows.Controls.ContentPresenter ""
4             SimpleVisualTree.MainPage ""
5                 System.Windows.Controls.Grid "LayoutRoot"
6                     System.Windows.Controls.StackPanel "TitlePanel"
7                         System.Windows.Controls.TextBlock ""
8                         System.Windows.Controls.TextBlock ""
9                         System.Windows.Controls.Grid "ContentPanel"
10                             System.Windows.Controls.Button ""
11                                 System.Windows.Controls.Grid ""
12                                     System.Windows.Controls.Border "ButtonBackground"
13                                         System.Windows.Controls.ContentControl "ContentContainer"
14                                             System.Windows.Controls.ContentPresenter ""
15                                                 System.Windows.Controls.Grid ""
16                                                     System.Windows.Controls.TextBlock ""
```

The *VisualTreeHelper* class also includes a useful *GetParent* method so that you can walk the tree in reverse order, if you like. Printing out the tree is instructive, but it's not a common requirement for a real application. What *is* quite common is walking the tree to find a particular element. If you have declared the element in XAML, then you can also declare a name for it, which means you'll have a field that you can use in code. However, it should be clear from the foregoing that sometimes you have visual elements that you might want to work with but which were implicit in some composite control—and for which, therefore, you have no opportunity to define a field. If the implicitly defined (well, actually, explicitly defined—just not explicitly in your application code) element is exposed as a property of one of your code elements, there's also no problem. Sometimes, however, such a non-explicit element is also not exposed as an accessible property of any of your explicit elements.

One scenario for which the *VisualTreeHelper* technique is useful is when you want to access elements in a dynamic visual collection, such as a *ListBox*. In this scenario, you don't necessarily have fields defined for the controls within the *ListBoxItem* template. Another scenario is when you want to carry out some common processing for multiple elements of the same type all at once, even though you might or might not have fields for each element, and the elements might be in arbitrary control hierarchies on your page. Here's a simple example to illustrate this scenario. The following application (the *GlobalElementChange* solution in the sample code), which is shown in Figure 2-3, has three *TextBlock* controls and three *TextBox* controls. The *Button* at the bottom toggles the *IsReadOnly* state of all three *TextBox* controls.



FIGURE 2-3 One use of the visual tree is to manipulate multiple, unnamed elements.

The code-behind defines a *GetChildren* method, which uses the *VisualTreeHelper* to get an enumeration of all children of the *LayoutRoot* control. Next, the *IEnumerable<T>.OfType<T>* method filters the list to the *TextBox* controls. As it happens, all three *TextBox* controls are children of the same parent panel, but this technique would work regardless of the layout, so long as you start walking the tree at a high enough level.

```
private void ChangeElements_Click(object sender, RoutedEventArgs e)
{
    GetChildren(LayoutRoot).OfType<TextBox>().ToList().ForEach(
        t => t.IsReadOnly = !t.IsReadOnly);
}

private IEnumerable<DependencyObject> GetChildren(DependencyObject obj)
{
    int count = VisualTreeHelper.GetChildrenCount(obj);
    for (int i = 0; i < count; i++)
    {
        var child = VisualTreeHelper.GetChild(obj, i);
        yield return child;
        foreach (var descendent in GetChildren(child))
            yield return descendent;
    }
}
```

Wrapping the *VisualTreeHelper* methods like this is a common enough requirement that many developers find themselves building a library of such wrappers. Given the reusability of such a library, it makes sense to implement such wrappers as extension methods. For example, you could rewrite (and reuse) the *GetChildren* method as an extension method. Extension methods are a standard C# feature; by definition these are implemented as static methods in a static class, and the first parameter is an object of the type that you're extending.

```
private void ChangeElements_Click(object sender, RoutedEventArgs e)
{
    //GetChildren(LayoutRoot).OfType<TextBox>().ToList().ForEach(
    //    t => t.IsReadOnly = !t.IsReadOnly);
    LayoutRoot.GetChildren().OfType<TextBox>().ToList().ForEach(
        t => t.IsReadOnly = !t.IsReadOnly);
}

public static class VisualTreeExtensions
{
    public static IEnumerable<DependencyObject> GetChildren(
        this DependencyObject obj)
    {
        int count = VisualTreeHelper.GetChildrenCount(obj);
        for (int i = 0; i < count; i++)
        {
            var child = VisualTreeHelper.GetChild(obj, i);
            yield return child;
            foreach (var descendent in GetChildren(child))
                yield return descendent;
        }
    }
}
```

A final enhancement that's worth making is to provide a templated layer on top of the custom methods. You can essentially move the behavior of the *OfType<T>* filter into the method itself by using a Language-Integrated Query (LINQ) expression, and then expose this by changing the signature to *GetChildren<T>*.

```
private void ChangeElements_Click(object sender, RoutedEventArgs e)
{
    //GetChildren(LayoutRoot).OfType<TextBox>().ToList().ForEach(
    //    t => t.IsReadOnly = !t.IsReadOnly);
    //LayoutRoot.GetChildren().OfType<TextBox>().ToList().ForEach(
    //    t => t.IsReadOnly = !t.IsReadOnly);
    LayoutRoot.GetChildren<TextBox>().ToList().ForEach(
        t => t.IsReadOnly = !t.IsReadOnly);
}

public static class VisualTreeExtensions
{
    public static IEnumerable<T> GetChildren<T>(
        this DependencyObject obj) where T : DependencyObject
    {
        return GetChildren(obj).Where(child => child is T).Cast<T>();
    }
}
```

```

public static IEnumerable<DependencyObject> GetChildren(
    this DependencyObject obj)
{
    int count = VisualTreeHelper.GetChildrenCount(obj);
    for (int i = 0; i < count; i++)
    {
        var child = VisualTreeHelper.GetChild(obj, i);
        yield return child;
        foreach (var descendent in GetChildren(child))
            yield return descendent;
    }
}
}

```



Note The Silverlight Toolkit contains a rich set of *VisualTreeExtensions*, which you are encouraged to use rather than crafting your own. See Chapter 1, “Vision and Architecture,” for details about how to get the Toolkit.

Screen Layout

The Silverlight layout system supports three standard types of layout: absolute, relative, and dynamic. These are actually fairly arbitrary categorizations, but they do map to the layout characteristics of the primary control container types (*Panels*). With absolute layout, you specify absolute values for the size and position of all the visual elements. With relative layout, you specify some values for elements, relative to each other. With dynamic layout, you specify very little; instead, you allow the system to size and position elements based on calculations of the elements’ contents. In all cases, you can specify explicit values in the application XAML or the code-behind, and/or values that are specified elsewhere in style resources. Per Metro guidelines, you are encouraged to use the standard style resources for elements such as fonts, font sizes, margins, padding, and stroke thicknesses. Note that you’re also free to specify your own custom layouts by using whatever behavior you like.

The three types of layout are represented by three container controls, all derivatives of the base *Panel* class:

- *Canvas*, for absolute layout
- *Grid*, for relative layout
- *StackPanel*, for dynamic layout

The screenshots in Figure 2-4 show the *SimpleLayout* solution in the accompanying sample code. This is an application that uses all three standard types of layout. Note that the two *Grid* controls on the *MainPage* have their *ShowGridLines* property set to *True*, which is a useful visual aid during development, although you would usually remove this prior to publication.

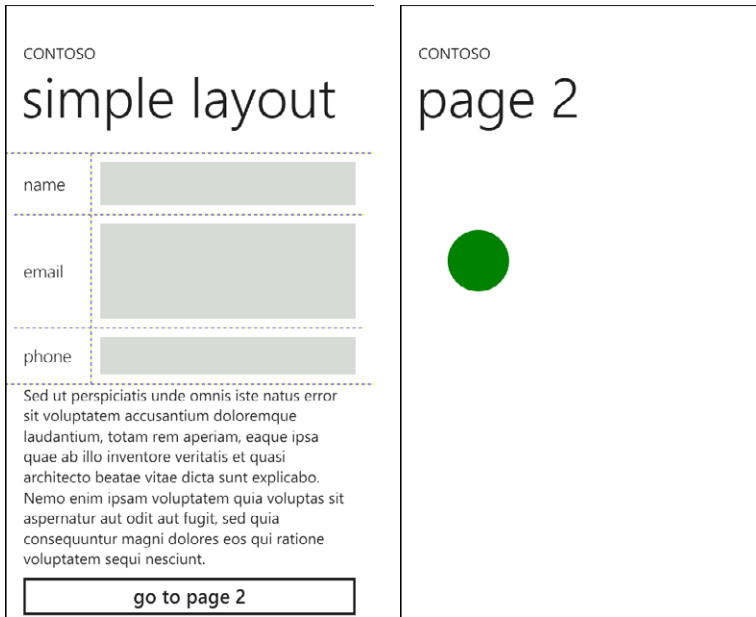


FIGURE 2-4 An application with *Grids* and *StackPanels* on Page 1, and a *Canvas* on Page 2.

The *MainPage* uses relative layout overall and is divided into three areas vertically: the top area uses dynamic layout, the middle area uses relative, and the bottom area uses dynamic layout again. *Page2* uses absolute layout. The *MainPage* XAML has an outer *Grid* (named *LayoutRoot*), within which it defines three areas: one represented by a *StackPanel*, the second by a *Grid*, and the third by another *StackPanel*.

```
<Grid x:Name="LayoutRoot" ShowGridLines="True">
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto"/>
        <RowDefinition Height="300"/>
        <RowDefinition Height="*/>
    </Grid.RowDefinitions>

    <StackPanel x:Name="TitlePanel" Grid.Row="0">
        <TextBlock x:Name="ApplicationTitle" Text="CONTOSO"/>
        <TextBlock x:Name="PageTitle" Text="simple layout"/>
    </StackPanel>

    <Grid x:Name="ContentPanel" Grid.Row="1">
        <Grid.RowDefinitions>
            <RowDefinition Height="80"/>
            <RowDefinition Height="2*/>
            <RowDefinition Height="*/>
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="100"/>
            <ColumnDefinition Width="*/>
        </Grid.ColumnDefinitions>
```

```

<TextBlock Grid.Row="0" Grid.Column="0" Text="name"/>
<TextBlock Grid.Row="1" Grid.Column="0" Text="email"/>
<TextBlock Grid.Row="2" Grid.Column="0" Text="phone"/>

<TextBox Grid.Row="0" Grid.Column="1"/>
<TextBox Grid.Row="1" Grid.Column="1"/>
<TextBox Grid.Row="2" Grid.Column="1"/>
</Grid>

<StackPanel Grid.Row="2">
  <TextBlock TextWrapping="Wrap" Text="Sed ut perspiciatis unde omnis iste
atus error sit voluptatem accusantium doloremque laudantium, totam rem aperiam, eaque ipsa
uae ab illo inventore veritatis et quasi architecto beatae vitae dicta sunt explicabo. Nemo
nim ipsam voluptatem quia voluptas sit aspernatur aut odit aut fugit, sed quia consequuntur
agni dolores eos qui ratione voluptatem sequi nesciunt. "/>
  <Button Content="go to page 2" x:Name="Page2" Click="Page2_Click"/>
</StackPanel>
</Grid>

```

StackPanel controls use dynamic layout, in the sense that the Silverlight runtime computes the layout for each element based on the size of the contents. The two *TextBlock* controls in the first *StackPanel* contain different text values and use different fonts and sizes. The default stacking mode in a *StackPanel* is vertical, so the second *TextBlock* starts wherever the first one left off. As it happens, the second *Grid* starts wherever the first *StackPanel* (and the second *TextBlock*) left off, but for a different reason. The *StackPanel* and *Grid* are both children of an outer *Grid*, and the outer *Grid* specifies row heights. In this example, the first row height is set to *Auto*, so it takes up whatever space is needed by the child *StackPanel*. This is why the second *Grid* effectively starts where the *StackPanel* ended. Using the same mechanism, the *Button* in the second *StackPanel* is pushed down by the size of the *Text* value above it. So, the *Button* positioning is relatively arbitrary, and the result might or might not end up being aesthetically pleasing.

In contrast, grids use relative layout, with which you have more explicit control over the sizes and relative positions of each child element. Consider the first *Grid*, which specifies three *RowDefinitions*, each with their heights set to *Auto*, *300*, and ***, respectively:

```

<Grid.RowDefinitions>
  <RowDefinition Height="Auto"/>
  <RowDefinition Height="300"/>
  <RowDefinition Height="*/>
</Grid.RowDefinitions>

```

- *Auto* specifies that the row height will be whatever it needs to be for the height of the content. In this case, the content is a *StackPanel* made up of two *TextBlock* controls, with their heights, margins, and padding determined by the style resources defined for them.
- *300* is the height in pixels of the second grid row, which is occupied by an inner *Grid*. This will be constrained to 300 pixels, regardless of the sizes of its children.
- *** indicates that this row will be allocated whatever height is left after the first two rows. That is, $(Page\ Height - (ApplicationTitle + PageTitle + Margins) - 300)$.

The inner *Grid* (named *ContentPanel*), is divided into three rows and two columns:

```
<Grid.RowDefinitions>
  <RowDefinition Height="80"/>
  <RowDefinition Height="2*"/>
  <RowDefinition Height="*/>
</Grid.RowDefinitions>
<Grid.ColumnDefinitions>
  <ColumnDefinition Width="100"/>
  <ColumnDefinition Width="*/>
</Grid.ColumnDefinitions>
```

Row 0 is 80 pixels high. Rows 1 and 2 are allocated heights using a weighting of 2 and 1, respectively. This is a 2:1 ratio of the remaining space; that is, $300 - 80 = 220$, yielding values of 147 and 73.

Finally, *Page2* has the traditional outer *Grid* and inner *StackPanel* for the *ApplicationTitle* and *PageTitle*. Beyond that, however, the remaining space is taken up by a simple *Canvas*. The *Ellipse* is positioned by using absolute values for the X and Y coordinates, specified as values for the *Canvas.Left* and *Canvas.Top* properties. The upper-left corner of the phone screen is X,Y coordinate 0,0. Note that these are attached properties (discussed later in this chapter).

```
<Canvas x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
  <Ellipse Width="80" Height="80" Fill="Green" Canvas.Left="50" Canvas.Top="100"/>
</Canvas>
```

A *Canvas* clearly gives you absolute control over the layout of the child elements. The corollary is that you must specify layout values explicitly, because nothing is done for you. If none of the standard *Panel* classes give you the layout control you want, you can consider creating a custom panel.

UserControl vs. Custom Control

The difference between *UserControls* and Custom Controls is a common source of confusion. Both are control types that you define; both are based on a standard type in the base class library; both are supported in Silverlight; and you can use either or both in a Windows Phone application. So much for the commonalities; the major differences are listed in Table 2-2.

TABLE 2-2 *UserControl* vs. Custom Control

Feature	UserControl	Custom Control
Derivation	You typically derive a class from <i>UserControl</i> . Visual Studio will only generate a type derived from <i>UserControl</i> , but you can derive from another type such as <i>ChildWindow</i> or <i>PhoneApplicationPage</i> (which is itself a <i>UserControl</i>), by manually editing the generated XAML and code files.	You can derive a class from any <i>Control</i> -based class, and for complete freedom, you would derive from the base <i>Control</i> class itself. You can derive from <i>ItemsControl</i> if you're building a custom control for a collection (think, <i>ListBox</i>). Derive from <i>ContentControl</i> if your custom control will have only one piece of content (think, <i>Button</i>). Derive from <i>Panel</i> if your control will be used as a layout container (think, <i>StackPanel</i>).

Feature	UserControl	Custom Control
Tool support	Visual Studio provides a project item template for a Windows Phone <i>UserControl</i> .	Visual Studio provides a project item template for a <i>class</i> .
Starter code	Visual Studio generates the same code as for a desktop Silverlight <i>UserControl</i> , there's nothing phone-specific about it in the starter code.	Visual Studio generates the same code as for a desktop Silverlight class, there's nothing phone-specific about it in the starter code.
Designer support	You get a XAML file and a standard code-behind code file. This means you get XAML visual designer support.	You get only a code file. There is no visual designer support. Note that you can create the template visually in Microsoft Expression Blend.
Canonical use case	Use when you want a composite control that contains other controls, which you can declare in XAML (or programmatically).	Use when you want completely custom behavior that might or might not include any other controls. Typically used to extend the behavior of a standard control via subclassing. Also required if you want to support retemplating.
Developer experience	Rapid application development (RAD).	Requires more work, and a better understanding of Silverlight.

Figure 2-5 is a screenshot of an application that uses a custom layout control (*Panel*). This is the *CustomPanel* solution in the sample code.



FIGURE 2-5 A custom layout panel.

The behavior of this control becomes more apparent if you examine the XAML for the application in which it is being consumed.

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <local:MyCustomPanel Background="Transparent" Width="456" Height="500">
        <Rectangle Fill="#FFFF0097"/>
        <Rectangle Fill="#FFA200FF"/>
        <Rectangle Fill="#FF00ABA9"/>
        <Rectangle Fill="#FF8CBF26"/>
    </local:MyCustomPanel>
</Grid>
```

You can see that the custom control, named *MyCustomPanel*, contains four *Rectangle* objects. None of the children define a value for *Width* or *Height*, yet the control lays out the children using a scaling factor so that they decrease in size toward the right. If you want this kind of full control over layout, you must override the *MeasureOverride* and *ArrangeOverride* methods. These are called in sequence. First *MeasureOverride* is called—and this is your opportunity to specify the size available for each of the control's children. If a child already has a width and height defined, then it is common to specify the smaller of the defined size and the available size. Of course, this is only a guideline; there's nothing to stop you from specifying any arbitrary size you like, including a zero size. If you specify a zero size for any child, then that child will not be displayed. It might even be appropriate to specify a size that is larger than the available size, if the intended purpose of the control is, for example, to provide a viewport onto a larger item. Finally, you return the overall size available for the control itself, which, of course, should be equal to or less than the available size that the system passes in to *MeasureOverride*. It should be greater than or equal to the total size of all the children.

After the measure pass is complete, the system calls *ArrangeOverride*. This is your opportunity to specify the position and size of each child, and the final size of the control itself. During this call, you would typically calculate the position of the children by using their desired sizes as a factor. The desired sizes were set previously in the *MeasureOverride*. You are free to lay out your children in any way you like, including overlapping, off the screen, and not at all.

In this custom control, you define a scale factor, which is applied in the *MeasureOverride* to the size of the first child and progressively reduced for each subsequent child. In the *ArrangeOverride*, you simply lay out all the children in a single row.

```
public class MyCustomPanel : Panel
{
    private Size maxSize;
    private double baseScaleFactor = 0.9;

    protected override Size MeasureOverride(Size availableSize)
    {
        if (Children.Count > 0)
        {
            double childWidth = Width / Children.Count;
            double childHeight = Width / Children.Count;
        }
    }
}
```

```

        maxSize = new Size(childWidth, childHeight);
        int i = 0;
        double currentScaleFactor = 1.0;
        foreach (FrameworkElement child in Children)
        {
            child.Width = maxSize.Width;
            child.Height = maxSize.Height;

            currentScaleFactor *= baseScaleFactor;
            child.Measure(new Size(
                childWidth * currentScaleFactor, childHeight * currentScaleFactor));
            i++;
        }
    }
    return new Size(Width, Height);
}

protected override Size ArrangeOverride(Size finalSize)
{
    for (int i = 0; i < Children.Count; i++)
    {
        double dw = Children[i].DesiredSize.Width;
        double dh = Children[i].DesiredSize.Height;
        Children[i].Arrange(new Rect(maxSize.Width * i, 0, dw, dh));
    }

    return new Size(Width, Height);
}
}

```



Note One drawback of this implementation is that it uses the panel's *Width* and *Height* property values inside both methods. However, this requires that these are set in XAML—and they might not be, if the developer chooses to specify size and position by *HorizontalAlignment*/*VerticalAlignment*, for example.

Routed Events

UI events in Silverlight are represented by the *RoutedEvents* class. This is slightly confusing because it covers both events that are routed and events that are not routed. Or, perhaps more strictly, it covers events that are fully routed and events that are only partially routed. If you take the perspective of events that are surfaced for the application code to handle, then a good example of a “routed” event is the *MouseLeftButtonDown* event. A good example of a “non-routed” event is the *Click* event on a *Button*. In fact, as you’ll see shortly, the events surfaced to application code are often a façade for other events.

Figure 2-6 is a screenshot of a simple application (the *SimpleEvents* solution in the sample code) that responds to tap events. The page has a standard “starter” layout. That is to say, it consists of a hierarchical structure, as shown and described here:

- An unnamed *PhoneApplicationPage* at the base (outermost) level of the visual tree, which contains
 - A *Grid* named *LayoutRoot*, which in turn contains
 - A *StackPanel* named *TitlePanel* (with two *TextBlock* controls that are not interesting for this exercise)
 - A *StackPanel* named *ContentPanel* (with a green background), and finally
 - A *Button* named *MyButton*

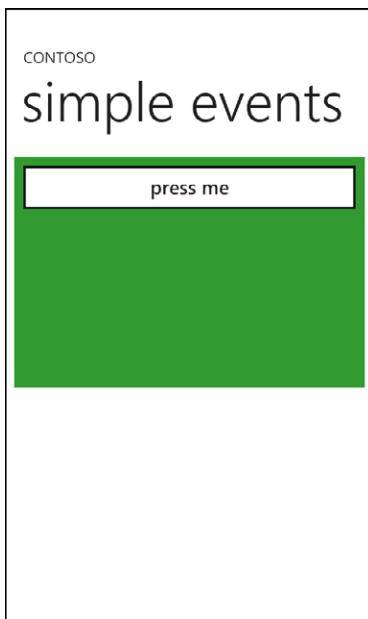


FIGURE 2-6 This simple application demonstrates event routing and the visual tree.

If the user taps the *Button*, this raises a non-routed *Click* event. If the user taps anywhere else, it raises a routed *MouseLeftButtonDown* event. On the emulator, of course, this will actually be a physical mouse left-button down, but on the device, it will be a tap touch event. Under the covers in fact, for both the emulator and device—and for both the *Button* and non-*Button* areas—the initial event raised as a result of user input is a *MouseLeftButtonDown* event. The *Button* class handles this internally and then raises the *Click* event as a result. The *Click* event is routed only as far as the class where the *Button* instance is declared, and there is no automatic onward routing.

Contrast this with the raw *MouseLeftButtonDown* event raised when the user taps any of the other visual elements. Unlike the *Button* class, neither the *StackPanel*, *Grid*, nor *PhoneApplicationPage* have any special logic to handle this event.

In the page class in this example, there are event handlers at every level in the visual tree. However, if you examine the debug output from these handlers, you'll see that when the user taps the *Button*, only the *MyButton_Click* handler is invoked. Conversely, if the user taps in the *StackPanel* outside the *Button*, the event is first handled in the *ContentPanel*'s handler. After that, it is automatically routed to the next handler in the tree (the *LayoutRoot*); from there, it is routed to the outermost *Page* handler. At any time, you could stop the routing by setting the *Handled* property of the *MouseButtonEventArgs* to *True*. In contrast, the *RoutedEventArgs* that is passed in the *Click* event does not expose a *Handled* property; there is no need to because the event is not automatically onward-routed.

```
private void MyButton_Click(object sender, RoutedEventArgs e)
{
    Debug.WriteLine("{0} - {1}: MyButton_Click\n",
        sender.GetType(), e.OriginalSource.GetType());
}

private void MyButton_MouseLeftButtonDown(object sender, MouseButtonEventArgs e)
{
    Debug.WriteLine("{0} - {1}: MyButton_MouseLeftButtonDown",
        sender.GetType(), e.OriginalSource.GetType());
}

private void ContentPanel_MouseLeftButtonDown(object sender, MouseButtonEventArgs e)
{
    Debug.WriteLine("{0} - {1}: ContentPanel_MouseLeftButtonDown",
        sender.GetType(), e.OriginalSource.GetType());
}

private void LayoutRoot_MouseLeftButtonDown(object sender, MouseButtonEventArgs e)
{
    Debug.WriteLine("{0} - {1}: LayoutRoot_MouseLeftButtonDown",
        sender.GetType(), e.OriginalSource.GetType());
    //e.Handled = true;
}

private void PhoneApplicationPage_MouseLeftButtonDown(object sender, MouseButtonEventArgs e)
{
    Debug.WriteLine("{0} - {1}: PhoneApplicationPage_MouseLeftButtonDown\n",
        sender.GetType(), e.OriginalSource.GetType());
}
```

Also note that the event handler signature follows the Microsoft .NET standard. The first parameter is an object that represents the current source of the event—that is, the current source in the routing path. The second parameter is the *EventArgs* (or *EventArgs*-derived) object that carries any interesting payload. In the case of routed events, it exposes the *Handled* property. To be clear, although *MouseButtonEventArgs* derives indirectly from *RoutedEventArgs*, it is the *MouseButtonEventArgs* where the *Handled* property is defined, not in the *RoutedEventArgs* class. The only additional information that the *RoutedEventArgs* provides over and above its base *EventArgs* type is the *OriginalSource* property, which is the original source of the event.

So, if the user taps within the *ContentPanel* outside the *Button*, the *OriginalSource* is always the *ContentPanel*, and the sender will be the object in which the event is being handled. This will vary, depending on where in the routing path the event has reached.

It is also interesting to note that you have choices about how to connect event handlers. So far, this example has hooked up the *MouseLeftButtonDown* event at the level of the *ContentPanel* in XAML, as shown in the following:

```
<StackPanel Background="#FF339933"
    x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0" Height="300"
    MouseLeftButtonDown="ContentPanel_MouseLeftButtonDown">
```

Instead, you could hook it up in code, as shown in the snippet that follows, which would achieve exactly the same behavior:

```
ContentPanel.MouseLeftButtonDown +=
    new MouseButtonEventHandler(ContentPanel_MouseLeftButtonDown);
```

However, hooking up events in code gives you one further option: you could use the explicit *AddHandler* method instead of the `+=` approach, and if you do so, you can then pass an additional parameter:

```
ContentPanel.AddHandler(UIElement.MouseLeftButtonDownEvent,
    new MouseButtonEventHandler(ContentPanel_MouseLeftButtonDown), true);
```

The difference this makes is in the final parameter; if this is true, the framework will invoke the handler for routed events that have already been marked as handled by another element along the event route, for example, the *MouseLeftButtonDown* event that was “handled” by the *Button* to produce the *Click* event.

It is important to note that the routing path follows the visual tree, from the most local (leaf-node) object to the outermost (root-node). As a performance optimization, therefore, it makes sense to set *Handled=true* when you’re sure that you have in fact handled the event and want to prevent any unnecessary onward routing. Also remember that some visual elements in an application are not part of the visual tree (Popups, and the Application Bar are the common examples), so they will not be involved in event routing. Another quirk to note is that if you don’t explicitly set a *Background* on a *Panel* (including *StackPanel* and *Grid*), then the event is not routed to the *Panel*. A *Transparent Background* is sufficient—it doesn’t need to be visible—but you do also need *IsHitTestVisible* set to *True* (which is the default). In this sample, the *ContentPanel* has an explicit background and receives events, but the *TitlePanel* does not.

Resources

An application typically consists of code plus data. The data can include image files, media files (audio or video), and text files—indeed, any kind of file in any format. All of these are considered to be resources. In addition, the term “resource” has two more specific meanings in the context of Windows Phone development. One meaning relates to the way in which you package the resource in your

application's .xap file. The other meaning relates specifically to code resources, which will be incorporated within the application's XAML. In the following sections, you'll see the different ways of packaging your resource files, and your options for defining XAML resources.

Content vs. Resource

There are two ways to include resources in a Windows Phone project: either with the build action set to *Content*, or with the build action set to *Resource*. The key differences are listed in Table 2-3, using as an example an image resource named "MyImage.jpg" in the folder "Images," at the root of a project with the assembly name "MyAssembly." Note that Visual Studio and Expression Blend have varying defaults for when you add different types of content, so you should always double-check that you have the value that you want. Also, you can select multiple items in the Visual Studio Solution Explorer and set the value for all of them in one go. Finally, there are several other build actions available, as listed in the Visual Studio property grid, but none of the other actions are relevant for phone applications.

TABLE 2-3 Content vs. Resource

Issue	Content	Resource
Location in the XAP	Loose in the XAP, in the specified folder	Embedded in the assembly itself
Source path (current assembly)	/Images/MyImage.jpg (the leading slash is optional)	Images/MyImage.jpg (note <i>no</i> leading slash)
Source path (external assembly)	n/a	/MyAssembly;component/Images/MyImage.jpg (note the leading slash)
Performance	Faster startup, slower to load the image	Slower startup, faster to load the image
Performance (media)	Faster startup, faster to play the media	Slower startup, slower to play the media
Assembly size	Smaller	Bigger
Loading behavior	Asynchronous	Synchronous
XAP size	(Same)	(Same)
Optimal use case	Application	Library
Used for system UI (Application Bar, SplashScreen, LiveTile, and so on)	Must use Content	Cannot use Resource

The following XAML shows valid syntax for loading both Content and Resource images from the current assembly, and a Resource image from an external assembly:

```
<!-- Content resource -->
<Image
  Width="200" Height="150"
  Source="/Images/Palms1.jpg"/>

<!-- Internal Resource resource -->
<Image
  Width="200" Height="150"
  Source="Images/Palms2.jpg"/>
```

```
<!-- External Resource resource -->
<Image
    Width="200" Height="150"
    Source="/ImageLibrary;component/Images/Coconuts.jpg"/>
```

Once a resource has been read into memory, it can be cached—this is especially true of image resources. So, if you have only a few small resources, it might be worth taking the load-time performance hit incurred when you embed them as resources in the assembly. You need to balance this against the marketplace certification requirement that your application must show its first screen within 5 seconds of launch, and be responsive to user interaction within 20 seconds. If you embed too many resources, you can easily exceed these startup limits. There’s an additional twist in the case of media (audio and video) resources: if these are embedded in the assembly, they will nevertheless be copied out to files in isolated storage before they are played back. The underlying reason for this is because the media functionality on the phone is optimized for playback from network streaming and from disk file, but not from memory. In general, therefore, you should never mark audio/video as *Resource*.

Resource Dictionaries

In Windows Phone development, there are two different concepts of resources:

- Data resources, such as images, text files, and audio and video files (discussed above).
- Reusable XAML or code resources, such as styles, templates, brushes, colors, animations, and so on.

For example, consider the case for which you define a *Brush* for the *Foreground* in a *TextBlock*. The simple approach is to define this *Brush* inline with the definition of the *TextBlock*. You can see this at work in the *SimpleResources* solution in the sample code. The XAML that follows results in the first piece of text, “Monday”, in Figure 2-7.

```
<TextBlock Text="Monday">
    <TextBlock.Foreground>
        <LinearGradientBrush>
            <GradientStop Color="#FF339933" Offset="0"/>
            <GradientStop Color="#FF09609" Offset="1"/>
        </LinearGradientBrush>
    </TextBlock.Foreground>
</TextBlock>
```

If, however, you find that you’re using the same *Brush* for multiple *TextBlock* controls, then it makes sense to abstract the definition to a resource. Typically, you would define this in a *Resources* section at the page level. When you define a resource it must have a *Key* name. It is common to use the explicit “x:” prefix, where “x” is defined in the application as the namespace for the standard Silverlight XAML schema. A resource can have a *Name* instead of or in addition to a *Key*. If *Key* is not specified, then *Name* is used as the key. The reason for this is that *Name* is a legacy resource syntax that is maintained for backward compatibility to support existing Silverlight usage. This is not strictly relevant for Phone applications, and the preferred approach is to use *Key* for resource identifiers. Moreover, recall that when you declare a *Name* for an element in XAML, this generates a corresponding class member,

plus code in the *Initialize* method to call *FindName* to match up the named element with the code variable. In the case of a resource, it is unlikely that you'll want to use this class member in code, so that's all a waste of time.

```
<phone:PhoneApplicationPage.Resources>
  <LinearGradientBrush x:Key="MyGradientBrush">
    <GradientStop Color="#FF339933" Offset="0"/>
    <GradientStop Color="#FFF09609" Offset="1"/>
  </LinearGradientBrush>
</phone:PhoneApplicationPage.Resources>
```

Having defined the resource, you would then consume it by using the *{StaticResource}* syntax. The following code consumes the *Brush* resource, and also consumes two other resources: a *Margin* and a *FontSize*. This is the second piece of text, "Tuesday", in Figure 2-7.

```
<TextBlock
  Text="Tuesday" Foreground="{StaticResource MyGradientBrush}"
  Margin="{StaticResource PhoneMargin}"
  FontSize="{StaticResource PhoneFontSizeExtraExtraLarge}"/>
```

The two standard resources used here are defined in the standard theme resources.

```
<Thickness x:Key="PhoneMargin">12</Thickness>
<!--54pt-->
<System:Double x:Key="PhoneFontSizeExtraExtraLarge">72</System:Double>
```

Note that *PhoneApplicationPage.Resources* is of type *ResourceDictionary*. A *ResourceDictionary* is a *DependencyObject* that implements *IDictionary*, so it contains a regular collection of key-value pairs. The key is an arbitrary string that the application defines. The value is the resource itself. There is a finite list of types that can be put into a resource dictionary. For an object to be defined in a resource dictionary, it must be shareable. This is required because when the object tree of an application is constructed at runtime, any given object cannot exist at multiple locations in the tree. So, it must exist at one location in the tree, and therefore be shareable from that location to all its consumers in the tree. The following types are supported:

- *Styles* and templates
- *Brushes* and colors
- *Animation* types, including storyboards
- *Transforms*, *Matrix*, *Matrix3D*, and *Point* structure values
- Custom types defined in the application code and instantiated in XAML as a resource, including resource and value converters
- Strings and basic numeric values such as double and int

It's worth mentioning that any type of *UIElement* (such as a *Control*) is not supported.

Although you typically define such resources at the page level, you can in fact define resources for any *FrameworkElement*—that is to say, for any element in the logical tree. For example, you could define the resource at the level of the *StackPanel* that contains the *TextBlock*. Just to illustrate the point, the following definition uses different colors:

```
<StackPanel.Resources>
  <LinearGradientBrush x:Key="MyGradientBrush">
    <GradientStop Color="#FFA200FF" Offset="0"/>
    <GradientStop Color="#FF00ABA9" Offset="1"/>
  </LinearGradientBrush>
</StackPanel.Resources>
```

Note, however, that this definition specifies the same *Key* name. If a different *Key* name is used, then the consuming code could choose to use either of the two resources. If the same resource (that is, with the same *Key* name) is defined at two or more different levels in the tree, then the system uses the one that is most local to the consuming element. In the current example, the visual tree is essentially *Page*→*StackPanel*→*TextBlock*. So, if the same resource is defined at both *Page* and *StackPanel* levels, the system will use the more local *StackPanel* version.

In addition to *FrameworkElements*, you can also define resources for the *Application* class. Application-level resources are clearly available to all pages within the application. The example that follows defines a *Brush* resource with the key "MyAppGradientBrush". In the page code, you consume this on a third *TextBlock*. In the screenshot in Figure 2-7, this is the third piece of text ("Wednesday").

```
<Application.Resources>
  <LinearGradientBrush x:Key="MyAppGradientBrush">
    <GradientStop Color="#FFE671B8" Offset="0"/>
    <GradientStop Color="#FF8CBF26" Offset="1"/>
  </LinearGradientBrush>
</Application.Resources>
```

To resolve a resource reference, the Silverlight runtime walks the logical tree outward in scope, from child to parent, starting with the consuming element. So, it will start with the object where the actual usage is applied and that object's own *Resources* property. If that resource dictionary is not null, then the system searches in that dictionary for the specified resource, based on its key. If it is found, the lookup stops and the resource is applied. Note that it is not generally useful to define a resource within the object where it is consumed; the value is only accessible within that object, thus you might as well define the values inline, instead of taking the performance hit of resource resolution. If the resource is not found in the immediate object's resource dictionary, the lookup then proceeds to the next outer (parent) object in the tree and searches there. This sequence continues until the root element of the XAML is reached, exhausting the search of all possible immediate (that is, page or frame-level) resource locations. If the requested resource is not found in the page-level resources, then the runtime checks the *Application* resources. If the resource is not found there, you'll get a XAML parse error.

Clearly, you would define your resources at a level that provides the reuse you want. Typically, this means at the page level if the resource is used by multiple elements on that page, or at the application level if it is used across multiple pages. In addition, you can define resources in external files, and

then merge those files into your application-level or page-level resource dictionaries. For example, you could add a new XML file to the project, naming it perhaps *MyResources.xaml*. The default build action for a XAML file is *Page*, which is what you need in order to have the external resources built into the assembly. In this file, you can define one or more resources in a *ResourceDictionary*. The following example defines a *Brush* resource with the key "MyOtherBrush":

```
<ResourceDictionary
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
    <LinearGradientBrush x:Key="MyOtherBrush">
        <GradientStop Color="#FFFF0097" Offset="0"/>
        <GradientStop Color="#FFA05000" Offset="1"/>
    </LinearGradientBrush>
</ResourceDictionary>
```

In the application, you instruct the XAML parser to merge this resource dictionary into any of the resource dictionaries in the application, either for any *FrameworkElement* or at the *Application* level.

```
<Application.Resources>
    <ResourceDictionary>
        <LinearGradientBrush x:Key="MyAppGradientBrush">
            <GradientStop Color="#FFE671B8" Offset="0"/>
            <GradientStop Color="#FF8CBF26" Offset="1"/>
        </LinearGradientBrush>
        <ResourceDictionary.MergedDictionaries>
            <ResourceDictionary Source="MyResources.xaml" />
        </ResourceDictionary.MergedDictionaries>
    </ResourceDictionary>
</Application.Resources>
```

Then, you can consume this just like any other accessible resource in the project. This results in the text "Thursday" in Figure 2-7.

```
<TextBlock
    Text="Thursday" Foreground="{StaticResource MyOtherBrush}"
    Margin="{StaticResource PhoneMargin}"
    FontSize="{StaticResource PhoneFontSizeExtraExtraLarge}"/>
```

Note that using merged resource dictionaries affects both the lookup sequence and also the key uniqueness requirements. In the lookup sequence, a merged resource dictionary is checked only after checking all the resources of the local resource dictionary.

You can specify more than one *ResourceDictionary* within *MergedDictionaries*. Once the lookup behavior exhausts the search in the main dictionary and reaches the merged dictionaries, each item in *MergedDictionaries* is checked, but in the inverse order that they are declared in the *MergedDictionaries* property—that is, in last-in, first-out (LIFO) order. Why is this? You might expect that a more logical sequence would be first-in, first-out (FIFO), in the order in which dictionaries are declared. The order is reversed to allow for dynamic additions to the collection. The classic scenario for which this is used is user preferences. These would be known only at runtime, and would need to take precedence over any static resources. For this reason, you must search the user preferences first, and for that to happen, the search order must be LIFO.

Also, the key uniqueness requirement does not extend across merged dictionaries. This means that you could define the same key in multiple merged dictionaries. As always, the search for keys stops when a match is found, so any duplicate keys later in the search sequence are irrelevant.

A resource can reference another resource, but only if that other resource has already been encountered in the lookup sequence. Forward references are illegal; thus, you need to understand the lookup sequence in order to avoid forward references. So, any resources that will be referenced by other resources need to be defined at an earlier point in the lookup sequence—which translates to a wider scope.

Finally, you can consume resources that are defined in an external assembly. This takes the reuse aspect of resource dictionaries to its logical conclusion. Beyond defining resources in an external XML file, you can build that file (or multiple such files) into a separate library assembly. The definition of the resources doesn't change; you could build the exact same *MyResources.xaml* file into a separate assembly. The consumption of these resources does change: you must use the `"/<AssemblyName>;component/<path-to-resource-XAML>"` syntax:

```
<Application.Resources>
  <ResourceDictionary>
    <ResourceDictionary.MergedDictionaries>
      <ResourceDictionary
        Source="/ResourceLibrary;component/MyLibraryResources.xaml" />
    </ResourceDictionary.MergedDictionaries>
  </ResourceDictionary>
</Application.Resources>
```

In the *SimpleResources* solution (in the sample code), this last resource is used for the fifth *Text Block* ("Friday"). The resulting UI from all the foregoing resource approaches is shown in Figure 2-7.

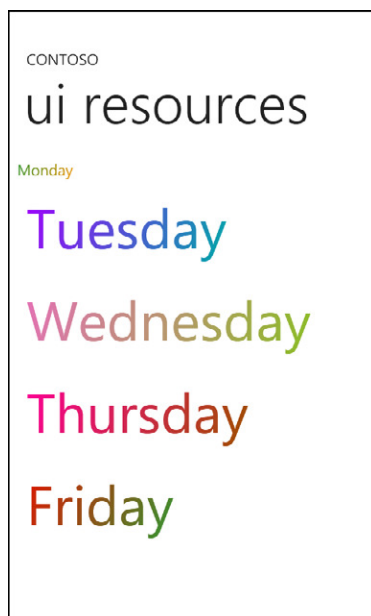


FIGURE 2-7 Using XAML resources and resource dictionaries.

Dependency and Attached Properties

The Common Language Runtime (CLR) supports a code design in which a class can expose properties. Silverlight adds two additional features on top of CLR properties: dependency properties and attached properties. You can use these features to make your classes interoperate more seamlessly with the Silverlight runtime, and they're especially useful if you have highly customized UI requirements.

Dependency Properties

Dependency properties are part of the Silverlight infrastructure. They are designed to augment basic CLR properties with features such as visual inheritance, animation, and data binding. They provide a structured way to give properties a default, inherited, or animated value, and to provide callbacks that are invoked when the value of the property changes. Almost all Silverlight class properties are dependency properties. If you want to create a custom control that exposes properties—and you want those properties to be settable in styles, templates, transforms, data bindings, or as targets of animation—then they must be set up as dependency properties.

In this example, you implement a custom *Button* that exposes a *Direction* property. This *Direction* property is used internally to govern which one of two different images to use for the *Button* content.

Any class that implements dependency properties must derive (directly or indirectly) from *DependencyObject*. The custom class derives from *Button*, which has *DependencyObject* in its hierarchy. Your class has a regular property, *Direction* (of type *Direction*, which might be a little confusing to humans, but not to the compiler), which uses the *DependencyObject.GetValue/SetValue* methods. The regular property is not backed by a private field; instead, it is backed by a Silverlight-maintained property repository. A dependency property has a public static field of type *DependencyProperty*, which has the same name as the underlying regular property but with "Property" appended to it. You set up a dependency property by registering its property name, type, the type of the enclosing class, the default value, and the callback to be invoked when the property value changes, as demonstrated in the following:

```
public enum Direction { Up, Down }

public class CustomButton : Button
{
    private static Image arrowUp;
    private static Image arrowDown;

    public Direction Direction
    {
        get { return (Direction)GetValue(DirectionProperty); }
        set { SetValue(DirectionProperty, value); }
    }

    public static readonly DependencyProperty DirectionProperty =
        DependencyProperty.Register("Direction",
            typeof(Direction),
            typeof(CustomButton),
            new PropertyMetadata(Direction.Up, OnDirectionChanged));
}
```


In the constructor, you initialize two alternative *Image* objects that correspond to the two possible *Direction* property values.

```
public CustomButton()
{
    arrowUp = new Image();
    BitmapImage bmp = new BitmapImage(new Uri(
        "/Images/black_arrow_up.png", UriKind.Relative));
    arrowUp.Source = bmp;

    arrowDown = new Image();
    bmp = new BitmapImage(new Uri(
        "/Images/black_arrow_down.png", UriKind.Relative));
    arrowDown.Source = bmp;

    this.Content = arrowUp;
}
```

In the *OnDirectionChanged* callback, you fetch the new *Direction* property value from the *EventArgs* provided in the call and switch the *Content* to the up or down image, accordingly.

```
static void OnDirectionChanged(
    DependencyObject sender, DependencyPropertyChangedEventArgs e)
{
    CustomButton button = sender as CustomButton;
    Direction d = (Direction)e.NewValue;
    if (d == Direction.Up)
    {
        button.Content = arrowUp;
    }
    else
    {
        button.Content = arrowDown;
    }
}
```

In the XAML, set up a namespace for the current assembly so that you can access the types in the XAML itself.

```
<phone:PhoneApplicationPage
...
    xmlns:local="clr-namespace:TestDependencyProps"
>
```

There are various ways you can use your custom dependency property. One is to set up a *Style* resource, as shown here:

```
<phone:PhoneApplicationPage.Resources>
    <Style x:Key="DownButtonStyle" TargetType="local:CustomButton">
        <Setter Property="Direction" Value="Down"/>
    </Style>
</phone:PhoneApplicationPage.Resources>
```

You can also set the property directly at the point where you declare an instance of the custom button. In the following example, the first *Button* has its *Direction* property set directly, the second one falls back on the default value (Up), and the third one uses the custom style:

```
<StackPanel x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <local:CustomButton Width="100" Direction="Down"/>
    <local:CustomButton Width="300"/>
    <local:CustomButton Width="150" Style="{StaticResource DownButtonStyle}"/>
</StackPanel>
```

Figure 2-8 shows the results of these three custom buttons. This is the *TestDependencyProps* solution in the sample code. This might seem like a lot of work to expose a property, but it does allow the property to be used in ways that are not possible for a standard CLR property. For example, you cannot use a standard CLR property in the *Setter* element of a *Style*, nor can you reference a standard CLR property in the XAML attributes for an element instance (unless you construct a custom object as part of a *ResourceDictionary*). Plus, of course, you cannot use it directly in data binding or animation.

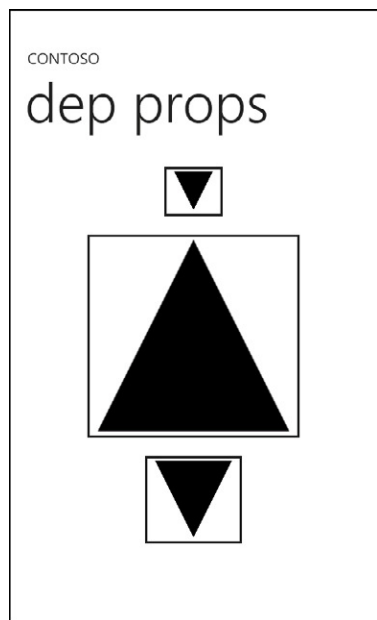


FIGURE 2-8 A demonstration of dependency properties.

Attached Properties

Earlier in this chapter, you learned how to create a custom *Panel* type that lays out its child controls by using a decreasing scale factor. It is often useful to be able to define the value of a parent control's property in the definition of the child. This is especially applicable to custom *Panel* types. To continue the earlier example, suppose that instead of having a fixed scaling factor, you want to allow each child to specify its own scale factor. Figure 2-9 shows the desired effect (see the *CustomPanel_AttachableProperty* solution in the sample code).

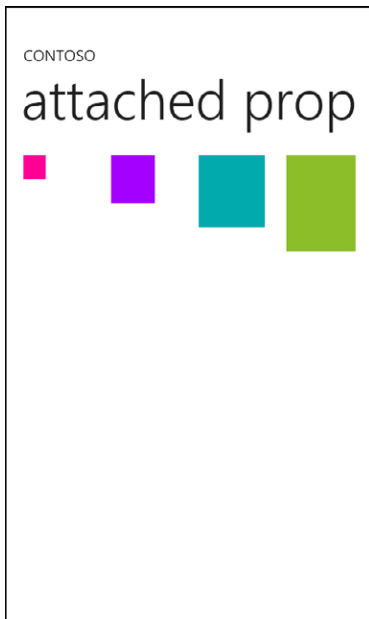


FIGURE 2-9 An application with a custom panel, using attached properties.

The listing that follows shows how you want to use the modified custom *Panel* in the application code. In this scenario, you don't want to specify a scaling factor for the control itself, because then it would apply to all children. Instead, you want to specify a different scaling factor for each child. In XAML, you can provide a value for the *ScaleFactor* for each child as though it were a property of the child itself. You're effectively attaching the *ScaleFactor* property of the parent control to the child element.

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
  <local:MyCustomPanel Background="Transparent" Width="456" Height="500">
    <Rectangle Fill="#FFFF0097" local:MyCustomPanel.ScaleFactor="0.25"/>
    <Rectangle Fill="#FFA200FF" local:MyCustomPanel.ScaleFactor="0.5"/>
    <Rectangle Fill="#FF00ABA9" local:MyCustomPanel.ScaleFactor="0.75"/>
    <Rectangle Fill="#FF8CBF26" local:MyCustomPanel.ScaleFactor="1.0"/>
  </local:MyCustomPanel>
</Grid>
```

To use this scale factor value in the control's layout computations, you can make a simple change to the *MeasureOverride*: use the *ScaleFactor* dependency property for the child instead of the previous fixed value.

```
protected override Size MeasureOverride(Size availableSize)
{
    if (Children.Count > 0)
    {
        double childWidth = Width / Children.Count;
        double childHeight = Height / Children.Count;
```

```

        maxSize = new Size(childWidth, childHeight);
        int i = 0;
        double currentScaleFactor = 1.0;
        foreach (FrameworkElement child in Children)
        {
            child.Width = maxSize.Width;
            child.Height = maxSize.Height;

            currentScaleFactor =
                //baseScaleFactor;
                (double)child.GetValue(ScaleFactorProperty);

            child.Measure(new Size(
                childWidth * currentScaleFactor, childHeight * currentScaleFactor));
            i++;
        }
    }
    return new Size(Width, Height);
}

```

To make the property attachable in the first place, you need to enhance the custom control class with an additional public static field of type, *DependencyProperty*. For a regular dependency property, you would invoke the *Register* method to register this with the runtime. For an attachable dependency property, you instead invoke the *RegisterAttached* method. Also, instead of a regular CLR property backing this dependency property, you need to specify a pair of *GetXxx* and *SetXxx* methods.

```

public static readonly DependencyProperty ScaleFactorProperty =
    DependencyProperty.RegisterAttached(
        "ScaleFactor", typeof(double), typeof(MyCustomPanel), null);

public static double GetScaleFactor(DependencyObject obj)
{
    return (double)obj.GetValue(ScaleFactorProperty);
}

public static void SetScaleFactor(DependencyObject obj, double sf)
{
    obj.SetValue(ScaleFactorProperty, sf);
}

```

Note that passing null as the last parameter to *RegisterAttached* means that the panel will show items as zero width and height if you omit the attached property. This might be acceptable, but it is generally more useful to provide a non-zero default value, especially where the property relates to some size variable. So, a better registration would be the following:

```

public static readonly DependencyProperty ScaleFactorProperty =
    DependencyProperty.RegisterAttached(
        "ScaleFactor", typeof(double), typeof(MyCustomPanel), new PropertyMetadata(1.0));

```

With this updated registration, the following two lines of XAML would then be effectively identical:

```

<Rectangle Fill="#FF8CBF26" local:MyCustomPanel.ScaleFactor="1.0"/>
<Rectangle Fill="#FF8CBF26"/>

```

The model of attachable properties allows you to extend existing controls with custom characteristics. In the preceding example, you could use any control type (*Button*, *CheckBox*, *ListBox*, and so on) as a child within the custom *Panel* and define a value for the attached property. In fact, you could use any *DependencyObject* type. But what is the thinking behind this slightly unusual model? You want a property of some parent control to be attached to its children and definable in the definition of the child instances. How else could this be achieved? Consider the alternatives. Inheritance wouldn't work because you'd have to define the required property in some base class so that all children would inherit it. For example, perhaps provide the property in the base *Control* class. This might serve the purpose where child controls such as *Button* and *CheckBox* would inherit it. It would not work for *Panels* or shapes such as *Rectangle*, because *Panel* and *Shape* derive from *FrameworkElement* not *Control*. One problem is that the parent-child *visual* relationship has nothing to do with the parent-child *inheritance* relationship. For inheritance to be a suitable solution, you'd have to ensure that all properties you want to include in the model are defined on all base classes of all types that could be used as visual children. You'd end up with an inverted hierarchy, where base classes are stuffed with properties that are used only by some subset of children. Worse, the system would not be extensible.

In addition to the logical perspective, attachable properties are actually physically attached to the object for which they're defined. So, in the example, the memory for the *Rectangle* objects includes their attached properties. However, the attached properties do not form part of the object type. To illustrate this, you could declare the XAML that follows, in which the *ScaleFactor* property is attached to a *Rectangle*, but that *Rectangle* does not live in the context of a *MyCustomPanel*. This will compile and run quite happily, but there might be nothing that will ever read the attached property value.

```
<Grid Grid.Row="2" Margin="12,0,12,0" Width="456" Height="200">
  <Rectangle Fill="#FFA05000" Local:MyCustomPanel.ScaleFactor="0.25"/>
</Grid>
```

This is validation of the extensibility of the model: to attach a property, you don't have to make any changes to the target type, and nothing breaks if you attach a property, regardless of whether it is actually used.

Summary

This chapter explored the basic UI infrastructure on the phone and looked at the similarities and differences between Windows Phone development and Silverlight desktop development as well as the various options for layout, controls, resources, and coding approaches to building the UI. You saw some of the subtleties involved in using resource dictionaries and dependency properties, both of which make it easier for designers and developers to work on the same project.

Controls

Chapter 2, “UI Core,” illustrated how you can create your own custom controls when necessary. However, the Windows Phone application platform includes a set of standard controls that are based on the standard Microsoft Silverlight controls. In addition, the SDK includes three additional controls, which, for all intents and purposes, you can treat as if they were a part of the platform. If that’s not enough, you can also use the set of controls that are in the Silverlight Toolkit, which is a separate download. All of these standard controls conform to the Metro design and usability principles, so you should use them in preference to custom controls wherever possible. The UI model, in conjunction with the phone-specific UI controls, not only make phone development easier, but also allow developers to build applications that are more engaging, user-friendly, and actually perform better than desktop applications.

Standard Controls

There is a rich set of standard controls that meet the requirements of most applications, avoiding the need to build custom controls. The full set of standard controls includes the controls in the Windows Phone platform itself, plus the controls in the Windows Phone SDK and the controls in the Silverlight Toolkit, as described in the following sections.

Platform, SDK, Toolkit

Table 3-1 presents the three categories of “standard” libraries that implement controls used in Windows Phone applications.

TABLE 3-1 Standard, SDK, and Toolkit Control Assemblies

Source	Assembly	Compile-Time Reference Location
Platform	System.Windows.dll, Microsoft.Phone.dll	%ProgramFiles%\Reference Assemblies\Microsoft\Framework\Silverlight\v4.0\Profile\WindowsPhone\
SDK	Microsoft.Phone.Controls.dll, Microsoft.Phone.Controls.Maps.dll	%ProgramFiles%\Microsoft SDKs\Windows Phone\v7.x\Libraries\Silverlight\
Toolkit	Microsoft.Phone.Controls.Toolkit.dll	%ProgramFiles%\Microsoft SDKs\Windows Phone\v7.x\Toolkit\<release-date>\Bin\

Note that the Standard and SDK controls are part of the Windows Phone SDK, but the Toolkit is a separate download (see Chapter 1, “Vision and Architecture,” for details). Table 3-2 provides a list of controls that are available in each category.

TABLE 3-2 Platform, SDK, and Toolkit Controls

Source	Controls
Standard	<i>Border, Button, Canvas, CheckBox, Grid, HyperlinkButton, Image, ListBox, MediaElement, PasswordBox, ProgressBar, RadioButton, ScrollViewer, Slider, StackPanel, TextBlock, TextBox, WebBrowser</i>
SDK	<i>Map, Panorama, Pivot</i>
Toolkit	<i>AutoCompleteBox, ContextMenu, DatePicker, GestureService, ListPicker, LongListSelector, PageTransitions, TimePicker, ToggleSwitch, WrapPanel</i>

Apart from the final assemblies where the controls are implemented, the supported controls can also be categorized slightly differently, based on how they are implemented under the covers:

- **Customization of Existing Silverlight Controls** Many controls have a corollary in Silverlight; for example, the *Button* family contains *Push*, *Radio*, and *Checkbox*. These are based on the existing Silverlight control in the *System.Windows.Controls* namespace, enhanced with a new control template to give them the Metro look and feel.
- **New Controls** In some cases, there was no existing Silverlight control suitable for skinning. For these, a new control was built in the *Microsoft.Phone.Controls* namespace. The best examples of these new controls are *Pivot* and *Panorama*. Note that they were not built from scratch. Silverlight has a rich control infrastructure that emphasizes reuse. All but the most basic Silverlight controls are composed of reusable primitives. For example, the *ListPicker* is based on the Silverlight *ItemsControl* that is the heart of all Silverlight list controls.

To implement the Metro user experience (UX) in Silverlight, Microsoft performed the following additional work across the Silverlight control set:

- **Software Input Panel (SIP) and Input Method Editor(IME)** IME is the feature with which users can add characters or symbols that are not on the keyboard/SIP. These were both integrated for *TextBox*.
- **Touch/gesture enabling** Silverlight enabled this off the *UIElement* class, which is very high up in the Silverlight framework hierarchy, thereby enabling this support for any visual element.
- **Scrolling** To support the Metro scrolling behavior and visual indicator, Silverlight updated the *ScrollViewer* and *Scrollbar*.

SDK Controls: *Pivot*

The *Pivot* control is designed to provide a delightful way for the user to switch between alternate views in an application or to filter large datasets on the small screen of a mobile device. The control's experience is optimized for the following uses:

- Filtering large datasets, where it is typically used in conjunction with a *ListBox*. Data from the same data set can be presented in different ways (for example, a collection of songs filtered by title, artist, genre, and so on).
- Displaying related content from different data sets (for example, viewing different segments of information about the same contact).
- Switching application views without the use of multiple pages (to provide a sub-page-level transition experience).

The *Pivot* control is designed to behave in a fast, fluid fashion, and to be enjoyable to use. In addition to standard touch gesture support, it is highly responsive and incorporates carefully designed animations and transitions. It participates in the device orientation experience and includes smart reflow of the UI when a device is rotated.



Note Usability studies arrived at the conclusion that the *Pivot* control should not contain more than seven pages. This is intended to prevent cognitive overload to the user. While the developers of the *Pivot* have not put any hard limit on the number of *PivotItem*s a *Pivot* can contain, there was no extra effort expended to support scenarios that use more than seven items.

Conceptually, the *Pivot* control is divided into the following three logical panes, as defined in the XAML for the *Pivot* control itself:

- A title, which is the same as any other page title, represented in code by the *Pivot.Title* property
- A header for each item, represented by the *PivotItem.Header* property
- A “content pane” represented by the *PivotItem* itself
- These three panes are animated in different ways, and the user will therefore understand that there are three panes. However, in the application code where a *Pivot* is used, these three panes are represented by two entities (in XAML and code-behind): the *Pivot* control and the *PivotItem* control. The *Pivot* exposes a *Title* property, and the *PivotItem* exposes a *Header* property. As far as application code is concerned, the *Header* is part of each *PivotItem*, although at runtime the *Header* is processed as part of the collection of headers within the *Pivot* control. A page can contain one or more *Pivot* controls, but you are encouraged to restrict yourself to one per page, and to make your *Pivot* the full height and width of the page. A *Pivot* control may in turn contain an arbitrary number of *PivotItem* controls. The basic elements of a *Pivot* are shown in Figure 3-1.

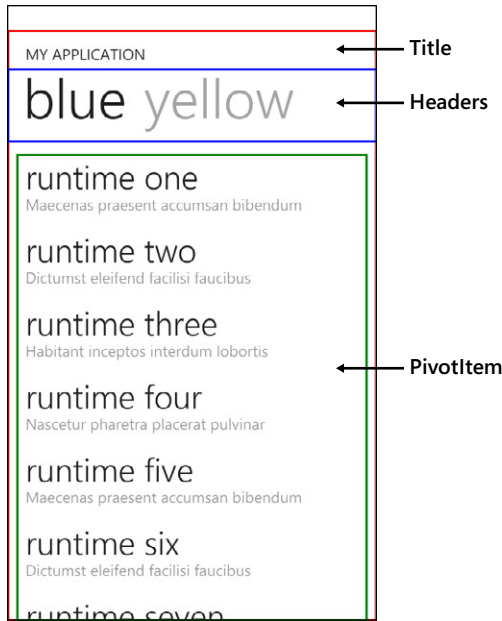


FIGURE 3-1 The fundamental elements of the *Pivot* control, using the standard project template.

Note that if you supply header text that is too long, it is simply cut off at the end of the screen. If this happens, the user will not be able to tap on another *Header* to transition out of the current item. Instead, she would have to pan or flick the current *Header* or *PivotItem* body. Also note that the *PivotItem Header* is a fixed height across all *PivotItems* in the same *Pivot* control.

The *Pivot* control supports the following special animations and transitions:

- Initial fly in, with the title and headers appearing first, followed by the content for the first (or last visited) *PivotItem*.
- When transitioning between items, the new header and any new headers to the right or left are rendered first, followed by a custom animation on the *PivotItem* content, and lastly, the old header in the case where this has wrapped to the end.
- When moving to a different item, both the *PivotItem Header* and *PivotItem* itself visually update simultaneously.
- When the user flicks left/right from one pivot item to another, the display always comes to rest locked on one whole pivot item.
- When the user flicks right beyond the last item, the display wraps to the first item.

By default, user input is first handled by the *PivotItem* if the input starts in the *PivotItem*, and by the *Header* if started in the header area. If the input is not handled by the *PivotItem*, it is passed to the *Pivot* control. If the *Pivot* control does not handle it, it is passed to the parent. Touch input is defined for the entire *Pivot* control, unless specified otherwise. Table 3-3 assumes that the *PivotItem* did not handle the input—that is, the input was passed to the *Pivot* control.

TABLE 3-3 *Pivot* Control Behavior

Gesture	Header/ <i>PivotItem</i>	Behavior
Tap, Double-Tap, or Press-and-Hold	<i>Header</i>	If the currently active item is tapped, nothing happens. If a different header is tapped, this moves to the tapped item.
	<i>PivotItem</i>	Handled by the content of the item.
Pan Right/Left	Both	Moves to the next item.
Pan Up/Down	<i>Header</i>	Nothing.
	<i>PivotItem</i>	Handled by the content of the item.
Flick Right/Left	Both	Moves to next item.
Flick Up/Down	<i>Header</i>	Nothing.
	<i>PivotItem</i>	Handled by the content of the item.
Pinch-and-Stretch	<i>Header</i>	Nothing
	<i>PivotItem</i>	Handled by the content of the item.

The following example (the *TestPivot* solution in the sample code) shows the barest minimum use of a *Pivot* control so that you can focus on the key features. The page has one *Pivot* control, which in turn has three *PivotItem* elements. Each *PivotItem* has a simple shape (*Ellipse*, *Rectangle*, or *Path*).

```
<controls:Pivot Title="CONTOSO">
  <controls:PivotItem Header="one">
    <Ellipse Width="100" Height="100" Fill="#339933"
      VerticalAlignment="Top" HorizontalAlignment="Left"
      Margin="{StaticResource PhoneHorizontalMargin}"/>
  </controls:PivotItem>
  <controls:PivotItem Header="two">
    <Rectangle Width="100" Height="100" Fill="#F09609"
      VerticalAlignment="Top" HorizontalAlignment="Left"
      Margin="{StaticResource PhoneHorizontalMargin}"/>
  </controls:PivotItem>
  <controls:PivotItem Header="three">
    <Path Fill="#1BA1E2" Data="M 50,0 L 100,100 L 0,100 Z"
      Margin="{StaticResource PhoneHorizontalMargin}"/>
  </controls:PivotItem>
</controls:Pivot>
```

The three *PivotItems* are shown in Figure 3-2.

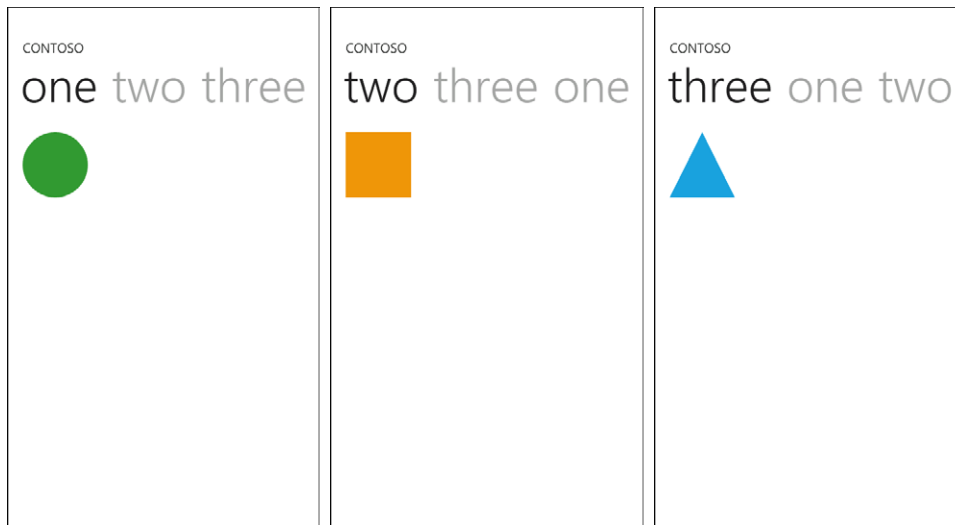


FIGURE 3-2 A basic pivot application with three pivot items.

Note that the standard text styles all use a left and right margin of 12 pixels, defined as the *Phone HorizontalMargin* resource. (That's why the shapes in this example are set to this same margin.) You can also set a background image on a *Pivot*. This should be between 480x696 and 480x800 in size, depending on your configuration of the *SystemTray* and *AppBar* (see Table 2-1, in Chapter 2). If you supply a different size, the image will be scaled to one screen, which does not change as the user flicks from one pivot item to another.

```
<controls:Pivot Title="My Pivot App">
  <controls:Pivot.Background>
    <ImageBrush ImageSource="PivotBackground.jpg"/>
  </controls:Pivot.Background>
```

It is possible to create a page that has more than one *Pivot* control on it; however, this is a technique that you should use judiciously. No matter how big the emulator looks on your huge desktop monitor, never forget the extreme constraints of the real phone device screen. With that caveat, it is actually very simple to create a multi-*Pivot* page. No technical restriction was imposed here, because it was felt that there were legitimate scenarios in which this might be useful. It comes under the heading of “things we let the developer do, even though he could shoot himself in the foot with it if he’s not very careful.”

Figure 3-3 illustrates such an application (the *TwoPivots* solution in the sample code). There are two *Pivot* controls on the main page, each with two *PivotItem* elements. The top *Pivot* displays simple lists of strings (people names and city names). The first *PivotItem* in the bottom *Pivot* displays a “details” page of text for the currently selected item in the people list in the top *Pivot*. The second *PivotItem* in the bottom *Pivot* displays a collection of photos associated with the selected person. The user can interact with each *Pivot* independently.

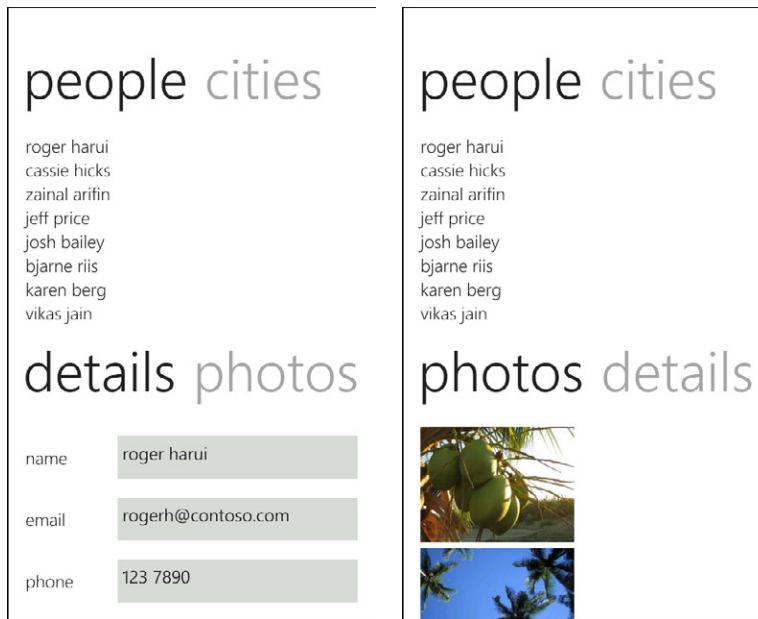


FIGURE 3-3 A page with two linked *Pivot* controls.

The item-level viewmodel is a simple *PersonViewModel* type that represents a person, with properties exposed for ID, Name, Email, Phone, and City. A set of these items is created in the *LoadData* method. In addition to the simple string fields, each *PersonViewModel* has a collection of photos, which are identified by a string path value.

```
List<string> photos = new List<string>();
photos.Add("/Images/Coconuts.jpg");
photos.Add("/Images/Palms.jpg");
photos.Add("/Images/Sea.jpg");
People.Add(new PersonViewModel()
{
    ID = 1,
    Name = "roger harui",
    Email = "rogerh@contoso.com",
    Phone = "123 7890",
    City = "Seattle",
    Photos = photos
});
```

... etc for the other *PersonViewModel* items.

From the XAML, you can see that the default *Grid* is replaced with a *StackPanel*. This is done to simplify positioning of the child elements. Then, two *Pivot* controls are stacked up, one on top of the other, each with its own collection of *PivotItem* elements. The top *Pivot* has its *ItemsSource* set to the collection of *PersonViewModel* items in the *MainViewModel*, and the two *PivotItem* elements each have a simple *ListBox* with data bound to the *Name* and *City* of the respective item. The bottom *PivotItem* elements bind to the *Name*, *Pivot*, *Phone*, and *Photos* collection items.

```

<StackPanel x:Name="LayoutRoot" Background="Transparent">

    <controls:Pivot Height="380">
        <controls:PivotItem Header="people">
            <ListBox
                x:Name="PeopleListBox" ItemsSource="{Binding People}"
                SelectionChanged="PeopleListBox_SelectionChanged">
                <ListBox.ItemTemplate>
                    <DataTemplate>
                        <TextBlock Text="{Binding Name}"/>
                    </DataTemplate>
                </ListBox.ItemTemplate>
            </ListBox>
        </controls:PivotItem>

        <controls:PivotItem Header="cities">
            <ListBox
                x:Name="CitiesListBox" ItemsSource="{Binding People}">
                <ListBox.ItemTemplate>
                    <DataTemplate>
                        <TextBlock Text="{Binding City}"/>
                    </DataTemplate>
                </ListBox.ItemTemplate>
            </ListBox>
        </controls:PivotItem>
    </controls:Pivot>

    <controls:Pivot>
        <controls:PivotItem Header="details" x:Name="DetailsPivot">
            <Grid>
                <TextBlock Text="name " />
                <TextBox Text="{Binding Name}"/>
                <TextBlock Text="email " />
                <TextBox Text="{Binding Email}" />
                <TextBlock Text="phone " />
                <TextBox Text="{Binding Phone}" />
            </Grid>
        </controls:PivotItem>
        <controls:PivotItem Header="photos" >
            <ListBox x:Name="PhotosList">
                <ListBox.ItemTemplate>
                    <DataTemplate>
                        <StackPanel>
                            <Image Source="{Binding}" Height="150"/>
                            <Grid Height="8"/>
                        </StackPanel>
                    </DataTemplate>
                </ListBox.ItemTemplate>
            </ListBox>
        </controls:PivotItem>
    </controls:Pivot>

</StackPanel>

```

To ensure that the bottom *Pivot* correctly populates its elements, the *SelectionChanged* handler on the *PeopleListBox* sets the current item as the *DataContext* for the “details” *PivotItem* in the bottom *Pivot*, and sets the *ItemsSource* of the photos *ListBox* to the *Photos* collection of that item.

```
private void PeopleListBox_SelectionChanged(  
    object sender, SelectionChangedEventArgs e)  
{  
    PersonViewModel pe = (PersonViewModel)((ListBox)sender).SelectedItem;  
    DetailsPivot.DataContext = pe;  
    PhotosList.ItemsSource = pe.Photos;  
}
```

SDK Controls: *Panorama*

The *Panorama* control is essentially a wide horizontal canvas, which hosts arbitrary child controls. It is intended to be beautiful and engaging, and is primarily for use in high-visibility areas like the entry point page of an application. Figure 3-4 illustrates the logical user experience of an application with a *Panorama* that has three *PanoramaItem* children. You can see this behavior at work in the *TestPanorama* solution in the sample code.



FIGURE 3-4 The *Panorama* control provides an engaging user experience.

The *Panorama* is provided to support parts of the application that are inspired by user experiences such as:

- Casual browsing in a magazine, where the user is encouraged to explore content in a non-task-directed way.
- More data-dense information hubs for which the bigger logical real estate of a *Panorama* enables the user to achieve more targeted goals such as looking for a specific contact or game to play.

The *Panorama* contains the following attributes that distinguish it from a standard view:

- It is wider than the normal 800x480 screen window to facilitate smooth left/right panning, as opposed to the discrete jumps offered by the *Pivot* control. The width is variable, so it will be different from one *Panorama* to another.
- Specific use of background and foreground animation layers while panning. The layers pan at different speeds, similar to parallax (depth of field) effects but with a somewhat different physics model.

Table 3-4 summarizes the gestures and navigational effects supported in the *Panorama* control. Note that these all follow the standard Metro gesture rules.

TABLE 3-4 Panorama Control Behavior

Gesture	Description
horizontal pan	Draggable space is anywhere on the screen even if assigned as the hit area of a hosted control.
horizontal flick	Flick can span hosted controls and even (with sufficient velocity) section boundaries.
vertical pan or flick	Vertical scrolling is triggered according to the content of the <i>Panorama</i> , such as when the gesture falls over a list or grid that has more vertical items than will fit in the visible screen area.

If a *Panorama* hosts a vertically scrolling control, the *Panorama* only scrolls the hosted control contents plus the control label, not the *Panorama* headers, other labels, or controls. The horizontal *Panorama* canvas therefore remains in focus at all times around the edges. It never scrolls up or down, only the list itself moves, cropping under the *Panorama* header.

The *Panorama* never scrolls both vertically and horizontally at the same time. There is no diagonal motion; rather, the *Panorama* locks to either vertical or horizontal as soon as it computes the direction, angle, and location span of the gesture. The baseline UX is that the *Panorama* itself scrolls horizontally, while hosted list controls scroll vertically. This can be made more complex if the *Panorama* hosts controls such as a horizontal *WrapPanel*, which can scroll horizontally within the broader *Panorama* horizontal scrolling.

A *Panorama* is made up of four layers, each with its own animation logic, as described in Table 3-5.

TABLE 3-5 Panorama Animation Logic

Layer	Description	Animation Rules
Background Image	<ul style="list-style-type: none"> ■ Optional. Spans the entire <i>Panorama</i>. Must be at least 480 pixels wide, and can be much wider—up to 1024 is recommended. Full bleed top to bottom. ■ Has a ~30 percent transparent black or white opacity filter applied (depending on the active theme), to aid in text legibility. 	The rate of motion relative to the panning gesture is determined by the total width of the (top) content layer compared to the width of the background art. In other words, the narrower the background, the slower the motion across it. When wrapping end-to-end, it animates completely off and then back onto the visible area.
Panorama Title	<ul style="list-style-type: none"> ■ Optional. This is the name of the entire <i>Panorama</i> screen, which spans all sections of the <i>Panorama</i> (via motion). 	Left-aligned, and stretched across the full <i>Panorama</i> width, such that some part of it is always visible, regardless of how many <i>Panoramaltem</i> child items there are. If there are very few items (perhaps only one), then the <i>Title</i> will extend beyond the item(s), as they wrap beneath it. The <i>Title</i> tends to move slowly in relation to the top content layer, but faster than the Background Image. It animates completely off and then back onto the visible area when wrapping end-to-end.
<i>Panoramaltem</i> Header	<ul style="list-style-type: none"> ■ Optional, on a per-item basis. Spans the individual section (via motion). 	Left-aligned across the item width. The <i>Header</i> animates in sync with the item content, and completely off when user navigates to a new item.
Content	<ul style="list-style-type: none"> ■ Optional, on a per-item basis. 	The rate of motion matches finger drag as closely as possible.

The standard *SystemTray* would interfere with the immersive parallax effect of the *Panorama*, so this is not displayed on a *Panorama* in Windows Phone 7. In version 7.1, The *SystemTray* exposes an *Opacity* property, so the guidance here is to keep it visible but set its *Opacity* to 0 so that the user can see the critical *SystemTray* information while minimizing interference with the *Panorama*. It is possible to use an Application Bar on a *Panorama*, but the effect is suboptimal, so this is discouraged in version 7 projects. This guideline is consistent with other fullscreen views such as “video now playing,” browser, or maps. If you must show an Application Bar on a *Panorama*, you should set its *Opacity* to about 50 percent. Also, version 7.1 introduces a new Application Bar minimized mode, which you can use instead. Notifications and other *SystemTray* activity such as low battery are gracefully unfolded as they arise, but they are then put away again so as not to interfere with the layer motion effects (so long as the *SystemTray* is set to visible).



Note There is no support for a landscape mode for *Panorama* controls, although this might be introduced in a later version of Windows Phone.

Individual *Panorama* designers can choose to include arbitrary items and controls as best fit the needs of the user in that context. They can also put their items in any order they see fit. That said, you’re strongly encouraged to avoid controls that need horizontal gestures—such as *ToggleSwitch*, *Slider*, *WebBrowser*, or *Map* controls—because this will result in gesture conflicts with the underlying *Panorama*.

To provide the rich “magazine cover” feel in the panoramas, designers/developers are encouraged to use full bleed background images behind all of the other title and content layers. However, if not chosen well, this art could become an unwelcome distraction. Here are some guidelines to keep in mind for background images in panoramas.

- **You don’t always need to use it.** If the content doesn’t suggest appropriate art, then don’t simply insert random images there; background images should be contextually appropriate. The People Hub is a good example of this, where a neutral background that respects the phone’s color theme is the most appropriate choice.
- **Branding can override content.** It is reasonable to use colors or graphics that are brand appropriate, but that don’t actually represent specific content. The Games Hub is a good example of this, employing the Xbox color scheme.
- **Photographic backgrounds work well.** If your *Panorama* is all about music, then by all means use band images; if it is all about the photo experience then use photos from the user’s collection. Illustrations can *potentially* work, but photos look good.
- **Don’t include embedded text and logos.** Because we have so many typographic and iconic elements overlaying the backgrounds, all moving at different speeds, any text or icons or logos embedded in the background image can be visually overwhelming.
- **Use dark, soft, and low-contrast images.** Darker images with soft edges and lower contrast will generally work better as panorama backgrounds than the opposite.
- **Avoid composite images.** A background image that is made up from smaller images will tend to look odd as the parallax effect forces the edges to get out of sync with the content layers in the front. On the other hand, it is common to use *Image* child controls—for example, album art in a *ListBox*—in a *PanoramaItem*.

You set up a *Panorama* in a very similar way to a *Pivot*, but it behaves more like a scrolling canvas. In addition, the title stretches across the width of the panorama space (and therefore uses a bigger font size). From a user’s perspective, it behaves as if the phone screen is panning across a wide background, bringing each item into view. As with *Pivot*, the *Panorama* wraps in a loop. However, as the user flicks left/right, while the display will always come to rest with one *PanoramaItem* aligned to the left, the next item will be partially visible. The following XAML defines a *Panorama* set up with the same content as the earlier *Pivot* example. You can see this at work in the *SimplePano* solution in the sample code.

```
<controls:Panorama Title="My Panorama App">
  <controls:PanoramaItem Header="One">
    <Ellipse
      Width="100" Height="100" Fill="#339933"
      VerticalAlignment="Top" HorizontalAlignment="Left"
      Margin="{StaticResource PhoneHorizontalMargin}"/>
  </controls:PanoramaItem>
  <controls:PanoramaItem Header="Two">
    <Rectangle
      Width="100" Height="100" Fill="#F09609"
      VerticalAlignment="Top" HorizontalAlignment="Left"
```

```

        Margin="{StaticResource PhoneHorizontalMargin}"/>
</controls:PanoramaItem>
<controls:PanoramaItem Header="Three">
    <Path Fill="#1BA1E2" Data="M 50,0 L 100,100 L 0,100 Z"
        Margin="{StaticResource PhoneHorizontalMargin}"/>
</controls:PanoramaItem>
</controls:Panorama>

```

Compare Figure 3-5 with the screenshots of the earlier *Pivot* example to see the similarities and differences between a *Panorama* and a *Pivot* that are set up with the same content.

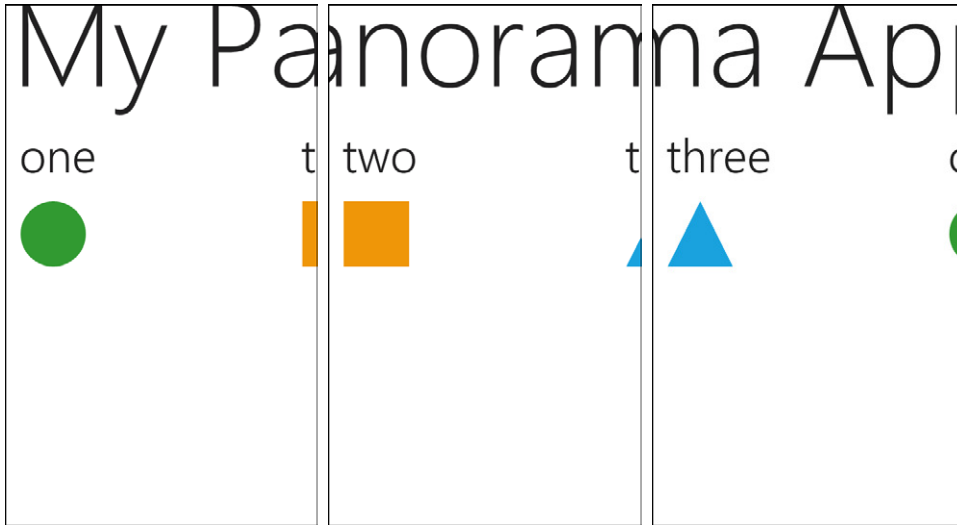


FIGURE 3-5 A panorama application with three items.

If you specify a background image for a *Panorama*, the image should be between 480x800 pixels and 1024x800 pixels to ensure good performance, minimal load time, and minimal scaling. The image is resized to 1024x800 (by stretching rather than scaling), although, curiously, the Microsoft Visual Studio Panorama Application template generates an image sized 1024x768. This size is wider than the screen, so part of the image scrolls into view only as the user flicks the *Panorama*. This scale is applied regardless of how many items you have in the *Panorama*.

```

<controls:Panorama Title="My Panorama App">
    <controls:Panorama.Background>
        <ImageBrush ImageSource="PanoramaBackground.jpg"/>
    </controls:Panorama.Background>

```

The *Panorama* control is intended to be used in a way that entices users to explore further in your application. There should be a lot of empty space, which is why the heading is quite large, deliberately leaving little leeway for cluttering the space with a lot of small items. The *Panorama* control itself is nominally 480x800, and the usable space on the individual items will be 432x618. The parallax effect of the *Panorama* is achieved by manipulating the foreground item content and the background image independently, within a logical view that is the size of all the items. So, if you have three items,

each 432x618 (totaling 1306 pixels wide), and a background image at 1024x800, then clearly the image is panned at a different rate from the individual items. Figure 3-6 illustrates the mechanism.

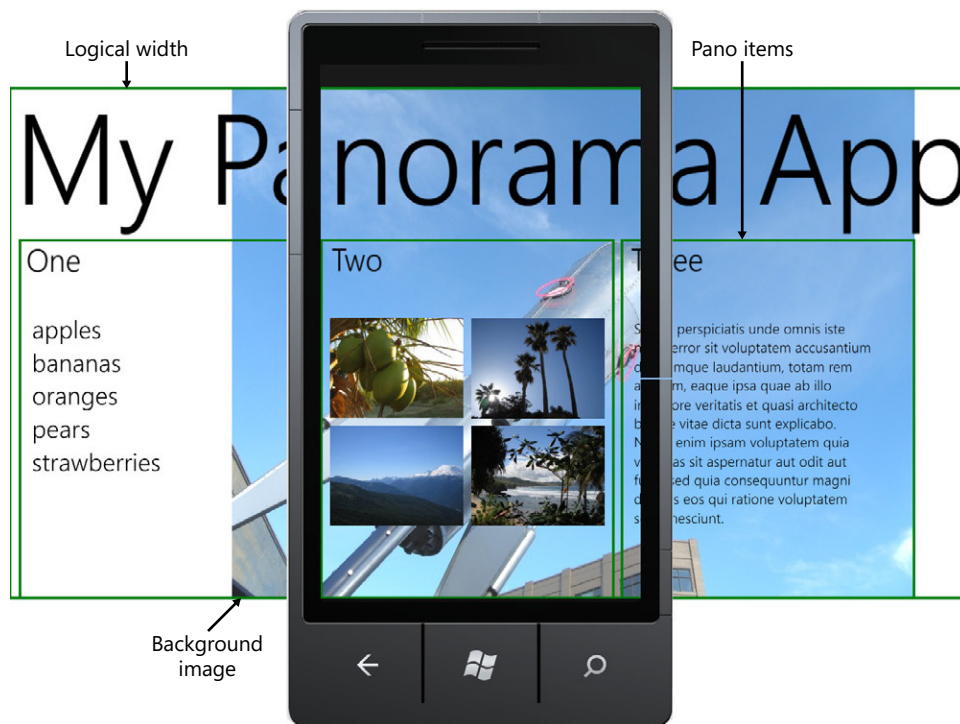


FIGURE 3-6 A breakdown of the panorama panning behavior.

You can also create a wide *Panorama* item, by setting the item *Orientation* property to *Horizontal*. The following example constructs an item with eight rectangles, each 100x100 pixels, stacked horizontally (in the *TestPanorama_WidelItem* solution in the sample code):

```
<phone:PhoneApplicationPage.Resources>
    <Style x:Key="RectangleStyle" TargetType="Rectangle">
        <Setter Property="Width" Value="100"/>
        <Setter Property="Height" Value="100"/>
        <Setter Property="VerticalAlignment" Value="Top"/>
        <Setter Property="HorizontalAlignment" Value="Left"/>
        <Setter Property="Fill" Value="#F09609"/>
        <Setter Property="Margin"
            Value="{StaticResource PhoneHorizontalMargin}"/>
    </Style>
</phone:PhoneApplicationPage.Resources>

<controls:Panorama Name="MyPano" Title="My Panorama App">
    <controls:Panorama.Background>
        <ImageBrush ImageSource="PanoramaBackground.jpg"/>
    </controls:Panorama.Background>
</controls:Panorama>
```

```

<controls:PanoramaItem Header="One" Name="Item1">
    <Ellipse
        Width="100" Height="100" Fill="#339933"
        VerticalAlignment="Top" HorizontalAlignment="Left"
        Margin="{StaticResource PhoneHorizontalMargin}"/>
</controls:PanoramaItem>

<controls:PanoramaItem Header="Two" Name="Item2" Orientation="Horizontal">
    <StackPanel Orientation="Horizontal">
        <Rectangle Style="{StaticResource RectangleStyle}"/>
        <Rectangle Style="{StaticResource RectangleStyle}"/>
        <Rectangle Style="{StaticResource RectangleStyle}"/>
        <Rectangle Style="{StaticResource RectangleStyle}"/>
        <Rectangle Style="{StaticResource RectangleStyle}"/>
        <Rectangle Style="{StaticResource RectangleStyle}"/>
        <Rectangle Style="{StaticResource RectangleStyle}"/>
        <Rectangle Style="{StaticResource RectangleStyle}"/>
    </StackPanel>
</controls:PanoramaItem>

<controls:PanoramaItem Header="Three" Name="Item3">
    <Path Fill="#1BA1E2" Data="M 50,0 L 100,100 L 0,100 Z"
        Margin="{StaticResource PhoneHorizontalMargin}"/>
</controls:PanoramaItem>
</controls:Panorama>

```

Figure 3-7 shows the result; the user will pan a longer distance across the second item. The background image, as always, pans at a different rate to cover the full logical width.



FIGURE 3-7 A wide horizontal *PanoramaItem*.

When the user navigates away and then back to a page with a *Panorama*, the *Panorama* will be positioned on the item she was last viewing, with the *Title* aligned to start with that item. You can see this behavior in action, using the standard pictures hub on the phone:

1. Open the pictures hub, and note that the page title and background are aligned at the left, and the default item is the first item (with the menu).
2. Pan across to the “what’s new” item, and note that the page title and background are offset.
3. Tap Start and then tap Back. Note that you return to the same “what’s new” item, but that the page title and background are reset to align to the left.

The default item is normally the first item, but you can change this behavior. The *Panorama* control exposes a *DefaultItem* property. This can be used to get or set the item that is displayed by default when you navigate to the *Panorama*. The default value is 0—that is, the first *PanoramaItem* in the collection—but as shown in the code snippet that follows, you can set it to a different value, if that makes sense in your application. Figure 3-8 illustrates how the *DefaultItem* is now the second item in the collection.

```
MyPano.DefaultItem = MyPano.Items[1];
```



FIGURE 3-8 You can specify a value for the *DefaultItem* property of the *Panorama* control.

You are strongly encouraged to follow the Metro guidelines. This includes adopting the fonts and styles used in the panorama *Title* and item *Headers*. If you explicitly set a *FontSize* for either of these, it will be ignored. However, it is possible to retemplate them, instead.

The following code provides a custom *DataTemplate* for the panorama *Title*, using a stacked *Image* and *TextBlock*, with custom text attributes:

```
<controls:Panorama Name="MyPano" Title="My Panorama App">
  <controls:Panorama.TitleTemplate>
    <DataTemplate>
      <StackPanel Orientation="Horizontal" Margin="12,80,0,0">
        <Image
          Source="/Images/rocket-logo.png" Height="190" Width="190"/>
        <TextBlock
          Text="corporate app"
          FontSize="{StaticResource PhoneFontSizeHuge}"
          VerticalAlignment="Center"/>
      </StackPanel>
    </DataTemplate>
  </controls:Panorama.TitleTemplate>
  ... irrelevant code omitted for brevity.
</controls:Panorama>
```

Figure 3-9 shows the result (see the *TestPanorama_Template* solution in the sample code). The idea here is to build a custom *Title*—in this example, using a corporate logo—while still maintaining faith with Metro.

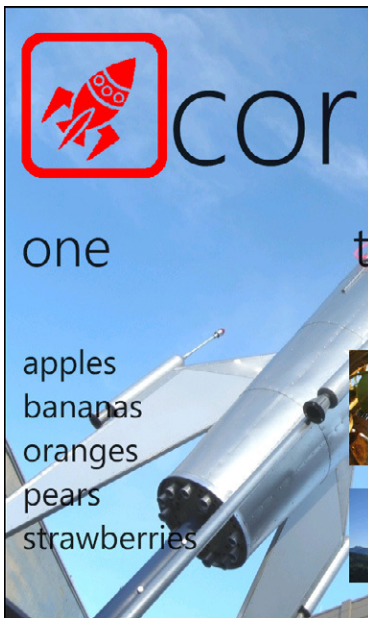


FIGURE 3-9 The retemplated panorama *Title*.

Toolkit Controls

While the standard Silverlight *ComboBox* is available, it is not Metro-styled, and is hidden in the Visual Studio toolbox. So, although you could add it manually to your XAML, you should probably avoid using it. A suitable alternative is the *ListPicker* control in the Silverlight Toolkit. This offers both the same developer experience and a very similar user experience.

The standard *ListBox* is suitable for short lists. If the list contains a lot of items, you should consider using the *LongListSelector* from the Toolkit, instead. Exactly how many is “a lot” depends on the complexity of each item, where you’re getting the data from, and any processing (*IValueConverter*, for example) that you do on the data before rendering in the UI. If you want group headers with a jump list (like the People Hub or a long Start menu), then you should consider using the *LongListSelector*. Even then, you should probably only use it if you have at least 45 or so items.

The *LoopingSelector* is a good representative example of a custom control from the Toolkit. When you set an alarm on the phone, using the standard Alarms application, you can spin a virtual dial for hours and minutes. The *LoopingSelector* in the Toolkit implements much the same thing. Figure 3-10 shows a simple example (the *TestLoopingSelector* solution in the sample code), which displays the alphabet in a *LoopingSelector*. The user can spin the list in an endless loop. When the list comes to rest on a letter, that letter is placed into the *TextBlock* at the top.



FIGURE 3-10 An example of a *LoopingSelector*.

The page defines a *TextBox* and a *LoopingSelector*, as follows:

```
<Grid x:Name="ContentPanel">
  <Grid.RowDefinitions>
    <RowDefinition Height="Auto"/>
```



```

        <RowDefinition Height="20"/>
        <RowDefinition Height="480"/>
    </Grid.RowDefinitions>
    <TextBox
        Grid.Row="0" Name="currentAlpha"
        Height="150" Width="150" Text="*"
        FontSize="{StaticResource PhoneFontSizeExtraExtraLarge}"
        Foreground="{StaticResource PhoneAccentBrush}"/>
    <toolkit:LoopingSelector
        Grid.Row="2" x:Name="selector"
        ItemMargin="3" ItemSize="74,74"
        FontSize="{StaticResource PhoneFontSizeExtraLarge}"/>
</Grid>

```

You sink the *SelectionChanged* event such that when the user selects an item from the *LoopingSelector*, you copy the value into the *TextBox*.

```

public partial class MainPage : PhoneApplicationPage
{
    AlphaLoopingDataSource data;

    public MainPage()
    {
        InitializeComponent();
        data = new AlphaLoopingDataSource();
        data.SelectionChanged +=
            data_SelectionChanged;
        this.selector.DataSource = data;
    }

    private void data_SelectionChanged(object sender, SelectionChangedEventArgs e)
    {
        this.currentAlpha.Text = (String)this.data.SelectedItem;
    }
}

```

For most collection controls—that is, controls derived from *ItemsControl*—such as the *ListBox* and *ListPicker*, you set the *ItemsSource* property to any collection type (including simple arrays, *List<T>*, *ObservableCollection<T>*, and so on). On the other hand, to use the *LoopingSelector* control, you must define a class that implements *ILoopingSelectorDataSource*, and set it as the *DataSource* for the *LoopingSelector* control. Here’s how that interface is declared:

```

public interface ILoopingSelectorDataSource
{
    object SelectedItem { get; set; }
    event EventHandler<SelectionChangedEventArgs> SelectionChanged;
    object GetNext(object relativeTo);
    object GetPrevious(object relativeTo);
}

```

In this example, a custom *AlphaLoopingDataSource* class is defined. This contains a list of strings corresponding to the letters of the alphabet (plus “*”). *GetNext* returns the item after the one supplied, or the first item (“*”) if you’re supplied with the last item in the list (“Z”). *GetPrevious* does the same in reverse. Next, *SelectedItem* returns the new selected item, and also raises a *SelectionChanged*

event—this is what the consuming application uses to determine that the user has changed the selection in the list. Note that the previously selected item is saved before setting the new value, so that you can use it to construct the *EventArgs* for the *SelectionChanged* event.

```
public class AlphaLoopingDataSource : ILoopingSelectorDataSource
{
    private static List<String> alphabet = new List<string>
    {
        "*", "A", "B", "C", "D", "E", "F", "G", "H",
        "I", "J", "K", "L", "M", "N", "O", "P", "Q",
        "R", "S", "T", "U", "V", "W", "X", "Y", "Z" };
    private static int count = alphabet.Count;
    private object selectedItem;

    public AlphaLoopingDataSource()
    {
        this.SelectedItem = alphabet[0];
    }

    public object GetNext(object relativeTo)
    {
        int currentIndex = alphabet.IndexOf((String)relativeTo);
        return alphabet[(currentIndex + 1) % count];
    }

    public object GetPrevious(object relativeTo)
    {
        int currentIndex = alphabet.IndexOf((String)relativeTo);
        return alphabet[(currentIndex - 1 + count) % count];
    }

    public object SelectedItem
    {
        get
        {
            return this.selectedItem;
        }
        set
        {
            if (this.selectedItem != value)
            {
                object previousSelectedItem = this.selectedItem;
                this.selectedItem = value;
                this.OnSelectionChanged(
                    previousSelectedItem, this.selectedItem);
            }
        }
    }

    public event EventHandler<SelectionChangedEventArgs> SelectionChanged;

    protected virtual void OnSelectionChanged(
        object oldSelectedItem, object newSelectedItem)
    {
        if (this.SelectionChanged != null)
        {
            this.SelectionChanged(this, new SelectionChangedEventArgs(
                new object[] { oldSelectedItem },
```

```

        new object[] { newSelectedItem }));
    }
}

```

Transient Panels

In addition to using controls on a page, you can also create transient panels—that is, popups, child windows, and other visual elements that are transient in nature, can take up the whole screen (but typically merely obscure part of it), and can all be dismissed in a consistent manner.

First, why is there no *Dialog* class in Windows Phone, or indeed in Silverlight? The main reason for this is that Silverlight was designed originally for use in web browsers, and in that context responsiveness is very important. Much of Silverlight is fundamentally asynchronous. This becomes even more important on the Phone, where it is critical not to block the UI thread; therefore, asynchronous operation is the default. When developers think of traditional dialog boxes, they are generally thinking of modal dialog boxes. A modal dialog is anathema in the Silverlight/Windows Phone world because modal equals blocking. In Windows Phone, the use of dialog-like visuals is even more critical. Even if the dialog is not modal, there's a problem with real estate. The phone's small form factor simply doesn't accommodate the pattern by which an application shows the arbitrary dialog-like visuals that are so common in desktop applications.

That being said, there is one obvious modal dialog in Windows Phone: the *MessageBox*. This blocks until the user dismisses the dialog via one of its buttons, presses the hardware Back key, or switches away from the application. Again, we see that the Back key is used in a consistent manner to back out of a situation that the user doesn't want to be in. The system *MessageBox* can be used for prompts, warnings, informational dialogs, error reports, and the like. However, you should never call it inside the startup of your application or during any navigation sequences; otherwise, the system might terminate the application for being unresponsive. It supports arbitrary text and a restricted set of buttons. If the application needs something similar to a *MessageBox*, but with more scope for customizing the visual controls, there are at least three other choices:

- **Popup** A standard class used for hosting arbitrary content, typically composed in custom *UserControls*.
- **System.Windows.Visibility** Any visual (including *UserControls*) that is part of the page, but with its *Visibility* toggled conditionally at runtime.
- **ChildWindow** A standard Silverlight class (in *System.Windows.Controls.dll*) that can be used in place of *UserControl* to develop custom visuals.

The following application (the *TestPopup* solution in the sample code) illustrates all three approaches, plus *MessageBox*, as shown in Figure 3-11. All approaches attempt to end up with a very similar experience, which is one that is similar to the standard *MessageBox*. This serves to highlight the differences in how you code each technique.

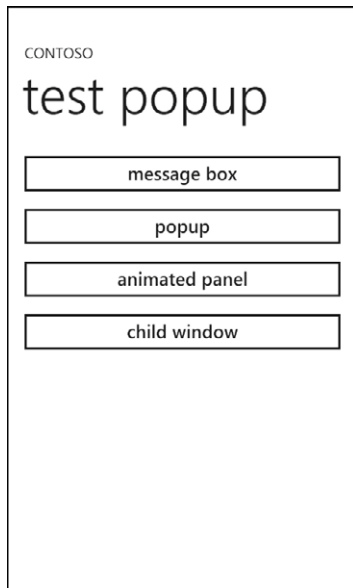


FIGURE 3-11 Using the *TestPopup* solution to test transient panels.

The first approach is the *MessageBox*. This is about as simple as it gets. In the *MainPage* code-behind, the *Click* handler for the “message box” button simply displays a *MessageBox*, as shown in Figure 3-12.

```
private void ShowMessageBox_Click(object sender, RoutedEventArgs e)
{
    MessageBox.Show("hello message box");
}
```



FIGURE 3-12 A standard, simple *MessageBox*.

The message box appears at the top of the screen, overlaying the previous page (which is also grayed-out), and is modal. It also has an elegant transition effect and plays a brief sound on entry (the sound was removed in version 7.1). To dismiss it, the user can tap OK (or whatever other buttons you initialize the *MessageBox* with), or he can press Back.

The second approach utilizes a *Popup*. This typically hosts a custom *UserControl*. So, you add a *UserControl* item to the project, and define its visual elements in XAML. To make this as close to the standard *MessageBox* as possible, simply provide a *TextBlock* and a *Button*. Obviously, if that's all you want, you would use the *MessageBox*, instead; the point of a *UserControl* is that you can put any custom visuals you want on it. However, for this illustration, it should be similar to the standard *MessageBox*.

```
<UserControl x:Class="TestPopup.PopupControl"
...

    <StackPanel
        Width="480" Height="226"
        Background="{StaticResource PhoneChromeBrush}" >
        <TextBlock
            Text="hello popup" Margin="22,110,0,0"
            Style="{StaticResource PhoneTextTitle3Style}"/>
        <Button
            x:Name="PopupClose" Content="close"
            Margin="12,0,0,0" Width="150" HorizontalAlignment="Left"/>
    </StackPanel>
</UserControl>
```

To use this control, declare a *Popup*, instantiate the control, and set it as the child of the *Popup*. Then show the *Popup* by setting *IsOpen* to *true*. The position of the *Popup* defaults to 0,0—that is, the upper-left corner of the parent page. In this example, that is exactly what you want, because that is what most closely resembles the standard *MessageBox*.

There are two ways to close this transient visual: the obvious technique is to set *IsOpen* to *false* in the *Click* handler for the button. To make the user experience consistent, ensure that the hardware Back button also dismisses the *Popup*. To do this, override the *OnBackKeyPress* for the page, and if the *Popup* is open, set *IsOpen* to *false*. It is important to be careful when handling the Back key, and to ensure that the behavior is always consistent with the standard behavior. Dismissing a modal dialog is one case for which it is legitimate to handle this key. It is absolutely not appropriate to handle the Back key to prevent the user backing out of an application.

```
private Popup p = new Popup();
private void ShowPopup_Click(object sender, RoutedEventArgs e)
{
    if (p.Child == null)
    {
        PopupControl pup = new PopupControl();
        pup.PopupClose.Click += new RoutedEventHandler(PopupClose_Click);
        p.Child = pup;
    }
    p.IsOpen = true;
}
```

```

private void PopupClose_Click(object sender, RoutedEventArgs e)
{
    if (p != null)
    {
        p.IsOpen = false;
    }
}

protected override void OnBackKeyPress(CancelEventArgs e)
{
    if (p != null && p.IsOpen)
    {
        p.IsOpen = false;
        e.Cancel = true;
    }
}

```

The third approach, a transient window with its *Visibility* property toggled, is illustrated in Figure 3-13. Again, this could be a *UserControl*, or it could simply be some parent control (*Grid*, *Panel*, and so forth) on the page itself. In this example, you want to make it a child of the *Grid* on the *MainPage*. One of the reasons a developer might want to take this approach instead of using a *MessageBox* is that this kind of visual can easily be animated. To implement this, you need a couple of *Storyboards*: one for animating the visual as it opens, and the other for animating it as it closes. In both cases, the target property is *RotationX*, which means that the visual will be rotated on its X axis.

```

<phone:PhoneApplicationPage.Resources>
    <Storyboard x:Name="AnimatedPanelOpenStory">
        <DoubleAnimation
            Storyboard.TargetName="AnimatedPanelPlaneProjection"
            Storyboard.TargetProperty="RotationX"
            From="-80" To="0" Duration="0:0:0.4"/>
    </Storyboard>
    <Storyboard x:Name="AnimatedPanelCloseStory">
        <DoubleAnimation
            Storyboard.TargetName="AnimatedPanelPlaneProjection"
            Storyboard.TargetProperty="RotationX"
            From="0" To="-80" Duration="0:0:0.4"/>
    </Storyboard>
</phone:PhoneApplicationPage.Resources>

<Grid x:Name="LayoutRoot" Background="Transparent">
    ...
    <StackPanel
        x:Name="AnimatedPanel" Visibility="Collapsed"
        Width="480" Height="226" VerticalAlignment="Top"
        Background="{StaticResource PhoneChromeBrush}">
        <StackPanel.Projection>
            <PlaneProjection x:Name="AnimatedPanelPlaneProjection" />
        </StackPanel.Projection>
        <TextBlock
            Text="hello animated panel" Margin="22,110,0,0"
            Style="{StaticResource PhoneTextTitle3Style}"/>
        <Button
            x:Name="AnimatedClose" Content="close"
            Margin="12,0,0,0" Width="250"

```

```

        HorizontalAlignment="Left"/>
    </StackPanel>
</Grid>

```

The actual declaration of the *StackPanel* must be at the bottom of the XAML so that it sits above everything else in the Z-order. Note that you should set the *VerticalAlignment* to position the visual at the top of the page, and set the background to the same brush that the *MessageBox* uses. This is represented in user-code by the *PhoneChromeBrush* resource. Also note that the control's *Visibility* is set to *Collapsed* initially.

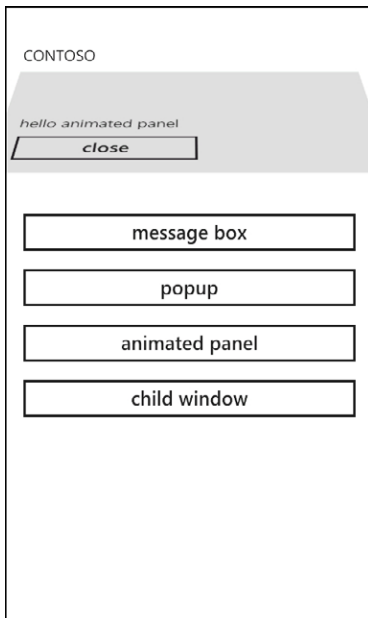


FIGURE 3-13 Using up an animated panel to closely mirror the standard message box behavior.

To show this control, first set its *Visibility* to *Visible*, hook up the “close” button *Click* event, and then start the “opening” animation. When the user taps the “close” button, you hook up the *Completed* event on the “closing” animation and start that animation. Only when the animation has completed do you make the control *Collapsed* again.

```

private bool isAnimatedConnected;
private bool isAnimatedOpen;
private void ShowAnimatedPanel_Click(object sender, RoutedEventArgs e)
{
    AnimatedPanel.Visibility = Visibility.Visible;
    AnimatedClose.Click +=
        new RoutedEventHandler(AnimatedClose_Click);
    AnimatedPanelCloseStory.Completed +=
        new EventHandler(popupCloseStory_Completed);
    AnimatedPanelOpenStory.Stop();
    AnimatedPanelOpenStory.Begin();
    isAnimatedOpen = true;
}

```

```

private void AnimatedClose_Click(object sender, RoutedEventArgs e)
{
    ClosePanel();
}

private void ClosePanel()
{
    AnimatedPanelOpenStory.Stop();
    AnimatedPanelCloseStory.Begin();
    isAnimatedOpen = false;
}

private void popupCloseStory_Completed(object sender, EventArgs e)
{
    AnimatedPanel.Visibility = Visibility.Collapsed;
}

```

To replicate the Back button behavior, you must ensure that the same close method is called in the override of *OnBackKeyPress*.

```

protected override void OnBackKeyPress(CancelEventArgs e)
{
    ...

    if (isAnimatedOpen)
    {
        ClosePanel();
        e.Cancel = true;
    }
}

```

The fourth and final approach is the *ChildWindow*. This is a class in the Silverlight Toolkit. Recall that this is not part of the Windows Phone SDK; rather, it is a separate download. Having installed the Toolkit, you need to add a reference to the *System.Windows.Controls.dll* assembly, which will typically be in a location such as %ProgramFiles%\Microsoft SDKs\Silverlight\v3.0\Libraries\Client\System.Windows.Controls.dll.

Next, add a namespace declaration for this in the XAML as shown in the following:

```
xmlns:sltk="clr-namespace:System.Windows.Controls;assembly=System.Windows.Controls"
```

Then, add another *UserControl* to the project, but this time, change the definition to *ChildWindow*. That is, change the following (where *TestPopup* is the name of the project, and *ChildControl* is the name of the custom *UserControl*).

```
<UserControl x:Class="TestPopup.ChildControl"
```

to this:

```
<sltk:ChildWindow x:Class="TestPopup.ChildControl"
```


The whole of the XAML file is listed in the code that follows. Note that the *Width*, *Height*, *Background* and *Margin* of the control itself is set, rather than the child *StackPanel*. The *StackPanel* offers the same *TextBlock* and *Button*, but this time, you handle the button *Click* event in the control itself rather than in the parent page.

```
<sltk:ChildWindow x:Class="TestPopup.ChildControl"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    xmlns:sltk="clr-namespace:System.Windows.Controls;assembly=System.Windows.Controls"
    mc:Ignorable="d"
    FontFamily="{StaticResource PhoneFontFamilyNormal}"
    FontSize="{StaticResource PhoneFontSizeNormal}"
    Foreground="{StaticResource PhoneForegroundBrush}"
    d:DesignHeight="480" d:DesignWidth="480"
    Width="480" Height="226"
    Background="{StaticResource PhoneChromeBrush}"
    Margin="0,-576,0,0"
>

    <StackPanel>
        <TextBlock
            Text="hello childwindow" Margin="22,80,0,0"
            Style="{StaticResource PhoneTextTitle3Style}"/>
        <Button
            x:Name="ChildClose" Content="close" Click="ChildClose_Click"
            Margin="12,0,0,0" Width="150" HorizontalAlignment="Left"/>
    </StackPanel>

</sltk:ChildWindow>
```

The code-behind also defines the inheritance to be *ChildWindow*, not *UserControl*. The only other thing we need do is to set the control's *DialogResult* to true when the user presses the "close" button:

```
public partial class ChildControl : ChildWindow
{
    public ChildControl()
    {
        InitializeComponent();
    }

    private void ChildClose_Click(object sender, RoutedEventArgs e)
    {
        DialogResult = true;
    }
}
```

Back in the *MainPage* code-behind, you display this control by instantiating it and calling its *Show* method. You also set a flag so that you can determine whether or not it is open. Thus, you need the following in the *OnBackKeyPress* override:

```
private ChildControl cc = new ChildControl();
private bool isChildOpen;
private void ShowChildWindow_Click(object sender, RoutedEventArgs e)
{
    cc.Show();
    isChildOpen = true;
}

protected override void OnBackKeyPress(CancelEventArgs e)
{
    ...

    if (isChildOpen)
    {
        cc.DialogResult = false;
        isChildOpen = false;
        e.Cancel = true;
    }
}
```

For all three types of custom transient visual, instead of overriding *OnBackKeyPress*, an alternative approach is to define a handler for the *BackKeyPress* event on the page.

```
<phone:PhoneApplicationPage
...
    BackKeyPress="PhoneApplicationPage_BackKeyPress">
```

Adding this to the XAML in the designer will add a stub for the handler in the code-behind (if you use Tab to invoke auto-complete; otherwise, you need to right-click and select *Navigate To Event Handler*). This could be implemented to execute exactly the same code as the previous override of *OnBackKeyPress*. If the application were to implement both methods (which would be unnecessary), the *OnBackKeyPress* is called first, and then the event handler. Note that neither method is ever called when the user presses Back to dismiss a simple *MessageBox*.

That's it; four ways to offer dialog-like behavior in an application. The two principles to keep in mind here are:

- Use a transient visual, employing whichever technique makes sense for the given scenario, instead of using a page. This avoids polluting the page backstack with visuals to which the user will not expect to return.
- Maintain consistency across all transient visuals by overriding *OnBackKeyPress* or handling the *BackKeyPress* event so that the user can always dismiss a "dialog" by using the Back key.

Note that while you can achieve *MessageBox*-like behavior up to a point, there are two key reasons that make *MessageBox* so different from the other approaches:

- *MessageBox* blocks the caller until the user makes a selection; all the other mechanisms are asynchronous.
- *MessageBox* offers very limited scope for customization, unlike all the other approaches.

The *MessageBox* also implements an *interaction guard*—that is, it grays out the rest of the application UI and suppresses touch input. You could implement this in any of the custom approaches described earlier by simply overlaying a full-screen rectangle filled with the *PhoneSemitransparent Brush*. Also note that in an XNA application, there's an alternative: *BeginShowMessageBox*. This is more configurable than the Silverlight *MessageBox*, although it does not block the way *MessageBox* does, and it should not be used in a non-XNA application unless you're already pulling in some XNA types for other reasons. Integration between Silverlight and XNA is streamlined in version 7.1, so it becomes more appropriate to use this approach in version 7.1 projects. The following code shows how this is used (demonstrated in the *TestXnaMBox* solution in the sample code):

```
private void XnaMessageBox_Click(object sender, RoutedEventArgs e)
{
    List<string> buttons = new List<string>();
    buttons.Add("foo");
    buttons.Add("bar");

    int? retVal = null;

    Guide.BeginShowMessageBox(
        "xna",                // Title.
        "hello world",        // Text.
        buttons,              // Buttons.
        1,                    // Index of the button with the focus.
        MessageBoxIcon.Warning, // Icon.
        result =>              // Async callback invoked on closing the message box.
        {
            retVal = Guide.EndShowMessageBox(result);
            Debug.WriteLine("return value = {0}", retVal);
        },
        123);                 // Arbitrary payload for this message.
}
```

Summary

This chapter examined the developer's choices for standard controls, including the built-in application platform controls, the SDK controls, and the Silverlight Toolkit controls. These are all conformant to the Metro design and usability guidelines, and should always be preferred over custom controls, wherever possible. In particular, you can use the *Pivot* and *Panorama* controls to build compelling, user-friendly applications that can seem to escape the bounds of the physical platform. Most of the UI in your application will be part of a page, but you can also use transient panels where those make sense, and you can choose from a range of different techniques when using them.

Data Binding and Layer Decoupling

At some point, almost any non-trivial application will present data to the user. Most modern programming frameworks support this in a variety of ways to make it easier, quicker, and more robust to render data in the UI. At the same time, these frameworks promote better engineering practices by cleanly separating the data from the UI, establishing standard mechanisms for connecting the data and UI in a loose fashion, and ensuring that components consuming the data are conveniently notified of any changes (either by the UI or from the underlying data source) so that the application can take appropriate action. This chapter examines the data binding support in Microsoft Silverlight for Windows Phone, the rationale for its existence, and the various ways that it supports both the functionality of mapping data and UI, and the engineering excellence of loose coupling between layers.

Life without Data Binding

A common scenario is to have some data (either originating in a back-end data repository, or computed dynamically in the application) that you want to display in the UI. It is also common to allow the user to modify the data in the UI and for any changes to be propagated back to the data repository. It is perfectly possible to do this without data binding. Indeed, the traditional programming model is to do just that. Figure 4-1 and the ensuing code present a simple example of a phone application that manually propagates data changes back and forth between the data source and the UI. This is the *NoDatabinding* solution in the sample code.

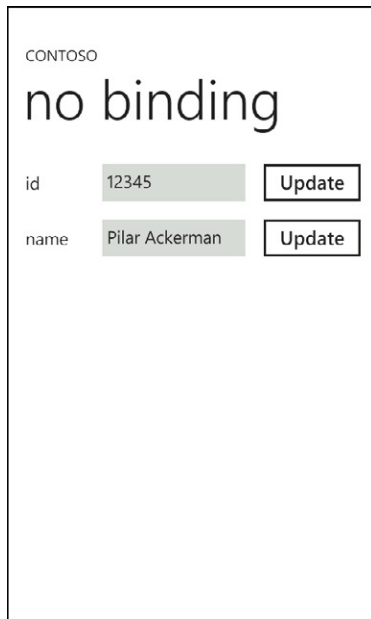


FIGURE 4-1 A simple application without data binding.

The XAML that follows is straightforward, offering two rows, each made up of a *TextBlock*, a *TextBox*, and a *Button*:

```
<TextBlock
    Text="id"
    FontSize="{StaticResource PhoneFontSizeMedium}"
    VerticalAlignment="Center"/>
<TextBox
    Name="viewID"
    FontSize="{StaticResource PhoneFontSizeMedium}"/>
<Button
    Height="Auto" Content="Update"
    Name="updateID" Click="updateID_Click" />

<TextBlock
    Text="name"
    FontSize="{StaticResource PhoneFontSizeMedium}"
    VerticalAlignment="Center"/>
<TextBox
    Name="viewName"
    FontSize="{StaticResource PhoneFontSizeMedium}"/>
<Button
    Height="Auto" Content="Update"
    Name="updateName" Click="updateName_Click" />
```

The data is represented in code by a trivial *Employee* class which exposes a couple of public fields.

```
public class Employee
{
    public string ID;
    public string Name;
}
```

In the *MainPage* code-behind, you initialize an *Employee* and manually set this data into the elements in the view. At runtime, the user can edit either of the *TextBox* controls. When she clicks the *updateID Button*, the code simulates a change originating at the underlying the data source and then propagates that change forward to the UI. Conversely, when she clicks the *updateName Button*, any changes they made to the *Name* data are propagated manually back to the data source. The *MessageBox* simply serves to verify that the data source was changed.

```
public partial class MainPage : PhoneApplicationPage
{
    private Employee data;

    public MainPage()
    {
        InitializeComponent();

        data = new Employee { ID = "12345", Name = "Pilar Ackerman" };
        viewID.Text = data.ID;
        viewName.Text = data.Name;
    }

    private void updateID_Click(object sender, RoutedEventArgs e)
    {
        data.ID = "98765";
        viewID.Text = data.ID;
    }

    private void updateName_Click(object sender, RoutedEventArgs e)
    {
        data.Name = viewName.Text;
        MessageBox.Show(data.Name);
    }
}
```

That's all very well, and for such an extremely simple data class, the work is not too onerous or difficult to maintain. However, with more complex data, and a higher volume of it, this manual back-and-forth data propagation will rapidly become a burden. It will also involve a lot of manual code, which inevitably increases the chance of introducing bugs. Furthermore, the data is very tightly coupled to the UI. If application requirements change over time that include modifications to the data model, this will require corresponding changes to the UI. Additionally, it will entail changes to all the code that's doing the manual value-change propagation. Similarly, even if only the UI changes, you will still need to change the propagation code. Such tight coupling makes the whole application very inflexible and fragile in the face of ongoing requirements changes.

Fortunately, Silverlight—and indeed, Windows Presentation Foundation (WPF)—includes support for automatically initializing the UI from backing data and for automatically propagating changes, in both directions. This support is called *data binding*.

Simple Data Binding and *INotifyPropertyChanged*

Data binding is supported as a standard feature in Silverlight. The goals of Silverlight data binding include:

- Enable a range of diverse data sources to be connected (web service calls, SQL queries, business objects, and so on). The underlying data sources are represented in the application code by a set of one or more Common Language Runtime (CLR) objects.
- Simplify the connection and synchronization of data so that the developer does not need to maintain manual value propagation code.
- Maintain the separation between the UI design and the application logic (and therefore between design tools such as Microsoft Expression Blend, and development tools such as Microsoft Visual Studio).

One way to think of data binding is as a pattern with which you can declare the relationship between your application's data and the UI that displays the data without hard-coding the specifics of how and when the data is propagated to the UI. This allows you to maximize the Separation of Concerns (SoC), ensuring that the UI code is as decoupled as possible from the business logic, and yet further decoupled from the underlying data source. Figure 4-2 illustrates the pattern.

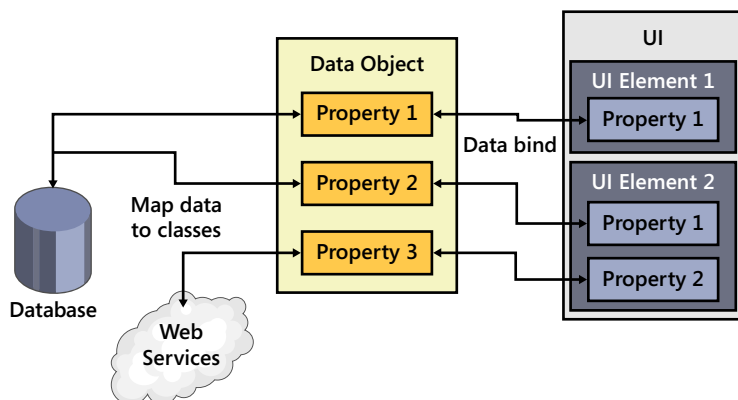


FIGURE 4-2 The data binding pattern.

In addition to separating concerns, the declarative relationship also takes advantage of change notifications. That is, when the value of the data changes in the underlying data source, the application does not need to render the change in the UI explicitly. Instead, it relies on the class that represents the data source raising a *change notification event*, which the Silverlight runtime picks up and

uses to propagate the change to the UI. The same happens in reverse; when the user changes the data interactively in the UI, that change is propagated back to the data source.

You'll adapt the previous application to use data binding. In this version (the *SimpleDataBinding* solution in the sample code), the two *TextBox* controls are each data-bound to properties of the data class. One *TextBox* is set to the *ID*; the other is set to the *Name*. First, the UI declarations in the XAML need to be updated to specify the binding for each *TextBox*. You do this by using the *{Binding}* syntax. In this example, the *ID* has a one-way binding, which means that the data is pulled from the data class into the UI only. Any changes made to the underlying data will be propagated to the UI. However, any changes made to the value in the UI will not be propagated back to the data source. On the other hand, the *Name* *TextBox* has a two-way binding. This means that changes are propagated in both directions. Note that *OneWay* mode is the default; thus it's unnecessary to specify it (the listing that follows only include it to emphasize that the two *TextBox* controls have different binding modes). Also note that the *TextBox* names are no longer required because you no longer access them directly in code. This also saves a little memory and initialization time.

```
<TextBox
    FontSize="{StaticResource PhoneFontSizeMedium}"
    Text="{Binding ID, Mode=OneWay}"/>

<TextBox
    FontSize="{StaticResource PhoneFontSizeMedium}"
    Text="{Binding Name, Mode=TwoWay}"/>
```

To connect the data object to the UI, you instantiate the data object class, and assign it to the *DataContext* of the *FrameworkElement* that you want to data-bind. You can specify an individual *FrameworkElement* such as a single control or some containing parent control. At its simplest, this *FrameworkElement* might even be the main page itself, as in this example. Setting the *DataContext* at the page level is a common strategy. All child elements of the page will inherit the same *DataContext*; however, it can also be overridden at any level if required.

Note that the *DataContext* *property* is typed as *object*, which is why you can assign an object of a custom type such as *Employee*—you can, of course, assign anything to an object. Assigning the *Employee* object to the *DataContext* of the page is how the data binding system resolves the references to *ID* and *Name* in the binding declarations of the individual elements, because these elements inherit the *DataContext* of the page. The target (UI element) of the binding can be any accessible property or element that is implemented as a *DependencyProperty* (the *DependencyProperty* mechanism is described in Chapter 2, “UI Core”). The source can be any public property or public field of any type.



Note One of the most common binding errors is to forget to make your properties public.

In the *Click* handler for the *updateID Button*, you merely display the value of the underlying data. Because one-way data binding is defined, this will not change, regardless of any changes the user makes in the UI for this field. On the other hand, the *Click* handler for the *updateName Button* first displays the current value of the underlying *Name*. If the user has made any changes in the UI, this will automatically be reflected in the underlying data. Once you have verified that any such change were indeed propagated (using a message box), you then go on to change the underlying data programmatically to an arbitrary value. The point of this is to propagate the change automatically to the UI.

```
public partial class MainPage : PhoneApplicationPage
{
    private Employee data;

    public MainPage()
    {
        InitializeComponent();
        data = new Employee { ID = "12345", Name = "Pilar Ackerman" };

        // viewID.Text = data.ID;
        // viewName.Text = data.Name;
        DataContext = data;
    }

    private void updateID_Click(object sender, RoutedEventArgs e)
    {
        // data.ID = viewID.Text;
        MessageBox.Show(data.ID);
    }

    private void updateName_Click(object sender, RoutedEventArgs e)
    {
        // viewName.Text = data.Name;
        MessageBox.Show(data.Name);
        data.Name = "David Pelton";
    }
}
```

If you want the system to propagate changes in data values for you, your data class needs to implement *INotifyPropertyChanged*. This defines one member: an event of type *PropertyChanged EventHandler*. For data binding to work, the public fields you had previously are no longer sufficient. They must be defined as public properties, not fields. You implement your property setters to raise this event, specifying by name the property that has changed. The invocation of the *PropertyChangedEventHandler* can also be usefully factored out to a custom method. This will be especially appropriate if there are many properties in the data class. Note that this is still a relatively simple design—you're supporting simple data binding, but the SoC could be better (see the MVVM section, later in this chapter). Also, although it looks like more code than before, it is very cookie-cutter code and there is nothing complex or risky about it. As your data becomes more complex, so you shall reap ever greater return on investment with the *INotifyPropertyChanged* approach.

```
public class Employee : INotifyPropertyChanged
{
    public event PropertyChangedEventHandler PropertyChanged;
```

```

private string id;
public string ID
{
    get { return id; }
    set
    {
        id = value;
        NotifyPropertyChanged("ID");
    }
}

private string name;
public string Name
{
    get { return name; }
    set
    {
        name = value;
        NotifyPropertyChanged("Name");
    }
}

private void NotifyPropertyChanged(string propertyName)
{
    PropertyChangedEventHandler handler = PropertyChanged;
    if (null != handler)
    {
        handler(this, new PropertyChangedEventArgs(propertyName));
    }
}
}

```

Notice also that the custom method declares a local *PropertyChangedEventHandler* variable instead of using the *PropertyChanged* field directly. Why is this? The reason is to make the code more robust in the face of multi-threaded calls. This code does not explicitly instantiate the *PropertyChanged* event; rather, this is done for you by the C# compiler. By the same token, removing event handler sinks is also done for you. When the last event sink is removed, the handler field becomes null, as a housekeeping strategy. Declaring a local variable doesn't protect against the field becoming null before or after you assign it—but it does protect against it being non-null before you assign it and then null after you assign it, but before you invoke it. Of course, “more robust” doesn't mean “perfect.” For a more comprehensive discussion of threading issues, refer to Chapter 14, “Go to Market.”

The ability to specify data binding in XAML confers significant benefits, but it is also possible to specify data binding in code. This might be appropriate if the binding is conditional upon some run-time behavior. For example, you could remove the *{Binding}* specifiers in your XAML and replace them with calls to *BindingOperations.SetBinding*.

```

public MainPage()
{
    InitializeComponent();
    data = new Employee { ID = "12345", Name = "Pilar Ackerman" };
    DataContext = data;
}

```

```

Binding b = new Binding("ID");
b.Mode = BindingMode.OneWay;
BindingOperations.SetBinding(viewID, TextBox.TextProperty, b);

b = new Binding("Name");
b.Mode = BindingMode.TwoWay;
BindingOperations.SetBinding(viewName, TextBox.TextProperty, b);
}

```

Data Binding Collections

It is very common to have collections of items to which you want to bind data. For example, multiple rows from a data set are commonly bound to some *ItemsControl* such as a *ListBox* or *ListPicker*. To bind to a collection, at a minimum, you need to do the following:

- Provide a data source that is a collection of some object type
- Use an *ItemsControl* (or derivative) element as the display container for your list, such as the *ListBox* control
- Set the *ItemsSource* property of the *ItemsControl* to the collection object

The example that follows (the *CollectionDataBinding* solution in the sample code) demonstrates a *ListBox* bound to a simple collection of strings, as shown in Figure 4-3.

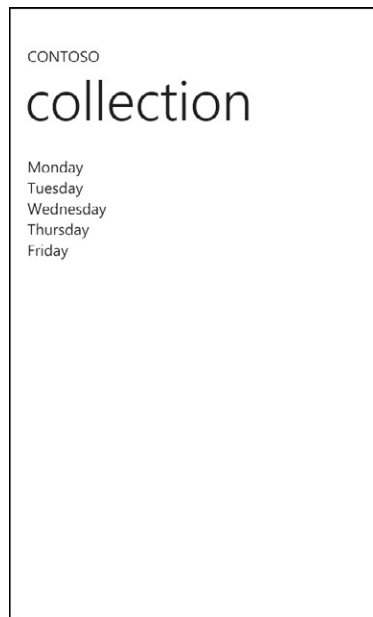


FIGURE 4-3 Data binding to a collection.

The XAML definition of the `ListBox` is very simple; apart from position and size, all you need to define is a name so that you can refer to it in code:

```
<StackPanel x:Name="ContentPanel" Margin="12,0,12,0">
    <ListBox
        x:Name="daysList" Width="300" Height="200"
        VerticalAlignment="Top" />
</StackPanel>
```

The code-behind is also very simple. There is a collection of strings in a `List<T>`, initialized in the `MainPage` constructor, and set as the `ItemsSource` of the `ListBox`:

```
public partial class MainPage : PhoneApplicationPage
{
    private List<string> myDays;

    public MainPage()
    {
        InitializeComponent();

        myDays = new List<string>();
        myDays.Add("Monday");
        myDays.Add("Tuesday");
        myDays.Add("Wednesday");
        myDays.Add("Thursday");
        myDays.Add("Friday");

        daysList.ItemsSource = myDays;
    }
}
```

Note that there is no need to assign the `DataContext` in this case, because you are explicitly assigning the collection data to the `ItemsSource`. Typically, you might do both, because typically, you would be data binding one or more collections (with explicitly assigned `ItemsSource` properties) and also individual items (which would rely on the `DataContext` to resolve their bindings). It is also possible to assign a more specific `DataContext` on a per-element basis (at any level in the visual tree). However, it is more common to set the `DataContext` at a page level, allowing each element in the page to inherit this, and then simply assign individual `ItemsSource` collections, as required.



Note Data binding large collections has a negative effect on performance because of all the housekeeping that the runtime's data binding framework does in the background. Chapter 14 discusses mitigation strategies for this.

Data Templates

The previous example works because each item in the collection is a simple string. By default, the *ListBox* will render a string representation of its items, calling *ToString* if necessary. Suppose, however, that your collection items were not simple strings, but items of a more complex data type. Even a trivially more complex type such as our simple *Employee* class would cause problems here.

```
public class Employee
{
    public string ID { get; set; }
    public string Name { get; set; }
}
```

Let's add a collection of *Employee* objects to our *MainPage*, and a second *ListBox*:

```
<StackPanel x:Name="ContentPanel" Margin="12,0,12,0">
    <ListBox
        x:Name="daysList" Width="300" Height="200"
        VerticalAlignment="Top" />
    <ListBox
        x:Name="employeesList" Width="300" Height="200"
        VerticalAlignment="Top" />
</StackPanel>

private List<Employee> myEmployees;

public MainPage()
{
    ... previously listed code omitted for brevity

    myEmployees = new List<Employee>();
    myEmployees.Add(new Employee { ID = "12345", Name = "Pilar Ackerman" });
    myEmployees.Add(new Employee { ID = "13344", Name = "David Pelton" });
    myEmployees.Add(new Employee { ID = "15566", Name = "Jay Hamlin" });
    myEmployees.Add(new Employee { ID = "17788", Name = "Lori Penor" });
    myEmployees.Add(new Employee { ID = "12299", Name = "Yun-Feng Peng" });

    employeesList.ItemsSource = myEmployees;
}
```

The result is shown in Figure 4-4.

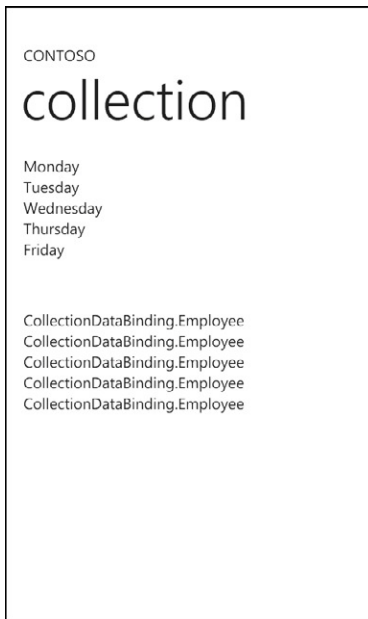


FIGURE 4-4 An example of collection items using the default *ToString* method.

Clearly, the *ListBox* is using the base object *ToString* to represent the *Employee*. There are two ways you could fix this:

- Provide an override of *ToString*, if all you want is a simple string
- Provide a data template, with which you can provide a more complex UI.

A *ToString* override is the minimum you could do—this would do the job, but it is very inflexible, because you still end up with just one single string for each item:

```
public override string ToString()
{
    return ID + " - " + Name;
}
```

The better approach is to provide a data template. This affords much greater control in formatting the UI. For example, you could have two columns—one for the *ID*, and one for the *Name*—and you could use different fonts, colors, styles, backgrounds, and so on for each column, for each row, and so forth. Figure 4-5 illustrates this approach. This is the *SimpleDataTemplate* solution in the sample code.

datatemplate

12345 *Pilar Ackerman*
 13344 *David Pelton*
 15566 *Jay Hamlin*
 17788 *Lori Penor*
 12299 *Yun-Feng Peng*

FIGURE 4-5 You can use a *DataTemplate* to control formatting.

The data template is defined in XAML, and assigned to the *ItemTemplate* property of the *ListBox*. In this example, the template is made up of a *Grid* that contains two *TextBlock* controls, each formatted slightly differently. These use the same *{Binding}* as before but don't need a *DataContext* because it is inherited from the current item in the list.

```
<StackPanel x:Name="ContentPanel" Margin="12,0,12,0">
  <ListBox
    Name="myItemsControl" VerticalAlignment="Top">
    <ItemsControl.ItemTemplate>
      <DataTemplate>
        <Grid>
          <Grid.ColumnDefinitions>
            <ColumnDefinition Width="100"/>
            <ColumnDefinition Width="*"/>
          </Grid.ColumnDefinitions>
          <TextBlock
            Grid.Column="0" Text="{Binding ID}"
            FontSize="{StaticResource PhoneFontSizeMedium}"
            FontWeight="Bold"/>
          <TextBlock
            Grid.Column="1" Text="{Binding Name}"
            FontSize="{StaticResource PhoneFontSizeMedium}"
            FontStyle="Italic"/>
        </Grid>
      </DataTemplate>
    </ItemsControl.ItemTemplate>
  </ListBox>
</StackPanel>
```


Dynamic Data-Bound Collections

With simple data binding, you want a class that implements *INotifyPropertyChanged* so that changes in value can be notified. When binding to collections, you should use a collection that implements *INotifyCollectionChanged* so that additions and deletions to the collection can be notified. If you want to be notified of changes both to the collection and to the values of the properties of the items within the collection, then you need something like this: *CollectionThatImplementsINotifyCollectionChanged<ItemThatImplementsINotifyPropertyChanged>*.

The following example uses such a collection. The individual items are a variation on the *Employee* class that has two string properties: a *Name* and a *Timestamp*. The *Employee* class implements *INotifyPropertyChanged* so that it can be used in two-way data binding. *Employee* items are collected in a custom collection class that derives from *ObservableCollection<T>*, which itself implements *INotifyCollectionChanged*. This custom class contains an arbitrary pool of names and exposes an *AddEmployee* method, which adds a new *Employee* to the underlying collection by using a random name from the pool and the current *DateTime*.

```
public class Employees : ObservableCollection<Employee>
{
    private string[] names =
    { "Sally", "Ajith", "Peter", "Ethel", "Doris", "Mike", "Raza" };

    Random rand = new Random();

    public void AddEmployee()
    {
        int i = rand.Next(0, names.Length - 1);
        Employee emp = new Employee
        { Name = names[i], Timestamp = DateTime.Now.ToLongTimeString() };
        this.Add(emp);
    }
}
```

The collection is bound to a *ListBox*, and the UI provides two *Button* controls to change the data. The Add Item button adds an item to the collection, which triggers the *NotifyCollectionChangedEvent*. The Change Item button updates the *Timestamp* on the currently selected item, which triggers the *NotifyPropertyChangedEvent*. The finished application (the *DynamicCollectionBinding* solution in the sample code) is shown in Figure 4-6.

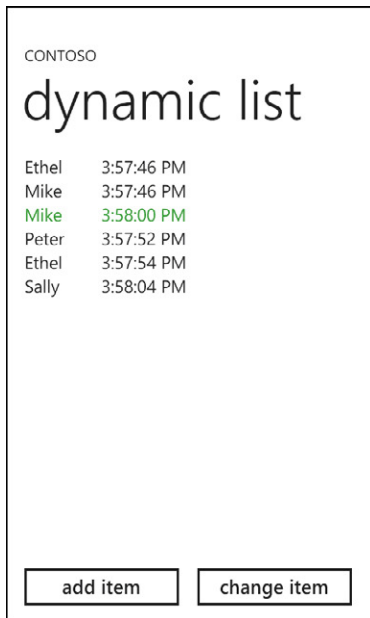


FIGURE 4-6 Data binding works with collections that change dynamically.

The *MainPage* code-behind, initializes the collection with a couple of *Employee* objects and binds it to the *ListBox*. It's also worth noting that you can retrieve a strongly typed object from the *ListBox*'s *IEnumerable* collection of data-bound items. For example, in the *changeButton_Click* handler, you retrieve the currently selected item and cast it to an *Employee* object.

```
public partial class MainPage : PhoneApplicationPage
{
    private Employees emps;

    public MainPage()
    {
        InitializeComponent();
        emps = new Employees();
        emps.AddEmployee();
        emps.AddEmployee();
        myList.ItemsSource = emps;
    }

    private void addButton_Click(object sender, RoutedEventArgs e)
    {
        emps.AddEmployee();
    }
}
```

```

private void changeButton_Click(object sender, RoutedEventArgs e)
{
    if (myList.SelectedIndex != -1)
    {
        Employee emp = (Employee)myList.SelectedItem;
        emp.Timestamp = DateTime.Now.ToLongTimeString();
    }
}
}

```

That is all you have to do. You have already implemented *INotifyPropertyChanged* in the individual item class, and the *ObservableCollection<T>* has an implementation of *INotifyCollectionChanged* that you're using under the covers.

Template Resources

As with other things such as styles, you can define a data template as a resource. This is useful if it's the kind of template that lends itself to reuse. The mechanism is straightforward. First, you define the template just as you would normally. The only difference is that you must declare a *Key* for each resource. The resource definition resides in the *Resources* section of the XAML element where you want it to be visible. This could be in the App.xaml, if you want to use the template across multiple pages, or locally in the XAML for the page in which it will be used, either at the page level or at the level of any child element at or above the level where it will be used.

```

<phone:PhoneApplicationPage.Resources>
    <DataTemplate x:Key="template1">
        <Grid>
            <Grid.ColumnDefinitions>
                <ColumnDefinition Width="100"/>
                <ColumnDefinition Width="*"/>
            </Grid.ColumnDefinitions>
            <TextBlock
                Grid.Column="0" Text="{Binding ID}"
                FontSize="{StaticResource PhoneFontSizeMedium}"
                FontWeight="Bold"/>
            <TextBlock
                Grid.Column="1" Text="{Binding Name}"
                FontSize="{StaticResource PhoneFontSizeMedium}"
                FontStyle="Italic"/>
        </Grid>
    </DataTemplate>
</phone:PhoneApplicationPage.Resources>

```

Next, in the element to which you want this template to apply, specify it by its *Key* name. You can see this at work in the *TemplateResource* solution in the sample code.

```

<ListBox
    x:Name="myList" VerticalAlignment="Top"
    ItemTemplate="{StaticResource template1}"/>

```

Type/Value Converters

You have seen that data binding makes it easy to associate data with the UI in a loosely coupled manner. You have also seen that you have a great degree of control over the UI formatting of data-bound data. In fact, it is even possible to convert a data value from one type to another as part of the data binding process. For example, suppose that you have a collection of *Employee* objects that expose two properties: *Name* and *Gender*. You want to bind the *Name* property in the conventional way (to a *TextBlock.Text* element). However, you want to bind the *Gender* property in a different way: rather than displaying a string representation of the *Gender*, you want to render the text for *Name* differently, depending on the value of *Gender*. Figure 4-7 shows two alternative implementations. In one example (on the left), you convert the *Gender* value to a *Brush* of a particular *Color* and render the name in *Red* for female, *Blue* for male. In the other example (on the right), you convert the *Gender* value to a *Font Weight* and render the name as *Light* for female, *Bold* for male. These screenshots are taken from the *BindingConverters_Color* and *BindingConverters_FontWeight* solutions in the sample code.

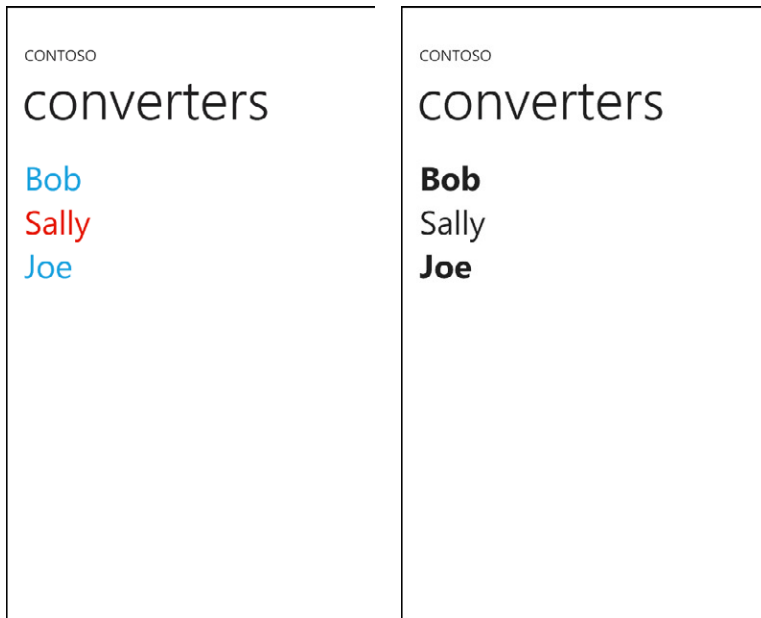


FIGURE 4-7 You can write value converters to customize your data-bound UI.

The *Employee* data object type simply exposes two properties. You also declare a simple collection to hold objects of this type. The only interesting code is the class that implements *IValueConverter*. This interface declares two methods: *Convert* and *ConvertBack*. If you want only one-way binding, you need only to implement the *Convert* method. For two-way binding, you would also need to implement *ConvertBack*.

The first example implements the *Convert* method to return a *Color* whose value is computed based on the incoming value parameter. This will be used in the data binding for the *Gender* property. In this way, you convert a *Gender* (that is, a string) value into a *Color* value.

```

public class GenderBrushConverter : IValueConverter
{
    public object Convert(
        object value, Type targetType, object parameter, CultureInfo culture)
    {
        if ((char)value == 'F')
            return new SolidColorBrush(Color.FromArgb(255,229,20,0));
        else
            return new SolidColorBrush(Color.FromArgb(255,27,161,226));
    }

    public object ConvertBack(
        object value, Type targetType, object parameter, CultureInfo culture)
    {
        // Implement this for two-way binding.
        throw new NotImplementedException();
    }
}

```

Note that the defined *Red* and *Blue* colors are part of the accent colors for the phone, which are different from the standard *Color.Red/Color.Blue* values. The converter is implemented within the application code, and you want to use it in the XAML for the same application. To make the converter accessible in the XAML, you need to declare a new XML namespace for this assembly, as part of the *PhoneApplicationPage* declaration, alongside all the other namespace declarations. Then, specify an *ItemsControl* (in this case, a *ListBox*), with a *GenderBrushConverter* resource. Bind the *TextBlock.Text* to the *Employee.Name* in the normal way. The interesting piece is binding the *TextBlock.Foreground* to the *Employee.Gender* via the converter, as shown here:

```

<phone:PhoneApplicationPage
...
    xmlns:local="clr-namespace:BindingConverters"
>
...
    <Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
        <ListBox
            Name="empList" HorizontalAlignment="Left">
            <ListBox.Resources>
                <local:GenderBrushConverter x:Key="myConverter"/>
            </ListBox.Resources>

            <ListBox.ItemTemplate>
                <DataTemplate>
                    <TextBlock Text="{Binding Name}"
                        FontSize="{StaticResource PhoneFontSizeExtraLarge}"
                        Foreground=
                            "{Binding Gender,
                                Converter={StaticResource myConverter}}"/>
                </DataTemplate>
            </ListBox.ItemTemplate>
        </ListBox>
    </Grid>
</phone:PhoneApplicationPage>

```

This listing places the converter resource within the *ListBox* itself—which is appropriate if you’re not intending to use it anywhere else. However, it is more common to declare converters at the page (or even application) level.

```
<phone:PhoneApplicationPage.Resources>
    <local:GenderBrushConverter x:Key="myConverter"/>
</phone:PhoneApplicationPage.Resources>
```

Finally, set the *ListBox.ItemsSource* property to the collection of Employee objects.

```
public MainPage()
{
    InitializeComponent();
    empList.ItemsSource = new EmployeeCollection();
}
```

The second example uses an almost identical approach, substituting a *FontWeight* for a *Color*, and is included mainly for the benefit of folks reading the paper version of this book, where different colors might not be so easily distinguishable. It also serves to illustrate how easy and flexible this technique is. The *Convert* method and the use of the converter in the item template are listed in the following snippet:

```
public object Convert(
    object value, Type targetType, object parameter, CultureInfo culture)
{
    if ((char)value == 'F')
    {
        return FontWeights.Light;
    }
    else
    {
        return FontWeights.Bold;
    }
}

<TextBlock Text="{Binding Name}"
    FontSize="{StaticResource PhoneFontSizeExtraLarge}"
    FontWeight =
        "{Binding Gender, Converter={StaticResource myConverter}}"/>
```

Element Binding

In addition to binding to data from a data source, you can also bind across elements in the UI. Here is an example that binds the *Text* property of a *TextBlock* to the value of a *Slider*. As the user moves the *Slider*, the value is propagated to the *TextBlock*. Note that this also uses a simple double-to-int value converter, which takes the double values of the *Slider* position and displays them as simple integer values in the *TextBlock*.

```
public class DoubleToIntConverter : IValueConverter
{
    public object Convert(
```

```

        object value, Type targetType, object parameter, CultureInfo culture)
    {
        return System.Convert.ToInt32((double)value);
    }

    public object ConvertBack(
        object value, Type targetType, object parameter, CultureInfo culture)
    {
        throw new NotImplementedException();
    }
}

```

The critical syntax in the XAML is to associate the *ElementName* property in the *TextBlock* with the name of the *Slider* element, and to specify that the name of the property on the source element to which you want to bind is the *Value* property (set to the *Path* property on the *TextBlock*). The result is shown in Figure 4-8 (screenshot taken from the *ElementBinding* solution in the sample code). It is also critical that you do not perform forward references—the *ElementName* must already be defined in the tree before you reference it, or else it won't work.

```

<StackPanel x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <Slider x:Name="mySlider" Maximum="100"/>
    <TextBlock
        Text="{Binding ElementName=mySlider,
            Path=Value, Converter={StaticResource myConverter}}"
        Style="{StaticResource PhoneTextTitle1Style}"
        Foreground="{StaticResource PhoneAccentBrush}"/>
</StackPanel>

```

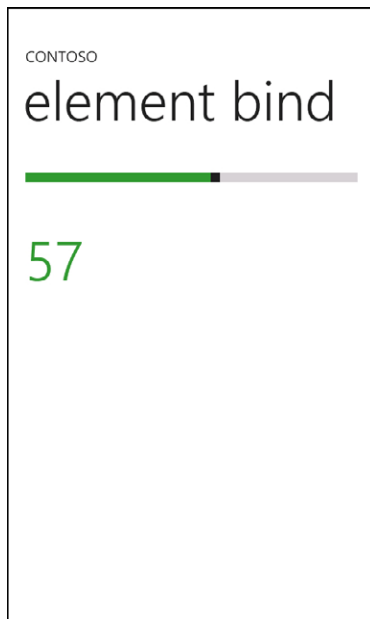


FIGURE 4-8 You can also bind data to UI elements. Here, the *Text* property of a *TextBlock* is bound to the value of a *Slider*.

Data Validation

Silverlight supports a simple level of data validation in two-way bindings. To make use of this validation, the simplest approach is to have your data class throw an exception in its property setter when it encounters invalid data (see the *BindingValidation* solution in the sample code). This variation of the *Employee* class throws an exception in the *ID* property setter if the string entered cannot be converted to an integer. There is no validation on the *Name* property in this example.

```
public class Employee
{
    private string id;
    public string ID
    {
        get { return id; }
        set
        {
            int tmp;
            if (Int32.TryParse(value, out tmp))
            {
                id = value;
            }
            else
            {
                throw new ArgumentOutOfRangeException();
            }
        }
    }

    public string Name { get; set; }
}
```

In the XAML for the *MainPage*, you set the *NotifyOnValidationError* and *ValidatesOnExceptions* properties of the *Binding* for the *ID TextBox* to *true*. This directs the binding engine to raise a *BindingValidationError* event when a validation error is added to or removed from the *Validation.Errors* collection. To handle this event, you need to create an event handler either in the *TextBox*, or on any of its parents in the hierarchy. It is common to handle validation errors on a per-page basis so that you can handle errors from multiple controls in a consistent manner for the whole page. Reading between the lines, it should be clear that this relies on the fact that the *BindingValidationError* is a routed event, which will bubble up the hierarchy from the control where the error occurs to the first parent that handles it.

In this example, however, you handle the event halfway up the hierarchy, in the parent *Grid*. The point being that you can short-circuit the routing and improve performance slightly, because you know that you have no controls outside the *Grid* that have any validation which could trigger a *BindingValidationError*.

```
<Grid
    x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0"
    BindingValidationError="ContentPanel_BindingValidationError">
    <Grid.RowDefinitions>
        <RowDefinition Height="80" />
    </Grid.RowDefinitions>
</Grid>
```



```

        <RowDefinition Height="80"/>
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="120"/>
        <ColumnDefinition Width="*" />
    </Grid.ColumnDefinitions>
    <TextBlock
        Grid.Row="0" Grid.Column="0" Text="ID: "
        Style="{StaticResource PhoneTextTitle3Style}"
        VerticalAlignment="Center"/>
    <TextBox
        x:Name="idText" Grid.Row="0" Grid.Column="1" >
        <TextBox.Text>
            <Binding
                Mode="TwoWay" Path="ID"
                NotifyOnValidationError="true"
                ValidatesOnExceptions="true"/>
        </TextBox.Text>
    </TextBox>
    <TextBlock
        Grid.Row="1" Grid.Column="0" Text="Name: "
        Style="{StaticResource PhoneTextTitle3Style}"
        VerticalAlignment="Center"/>
    <TextBox
        Grid.Row="1" Grid.Column="1"
        Text="{Binding Name, Mode=TwoWay}"/>
</Grid>

```

The implementation of the event handler is in the *MainPage* class. If an error has been added to the collection, the *TextBox* background displays *Red*. When the error is corrected, and therefore removed from the collection, the standard background for a Phone *TextBox* is restored.

```

private void ContentPanel_BindingValidationError(
    object sender, ValidationErrorEventArgs e)
{
    TextBox t = (TextBox)e.OriginalSource;
    if (e.Action == ValidationErrorEventAction.Added)
    {
        t.Background = new SolidColorBrush(Colors.Red);
    }
    else if (e.Action == ValidationErrorEventAction.Removed)
    {
        t.ClearValue(TextBox.BackgroundProperty);
    }
    e.Handled = true;
}

```

Note also the use of the *Control.ClearValue* method to reset the *Background Brush*. This is called on the *BackgroundProperty* in this case. An alternative would be to determine manually which *Brush* you should use (as shown in the following)—for example, if you know you’re using a standard *PhoneTextBoxBrush* resource—but that would clearly be less elegant.

```
t.Background = (Brush)Resources["PhoneTextBoxBrush"];
```

Figure 4-9 shows how the application looks in action. In this scenario, the user has typed in an invalid character and then moved the focus to another control. This triggers the validation engine in the data binding framework, which then invokes the error handler.

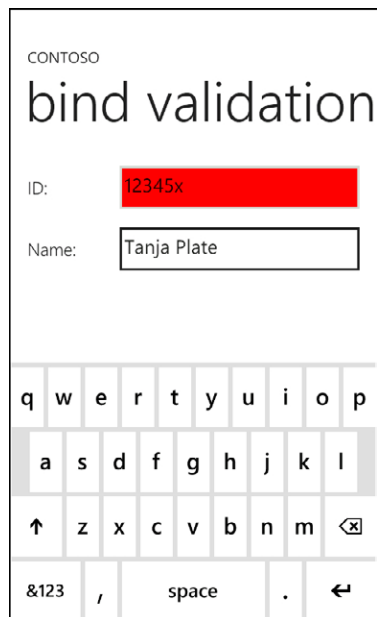


FIGURE 4-9 An invalid character triggers the validation engine in the data binding framework.

Note that it is also not uncommon to have multiple handlers at different levels in the visual tree. For example, you might have a complex set of visual elements, perhaps several *Grid* controls each containing multiple children, for which you want to handle validation errors for each *Grid* in a different fashion. You might also want to have a catch-all handler at the page level.

```
<phone:PhoneApplicationPage
...
    BindingValidationError="PhoneApplicationPage_BindingValidationError">

        <Grid
            x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0"
            BindingValidationError="ContentPanel_BindingValidationError">
...
        </Grid>
    </Grid>
</phone:PhoneApplicationPage>
```

To ensure that the event does not continue routing up the tree, you simply need to set *Handled* to *true* in any handler where you have in fact completely handled the event.

```
private void ContentPanel_BindingValidationError(
    object sender, ValidationErrorEventArgs e)
{
    Debug.WriteLine("ContentPanel_BindingValidationError");
}
```

```

        TextBox t = (TextBox)e.OriginalSource;
        if (e.Action == ValidationErrorEventAction.Added)
        {
            t.Background = new SolidColorBrush(Colors.Red);
        }
        else if (e.Action == ValidationErrorEventAction.Removed)
        {
            t.ClearValue(TextBox.BackgroundProperty);
        }

        e.Handled = true;
    }

    private void PhoneApplicationPage_BindingValidationError(
        object sender, ValidationErrorEventArgs e)
    {
        Debug.WriteLine("PhoneApplicationPage_BindingValidationError");
    }
}

```

Given this code, the handler at the page level would never be called, unless some other element outside the *Grid* also triggers a validation error.

Separating Concerns

So far, you've used simple collections of data that are part of the *MainPage* itself, and have been focused on functionality. Now it's time to pay a little more attention to engineering. In a more realistic application, you would want to further separate the code that represents data from the code that represents UI—at a minimum, by abstracting the collection object from the page code out to a separate class. You can adapt the earlier *DataTemplate* example to improve the SoC. This will be a first attempt to improve decoupling, and it will reap some benefits. Later, you'll see how to evolve this into the more standardized approach taken by the Visual Studio templates. Figure 4-10 illustrates the simple architecture that you're aiming for. You can see this at work in the *CollectionDataBinding_xaml* solution in the sample code.

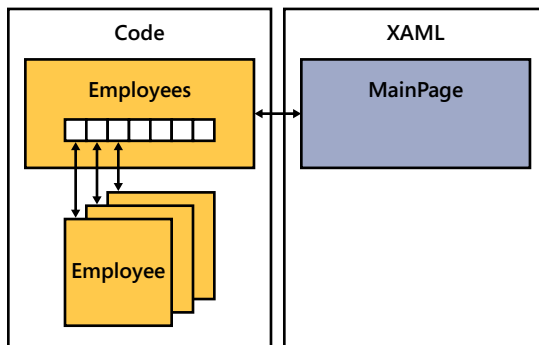


FIGURE 4-10 An example of simple separation of concerns.

First, create a new *Employees* class to represent the collection of *Employee* items. For the purposes of this example (for which you are not interested in collection changes), you can use a simple *List<T>*.

```
public class Employees
{
    public List<Employee> Items { get; set; }

    public Employees()
    {
        Items = new List<Employee>();
        Items.Add(new Employee { ID = "12345", Name = "Pilar Ackerman" });
        Items.Add(new Employee { ID = "13344", Name = "David Pelton" });
        Items.Add(new Employee { ID = "15566", Name = "Jay Hamlin" });
        Items.Add(new Employee { ID = "17788", Name = "Lori Penor" });
        Items.Add(new Employee { ID = "12299", Name = "Yun-Feng Peng" });
    }
}
```



Note It's common to define the property setter as private, so that clients can mutate the list but not completely replace it. However, in this example, you'll need to replace it later.

Having abstracted the data items from the UI, you can now streamline the *MainPage* class. All it needs now is for you to create an instance of the *Employees* collection, and then assign this to the *DataContext* for the page. This doesn't even need to be a class field, because the *DataContext* will keep the reference alive as long as it is needed. This one line of code is now the only connection in the UI code to the data code, providing much cleaner separation.

```
public MainPage()
{
    InitializeComponent();
    DataContext = new Employees();
}
```

One obvious advantage of separating concerns, even in this simple manner, is that the *ItemsSource* value can now be assigned declaratively in XAML, instead of in code, as demonstrated here:

```
<ListBox
    x:Name="employeesList" Width="300" Height="200"
    VerticalAlignment="Top"
    ItemsSource="{Binding Items}">
```

You could take this a step further, and assign the *DataContext* in XAML, also. To do this, you first need to add a namespace in the page's XAML for the current assembly so that you can subsequently refer to the *Employees* collection class. Second, declare a keyed resource for the *Employees* class. Third, set the *DataContext* to this resource in either the *ListBox* itself or in any of its parents.

```
<phone:PhoneApplicationPage
...
    xmlns:local="clr-namespace:CollectionDataBinding"
>
```

```

<phone:PhoneApplicationPage.Resources>
    <local:Employees x:Key="myDataContext"/>
</phone:PhoneApplicationPage.Resources>

<Grid
    x:Name="LayoutRoot" Background="Transparent"
    DataContext="{Binding Source={StaticResource myDataContext}}">
...
    <StackPanel x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
        <ListBox
            x:Name="employeesList" Width="300" Height="200"
            VerticalAlignment="Top"
            ItemsSource="{Binding Items}">
        </ListBox>
    </StackPanel>
...
</Grid>

</phone:PhoneApplicationPage>

```

Design-Time Data

Another benefit of separating concerns is that this promotes the separation of work between the design team and the development team. You can now easily set up dummy data for use by the designers, which is only used at design-time and does not form part of the final application. This gives the designers greater support in laying out the visual interface, based on realistic sample data. Here is how you do this.

First, declare the dummy data in a XAML file. In the following example, this is named *DesignTime Data.xaml*, but the name is arbitrary. This needs a namespace to reference the current assembly, which is where the *Employees* and *Employee* types are defined. In the XAML, define some *Employee* items that will be in the Items collection in the *Employees* object. Following is the entire contents of the file:

```

<local:Employees
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="clr-namespace:CollectionDataBinding"
    >

    <local:Employees.Items>
        <local:Employee ID="22334" Name="Dummy Name 1"/>
        <local:Employee ID="22445" Name="Dummy Name 2"/>
        <local:Employee ID="22556" Name="Dummy Name 3"/>
        <local:Employee ID="22667" Name="Dummy Name 4"/>
        <local:Employee ID="22778" Name="Dummy Name 5"/>
    </local:Employees.Items>

</local:Employees>

```



Note This is one of the development tasks in Silverlight that is much more easily done in Expression Blend rather than Visual Studio, and that is the primary environment in which design-time data will be used.

In the Properties window for this file, set the build action to *DesignData*. Finally, in the *MainPage.xaml*, declare this as design-time data by using the design-time namespace *d* that has already been defined (as <http://schemas.microsoft.com/expression/blend/2008>):

```
d:DataContext="{d:DesignData DesignTimeData.xaml}"
```

You will see this data rendered in the Design view in Visual Studio (and also in Expression Blend), as shown in Figure 4-11. This is the *CollectionDataBinding_DTDData* solution in the sample code.

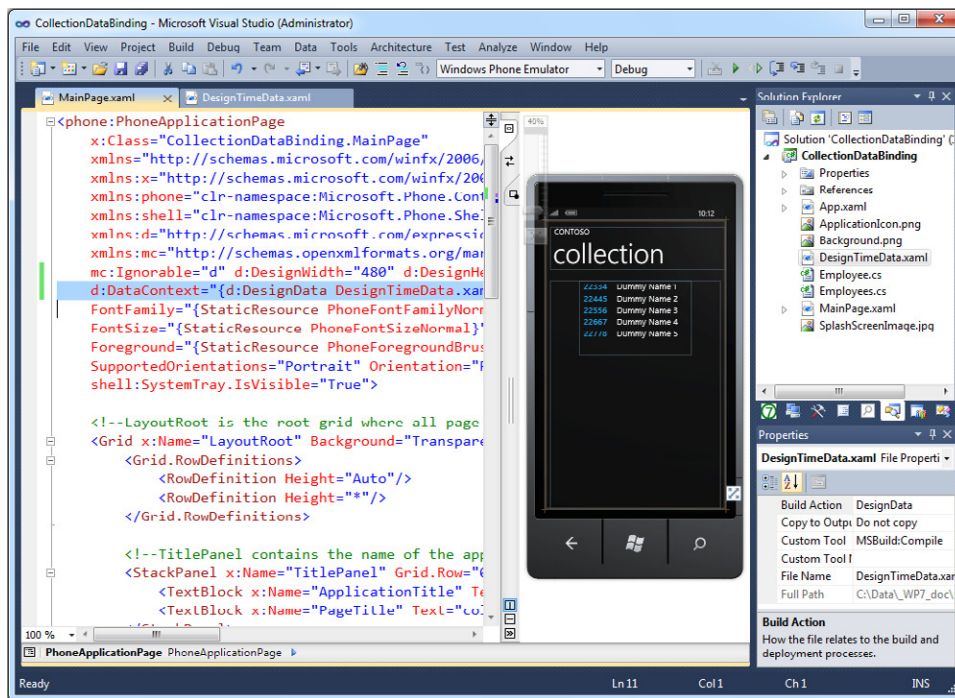


FIGURE 4-11 Design-time data in Visual Studio.

The Model-View ViewModel Pattern

The Model-View ViewModel (MVVM) pattern is extensively used in Silverlight (and therefore Silverlight applications for Windows Phone). This is an evolution of the Model-View Controller (MVC) pattern. One primary reason is to separate design from code. This supports the scenario where application UI designers work in Expression, whereas code developers work in Visual Studio—both working on the same application. It also makes testing a lot easier, in that you can build automated testing independently for each logical layer (UI, business logic, data layer, etc). The three parts of our application are decoupled:

- **View** This is the UI, represented by our XAML, and at a simple level by `mainpage.xaml`.
- **Model** These are the data objects, representing our connection to the underlying data source.
- **ViewModel** This is the equivalent to the controller in MVC, which mediates between model and view. Typically, the view's `DataContext` is bound to an instance of the viewmodel. The viewmodel, in turn, typically instantiates the model (or the model graph).

Silverlight also uses Dependency Injection (DI), which is one form of Inversion of Control (IoC). With DI, when a component is dependent on another component, it doesn't hard-code this dependency; instead, it lists the services it requires. The supplier of services can be injected into the component from an external entity, such as a factory or a dependency framework. In Silverlight, DI is used to provide the glue between the view, the viewmodel, and the model.

Instead of hard-coding the connections between the model, view, and viewmodel, you rely on the Silverlight runtime's DI capabilities. For example, you've seen several examples wherein you set the `DataContext` or `ItemsSource` of an element to some concrete object or collection, such as that illustrated here:

```
DataContext = new Employees();  
empList.ItemsSource = new EmployeeCollection();  
d:DataContext="{d:DesignData DesignTimeData.xaml}"
```

`DataContext` is of type *object*, and `ItemsSource` is of type *IEnumerable*. These afford extremely loose coupling—you can pretty much assign anything to a `DataContext`, and a very wide range of collection objects to an *IEnumerable*. You inject the specific concrete dependency that you want at some point, either at design-time or during unit testing with some mocked-up data, or at runtime in the final product with real data from the production source.

A high-level representation of the general case is shown in Figure 4-12. Not only is there separation between the model, viewmodel, and view, but there is also further decoupling between the viewmodel and view. Given the page-based UI model of Windows Phone applications, this is important to ensure that you can use the same viewmodel in multiple pages. For this reason, no view (page) is responsible for creating the viewmodel. Rather, the *App* creates the viewmodel and exposes it as a property, which is therefore accessible from any page.

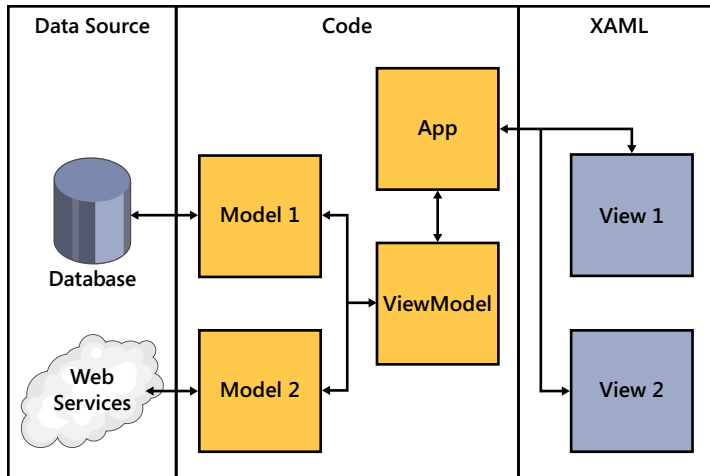


FIGURE 4-12 An overview of the MVVM layers.

Silverlight contains a dependency framework, whereby changes in the underlying model are propagated by the viewmodel to the view via *INotifyPropertyChanged*. The MVVM approach is encouraged in user code, and several of the Visual Studio project templates generate MVVM-based starter code.

Figure 4-13 illustrates a simple example (the *MvvmDataBinding* solution in the sample code). Note that this sample is kept deliberately simple; the limitations are discussed in the following:

- The view is represented by a *ListBox*.
- The viewmodel consists of a collection type that implements *INotifyCollectionChanged* and an item type that implements *INotifyPropertyChanged*.
- The model is represented by a simple class that offers just one piece of data: a *Name* string.

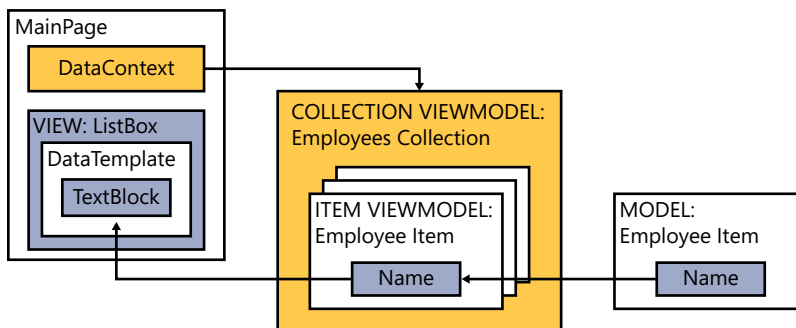


FIGURE 4-13 A simple MVVM application.

Working from the top down, in the *MainPage*, there is a *ListBox* in which each item is a *TextBlock* that is data-bound to the *Name* property in the *DataContext*.


```

<ListBox Name="empList" >
    <ListBox.ItemTemplate>
        <DataTemplate>
            <TextBlock Text="{Binding Name}"
                Style="{StaticResource PhoneTextLargeStyle}"/>
        </DataTemplate>
    </ListBox.ItemTemplate>
</ListBox>

```

In the *MainPage* constructor, you set the *DataContext* for the whole page to an instance of the collection viewmodel (*EmployeesViewModel*).

```

public MainPage()
{
    InitializeComponent();
    empList.ItemsSource = new EmployeesViewModel();
}

```

The collection viewmodel is an *ObservableCollection* of *EmployeeViewModel* items. The collection exposes a *GetData* method, with which you fetch the underlying data (represented in code by the *EmployeeModel* items) and use that data to initialize a set of *EmployeeModel* items. It might seem somewhat redundant to set up an array of *EmployeeModel* items only then to use them to create a set of *EmployeeViewModel* items to add to the collection. However, the point here is that you would more realistically fetch the data from some underlying data source (database, web service, and so on), so there would normally be an additional step to take each data (model) item and map it into an item-level viewmodel item, before adding it to the collection viewmodel.

```

public class EmployeesViewModel : ObservableCollection<EmployeeViewModel>
{
    public EmployeesViewModel()
    {
        GetData();
    }

    private void GetData()
    {
        List<EmployeeModel> data = new List<EmployeeModel>();
        data.Add(new EmployeeModel { Name = "humberto acevedo" });
        data.Add(new EmployeeModel { Name = "alfons parovszky" });
        data.Add(new EmployeeModel { Name = "yael peled" });

        foreach (EmployeeModel employee in data)
        {
            Add(new EmployeeViewModel(employee));
        }
    }
}

```

The *EmployeeViewModel* (item viewmodel) class implements *INotifyPropertyChanged*, which is critical for data binding. You separate the model from the viewmodel—this example illustrates one reason why it is useful to do this. The viewmodel is not just a straight pass-through of the data from the model to the view. Instead, you take in a model (*EmployeeModel*) item and perform some processing on the raw data to initialize the viewmodel (in this example, this ensures initial capital letters

for each word in the raw data). You should also take care to use the *Name* property when assigning the modified data, because the property has additional validation (checking for null, in this case) of which you want to take advantage. Finally, of course, the property setter only actually changes the data and raises the event if the incoming value is in fact different from the current value.

```
public class EmployeeViewModel : INotifyPropertyChanged
{
    public event PropertyChangedEventHandler PropertyChanged;

    private string name;
    public string Name
    {
        get { return name; }
        set
        {
            if (name != value)
            {
                name = TitleCase(value);
                if (PropertyChanged != null)
                {
                    PropertyChanged(
                        this, new PropertyChangedEventArgs("Name"));
                }
            }
        }
    }

    public EmployeeViewModel(EmployeeModel e)
    {
        Name = e.Name;
    }

    private static String TitleCase(String name)
    {
        string s = Regex.Replace(name, @"\w+", (n) =>
        {
            string tmp = n.Value;
            return
                char.ToUpper(tmp[0]) + tmp.Substring(1, tmp.Length - 1).ToLower();
        });
        return s;
    }
}
```

The raw data is represented by a simple *EmployeeModel* class. It is important to note that this is a true model class and not a viewmodel class—it has no code that has anything to do with UI at all. This example, therefore, has all three pieces of the MVVM architecture. Even in this very simple application, you can see a case for having all three pieces.

```
public class EmployeeModel
{
    public string Name { get; set; }
}
```

Note, however, that it will not always be strictly necessary (or even desirable) to follow this model to the letter. In many applications, it is sufficient to have separation between view and viewmodel, and there can well be no separate code representation of the model. Consider, for example, the very first example of a data-bindable *Employee* class you looked at earlier in this chapter. That version implemented *INotifyPropertyChanged*, which implies that it will take part in databinding to the UI. That makes it strictly a viewmodel class, not a model class.

The preceding example was kept deliberately simple, but you should now consider the gaps. You can also see the changes in the *MvvmDataBinding_Model* solution in the sample code. First, no reference was kept to the original *EmployeeModel* object when constructing each *EmployeeViewModel*. This effectively breaks the change propagation chain between the model and the viewmodel; thus, any changes made in the model directly would not get surfaced in the viewmodel, and vice-versa. Suppose that you want two-way data binding in the view, with an updated XAML declaration for the *ListBox* items, to use a *TextBox* instead of a *TextBlock*, such as in the following:

```
<ListBox x:Name="empList" >
    <ListBox.ItemTemplate>
        <DataTemplate>
            <TextBox
                Text="{Binding Name, Mode=TwoWay}"
                FontSize="{StaticResource PhoneFontSizeLarge}"/>
        </DataTemplate>
    </ListBox.ItemTemplate>
</ListBox>
```

Then, when the user changes any of the employee values in the UI, these will be changed in the *EmployeeViewModel*, as well, due to the two-way binding. To ensure that these are further propagated to the underlying *EmployeeModel*, you'd need to keep a reference to the model in the viewmodel.

```
private EmployeeModel model;
public EmployeeViewModel(EmployeeModel e)
{
    model = e;
    name = e.Name;
}
```

You'd also need to update the property setter to change the underlying model value.

```
public string Name
{
    get { return name; }
    set
    {
        if (name != value)
        {
            name = TitleCase(value);
            model.Name = value;
            if (PropertyChanged != null)
            {

```

```

        PropertyChanged(
            this, new PropertyChangedEventArgs("Name"));
    }
}
}
}

```

That would be enough to maintain the propagation chain from view through viewmodel to model. To fix the link in the other direction—from model to viewmodel—you’d need to expose some kind of change event from the model, and sink this in the viewmodel. Note that this should probably not be an *INotifyPropertyChanged* event, because that type implies a viewmodel-view relationship. Instead, you could define a custom event, as follows:

```

public class EmployeeModel
{
    public event EventHandler<EventArgs> NameChanged;

    private string name;
    public string Name
    {
        get { return name; }
        set
        {
            if (name != value)
            {
                name = value;
                if (NameChanged != null)
                {
                    NameChanged(this, new EventArgs());
                }
            }
        }
    }
}

```

Then, you’d sink this event in the viewmodel to update the viewmodel property value.

```

public EmployeeViewModel(EmployeeModel e)
{
    model = e;
    model.NameChanged += model_NameChanged;
    name = TitleCase(e.Name);
}

private void model_NameChanged(object sender, EventArgs e)
{
    Name = ((EmployeeModel)sender).Name;
}

```

Note that it’s important that the property setter verifies that the value is actually different from the current value of the field. If you don’t do this, you run the risk of an infinitely looping event sequence for any change notification. Finally, you should consider the case in which the viewmodel needs to be accessed from within multiple pages. So far, you’ve constructed the viewmodel in the *MainPage* constructor, which couples these two entities rather more tightly than they need to be. To improve

this, you should abstract a *ViewModel* property to a location shared by all pages—specifically, the *App* class.

```
public partial class App : Application
{
    private static EmployeesViewModel viewModel = null;
    public static EmployeesViewModel ViewModel
    {
        get
        {
            if (viewModel == null)
            {
                viewModel = new EmployeesViewModel();
            }
            return viewModel;
        }
    }
}
```

It's worth emphasizing that you're exposing the *ViewModel* as a property. Previously, consumers (the pages) had to instantiate the viewmodel via its constructor. Now, instead, they access the property. The point here is that, under the covers, the property is returning a singleton object—and consumers no longer use the constructor, making them even less tightly coupled than before. Having made these changes, you can now update the view (*MainPage*) code to refer to the *ViewModel* via the *App* class:

```
private EmployeesViewModel emps = App.ViewModel;
public MainPage()
{
    InitializeComponent();
    this.emplist.ItemsSource = emps;
}
```

There's one more potential problem with the design: you might want the *TitleCase* behavior to take place purely between the viewmodel and the view—that is, as a UI-only artifact. You might not want title case changes to be propagated back to the model. For instance, although you might display a name as "Pilar Ackerman," it might actually be acceptable for it to be stored in the backing database as "pilar ackerman" or "PILAR ACKERMAN." In this scenario, you'd want to perform an additional check to ensure that you don't propagate *TitleCase* changes back to the model, but still propagate other data changes. The same would apply for any changes that are UI or "cosmetic" only, as opposed to meaningful changes to the data itself.

MVVM is the pattern adopted in the Databound Application project template in Visual Studio. It's reasonable to assume that a Visual Studio template will generate code that follows best practices. This is almost entirely true—with one minor exception, as you shall see in the following section.

The Visual Studio Databound Application Project

The Databound Application template in Visual Studio generates a simple MVVM project (providing the view and viewmodel, but not the model). This is illustrated in Figure 4-14 (the *DataBoundApp* solution in the sample code). Take a moment to examine the anatomy of this project type. The *Main Page* includes a *ListBox* whose items are made up of two *TextBlock* controls. The *DetailsPage* includes two independent *TextBlock* controls.

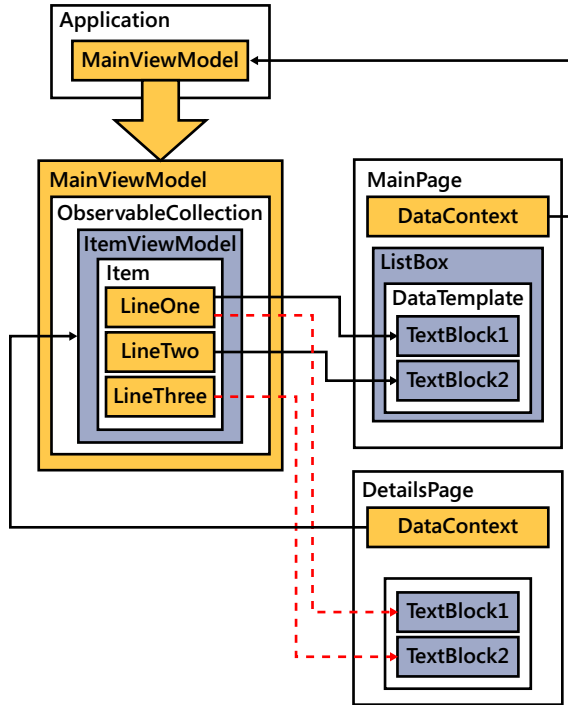


FIGURE 4-14 The Visual Studio Databound Application project template.

The *DataTemplate* for the *ListBox* contains two *TextBlock* controls, which are bound to the *LineOne* and *LineTwo* properties in the *ItemViewModel*.

```
<ListBox
  x:Name="MainListBox" Margin="0,0,-12,0"
  ItemsSource="{Binding Items}" SelectionChanged="MainListBox_SelectionChanged">
  <ListBox.ItemTemplate>
    <DataTemplate>
      <StackPanel Margin="0,0,0,17" Width="432">
        <TextBlock
          Text="{Binding LineOne}" TextWrapping="Wrap"
          Style="{StaticResource PhoneTextExtraLargeStyle}"/>
        <TextBlock
          Text="{Binding LineTwo}" TextWrapping="Wrap"
          Margin="12,-6,12,0"
          Style="{StaticResource PhoneTextSubtleStyle}"/>
      </StackPanel>
    </DataTemplate>
  </ListBox.ItemTemplate>
</ListBox>
```

```

        </StackPanel>
    </DataTemplate>
</ListBox.ItemTemplate>
</ListBox>

```

The individual items of data are modeled by the *ItemViewModel* class, which of course implements *INotifyPropertyChanged*. This class exposes the *LineOne*, *LineTwo* properties to which the *ListBox* items are bound.

```

public class ItemViewModel : INotifyPropertyChanged
{
    private string _lineOne;
    public string LineOne
    {
        get { return _lineOne; }
        set
        {
            if (value != _lineOne)
            {
                _lineOne = value;
                NotifyPropertyChanged("LineOne");
            }
        }
    }

    private string _lineTwo;
    public string LineTwo
    {
        ...
    }

    private string _lineThree;
    public string LineThree
    {
        ...
    }

    ...
}

```

The *MainViewModel* class contains an *ObservableCollection* of *ItemViewModel* items, and at run-time it creates an arbitrary set of items in its *LoadData* method (which you would typically replace with real data from your own model).

```

public class MainViewModel : INotifyPropertyChanged
{
    public MainViewModel()
    {
        this.Items = new ObservableCollection<ItemViewModel>();
    }

    public ObservableCollection<ItemViewModel> Items { get; private set; }

    public bool IsDataLoaded
    {

```

```

        get;
        private set;
    }

    public void LoadData()
    {
        this.Items.Add(new ItemViewModel() { LineOne = "runtime one", LineTwo = "Maecenas pr
esent accumsan bibendum", LineThree = "Facilisi faucibus habitant inceptos interdum lobortis
nascetur pharetra placerat pulvinar sagittis senectus sociosqu" });
        ...//etc
        this.IsDataLoaded = true;
    }
    ...
}

```

The *App* class has a field that is an instance of the *MainViewModel* class.

```

public partial class App : Application
{
    private static MainViewModel viewModel = null;
    public static MainViewModel ViewModel
    {
        get
        {
            if (viewModel == null)
                viewModel = new MainViewModel();
            return viewModel;
        }
    }

    private void Application_Activated(object sender, ActivatedEventArgs e)
    {
        if (!App.ViewModel.IsDataLoaded)
        {
            App.ViewModel.LoadData();
        }
    }
}

```

At runtime, the *DataContext* of the *MainPage* is set to refer to the *MainViewModel* in the *App* class. When the page is loaded, it ensures that there is data, loading it if necessary. The resulting list is shown in Figure 4-15 (on the left). Also, when the user selects an item from the *ListBox*, the application navigates to the *DetailsPage*, passing the selected item in the *QueryString*. Figure 4-15, right, shows an instance of the *DetailsPage*.

```

public partial class MainPage : PhoneApplicationPage
{
    public MainPage()
    {
        InitializeComponent();
        DataContext = App.ViewModel;
        this.Loaded += new RoutedEventHandler(MainPage_Loaded);
    }
}

```



```

private void MainPage_Loaded(object sender, RoutedEventArgs e)
{
    if (!App.ViewModel.IsDataLoaded)
    {
        App.ViewModel.LoadData();
    }
}

private void MainListBox_SelectionChanged(
    object sender, SelectionChangedEventArgs e)
{
    NavigationService.Navigate(new Uri(
        "/DetailsPage.xaml?selectedItem="
        + MainListBox.SelectedIndex, UriKind.Relative));
}
}

```

Down in the *DetailsPage* class, when the user navigates to the page, the code sets its *DataContext* to the item in the *MainViewModel.Items* collection that is specified in the *QueryString*.

```

protected override void OnNavigatedTo(NavigationEventArgs e)
{
    string selectedIndex = "";
    if (NavigationContext.QueryString.TryGetValue(
        "selectedItem", out selectedIndex))
    {
        int index = int.Parse(selectedIndex);
        DataContext = App.ViewModel.Items[index];
    }
}

```

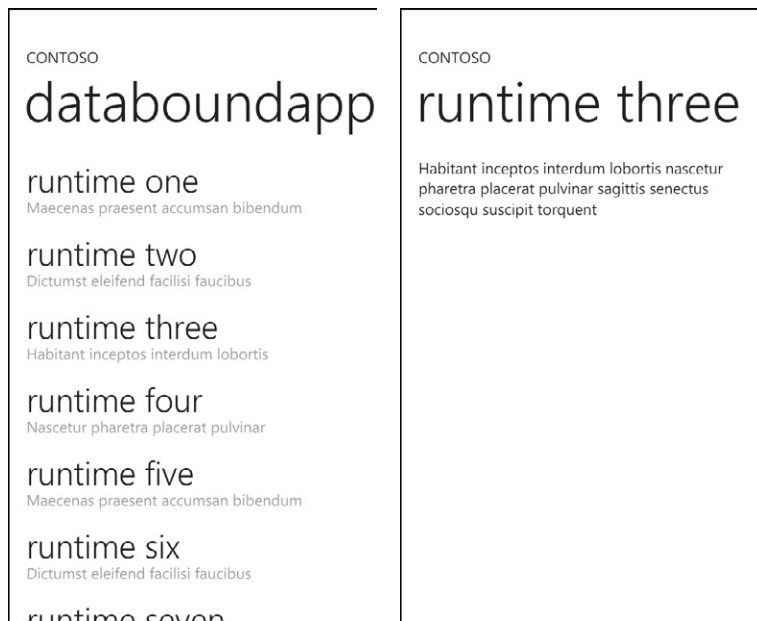


FIGURE 4-15 The *MainPage* and *DetailsPage* of the standard Databound Application project.

Note that at design time, in the XAML, the page's *DataContext* is set to a design-time data file (MainViewModelSampleData.xaml).

```
d:DataContext="{d:DesignData SampleData/MainViewModelSampleData.xaml}"
```

The standard Visual Studio template-generated *Pivot* and *Panorama* projects use the same MVVM approach. The *Pivot* project is especially interesting because it illustrates a good pattern for filtering data-bound data; all pivot items are bound to the same data source, but each one has a different “column filter” applied. This is represented in Figure 4-16, which you can see at work in the *PivotFilter* solution in the sample code.

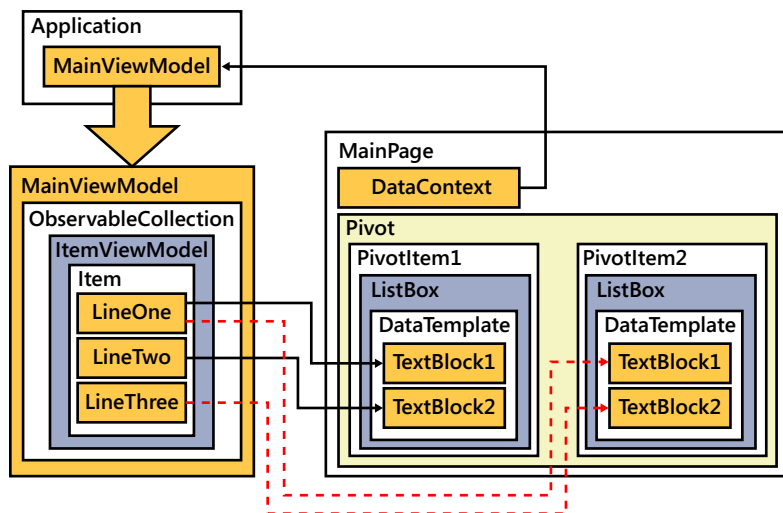


FIGURE 4-16 The Visual Studio *Pivot* application.

The *column filtering* is done in the XAML. The first *PivotItem* has a *ListBox* whose two *TextBlock* controls are bound to *LineOne* and *LineTwo* in the *ViewModel*. The second *PivotItem* has a *ListBox* whose two *TextBlock* controls are bound to *LineOne* and *LineThree*.

```
<controls:Pivot Title="MY APPLICATION">
  <controls:PivotItem Header="first">
    <!--Double line list with text wrapping-->
    <ListBox
      x:Name="FirstListBox" Margin="0,0,-12,0"
      ItemsSource="{Binding Items}">
      <ListBox.ItemTemplate>
        <DataTemplate>
          <StackPanel Margin="0,0,0,17" Width="432" Height="78">
            <TextBlock
              Text="{Binding LineOne}" TextWrapping="Wrap"
              Style="{StaticResource PhoneTextExtraLargeStyle}"/>
            <TextBlock
              Text="{Binding LineTwo}" TextWrapping="Wrap"
              Margin="12,-6,12,0"
              Style="{StaticResource PhoneTextSubtleStyle}"/>
          </StackPanel>
        </DataTemplate>
      </ListBox.ItemTemplate>
    </ListBox>
  </controls:PivotItem>
  <controls:PivotItem Header="second">
    <!--Single line list with text wrapping-->
    <ListBox
      x:Name="SecondListBox" Margin="0,0,-12,0"
      ItemsSource="{Binding Items}">
      <ListBox.ItemTemplate>
        <DataTemplate>
          <StackPanel Margin="0,0,0,17" Width="432" Height="78">
            <TextBlock
              Text="{Binding LineOne}" TextWrapping="Wrap"
              Style="{StaticResource PhoneTextExtraLargeStyle}"/>
            <TextBlock
              Text="{Binding LineThree}" TextWrapping="Wrap"
              Margin="12,-6,12,0"
              Style="{StaticResource PhoneTextSubtleStyle}"/>
          </StackPanel>
        </DataTemplate>
      </ListBox.ItemTemplate>
    </ListBox>
  </controls:PivotItem>
</controls:Pivot>
```

```

        </DataTemplate>
    </ListBox.ItemTemplate>
</ListBox>
</controls:PivotItem>

<controls:PivotItem Header="second">
    <ListBox
        x:Name="SecondListBox" Margin="0,0,-12,0"
        ItemsSource="{Binding Items}">
        <ListBox.ItemTemplate>
            <DataTemplate>
                <StackPanel Margin="0,0,0,17">
                    <TextBlock
                        Text="{Binding LineOne}" TextWrapping="NoWrap"
                        Margin="12,0,0,0"
                        Style="{StaticResource PhoneTextExtraLargeStyle}"/>
                    <TextBlock
                        Text="{Binding LineThree}" TextWrapping="NoWrap"
                        Margin="12,-6,0,0"
                        Style="{StaticResource PhoneTextSubtleStyle}"/>
                </StackPanel>
            </DataTemplate>
        </ListBox.ItemTemplate>
    </ListBox>
</controls:PivotItem>
</controls:Pivot>

```

You could obviously take this further and represent the UI of each pivot item differently, according to the nature of the data that is bound to the elements in that item. You could also pivot on the data via *row filtering*. For example, suppose the *ItemViewModel* class also provided an integer *ID* property. Then, you could easily filter the *PivotItem* contents based on the value of this *ID*. Instead of simply allowing the two *ListBox* controls to pick up the complete data set from the *ViewModel*, you could explicitly set each *itemsSource* to some filtered subset of the data, as illustrated in the following listing:

```

private void MainPage_Loaded(object sender, RoutedEventArgs e)
{
    if (!App.ViewModel.IsDataLoaded)
    {
        App.ViewModel.LoadData();
    }

    FirstListBox.ItemsSource =
        (from i in App.ViewModel.Items where i.ID % 2 != 0 select i);
    SecondListBox.ItemsSource =
        (from i in App.ViewModel.Items where i.ID % 2 == 0 select i);
}

```

Given this code, the first *PivotItem* would only list odd-numbered items (Figure 4-17, left), and the second would only list even-numbered items (Figure 4-17, right).

This will work fine for simple read-only data, which is all you have in the initial Databound Application project. However, for read-write data, you'd want to continue to take advantage of two-way data binding, including both *INotifyPropertyChanged* and *INotifyCollectionChanged*. To achieve filtering with these requirements, you'd need to change the simple Language-Integrated Query (LINQ) filters to use *CollectionViewSource* objects, instead. You can see this modified version in the *PivotFilter_CollectionViewSource* solution in the sample code.

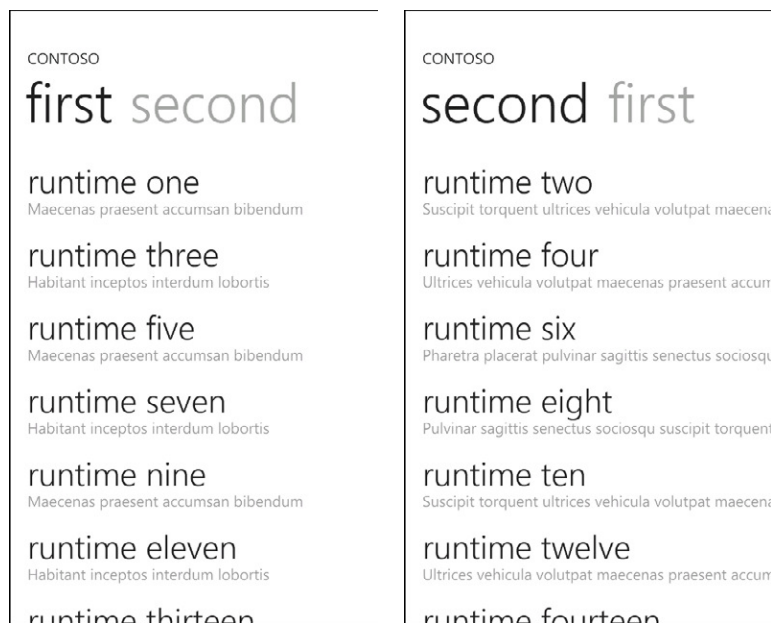


FIGURE 4-17 The first *PivotItem* (left) lists odd-numbered items, while the second *PivotItem* lists even-numbered items.

This class allows you to build a collection view on top of the binding source collection that you're using in the view—that is, a layer between the view and the viewmodel. This allows you to navigate and display the collection, based on sort, filter, and grouping queries, all without the need to manipulate the underlying source collection itself, and with propagation of *INotifyPropertyChanged* and *INotifyCollectionChanged* events.

```
private CollectionViewSource odds;
private CollectionViewSource evens;

private void MainPage_Loaded(object sender, RoutedEventArgs e)
{
    if (!App.ViewModel.IsDataLoaded)
    {
        App.ViewModel.LoadData();
    }

    //FirstListBox.ItemsSource =
    //    (from i in App.ViewModel.Items where i.ID % 2 != 0 select i);
    //SecondListBox.ItemsSource =
```

```
// (from i in App.ViewModel.Items where i.ID % 2 == 0 select i);

odds = new CollectionViewSource();
odds.Source = App.ViewModel.Items;
odds.Filter += (s, ev) =>
{
    ItemViewModel ivm = ev.Item as ItemViewModel;
    ev.Accepted = ivm.ID % 2 != 0;
};
FirstListBox.ItemsSource = odds.View;

evens = new CollectionViewSource();
evens.Source = App.ViewModel.Items;
evens.Filter += (s, ev) =>
{
    ItemViewModel ivm = ev.Item as ItemViewModel;
    ev.Accepted = ivm.ID % 2 == 0;
};
SecondListBox.ItemsSource = evens.View;
}
```

As a footnote, if you look closely, you might notice that the Visual Studio Databound Application template is not as optimal as it could be. Specifically, the code to test for loaded data in the *ViewModel* is duplicated—it's in the *Application_Activated* handler and also in the *MainPage_Loaded* handler. Not only that, but the code is also identical, using the *App* object reference in both places, even within the *App*'s *Application_Activated* handler where the *App* reference is clearly superfluous. Worse, it means that you're straining the decoupling somewhat, because the View now has a little more knowledge of the lifetime of the data than perhaps it should.

```
if (!App.ViewModel.IsDataLoaded)
{
    App.ViewModel.LoadData();
}
```

Why does the template generate this code? The reason is to accommodate the fact that the entry-point to the application varies, depending on circumstances. This arises from navigation and application lifecycle behavior—issues which are discussed thoroughly in Chapter 6, “Application Model.” The lifecycle aspect that is relevant here is that the application might start on the *MainPage*, and it might start on the *DetailsPage*. In normal circumstances, the user launches the application, and the *Application_Launching* event is raised, but not the *Application_Activated* event. Shortly after that, the *MainPage_Loaded* event is raised. It's for this code path that you need the loading code in either the *Application_Launching* or the *MainPage_Loaded* handlers.

The second scenario is where the user runs the application, navigates to the *DetailsPage*, and then navigates forward out of the application to another application. When he comes back to the first application, the *Application_Activated* event is raised, and the system navigates to the *DetailsPage*. The key point is that in this scenario, the *DetailsPage* is typically created before the *MainPage*. This is why you need the data loading code in either the *Application_Activated* handler or in a *PageLoaded* or *NavigatedTo* handler for the *DetailsPage*.

One improvement would be to remove the *MainPage_Loaded* handler (which is otherwise unused), and localize the code to the *App* class, which is in the *Application_Launching* and *Application_Activated* handlers. This would at least put it all in one class, and it would afford you the ability to remove the superfluous *App* object reference. An even more elegant solution would be to remove the duplication altogether. You could achieve this by simply putting the loading code in the *ViewModel* property getter. This would guarantee that any time the *ViewModel* is accessed, it will always have loaded data, regardless of the application's launch context, and regardless of how many pages need to access the data. The slight disadvantage is the very small performance cost of doing the *IsDataLoaded* test on each access. You can see these changes in the *DataBoundApp_modified* solution in the sample code.

```
public static MainViewModel ViewModel
{
    get
    {
        if (viewModel == null)
        {
            viewModel = new MainViewModel();

            // Code ported from MainPage_Loaded and Application_Activated.
            if (!viewModel.IsDataLoaded)
            {
                viewModel.LoadData();
            }

            return viewModel;
        }
    }
}
```

Or, if you assume that you always want to load the data on first initialization of the *ViewModel* object, you could perform both operations at the same time. To be safe, you could do this inside a lock, as shown in the following:

```
private static readonly object myLock = new object();

public static MainViewModel ViewModel
{
    get
    {
        lock(myLock)
        {
            if (viewModel == null)
            {
                viewModel = new MainViewModel();
                viewModel.LoadData();
            }

            return viewModel;
        }
    }
}
```



Note If your data comes in an asynchronous manner (for example, from the web), you would probably want to raise an event when you've received new data, and have the data consumers (*ViewModels*) subscribe to this event. You'd then have to implement a way to trigger loading, based perhaps on the first access to that event.

Note that while the MVVM pattern offers a number of benefits and is suitable for most applications that render data in the UI, it is not without its drawbacks. For a very small application, it can be overkill in terms of the additional effort required to set up and maintain additional *Model* and *View Model* classes and connections. The counter-argument from an engineering perspective is that once a developer (or developer organization) has gone to the effort of setting up the MVVM framework, development effort after that point for interoperating between data and UI is measurably reduced. This is typically true, and clearly the benefits increase as the complexity of the application increases, and can be only marginal or negative in a small application.

Developers should also consider the additional overhead: as is often the case, whereas a reusable framework such as MVVM—or indeed Silverlight data binding itself—tends to make development more RAD-like in the long run, this comes at the cost of runtime complexity and performance costs. Under the covers, the Silverlight runtime is doing work to handle *NotifyPropertyChanged* and *NotifyCollectionChanged* events, and then route them appropriately, including doing reflection to get the required data values (which is always a costly operation). It is also maintaining internal caches related to the data-bound objects. It's theoretically possible for the additional bookkeeping required for data binding an object even to exceed the memory consumption and processing for the object itself.

As in many other areas, developers should carefully consider the size and performance costs of any technique—the thresholds at which these might become critical are generally much lower on a mobile device than in a desktop application.

Summary

This chapter examined the data binding support in Silverlight for Windows Phone, the benefits it brings, and the various approaches you can take to customize the behavior by taking part in the data binding pipeline. Data binding works very well in combination with the MVVM pattern. This pattern helps to ensure clean SoC, such that the discrete functional parts of the application—the data, the view and the viewmodel—can be loosely coupled, and therefore, independently versioned.

Touch UI

This chapter looks at all the touch-related aspects of the phone UI, including the user's view of the logical gestures, and the various levels of real and virtual touch events and method overrides. You might be surprised to learn that some of the touch events use the term "mouse" in their names, but this is just another artifact of porting desktop Microsoft Silverlight to the phone. After a detailed examination of the touch events, the chapter then focuses on the keyboard, including both software and hardware versions. Finally, you'll see how to handle orientation behavior and the phone's application bar.

Logical Touch Gestures

Windows Phone supports a range of touch gestures. Table 5-1 provides the list of logical gestures from a user's perspective.

TABLE 5-1 Logical Touch Gestures

Gesture	Description
Tap	A finger touches the screen and then releases with minimal movement.
DoubleTap	Two taps in quick succession.
Hold	A finger touches the screen and holds it in place for some minimum period of time.
Pan/Drag	A finger touches the screen and then moves in any direction. Dragging moves some recognizably discrete content around the screen. Panning is really the same as dragging, except that the content is larger than the screen.
Flick	A finger drags across the screen and then lifts up without stopping the movement.
Pinch	Two fingers press on the screen and are then moved toward each other.
Stretch	Two fingers press on the screen and are then moved apart, relative to each other.

The Windows Phone chassis specification used by phone manufacturers dictates that all Windows Phones must support a true multi-touch input system capable of sensing and reporting a minimum of 4 and a maximum of 10 distinct touch points simultaneously. The baseline for a finger touch is a 7 mm diameter circle. There's also a recommendation that the touch screen should not support touches by objects that are not shaped like a finger (especially a palm touch), or which are greater than 30x30 mm. These measurements represent the boundary limits for any touch targets that you might include in your application.

Microsoft provides comprehensive guidelines on the use and usability of touch gestures, which are summarized in the following:

- Design and build your application so that the user can access as much functionality as possible using the simplest gestures. That is, opt for single-touch operations rather than multi-touch wherever possible; for example, choose single-tap over double-tap, and so on.
- Don't build excessive processing into touch event handlers. This ensures the minimum latency between the user touching the screen and your application carrying out the designated operation, with the appropriate visual feedback.
- If a user's touch gesture kicks off an operation that will take a noticeable amount of time to complete, you should provide immediate feedback that the gesture has been recognized and is being acted upon, and then present subsequent incremental feedback that the operation is in progress.
- When designing your visual UI, keep in mind that touch targets (controls, shapes, and so on) should not be smaller than 9x9 mm or 34x34 pixels. You should also provide at least 2 mm or 8 pixels between touch targets. For elements that are frequent touch targets in your application, you should actually make them larger—for these, a minimum size of 72x72 pixels is recommended. Controls smaller than 34 pixels are not categorically forbidden, although you might make it difficult for your application to pass certification testing. In any event, no touch target should ever be smaller than 7x7 mm or 26x26 pixels in area. For those touch targets for which hitting the wrong target by mistake would have a severe negative effect, you should make them even bigger.
- It is acceptable to make the touch target larger than the touch element, but you should not exceed 40 percent larger, and it should not be smaller. If you make it larger, don't forget that the minimum 8-pixel gap should be maintained between touchable targets, not between the elements. For standard controls, the touch target size relative to the control size is set and non-configurable; this guideline is only relevant if you're building custom controls or retemplating existing controls.

Note that the standard controls, such as *ListBox* and *Panorama*, also obey a set of Metro-inspired rules of motion, and you are encouraged to use the same rules when building your own controls and pages:

- When dragging, you should always track the element under the user's finger. In general, this means that the element that was under the user's finger when the gesture started must remain at the position currently under the user's finger as it moves around.
- When panning, the main element should move with the finger, but any minor elements on top of the main element can either remain stationary or move at a different rate. In the case of *Pivot* and *Panorama* controls, the main content moves around with the finger, whereas non-main elements, such as the title and headers, move at a different rate.
- When flicking, the user's expectation is that the movement continues on its own, so the application should identify a velocity and direction for that movement. The motion should continue with the same direction and speed as the gesture to give the perception that the visual element is a real object with a non-zero mass. The motion should decelerate gradually over time, eventually coming to a stop. In the case of composite controls (such as a *Panorama*), if a new flick gesture begins while a previous element is still in animation, then the previous animation should be immediately interrupted, and the most recent position of the previous element should be set as its final position. This does not apply, of course, for scenarios in which you have multiple, independent elements (for instance, multiple shapes on a canvas) that have no structural relationship with each other.
- Often, elements are restricted to some maximum limit of movement, which is typically dependent on the size of the element. For example, a standard *ListBox* can scroll only a certain distance, based on the number of items in the list and the minimum amount of items that must be visible at any time. On the other hand, a *LoopingSelector* control can scroll infinitely because it loops its contents. For some controls, it is reasonable to lock the elements so that they can move only along the X axis (for example, the *Panorama* control) or along the Y axis (for example, the *ListBox* control).
- Metro guidelines indicate that an out-of-bounds feedback effect should be applied whenever a user attempts to move the contents of a scrollable item past a fixed boundary, either by dragging or by flicking. Once the content encounters a boundary, the content is compressed in the direction of the motion, based on its velocity at the time it hits the boundary, and then decompressed to its original size. This behavior is evident in the standard *ListBox*, *ScrollViewer*, and related control types, and you should adopt it if you're building a custom list control of some kind.

You can handle the logical user gestures at any of six levels by handling four types of event and two types of virtual methods. These are summarized in Table 5-2, presented from the lowest level to the highest. Only the highest-level abstraction provides a close mapping to the logical gestures. In all other cases, the application is responsible for determining which touch gestures the user is performing by tracking the number, timing, and permutation of events. Note that internally, the platform takes the raw input messages and converts them to low-level *ManipulationXXX* events. Next, these are used to synthesize higher-level events such as the *MouseXXX* events and *FrameReported* events.

TABLE 5-2 Touch Events and Overrides

Event Type	Description
Low-level <i>ManipulationXXX</i> events, raised for each element, individually.	These are typically the first events to be raised, so handling them is the fastest approach. They are most useful for composite gestures—that is, pinch and stretch, drag and flick—rather than single <i>Tap</i> or <i>DoubleTap</i> gestures. They can be scoped to individual elements or set page-wide.
<i>OnManipulationXXX</i> virtual methods (not events).	If you override these, they are called before the corresponding <i>ManipulationXXX</i> event handlers (if any). You would typically override these in your <i>Page</i> class, so they will be used for all such events on the page rather than being scoped to any individual element or panel. Using these overrides is typically quicker than the event handlers; internally, if no event handler is connected in the delegate, then it is null and that set of work is skipped.
Slightly higher-level mouse events such as <i>MouseLeftButtonDown</i> and <i>MouseLeftButtonUp</i> that all <i>UIElement</i> types support.	These are routed events, so you need to be aware of the way routing works and know when to prevent onward routing and when not to. You should use these events only if you're building a cross-platform (for example, desktop Windows and/or web, plus phone) application.
Virtual methods (not events) such as <i>OnMouseLeftButtonDown</i> and <i>OnMouseLeftButtonUp</i> .	If you override these, they are always called after the lower-level <i>MouseXXX</i> event is raised. As with the <i>OnManipulationXXX</i> overrides, you would typically override these in your <i>Page</i> class. You should use these events only if you're building a cross-platform application.
Higher-level <i>FrameReported</i> events, which are raised for touch anywhere in the frame.	This is an application-wide event that cannot be more finely scoped. It is useful for handling multiple touch points, but not useful for individual elements or individual pages.
The Silverlight Toolkit provides an even higher-level abstraction via its <i>GestureService</i> .	This handles the lower-level <i>MouseXXX</i> events internally and uses them to synthesize higher-level abstractions that map directly to user gestures. Very easy to use, but comes at a price. Partially superseded in version 7.1 for tap, double-tap, and hold, but not for drag or flick.

Note that Windows Phone 7.1 introduces three additional events, exposed from the *UIElement* class (and all derivatives—which means all standard controls and UI elements in the platform), for *Tap*, *DoubleTap*, and *Hold*.

Manipulation Events: Single Touch (Tap)

At the lowest level exposed to applications, you can handle the *ManipulationXXX* events. You can use these events for all the Windows Phone manipulation types, both single-touch and multi-touch. You would typically handle a group of *ManipulationStarted*, *ManipulationDelta*, and *ManipulationCompleted* events.

First, let's examine a *ManipulationXXX* event-based, single-touch application (this is the *Test Manipulation* solution in the sample code). Figure 5-1 shows the application displaying a *TextBlock*. For this application, when the user touches the *TextBlock*, you'll increase the font size; when he touches anywhere else on the page, you'll decrease the font size.

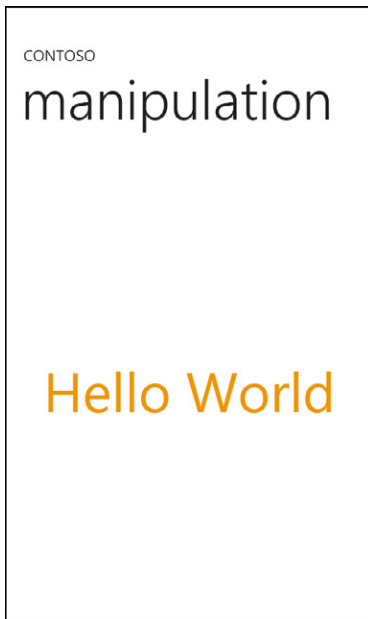


FIGURE 5-1 A sample application to demonstrate manipulation events.

You handle the *ManipulationStarted* event on the *TextBlock* and implement this to increment the *FontSize*. You also override the *OnManipulationStarted* method on the page itself and implement this to decrement the *FontSize*.

```
<Grid x:Name="ContentPanel">
    <TextBlock
        Name="helloText" Text="Hello World"
        ManipulationStarted=" helloText_ManipulationStarted" />
</Grid>
```

```
private void helloText_ManipulationStarted(object sender, ManipulationStartedEventArgs e)
{
    this.helloText.FontSize++;
    e.Complete();
    e.Handled = true;
}
```

```
protected override void OnManipulationStarted(ManipulationStartedEventArgs e)
{
    this.helloText.FontSize--;
    e.Complete();
    base.OnManipulationStarted(e);
}
```

Calling the *Complete* method directs the system not to process any more events for this manipulation. Without this, if the user holds the touch and then moves, this would raise one or more *ManipulationDelta* events. In this application, you don't care about these, so instruct the system that once you have the initial *ManipulationStarted* event, you don't care about anything else in this manipulation group. As always, setting *Handled* to *true* instructs the system not to continue routing this event up the visual tree. Note that the *OnManipulationXXX* override is always called before any *ManipulationXXX* event for that element.

Manipulation Events: Single Touch (Flick)

The next application offers a simple bouncing ball, whose movement is controlled by the user by means of flick and tap gestures, as shown in Figure 5-2. This is the *BallManipulation* solution in the sample code. When the user flicks the ball, you start it moving; when she taps the ball, you stop it.

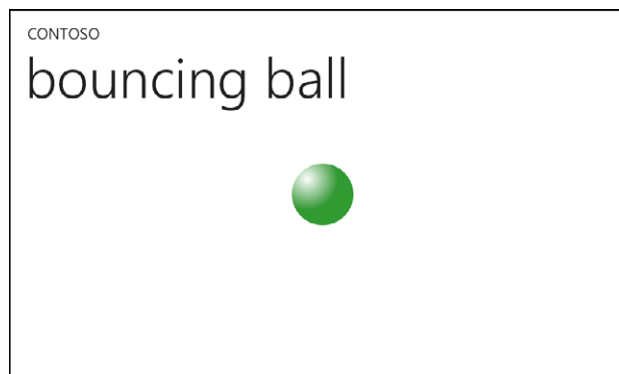


FIGURE 5-2 A sample application to demonstrate the flick gesture.

Recall that the standard Microsoft Visual Studio template generates a layout with an outer *Grid*. In this application, you change that to a *Canvas*, instead. This is because you want to render the bouncing ball anywhere on the screen, with absolute control over its position. The usual title panel is in a *Grid* inside this *Canvas*, and the *Canvas* contains an *Ellipse* to represent the ball. The *Ellipse* includes an event handler for the *ManipulationCompleted* event.

```
<Canvas x:Name="MainCanvas">
  <Ellipse
    x:Name="ball" Canvas.Left="368" Canvas.Top="200" Width="80" Height="80"
    ManipulationCompleted="ball_ManipulationCompleted">
    <Ellipse.Fill>
      <RadialGradientBrush GradientOrigin="0.25,0.25" Center="0.25,0.25">
        <RadialGradientBrush.GradientStops>
          <GradientStop Color="White" Offset="0"/>
          <GradientStop Color="#FF339933" Offset="1"/>
        </RadialGradientBrush.GradientStops>
      </RadialGradientBrush>
    </Ellipse.Fill>
  </Ellipse>
</Canvas>
```

Note that the initial position of the ball is centered on the canvas. To do this, you use absolute values for *Left* and *Top* that rely on having set the orientation to landscape for the application. To keep things simple, this application does not support portrait orientation. When the user flicks the ball, you handle the event by updating the horizontal and vertical velocity (which can be positive or negative); this gives you the new X,Y position to which to move the ball. You then start a *DispatcherTimer*. The timer has a 33 ms tick, which equates to 30 frames per second. In the *Tick* event handler, you move the ball by getting and setting its *Canvas.Left/Top* properties. If the ball hits the bounding box, it reverses direction. Note that you factor the size of the ball into the calculation, to make sure no part of it can go beyond the bounding box. When the user taps the ball, you stop it from moving by stopping the timer.

```
private double ballSize = 80;
private int horizontalVelocity;
private int verticalVelocity;

private DispatcherTimer timer;

public MainPage()
{
    InitializeComponent();
    timer = new DispatcherTimer();
    timer.Interval = new TimeSpan(0, 0, 0, 0, 33);
    timer.Tick += timer_Tick;
}

private void timer_Tick(object sender, EventArgs e)
{
    double xPos = (double)ball.GetValue(Canvas.LeftProperty);
    double yPos = (double)ball.GetValue(Canvas.TopProperty);

    if (xPos >= (MainCanvas.ActualWidth - ballSize) || xPos <= 0)
    {
        horizontalVelocity = -horizontalVelocity;
    }
    if (yPos >= (MainCanvas.ActualHeight - ballSize) || yPos <= 0)
    {
        verticalVelocity = -verticalVelocity;
    }

    xPos += horizontalVelocity;
    yPos += verticalVelocity;
    ball.SetValue(Canvas.LeftProperty, xPos);
    ball.SetValue(Canvas.TopProperty, yPos);
}
```

```
private void ball_ManipulationCompleted(object sender, ManipulationCompletedEventArgs e)
{
    if (e.IsInertial)
    {
        horizontalVelocity = (int)(e.FinalVelocities.LinearVelocity.X / 100);
        verticalVelocity = (int)(e.FinalVelocities.LinearVelocity.Y / 100);
        timer.Start();
    }
    else
    {
        timer.Stop();
    }
}
```

The only manipulation event you need to handle is *ManipulationCompleted*. You'll get this event for all kinds of touch gestures. In this application, you care only about flick and tap. Identifying a flick in this example is a simple matter of testing the *IsInertial* property of the *ManipulationCompletedEventArgs*. Of the different *ManipulationXXXEventArgs* types, only *ManipulationCompletedEventArgs* provides the *IsInertial* property. This is set to *true* in the case of a flick gesture. If this was a flick gesture, you can extract the linear velocity from the *FinalVelocities* property. Note that *IsInertial* will also be true in the context of a drag or pinch manipulation involving inertia, but you're not interested in that context in this example.

You stop the ball animating for any other event, but this is clearly a lazy and incomplete approach to handling just the tap event. If you really want to properly distinguish other non-inertial gestures, you need to do a little more work. Specifically, to distinguish pinch/stretch and pan/drag gestures, you additionally handle the *ManipulationDelta* event. To distinguish single-tap from double-tap or hold, you watch sequential *ManipulationStarted* and *ManipulationCompleted* events, and the elapsed time between them.

Manipulation Events: Multi-Touch

Figure 5-3 shows the next example (the *PinchAndStretch* solution in the sample code), which is an application that displays an image on the page that the user can shrink and enlarge via pinch and stretch gestures. He can also drag the image around the screen.

pinch+stretch



FIGURE 5-3 This application illustrates manipulation events for pinch-and-stretch and drag gestures.

The application handles pinch and stretch by applying a *ScaleTransform* to the image. It handles pan/drag by applying a *TranslateTransform*. The touch event of interest here is the *ManipulationDelta*. This event is raised when the touch input changes position during a manipulation.

```
<Image HorizontalAlignment="Center" VerticalAlignment="Center"
    Name="myImage" Source="TestImage.jpg"
    ManipulationDelta="myImage_ManipulationDelta">
    <Image.RenderTransform>
        <TransformGroup>
            <ScaleTransform x:Name="myScaleTransform" />
            <TranslateTransform x:Name="myTranslateTransform" />
        </TransformGroup>
    </Image.RenderTransform>
</Image>
```

Then, in the code, the *ManipulationDelta* event handler sets the scale of the *ScaleTransform* or the X/Y positions of the *TranslateTransform*. You can determine whether the user is pinching/stretching or panning/dragging by the values of the *DeltaManipulation* object: if the *Scale.X* and *Scale.Y* are non-zero, he must be pinching/stretching; if the *Translation.X* and *Translation.Y* are non-zero, he must be panning/dragging. In fact, the user could be doing both types of operation at the same time, so they are not treated as mutually exclusive in the code.

```
private void myImage_ManipulationDelta(object sender, ManipulationDeltaEventArgs e)
{
    if (e.DeltaManipulation.Scale.X != 0 || e.DeltaManipulation.Scale.Y != 0)
    {
        myScaleTransform.ScaleX *= e.DeltaManipulation.Scale.X;
        myScaleTransform.ScaleY *= e.DeltaManipulation.Scale.X;
    }

    myTranslateTransform.X += e.DeltaManipulation.Translation.X;
    myTranslateTransform.Y += e.DeltaManipulation.Translation.Y;
}
```

The application uses only the *X* property to modify the scale factor, which ensures that you keep the aspect ratio constant. However, there's nothing in the code to prevent the user from shrinking the image beyond the point at which he could get two fingers on it to enlarge it again; nor do you prevent him from moving it off the screen such that he can't get it back; nor do you prevent him flipping the image around the *X* or *Y* axis. In a more sophisticated solution, you might also want to accelerate the pan, based on the zoom factor. Finally, note that you could use a *CompositeTransform* in place of the scale and translate transforms. This is a more streamlined approach, which offers the same functionality.

```
<Image.RenderTransform>
  <!--<TransformGroup>
    <ScaleTransform x:Name="myScaleTransform" />
    <TranslateTransform x:Name="myTranslateTransform" />
  </TransformGroup-->
  <CompositeTransform x:Name="myCompositeTransform" />
</Image.RenderTransform>

private void myImage_ManipulationDelta(object sender, ManipulationDeltaEventArgs e)
{
    if (e.DeltaManipulation.Scale.X != 0 || e.DeltaManipulation.Scale.Y != 0)
    {
        //myScaleTransform.ScaleX *= e.DeltaManipulation.Scale.X;
        //myScaleTransform.ScaleY *= e.DeltaManipulation.Scale.X;
        myCompositeTransform.ScaleX *= e.DeltaManipulation.Scale.X;
        myCompositeTransform.ScaleY *= e.DeltaManipulation.Scale.X;
    }

    //myTranslateTransform.X += e.DeltaManipulation.Translation.X;
    //myTranslateTransform.Y += e.DeltaManipulation.Translation.Y;
    myCompositeTransform.TranslateX += e.DeltaManipulation.Translation.X;
    myCompositeTransform.TranslateY += e.DeltaManipulation.Translation.Y;
}
```

Mouse Events

At a slightly higher level, simple one-finger user input can be handled by handling the *MouseXXX* events: *MouseEnter*, *MouseLeave*, *MouseLeftButtonDown*, *MouseLeftButtonUp*, *MouseMove*, and *MouseWheel*. On the emulator, these are typically raised as a result of mouse input, whereas on the

device, they are raised as a result of touch input. For each of these events, the handler is passed a *MouseEventArgs*, which is a routed event (see Chapter 2, “UI Core,” for a discussion on routed events).

The following simple application exercises mouse events at various levels in the visual tree. The application is a *MouseXXX* version of the earlier *Manipulation* example. It displays a *TextBlock*, which when touched by the user causes the font size to increase; when she touches anywhere else on the page, you’ll decrease the font size. This is the *TestMouse* solution in the sample code.

Examine the XAML for this application. At the innermost scope, you declare a *MouseLeftButtonDown* handler for the *TextBlock* itself. Working outward, there is another *MouseLeftButtonDown* handler at the parent *ContentPanel* scope, and another at the *Page* scope. Note that if you want touch gestures to be registered on *Panel* controls, you must remember to set the *Background* explicitly to *Transparent* (or any other color). Both *Grid* controls in this example have this property set.

```
<phone:PhoneApplicationPage
...
    MouseLeftButtonDown="PhoneApplicationPage_MouseLeftButtonDown">

    <phone:PhoneApplicationPage.Resources>
        <SolidColorBrush Color="#FFF09609" x:Name="orangeBrush"/>
        <SolidColorBrush Color="#FF8CBF26" x:Name="limeBrush"/>
    </phone:PhoneApplicationPage.Resources>

    <Grid x:Name="LayoutRoot" Background="Transparent">
        <StackPanel x:Name="TitlePanel">
            <TextBlock x:Name="ApplicationTitle" Text="CONTOSO"/>
            <TextBlock x:Name="PageTitle" Text="test mouse"/>
        </StackPanel>
        <Grid
            x:Name="ContentPanel" Background="Transparent"
            MouseLeftButtonDown="ContentPanel_MouseLeftButtonDown">
            <TextBlock
                Name="HelloText" Text="Hello World"
                MouseLeftButtonDown="HelloText_MouseLeftButtonDown"/>
            </Grid>
        </Grid>
    </phone:PhoneApplicationPage>
```

The *TextBlock*-level handler is implemented to increment the *FontSize*, and the *ContentPanel*-level handler is implemented to decrement the *FontSize*. In both cases, you set *MouseButtonEventArgs.Handled* to *true*. If you don’t prevent onward routing, this event would be handled again by any handler at the next outer scope. In this example, if you don’t set *Handled=true* in the *TextBlock*-scoped handler, then the event would be onward routed to the *ContentPanel* handler. The net effect would be that the text reverts to the previous size, which to the user would look as if it never changed.

```
private void HelloText_MouseLeftButtonDown(object sender, MouseButtonEventArgs e)
{
    HelloText.FontSize++;
    e.Handled = true;
}
```

```
private void ContentPanel_MouseLeftButtonDown(object sender, MouseButtonEventArgs e)
{
    HelloText.FontSize--;
    e.Handled = true;
}
```

The XAML also defines a couple of *Brush* resources—note that these are defined with a *Name* instead of a *Key*. The advantage of this is that the XAML parser will generate class members for these resources automatically. The *TextBlock* is initialized to use the orange brush. In the *MouseLeftButtonDown* handler defined at page scope, you toggle the brush.

```
public MainPage()
{
    InitializeComponent();

    HelloText.Foreground = orangeBrush;
}

private void PhoneApplicationPage_MouseLeftButtonDown(object sender, MouseButtonEventArgs e)
{
    if (HelloText.Foreground == orangeBrush)
    {
        HelloText.Foreground = limeBrush;
    }
    else
    {
        HelloText.Foreground = orangeBrush;
    }
}
```

You're effectively modeling the logical tap gesture with just one mouse event. To be more correct, you should also handle the *MouseLeftButtonUp* event so that you can distinguish between different manipulations. With a little effort, you could use the *MouseLeftButtonDown*, *MouseMove*, and *MouseLeftButtonUp* events to model the user performing drag or flick gestures. These events are not, however, sufficient to model the pinch/stretch multi-touch gestures. Even for single-touch gestures, you'd have to handle the relevant events both within the element of interest and outside of it, as well.

This is because although the mouse down/up and enter/leave events occur in pairs, a given visual element doesn't necessarily receive paired events. For example, when the user holds the touch over the *TextBlock*, this raises a *MouseEnter* event, followed by *MouseLeftButtonDown*. If she continues to hold the touch and move her finger, this will raise *MouseMove* events—another event every time she moves more than a few pixels. If she leaves the area of the *TextBlock* (still keeping the touch on), this will raise a *MouseLeave* event, but no further *MouseMove* events while she's outside of the *TextBlock*. If she moves back into the *TextBlock* area (still keeping the touch on), this will raise a *MouseEnter*, followed by further *MouseMove* events. If instead, she keeps the touch on and leaves the area, but then releases the touch, there will be a *MouseLeftButtonUp* event—but for the *ContentPanel* (or the page), not for the *TextBlock*—because this action happened outside of the *TextBlock* area.

Note that the Silverlight *Control* class (from which most visual elements derive) provides virtual methods *OnMouseEnter*, *OnMouseLeftButtonDown*, and so on. So, instead of declaring event handlers directly, you could instead override any of these methods. The *OnMouseXXX* methods will always be

called after the corresponding event handlers. Another difference is that while all the raw *MouseXXX* events are cancelable routed events, only some of the *OnMouseXXX* overrides are cancelable events; some are passed a *MouseEventArgs* parameter, which does not expose a *Handled* property because it is not needed. In fact, the *MouseButtonEventArgs* type that the raw event handlers receive is actually derived from *MouseEventArgs*, and the only additional feature it exposes is the *Handled* property.



Note Even though there is a raw *MouseWheel* event and an *OnMouseWheel* virtual method in the *Control* class, you will never receive this event in a Windows Phone application. This is one of those things in the Silverlight *Control* class that is a redundant artifact of adapting desktop Silverlight for Windows Phone.

FrameReported Events

At a higher level, you can handle the *FrameReported* event. This is not applicable to individual visual elements. Rather, it is application-wide; the name does not refer to the Phone *ApplicationFrame* but instead to a discrete “frame” of input in a sequence of events. It is therefore also not a routed event and doesn’t traverse the visual tree. You handle this event anywhere in the application. You would typically put the code in one or more of your pages, because although this is application-wide, the subsequent processing is very much UI-related. The *FrameReported* event is exposed from the static *Touch* class. If you want to handle this event, you must hook it up in code, not XAML, because the XAML parser doesn’t support statics. The code that follows (the *TestFrameReported* solution in the sample code) is for an application that performs the same behavior as the previous mouse event example; that is, incrementing/decrementing the size of a *TextBlock*, and changing its foreground color.

In this version, you implement the *FrameReported* event handler to obtain the collection of touch points. You’re actually not interested in multi-touch here, so you only care about the first touch point in the collection. You can examine the *TouchDevice* property of the *TouchPoint* to determine which visual element the touch was directly over. Note that the *TouchDevice* does not refer to a physical device; rather, it refers to a touch gesture instance, for which each gesture instance consists of a group that can include a touch-down operation, and possibly also includes one or more touch-move operations and a touch-up operation.

```
public MainPage()
{
    InitializeComponent();
    Touch.FrameReported += new TouchFrameEventHandler(Touch_FrameReported);
}

private void Touch_FrameReported(object sender, TouchFrameEventArgs e)
{
    TouchPointCollection tpc = e.GetTouchPoints(this);
    TouchPoint tp = tpc.FirstOrDefault<TouchPoint>();
    if (tp.TouchDevice.DirectlyOver == HelloText)
```

```

    {
        HelloText.FontSize++;
    }
    else if (tp.TouchDevice.DirectlyOver == ContentPanel)
    {
        HelloText.FontSize--;
    }
    else
    {
        if (HelloText.Foreground == orangeBrush)
        {
            HelloText.Foreground = limeBrush;
        }
        else
        {
            HelloText.Foreground = orangeBrush;
        }
    }
}
}

```

Note that because this example cares only about the first touch point in the collection, you could use *GetPrimaryTouchPoint* instead of *FirstOrDefault*. However, although you don't make use of it in this application, you could examine multiple *TouchPoints* within the *TouchPointCollection* to work with multi-touch operations. Also note that the *TouchPoint* exposes both a *Position* and *Size* property. The *Position* provides the X,Y coordinates of the center of the *TouchPoint*, relative to the object passed in to the *GetXXXPoint* method. This allows for changes in orientation. In Windows Phone, the *Size* will always be 1x1 pixels—on the emulator, using a mouse, and on the device, regardless of how fat your fingers are. Also note that the technique of using *DirectlyOver* works nicely with a *TextBlock* because this is a primitive type; however, it won't work with more complex templated types such as the *Button* control.

Combining Manipulation and Mouse Events

It is appropriate to work with the different levels of touch event in different scenarios. It is not normally useful to combine events from more than one level. However, there's nothing to prevent you from doing this, and it is instructive to see how the various events at one level correspond to related events at another level. In the following example (the *MouseAndManipulation* solution in the sample code), you take the *TextBlock*-sizing application a step further by introducing handlers for both *ManipulationXXX* events and *MouseXXX* events. For both *ManipulationStarted* and *MouseLeftButtonDown* on the *TextBlock* itself, you increment the *FontSize*. For the same events on the parent panel, you decrement the *FontSize*.

```

private void HelloText_MouseLeftButtonDown(object sender, MouseButtonEventArgs e)
{
    HelloText.FontSize++;
    e.Handled = true;
}

```

```

private void HelloText_ManipulationStarted(object sender, ManipulationStartedEventArgs e)
{
    HelloText.FontSize++;
    e.Complete();
    e.Handled = true;
}

private void ContentPanel_MouseLeftButtonDown(object sender, MouseButtonEventArgs e)
{
    HelloText.FontSize--;
    e.Handled = true;
}

private void ContentPanel_ManipulationStarted(object sender, ManipulationStartedEventArgs e)
{
    HelloText.FontSize--;
    e.Complete();
    e.Handled = true;
}

```

Both *ManipulationStarted* and *MouseLeftButtonDown* events are raised, in that order. Setting *e.Handled* and/or calling *Complete* makes no difference to this. That is, *Handled* affects the onward routing up the visual tree; it does not affect the way the system internally raises both *ManipulationXXX* and *MouseXXX* events for the same physical touch input. As far as the application is concerned, the two event schemes are independent of each other. Note also that this means that in this example, the *FontSize* will be incremented or decremented twice for each tap. You can confirm this visually by simply setting the value of the *Text* property to the current *FontSize* by using *Self* data binding.

```

<Grid
    x:Name="ContentPanel" Background="Transparent"
    MouseLeftButtonDown="ContentPanel_MouseLeftButtonDown"
    ManipulationStarted="ContentPanel_ManipulationStarted"
    >
    <TextBlock
        Name="HelloText"
        HorizontalAlignment="Center" VerticalAlignment="Center"
        FontSize="{StaticResource PhoneFontSizeExtraExtraLarge}"
        Foreground="{StaticResource OrangeBrush}"
        MouseLeftButtonDown="HelloText_MouseLeftButtonDown"
        ManipulationStarted="HelloText_ManipulationStarted"
        DataContext="{Binding RelativeSource={RelativeSource Self}}"
        Text="{Binding FontSize}"
    />
</Grid>

```

Instead of simple “hello world” text, you data-bind the *Text* property to the *FontSize* of the current element. To do this, use the *{Binding RelativeSource}* syntax, specifying *{RelativeSource Self}*. Set the *Text* to *{Binding FontSize}*. Be aware that this must be done after setting the *FontSize* itself, so the ordering of attribute assignment in the XAML is important.

Click vs. Mouse/Manipulation Events

Many controls—such as *Button*, *CheckBox*, *RadioButton*, and so on—handle the *MouseXXX* events internally and then surface them as *Click* events for the application to consume. This is part of the package you buy into when you use any standard control: life is made easier for you but at the cost of flexibility. If you use a standard *Button*, you can handle the *Click* event, but you will not receive *MouseXXX* events, nor will your *OnMouseXXX* overrides be called. However, you can handle the *ManipulationXXX* events for the *Button* (either instead of or in addition to the *Click* event). You can always handle the *FrameReported* event, because this is application-wide and will be raised even for controls that swallow the *MouseXXX* events. You can see this at work in the *MouseClick* solution in the sample code. Also, you can use the *AddHandler(..., true)* approach to opt in to events that have already been handled.

The Silverlight Toolkit *GestureService*

The Silverlight Toolkit includes a *GestureService*. You use this service by attaching a *GestureListener* to the element(s) for which you want to retrieve gesture events. The *GestureListener* exposes events that map directly to the logical user gestures, as summarized in Table 5-3. The Event Sequence column indicates how the lower-level *MouseXXX* events are handled internally by the *GestureService* to synthesize higher-level user gestures.

TABLE 5-3 Silverlight Toolkit Gestures

Gesture	Event Sequence	Notes
Tap	<i>ManipulationStarted</i> <i>MouseLeftButtonDown</i> <i>ManipulationCompleted</i> <i>GestureBegin</i> <i>MouseLeftButtonUp</i> <i>Tap</i> <i>GestureCompleted</i>	Finger down and up again within 1.2 seconds, without moving more than a few pixels. The <i>GestureService</i> picks up the low-level <i>MouseLeftButtonDown</i> event, signals that some kind of gesture is beginning, and then recognizes the <i>MouseLeftButtonUp</i> within the specified time, which signifies a <i>Tap</i> , completing the gesture.
DoubleTap	<i>ManipulationStarted</i> <i>MouseLeftButtonDown</i> <i>GestureBegin</i> <i>ManipulationCompleted</i> <i>MouseLeftButtonUp</i> <i>Tap</i> <i>GestureCompleted</i> <i>ManipulationStarted</i> <i>MouseLeftButtonDown</i> <i>GestureBegin</i> <i>DoubleTap</i> <i>ManipulationCompleted</i> <i>MouseLeftButtonUp</i> <i>GestureCompleted</i>	The first <i>Tap</i> is recognized as a single <i>Tap</i> , but if it is followed within one second by a second discrete <i>Tap</i> , this is then modeled as a <i>Double Tap</i> .

Gesture	Event Sequence	Notes
<i>Flick</i>	<i>ManipulationStarted MouseLeftButtonDown GestureBegin MouseMove ManipulationDelta DragStarted DragDelta MouseLeftButtonUp Flick DragCompleted ManipulationCompleted GestureCompleted</i>	Finger down, finger move, and then finger up while still moving. It is common to treat the <i>Flick</i> as a natural extension of a <i>Drag</i> gesture. You can get a <i>Drag</i> without a <i>Flick</i> , but it is unusual to get a <i>Flick</i> without a previous <i>Drag</i> .
<i>Hold</i>	<i>ManipulationStarted MouseLeftButtonDown GestureBegin Hold</i>	Finger down for >1.2 seconds.
<i>GestureBegin, GestureCompleted</i>		Corresponds to the first finger down, and the last finger up. This pair of events will “book-end” all other high-level gesture types.
<i>DragStarted, DragDelta, DragCompleted</i>	<i>ManipulationStarted MouseLeftButtonDown GestureBegin ManipulationDelta MouseMove DragStarted DragDelta ManipulationDelta MouseMove DragDelta ManipulationDelta MouseMove ... ManipulationCompleted MouseLeftButtonUp DragCompleted GestureCompleted</i>	The sequence begins when the low-level events are recognized as <i>DragStarted</i> —when the touch moves beyond the threshold for a simple <i>Tap</i> . This is then followed by one or more pairs of <i>MouseMove/ManipulationDelta/DragDelta</i> events. This continues so long as the drag continues. Lifting the touch is recognized as <i>DragCompleted</i> .
<i>PinchStarted, PinchDelta, PinchCompleted</i>	<i>ManipulationStarted MouseLeftButtonDown GestureBegin MouseLeftButtonUp Tap GestureCompleted ManipulationDelta MouseLeftButtonDown MouseMove GestureBegin PinchStarted PinchDelta MouseMove PinchDelta PinchDelta PinchDelta PinchDelta PinchDelta PinchDelta ManipulationDelta MouseMove ... ManipulationCompleted PinchCompleted GestureCompleted</i>	Once pinching has started, there can be multiple <i>PinchDelta</i> events for each <i>MouseMove</i> . As with <i>Flick</i> , a <i>Drag</i> can sometimes evolve into a <i>Pinch</i> (when a second finger touches down). Similarly, a <i>Pinch</i> can turn into a <i>Drag</i> , if one of the fingers is raised. <i>Pinch</i> and <i>Drag</i> never interleave, but can run back to back.

Pinch and Drag

The following example is a variation on the earlier pinch-and-drag application, but instead of using the *ManipulationXXXEventArgs*, you're now working with more specific *Gesture* events. This is the *TestGestureService* solution in the sample code. To set this up, you first attach a *GestureListener* to the *Image* control in the page XAML.

```
<Image HorizontalAlignment="Center" VerticalAlignment="Center"
    Name="myImage" Source="TestImage.jpg">
    <Image.RenderTransform>
        <TransformGroup>
            <ScaleTransform x:Name="myScaleTransform" />
            <TranslateTransform x:Name="myTranslateTransform" />
        </TransformGroup>
    </Image.RenderTransform>
    <toolkit:GestureService.GestureListener>
        <toolkit:GestureListener x:Name="gl"
            PinchStarted="gl_PinchStarted"
            PinchDelta="gl_PinchDelta"
            DragStarted="gl_DragStarted"
            DragDelta="gl_DragDelta"/>
    </toolkit:GestureService.GestureListener>
</Image>
```

In the code-behind, you implement the selected event handlers as follows:

- Implement the *PinchStarted* event by caching the initial X-scale value of the *ScaleTransform* (that is, the *Width* of the *Image* before the user starts the pinch operation).
- Implement *PinchDelta* by updating the *ScaleX/ScaleY* values of the *ScaleTransform* according to the *DistanceRatio* value in the event arguments.
- Implement *DragStarted* by caching the initial X/Y values of the *TranslateTransform* (that is, the X/Y positions of the *Image* before the user starts the drag operation).
- Implement *DragDelta* by updating the X/Y values of the *TranslateTransform* according to the *HorizontalChange* provided in the event arguments.

```
public partial class MainPage : PhoneApplicationPage
{
    private double initialScale;
    private double initialPosX;
    private double initialPoxY;

    public MainPage()
    {
        InitializeComponent();
    }

    private void gl_PinchStarted(object sender, PinchStartedGestureEventArgs e)
    {
        initialScale = myScaleTransform.ScaleX;
    }
}
```

```

private void gl_PinchDelta(object sender, PinchGestureEventArgs e)
{
    myScaleTransform.ScaleX = initialScale * e.DistanceRatio;
    myScaleTransform.ScaleY = myScaleTransform.ScaleX;
}

private void gl_DragStarted(object sender, DragStartedGestureEventArgs e)
{
    initialPosX = myTranslateTransform.X;
    initialPoxY = myTranslateTransform.Y;
}

private void gl_DragDelta(object sender, DragDeltaGestureEventArgs e)
{
    myTranslateTransform.X += e.HorizontalChange;
    myTranslateTransform.Y += e.VerticalChange;
}
}

```

Flick and Tap

What follows is a *GestureListener* version of the earlier bouncing ball application, in which the ball's movement is controlled by the user with *Flick* and *Tap* gestures. This is the *BouncingBall* solution in the sample code. But instead of hooking up a *ManipulationCompleted* event handler for the *Ellipse*, you attach a *GestureListener*, with event handlers for the *Flick* and *Tap* events, as follows:

```

<Ellipse
  x:Name="ball" Canvas.Left="368" Canvas.Top="200" Width="80" Height="80" >
  <Ellipse.Fill>
    <RadialGradientBrush GradientOrigin="0.25,0.25">
      <RadialGradientBrush.GradientStops>
        <GradientStop Color="White" Offset="0"/>
        <GradientStop Color="#FF339933" Offset="1"/>
      </RadialGradientBrush.GradientStops>
    </RadialGradientBrush>
  </Ellipse.Fill>
  <toolkit:GestureService.GestureListener>
    <toolkit:GestureListener
      x:Name="gl" Flick="gl_Flick" Tap="gl_Tap"/>
  </toolkit:GestureService.GestureListener>
</Ellipse>
</Canvas>

```

The *HorizontalVelocity* and *VerticalVelocity* are exposed in a more convenient form from the *FlickGestureEventArgs*.

```

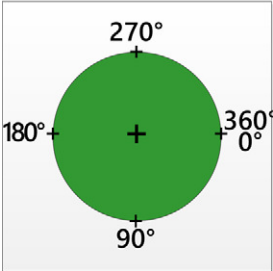
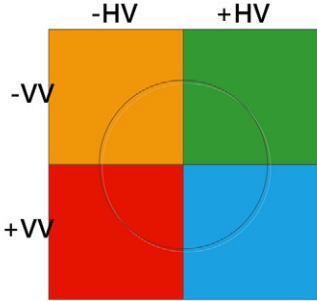
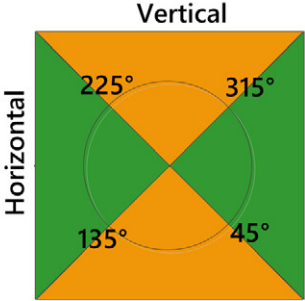
private void gl_Flick(object sender, FlickGestureEventArgs e)
{
    horizontalVelocity = (int)(e.HorizontalVelocity / 100);
    verticalVelocity = (int)(e.VerticalVelocity / 100);
    timer.Start();
}

```

```
private void gl_Tap(object sender, GestureEventArgs e)
{
    timer.Stop();
}
```

A summary of the most interesting properties exposed by *GestureListener* events is shown in Table 5-4.

TABLE 5-4 *GestureListener* Event Properties

Origin and Angle	Velocity	Direction
Most events expose an <i>Origin</i> property. <i>Flick</i> and <i>PinchXXX</i> expose an <i>Angle</i> , measured clockwise from 3 o'clock.	<i>Flick</i> and <i>DragXXX</i> expose <i>HorizontalVelocity</i> and <i>VerticalVelocity</i> , where positive horizontal is right from the origin, and positive vertical is down.	<i>Flick</i> and <i>DragXXX</i> also expose a <i>Direction</i> , where vertical is in the ranges 45°–135° and 225°–315°, and horizontal is in the ranges 135°–225° and 315°–45°.
		

Problems with the *GestureService*

The Toolkit’s *GestureService* is very useful and plugs a few gaps in the application platform itself. Specifically, without the *GestureService*, it is difficult in version 7.0 to distinguish a tap touch gesture from a tap-and-hold gesture, or from the start of a drag or other operation. The situation improves significantly in version 7.1, however. Also, although it is relatively easy to model drag and flick gestures with the *ManipulationXXX* events, it is considerably more work to model pinch and stretch.

However, even though it is definitely easier to work with the Toolkit’s *GestureService*, it is not without its drawbacks.

- The *GestureService* in the Toolkit is based on the XNA *TouchPanel*. In fact, XNA *TouchPanel* provides a very good set of gesture support for XNA games. However, bringing in XNA code means bringing in a very different model from the rest of the application that is using Silverlight. The *GestureService* does not play well with the rest of the input system for a Silverlight application and can cause performance issues.
- The type *GestureEventArgs* is defined both in the standard Silverlight *System.Windows.dll* and in the Toolkit’s *Microsoft.Phone.Controls.Toolkit.dll*. The main differences are that the Toolkit version provides a richer API but is not a routed event, whereas the Windows version is simpler

and routed. What's important here is that there is potential for a name clash, and you must be sure to be explicit which version you're using so as to avoid compiler errors—or worse, runtime exceptions—if you implicitly use the wrong one.

- There's the size of the Toolkit to consider. The Toolkit assembly itself adds 116 KB to your XAP size. This is acceptable if you're using several pieces of the Toolkit (perhaps some of the controls, in addition to the gesture service); if not, you should think hard about using it, if all you want is the gesture support. Also, because the Toolkit *GestureService* is based on the XNA *TouchPanel*, using the Toolkit also pulls in *Microsoft.Xna.Framework.dll* and *Microsoft.Xna.Framework.Input.Touch.dll* at runtime, which is another 700 KB overhead. Although this doesn't affect your XAP size, it does affect your working set size in memory.
- The Toolkit is unsupported and changes over time. Indeed, over time, pieces of the Toolkit find themselves getting rolled into the application platform itself. This is a good thing, and it allows Microsoft to gather feedback, use-cases, and priorities from users of the Toolkit as to what is important and worthwhile to include in the product. Another benefit is that the Toolkit is supplied in both binary and source-code format, so you can adapt the sources, if necessary. On the other hand, all the usual caveats apply when considering the use of unsupported code in production applications.

Behaviors

The behaviors feature in Expression Blend is primarily intended to support designers who want to add functionality to XAML elements without coding. Typically, a developer would create some custom behaviors in code, and then hand off the assembly (or just the source code) to the designer working in Expression Blend. The custom behaviors would show up in the Blend environment, allowing the designer to attach them to visual elements without touching the code.

So, if you want a high-level gesture abstraction, but you don't want to use the Toolkit *GestureService*, then one solution is to build custom behaviors that synthesize the high-level logical gestures from low-level *ManipulationXXX* or *MouseXXX* events. Using behaviors is a deliberately simple technique, but creating custom behaviors is a more advanced topic. In this section, you'll see how to create a custom behavior that mimics one of the standard Blend behaviors.

In this example (*TestBehaviors* in the sample code), the application displays an image in an *Image* control and responds to only one gesture: the *Drag* gesture, which is a custom behavior attached to the *Image* control in XAML. Recall that for the Toolkit *GestureService*, you can handle the drag manipulations by hooking up the *DragDelta* and *DragCompleted* events.

```
<Image
  Name="myImage" Source="TestImage.jpg">
  <Image.RenderTransform>
    <TranslateTransform x:Name="myTranslateTransform" />
  </Image.RenderTransform>
  <toolkit:GestureService.GestureListener>
```

```

        <toolkit:GestureListener
            x:Name="gl"
            DragDelta="gl_DragDelta"
            DragCompleted="gl_DragCompleted"
        />
    </toolkit:GestureService.GestureListener>
</Image>

```

You then implement the handler in the code-behind to perform the drag operation by applying the *TranslateTransform*.

```

private void gl_DragDelta(object sender, DragDeltaGestureEventArgs e)
{
    myTranslateTransform.X += e.HorizontalChange;
    myTranslateTransform.Y += e.VerticalChange;
}

```

You want to achieve a similar developer experience with your custom behavior. To attach it in XAML, you first need to declare an XML namespace for the *System.Windows.Interactivity.dll*—this is where the *Behaviors* types are defined.

```

xmlns:interact="clr-namespace:System.Windows.Interactivity;assembly=System.Windows.
Interactivity"
<Image
    Name="myImage" Source="TallPalms.jpg">
    <Image.RenderTransform>
        <TranslateTransform x:Name="myTranslateTransform" />
    </Image.RenderTransform>
    <interact:Interaction.Behaviors>
        <local:TouchGesture
            x:Name="DragBehavior"
            Drag="DragBehavior_Drag"/>
    </interact:Interaction.Behaviors>
</Image>

```

The *TouchGesture* type that is attached to the *Image* control is a custom *Behavior* defined in the application itself. Here's how this is built up. First, you define a class derived from *Behavior<T>*. *Behavior<T>* is the generic version of the base class for providing attachable state and commands to an object. The object to which this behavior is attached at runtime is provided in the *AssociatedObject* property. In this example, you want to be able to attach this behavior to any *UIElement* or derivative, so you derive from *Behavior<UIElement>*. You must override the *OnAttached* and *OnDetaching* methods to hook and unhook any necessary handlers from the *AssociatedObject*. In this example, you hook and unhook the *MouseLeftButtonDown*, *MouseMove*, and *MouseLeftButtonUp*. You'll need all three of these to be able to synthesize a *Drag* operation.

```

public class TouchGesture : Behavior<UIElement>
{
    protected bool isMouseDown { get; set; }

    protected override void OnAttached()
    {
        base.OnAttached();
        AssociatedObject.MouseLeftButtonDown += AssociatedObject_MouseLeftButtonDown;
    }
}

```

```

        AssociatedObject.MouseMove += AssociatedObject_MouseMove;
        AssociatedObject.MouseLeftButtonUp += AssociatedObject_MouseLeftButtonUp;
    }

    protected override void OnDetaching()
    {
        base.OnDetaching();
        AssociatedObject.MouseLeftButtonUp -= AssociatedObject_MouseLeftButtonUp;
        AssociatedObject.MouseMove -= AssociatedObject_MouseMove;
        AssociatedObject.MouseLeftButtonDown -= AssociatedObject_MouseLeftButtonDown;
    }

    private void AssociatedObject_MouseLeftButtonDown(object sender, MouseButtonEventArgs e)
    {
        lastPositionX = e.GetPosition(null).X;
        lastPositionY = e.GetPosition(null).Y;
        isMouseDown = true;
    }

    private void AssociatedObject_MouseLeftButtonUp(object sender, MouseButtonEventArgs e)
    {
        isMouseDown = false;
    }
    ... continued in the next code example.
}

```

You do nothing unless and until the mouse is down; at this point, you simply cache a flag to indicate that the mouse is down. When the mouse is raised again, you toggle this flag. In the meantime, if you get *MouseMove* events, you know the user must be dragging. All of the interesting work is done in the *MouseMove* handler: here, you extract the position data from the incoming *MouseEventArgs*, compute the change in position, and raise your custom *Drag* event (which you're expecting the consuming application to hook up).

```

public class TouchGesture : Behavior<UIElement>
{
    ... previously-listed code omitted for brevity.

    public event EventHandler<DragEventArgs> Drag;
    private double lastPositionX;
    private double lastPositionY;

    private void AssociatedObject_MouseMove(object sender, MouseEventArgs e)
    {
        if (isMouseDown)
        {
            Point position = e.GetPosition(null);
            double hChange = position.X - lastPositionX;
            double vChange = position.Y - lastPositionY;
            lastPositionX = position.X;
            lastPositionY = position.Y;
            OnDrag(new DragEventArgs(hChange, vChange));
        }
    }
}

```

```

private void OnDrag(DragEventArgs e)
{
    if (Drag != null)
    {
        Drag(AssociatedObject, e);
    }
}
}

```

In standard event form, the first argument to the custom *Drag* event is the sending object (in this case, the *AssociatedObject* property of this behavior). The second argument is a custom *DragEventArgs*, which allows you to pass the horizontal and vertical change data. Note that the *Drag* event is declared by using the generic *EventHandler<T>*, specifically so that you can define it to use this custom *DragEventArgs* type.

```

public class DragEventArgs : EventArgs
{
    public double HorizontalChange { get; set; }
    public double VerticalChange { get; set; }

    public DragEventArgs(double h, double v)
    {
        HorizontalChange = h;
        VerticalChange = v;
    }
}

```

Finally, in the page code for the application, you implement the *Drag* event handler by applying the *TranslateTransform* in almost exactly the same way as you did previously with the Toolkit *GestureService*.

```

private void DragBehavior_Drag(object sender, DragEventArgs e)
{
    myTranslateTransform.X += e.HorizontalChange;
    myTranslateTransform.Y += e.VerticalChange;
}

```

So, synthesizing your own high-level gesture events from lower-level *MouseXXX* or *ManipulationXXX* events is a valid alternative to using the Toolkit *GestureService*. You should be aware that although this example provides a custom implementation of drag manipulation handling, there is in fact a standard behavior shipped with Expression Blend that performs the same functionality. You can see this at work in the *TestBehaviors_Standard* solution in the sample code. To use this feature in a Visual Studio project, you need to add a reference to *Microsoft.Expressions.Interaction.dll* and add a corresponding XML namespace declaration.

```

xmlns:ilayout="clr-namespace:Microsoft.Expression.Interactivity.Layout;assembly=Microsoft.
Expression.Interactions"

```

Then, for the *Image* element in your XAML, in the *Interaction.Behaviors* child element, replace the custom behavior with the standard *MouseDragElementBehavior*, and then hook up a handler for the *Dragging* event.


```
<interact:Interaction.Behaviors>
  <ilayout:MouseDragElementBehavior Dragging="MouseDragElementBehavior_Dragging" />
</interact:Interaction.Behaviors>
```

The *Dragging* event handler would then have effectively the same implementation as the previous custom handler.

```
private void MouseDragElementBehavior_Dragging(object sender, MouseEventArgs e)
{
    myTranslateTransform.X += e.GetPosition(myImage).Y;
    myTranslateTransform.Y += e.GetPosition(myImage).X;
}
```

If there is a standard behavior that you can use, this would obviously save a lot of work. In fact, it gets even easier: recall that the behaviors feature is a Blend feature, and therefore, it's intended for use by designers, not developers. So, you can in fact use a standard behavior without any code. You would use the Blend UI to generate the XAML for you, and that's all you need to do. In Visual Studio, you only need to add the declaration of the behavior itself; you do not need to specify events. Of course, without a declared event, you also do not need an event handler in the code at all. So, all the preceding work could be replaced with this one XAML declaration:

```
<interact:Interaction.Behaviors>
  <ilayout:MouseDragElementBehavior />
</interact:Interaction.Behaviors>
```

Keyboard Input

A hardware keyboard is an optional component in the standard Windows Phone 7 chassis specification. On the other hand, the Software Input Panel (SIP) is a part of the platform itself, and is therefore available on all Windows Phone 7 devices. The SIP was also previously known as the On-Screen Keyboard (OSK). Just to complete the list of TLAs, the auto-correct and word suggestion features are collectively known as the Input Method Editor (IME). The SIP is popped up at appropriate times on your behalf—such as when the user taps an editable *TextBox*. When displayed, the SIP pushes your visual elements up and off the top of the screen. Exactly how much space it takes up on the screen depends on whether it is in portrait or landscape orientation, whether the word suggestions feature is engaged, the specific *InputScope* that you're using, and whether there's anything in the clipboard. In landscape orientation, the SIP occupies more space horizontally but less space vertically (because there is less vertical real estate overall in landscape). It's also important to remember that even if you detect that a hardware keyboard has been deployed, you can't assume that there's no SIP-related UI on screen, because there might be something in the clipboard, and that requires SIP-related UI.

The Windows Phone SIP actually has a lot of very useful features, including auto-correction, word suggestion, visual and audio feedback, accent key pickers, shift and shift-lock management, compensation for finger shake, and so on. However, these are all user features rather than developer features. Relatively little functionality is exposed to developers. This is partly because there is very little need for developers to program the SIP directly, and partly it is to ensure a consistent user experience (UX) by not enabling arbitrary programmatic manipulation.

You can configure the input scope of the SIP by setting the *InputScope* property on the element (for example, the *TextBox*) itself. There are 62 possible values for *InputScope*, but many of these are synonyms, and they map to 11 distinct modes. Why is this? There are a couple of reasons:

- To allow the developer to declare the intent of the scope. For example, the *CurrencyAmount InputScope* is synonymous with the *Number InputScope*, but the developer can specify *CurrencyAmount* to make his intentions clear.
- To allow for future tuning of the input scopes. For example, right now Bopomofo is synonymous with Default, but in some future release there might be a distinct Bopomofo SIP.

The following application (the *TestSip* solution in the sample code) provides a *ListBox* of the full set of input scopes, and data binds the *TextBox InputScope* to the selected item from that list. Figure 5-4 shows the *Chat* input scope on the left and the *TelephoneNumber* input scope on the right.



FIGURE 5-4 The SIP with *Chat* (left) and *TelephoneNumber* input scopes.

The *ListBox* declaration in XAML is trivial; the *TextBox* declaration includes data binding the *InputScope* property to the currently selected item in the *ListBox* element.

```
<StackPanel x:Name="ContentPanel">
    <ListBox x:Name="ScopeList" Height="300"/>
    <TextBox
        InputScope="{Binding ElementName=ScopeList, Path=SelectedItem}"/>
</StackPanel>
```

To populate the *ListBox*, you need to do a little reflection. Internally, the input scope values are declared as an enum named *InputScopeNameValue* in *System.Windows.dll*. You use *Type.GetFields* to

get each enum value, and extract the *FieldInfo.Name* to populate a simple list, which is then sorted alphabetically and set as the *ItemsSource* of the *ListBox*, as demonstrated here:

```
public MainPage()
{
    InitializeComponent();

    List<string> inputScopeNames = new List<String>();
    FieldInfo[] inputScopeEnumValues = typeof(InputScopeNameValue).GetFields(
        BindingFlags.Public | BindingFlags.Static);
    foreach (FieldInfo fi in inputScopeEnumValues)
    {
        inputScopeNames.Add(fi.Name);
    }
    inputScopeNames.Sort();
    ScopeList.ItemsSource = inputScopeNames;
}
```



Note Tuning the SIP display to the keys that you want the user to use doesn't prevent her from using other keys. If she has a hardware keyboard, or if she uses the clipboard, she can ignore the *InputScope* altogether.

You can emulate a hardware keyboard on the emulator by pressing the Pause key on the computer keyboard. This toggles the keyboard to act as the hardware keyboard for the phone emulator; it remains set until you press Pause again (or restart the emulator). The emulator window must have focus before you do this—you need to click inside the emulated screen (not on the chrome) for the keystroke to work (the same applies for other special keys such as F1 for back, F2 for start, and so on). If you want to filter out unacceptable keystrokes, you can handle the *LostFocus* event to remove unwanted characters after they've been entered. For example, the following code removes anything that's not a numeric digit:

```
private void MyTextBox_LostFocus(object sender, TextChangedEventArgs e)
{
    MyTextBox.Text = Regex.Replace(MyTextBox.Text, "[^0-9]", "");
}
```

The design of the SIP was the result of collaboration between Microsoft Research and the Windows Phone product team. The SIP has to work even if the application constrains itself to portrait mode, wherein the SIP and its individual keys will be relatively small, and with small gaps between keys. Moreover, because everything is so small, it is not possible to use classic “all-finger” typing on a mobile device, and one or two-digit typing is slow and laborious. To mitigate this, users expect predictive word-completion suggestions and auto-completion. Modern smartphones take this a step further by dynamically sizing the keys—or, typically, the touch targets—according to their prediction of the next character. For example, if you type “Targe”, the system will likely predict that the most probable next character is “t”, and will make the touch target for the “t” key larger, and the surrounding keys’ touch targets correspondingly smaller. This helps the user to hit the right key.

In fact, the designers of the Windows Phone 7 keyboard took this yet another step further. They found that although simple key-target resizing does improve typing accuracy in the general case, there are scenarios for which it can be counter-productive. For example, suppose instead of “Target”, you want to type “Targer” (someone’s name). Standard key-target resizing would increase the size of the “t” and reduce the size of its neighbor, “r”, making it even more difficult to enter the key that you want, not easier. What the Microsoft Research and the Windows Phone product team folks came up with is an *anchored* key-sizing approach, wherein each key has a central anchor that is always included in its target area regardless of predictions. Details of this approach are published in the paper *Usability Guided Key-Target Resizing for Soft Keyboards*, which is available at <http://research.microsoft.com/apps/pubs/default.aspx?id=118375>.

As part of its investigations and to fine-tune its design, the team published a game called Text Text Revolution!, which is available on the Windows Phone Marketplace. This game serves a dual purpose of helping users to train themselves to type more accurately and also to provide feedback data to fine-tune the prediction model itself.

Note that a sophisticated application could take a similar approach to improve the usability of the application’s touch input. This is not to suggest that you build your own SIP—that would be a bad idea. Rather, you could use the approach for non-key input, perhaps if you have a number of small buttons, or a number of game sprites positioned close together.

Orientation

If you want to handle changes in device orientation, you first need to set the application’s *Supported Orientations* to *PortraitOrLandscape* (typically in the *mainpage.xaml*), as shown here:

```
<phone:PhoneApplicationPage
...
    SupportedOrientations="PortraitOrLandscape" Orientation="Portrait"
>
```

In this simple example (*TestOrientation* in the sample code), you’ll add a series of image controls, just to see what the UX is as the orientation changes. These are placed in a 2x2 *Grid*. The first row of the *Grid* contains one large image; the second row contains a *StackPanel* with two smaller images stacked vertically. Note that the initial orientation of the application is *Portrait* and that the *StackPanel* is initialized at row 1, column 0 in the *Grid*.

```
<Grid x:Name="ContentPanel" Margin="12,0,12,0">
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto"/>
        <RowDefinition Height="*" />
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="Auto"/>
        <ColumnDefinition Width="*" />
    </Grid.ColumnDefinitions>
```

```

<Image x:Name="Image1" Grid.Row="0" Grid.Column="0" Stretch="Fill"
    HorizontalAlignment="Center" Source="Images/TestImage.jpg"
    Height="310" Width="456"/>

<StackPanel
    x:Name="imagePanel" Grid.Row="1" Grid.Column="0"
    HorizontalAlignment="Center" >
    <Image
        HorizontalAlignment="Center" Stretch="Fill"
        Height="125" Width="175" Source="Images/Coconuts.jpg"/>
    <Image
        HorizontalAlignment="Center" Stretch="Fill"
        Height="125" Width="175" Source="Images/PalmTrees.jpg"/>
</StackPanel>

</Grid>

```

The result is shown in Figure 5-5.

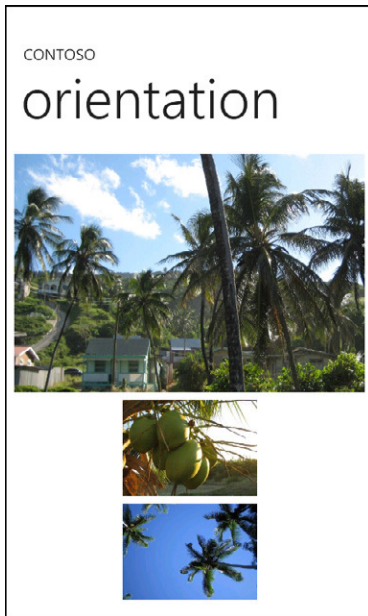


FIGURE 5-5 A layout in portrait orientation.

This looks acceptable in portrait mode. The problem is that when the user turns the phone sideways, you cannot see the images at the bottom, as illustrated in Figure 5-6.



FIGURE 5-6 The same layout in landscape orientation.

One solution is to put everything into a *ScrollView*, as follows:

```
<ScrollView x:Name="ContentGrid" Grid.Row="1" VerticalScrollBarVisibility="Auto">
  <Grid x:Name="ContentPanel" Margin="12,0,12,0">
    ...
  </Grid>
</ScrollView>
```

By doing this, you can scroll the controls into view, as demonstrated in Figure 5-7.

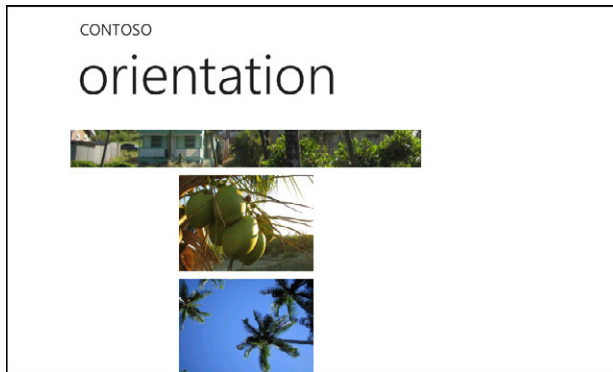


FIGURE 5-7 Landscape orientation and using *ScrollView*.

However, the layout is less aesthetically pleasing when the device is in landscape mode than it is when in portrait. A common solution to this is to handle the *OrientationChanged* event to rearrange the layout. You handle this event at the page level. If the user is changing to *Portrait*, you set the *Grid* row/column values for the *StackPanel* to row 1, column 0. Conversely, if he is changing to *Landscape*, you set them to row 0, column 1.

```
<phone:PhoneApplicationPage
...
  SupportedOrientations="PortraitOrLandscape" Orientation="Portrait"
  OrientationChanged="PhoneApplicationPage_OrientationChanged"
>
```

```

private void PhoneApplicationPage_OrientationChanged(
    object sender, OrientationChangedEventArgs e)
{
    if ((e.Orientation & PageOrientation.Portrait) == (PageOrientation.Portrait))
    {
        Grid.SetRow(imagePanel, 1);
        Grid.SetColumn(imagePanel, 0);
    }
    else
    {
        Grid.SetRow(imagePanel, 0);
        Grid.SetColumn(imagePanel, 1);
    }
}

```

The final result is shown in Figure 5-8.



FIGURE 5-8 Now you have a dynamically changing grid layout.

There are no hard guidelines on which orientation(s) to support in an application, except that if your application includes keyboard input (via SIP or hardware keyboard), you should allow for the user to switch to *Landscape*. The SIP in *Portrait* mode has very small keys and is therefore harder to use than in *Landscape* mode. Conversely, the problem with *Landscape* mode is that it leaves very little space for your *TextBox* and other controls, which makes multi-line editing especially problematic. Also consider that the phone might have a hardware keyboard, which could be *Portrait* or *Landscape*, and it would clearly be quite difficult for the user to use the *Landscape* hardware keyboard if your application supports only *Portrait* mode. It is also rare to support only *Landscape* mode; the common exceptions to this are all games and video-based applications.

If you specify *SupportedOrientations* of *Landscape* only, but at the same time set *Orientation* to *Portrait*, the *SupportedOrientations* is honored and the *Orientation* is ignored, and vice versa. Also, there are actually two *Landscape* modes (*LandscapeLeft* and *LandscapeRight*), which function as you would expect. There is no way to force or disable either one of these. There are also two *Portrait* modes (*PortraitUp* and *PortraitDown*), but only *PortraitUp* is used—there is no “upside-down” *Portrait* mode. Note that there is no programmatic way to switch orientations. You can do this to a degree by updating your *SupportedOrientations* property in code. The device will comply with this, but you cannot force *Left* or *Right*.

Another way to solve this type of layout problem might be to use a *WrapPanel*, which you can get from the Silverlight Toolkit. This is most suitable if you have a list of items that are all the same size, and you want them to re-arrange themselves according to orientation changes. The screenshots in Figures 5-9 and 5-10 illustrate this. This is the *WrapOrientation* solution in the sample code. In *Portrait* mode, the application displays two columns and three rows, whereas in *Landscape* mode, it lays out the elements in three columns and two rows.

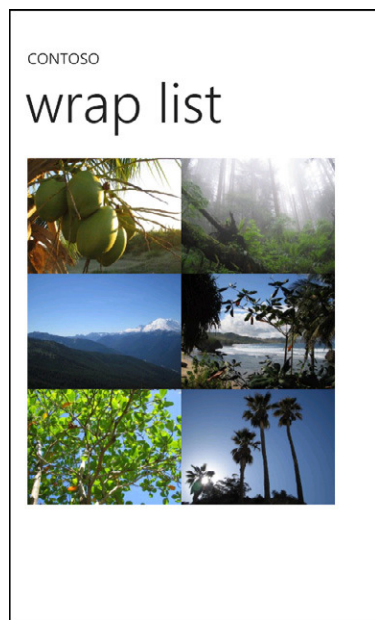


FIGURE 5-9 A *WrapPanel* in portrait mode.



FIGURE 5-10 The same *WrapPanel* in landscape mode.

The application has a list of photos that are displayed in a *ListBox*, which in turn is hosted in a *WrapPanel*.


```

<ListBox
    Margin="{StaticResource PhoneHorizontalMargin}" x:Name="PhotoList">
    <ListBox.ItemsPanel>
        <ItemsPanelTemplate>
            <toolkit:WrapPanel x:Name="PhotoPanel"/>
        </ItemsPanelTemplate>
    </ListBox.ItemsPanel>
    <ListBox.ItemTemplate>
        <DataTemplate>
            <Image
                Width="200" Height="150"
                Source="{Binding}"
                Stretch="UniformToFill" HorizontalAlignment="Left" />
        </DataTemplate>
    </ListBox.ItemTemplate>
</ListBox>

```

The *Source* data for each *Image* is set up as a simple string that refers to an *Image* resource in the assembly, as presented in the following:

```

public partial class MainPage : PhoneApplicationPage
{
    private List<string> photos = new List<string>();

    public MainPage()
    {
        InitializeComponent();
        photos.Add("/Images/Coconuts.jpg");
        photos.Add("/Images/Forest.jpg");
        photos.Add("/Images/Mountain.jpg");
        photos.Add("/Images/Sea.jpg");
        photos.Add("/Images/SunnyTree.jpg");
        photos.Add("/Images/TallPalms.jpg");
        PhotoList.ItemsSource = photos;
    }
}

```

There is no need to handle *OrientationChanged* events manually, because the *WrapPanel* dynamically adjusts its size according to the new *Width* and *Height* of the screen whenever the orientation changes. It also lays out its child elements within the constraints of its overall size and doesn't allow any of them to be clipped or omitted.

The Application Bar

Windows Phone offers an application bar that performs the same role as a menu or toolbar in a desktop application. This is represented by the *ApplicationBar*, *ApplicationBarIconButton*, and *ApplicationBarMenuItem* classes. The Application Bar—more commonly known as the App Bar—is only minimally customizable: you can use up to four buttons (with your own or standard images). You can also have a single, flat menu. These should be short items in a short menu. There's actually no technical limit imposed on the number of items that you can have, but the menu slides up to show

only five items completely. If you have more, the user would need to scroll to see the rest, and this compromises the UX.

An *AppBar* property is exposed from the *PhoneApplicationPage* class, so you can define App Bar items on a per-page basis. Note, however, that this is a virtualization on top of what under the covers is really a singleton object, managed by the Phone shell frame.

Microsoft supplies a suitable set of icons for use in the App Bar, installed as a part of the Windows Phone Developer Tools in %ProgramFiles%\Microsoft SDKs\Windows Phone\v7.0\Icons. You can use these directly or as a base for your own images. Both light and dark-themed versions are provided. You should use the dark-themed version (white images); the application platform will convert on the fly, as needed, if the user changes the theme on the device. The light-themed versions (black images) are provided, but not for use in the App Bar; they are available in case you want to use them elsewhere in your application, outside of the App Bar. The total image size is 48x48 pixels, and the customizable area within that is 26x26 pixels. The light-themed versions are shown in Table 5-6.

TABLE 5-6 Standard App Bar Icons



More Info Several third parties have also made icon sets available (both commercial and free), including:

- <http://yankoa.deviantart.com>
- <http://www.smartypantscoding.com/content/metro-icons-windows-phone-7>
- <http://metro.windowswiki.info/>

You can optionally set the opacity for the App Bar, and if you do so, you should limit yourself to values of 0, 0.5 and 1. If opacity <1, the App Bar will overlay the UI. If opacity=1, the page size is reduced by the 72-pixel size of the App Bar. If you don't provide images, the application will work just fine without images (showing the underlying default "x in a circle" image). If you attempt to provide more than four buttons, or if your buttons do not have text specified, the application will crash on startup.

In the following example (*AppBarAnimator* in the sample code), there is a single *TextBlock* and two App Bar buttons. Figure 5-11 shows that one button rotates the text on the X axis; the other rotates it on the Y axis.



FIGURE 5-11 The App Bar buttons in portrait mode.

When the phone orientation is changed, the App Bar icons are animated so that the images are always displayed in the same relative orientation. This is shown in Figure 5-12.

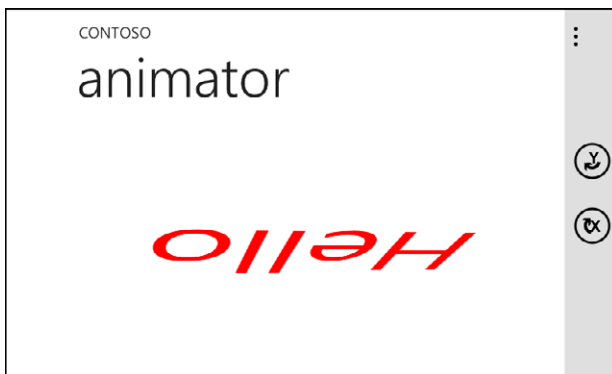


FIGURE 5-12 The same App Bar buttons in landscape mode.

The *TextBlock* specifies a *PlaneProjection* so that this can be the target of animation.

```
<TextBlock Name="helloText" Text="Hello"
    FontSize="{StaticResource PhoneFontSizeHuge}"
    HorizontalAlignment="Center" VerticalAlignment="Center">
  <TextBlock.Projection>
    <PlaneProjection x:Name="textPlane" />
  </TextBlock.Projection>
```

```

        <TextBlock.Foreground>
            <SolidColorBrush x:Name="textBrush" Color="Red" />
        </TextBlock.Foreground>
    </TextBlock>

```

To achieve the animation, the application has two *Storyboard* objects, which target the *RotationX*/*RotationY* properties of the named *PlaneProjection*.

```

<phone:PhoneApplicationPage.Resources>
    <Storyboard x:Name="rotateX">
        <DoubleAnimation Storyboard.TargetName="textPlane"
                        Storyboard.TargetProperty="RotationX"
                        From="0" To="360" Duration="0:0:3" />
    </Storyboard>
    <Storyboard x:Name="rotateY">
        <DoubleAnimation Storyboard.TargetName="textPlane"
                        Storyboard.TargetProperty="RotationY"
                        From="0" To="360" Duration="0:0:3" />
    </Storyboard>
</phone:PhoneApplicationPage.Resources>

```

The App Bar has two buttons, with suitable icons and text.

```

<phone:PhoneApplicationPage.ApplicationBar>
    <shell:ApplicationBar IsVisible="True" IsMenuEnabled="True">
        <shell:ApplicationBarIconButton
            IconUri="/Images/appBar.rotateX.png" Text="rotate X"
            x:Name="appBarRotateX" Click="appBarRotateX_Click"/>
        <shell:ApplicationBarIconButton
            IconUri="/Images/appBar.rotateY.png" Text="rotate Y"
            x:Name="appBarRotateY" Click="appBarRotateY_Click"/>
    </shell:ApplicationBar>
</phone:PhoneApplicationPage.ApplicationBar>

```

In the main page, you implement the two *Click* handlers to start the corresponding storyboard, as shown in the following:

```

private void appBarRotateX_Click(object sender, EventArgs e)
{
    rotateX.Begin();
}

private void appBarRotateY_Click(object sender, EventArgs e)
{
    rotateY.Begin();
}

```

The App Bar also supports menu items. The user accesses the menu via the ellipsis image in the App Bar, or by flicking up on the App Bar. In this example, you add a menu with two items, to change the color of the text.

```

<phone:PhoneApplicationPage.ApplicationBar>
    <shell:ApplicationBar IsVisible="True" IsMenuEnabled="True">
        <shell:ApplicationBarIconButton
            IconUri="/Images/appBar.rotateX.png" Text="rotate X"
            x:Name="appBarRotateX" Click="appBarRotateX_Click"/>

```

```

<shell:AppBarIconButton
    IconUri="/Images/appBar.rotateY.png" Text="rotate Y"
    x:Name="appBarRotateY" Click="appBarRotateY_Click"/>
<shell:AppBar.MenuItems>
    <shell:AppBarMenuItem
        x:Name="appBarRed" Text="Red" Click="appBarRed_Click"/>
    <shell:AppBarMenuItem
        x:Name="appBarGreen" Text="Green" Click="appBarGreen_Click"/>
</shell:AppBar.MenuItems>
</shell:AppBar>
</phone:PhoneApplicationPage.AppBar>

private void appBarRed_Click(object sender, EventArgs e)
{
    this.textBrush.Color = Colors.Red;
}

private void appBarGreen_Click(object sender, EventArgs e)
{
    this.textBrush.Color = Colors.Green;
}

```

Figure 5-13 shows how the menu slides up from the bottom when the user taps the menu ellipsis.

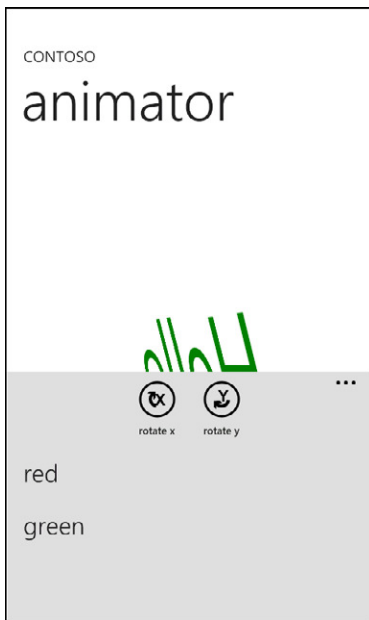


FIGURE 5-13 The App Bar menu items appear when the user taps the menu ellipsis.

By default, if you try to access App Bar elements by name, they will be null. When you build a Silverlight project, the build system produces an interim C# code file for each XAML page, named <page>.g.cs—for example, mainpage.g.cs. Note that the interim version of a page that defines App Bar elements includes code in the *InitializeComponent* to retrieve XAML-declared objects into code objects, including the App Bar buttons and menu items.

```

this.appBarRotateX = ((Microsoft.Phone.Shell.ApplicationBarIconButton)
    (this.FindName("appBarRotateX")));
this.appBarRotateY = ((Microsoft.Phone.Shell.ApplicationBarIconButton)
    (this.FindName("appBarRotateY")));
this.appBarRed = ((Microsoft.Phone.Shell.ApplicationBarMenuItem)
    (this.FindName("appBarRed")));
this.appBarGreen = ((Microsoft.Phone.Shell.ApplicationBarMenuItem)
    (this.FindName("appBarGreen")));

```

However, remember that the App Bar is not in the application's visual tree, and you cannot retrieve App Bar buttons or menu items by name, only by index, so the fields will still be null after these assignments. The *FindName* method walks the visual tree to find a *UIElement* with the specified name, and of course, it won't find any App Bar elements in the tree. Note that this applies only to the App Bar controls; other fields will be successfully assigned in *InitializeComponent*. If you need programmatic objects for the App Bar buttons (for example, if you need to dynamically enable/disable them), you need to fetch them from the collection by index.

```

public MainPage()
{
    InitializeComponent();

    // Assign fields by index, not name.
    appBarRotateX = this.ApplicationBar.Buttons[0] as ApplicationBarIconButton;
    appBarRotateY = this.ApplicationBar.Buttons[1] as ApplicationBarIconButton;
}

```

So, why is the App Bar not in the application's visual tree? The answer is that under the covers, the App Bar is more closely aligned with the phone's hardware buttons (such as the camera and lock-screen buttons) than with anything in your application. The App Bar is really a part of the phone's shell. It's just that the application platform provides convenient managed classes to represent it so that you can work with it easily in your application code. It is also designed to bridge your application and the standard phone chrome in a way that appears seamless to the user. Note that the *ApplicationBar* class that you use in your code implements the *IApplicationBar* interface, which is used internally and is not intended for you to use directly.

Summary

Windows Phone supports a range of touch gestures, and the application platform provides a rich set of choices for interacting with this, at six different levels. The Metro vision was factored into the selection and design of the underlying phone hardware, and this helped to solidify the precise numeric limitations of various touch and UI elements. The Metro guidelines are very clear on what you should do to optimize the user's experience, in terms of sizing and positioning controls, speed of response to touch input, and expected motion dynamics. The phone also has orientation sensors that are surfaced to the application as orientation events, and you examined alternatives for handling these changes to maintain the best UX. Finally, you looked at how you can work with the software App Bar, which is a part of the system chrome.

PART II

Application Model

CHAPTER 6	Application Model	175
CHAPTER 7	Navigation State and Storage	199
CHAPTER 8	Diagnostics and Debugging	243

This part examines the application's end-to-end user experience, focusing on how the user navigates between multiple pages in an application, as well as across multiple applications. This also includes the way an application should behave in the face of lifecycle events, plus a look at how to build diagnostics and debugging support in your application.

Application Model

This chapter focuses on the application model and explores at depth an application's lifecycle and events. On a desktop computer, users can have multiple applications open, performing work at the same time. Users can switch between running applications in any arbitrary order and can start and explicitly stop applications at any time. A smartphone, however, has significant constraints in terms of memory, processor capacity, disk storage, screen size, input capabilities, and even power. The severity of these physical constraints mandates that the application platform imposes corresponding constraints on applications. The application platform and the underlying operating system both execute a number of services in the background, and several of these will be executing at any given period of time. On the other hand, there will be only one application running in the foreground at any one time. There may be additional background applications running at the same time, notably background tasks such as email sync, but these are allocated a significantly smaller resource set (memory quota and CPU time). Despite this, users are presented with an experience by which they can switch from one application to another and back again, seamlessly. This chapter considers how this behavior is implemented, examines the hooks that an application developer can use to take a proactive part in the application lifecycle, and offers some guidance on how best to take advantage of the application model.

Lifetime Events and Tombstoning

The application platform does an excellent job of presenting users with a seamless multi-tasking application experience. For example, they can use the Start and Back buttons to launch new applications and switch back to previously running applications, which provides the illusion that multiple applications are running at the same time. In fact, in Windows Phone, only one application at a time is allowed to run in the foreground. The design of the application platform leads to a number of other constraints, listed in the following:

- Every launched application instance is either in an *active* or *dormant* state. The dormant state (also known as "paused") actually covers two different possible states: paused and *tombstoned*, which is discussed a bit later on.
- Windows Phone also distinguishes between the *foreground* and *non-foreground* states. In all cases, the foreground application is also active. On the other hand, in only some cases is the non-foreground application also dormant. Specifically, if the foreground application invokes certain Launchers and Choosers, the Launcher/Chooser becomes the foreground application,

and the initial application becomes the non-foreground application; however, crucially, it is not deactivated and remains in the active state, not dormant. Note that this is strictly a version 7 corner case. It was eliminated in version 7.1.

- Apart from the aforementioned corner case, there can be only one active application at a time, and only the active application (or the user) can initiate navigation from one page to another in an application or from one application to another.
- There can be only one active page at a time, and the only way to activate a page is to navigate to it.
- When an application is activated, the platform navigates to the page marked in the application as its initial or default page. Therefore, every application must have at least one page.

If the user navigates away from an application, it will always be either deactivated or closed, altogether. Specifically, if the user navigates away by pressing the Back key, it will be closed; if he navigates away by any other mechanism, such as pressing Start or launching a chooser, it will be deactivated. If the user subsequently navigates back to that application, it might be reactivated from a dormant state, and it might in fact be relaunched after tombstoning. Note that, unlike desktop Windows, the application lifetime is not a 1:1 mapping with the process lifetime. The platform maintains a backstack of previously running applications, implemented as a last-in-first-out (LIFO) queue as well as a minimal set of information about each application in the backstack so that it can quickly reactivate an application, if called upon to do so.

Therefore, if a user navigates back to an application that was previously running—one that is still on the backstack—then the system reactivates it. On the other hand, if the previously running application is no longer on the backstack, then the user will not find it by navigating backward through the backstack. In this scenario, the only way to reach the application is to launch the process again. Conversely, even if the previously running application *is* still on the backstack, but the user uses any mechanism other than the Back key, such as launching the same application from the Start tiles, then a new application instance will be started. All marketplace-downloadable applications are single-instance, so starting a fresh copy of an application always removes any previous instance from the backstack. However, this is not the case with certain built-in applications, such as email, for which you can have multiple instances on the backstack.

This combination of the user interaction model and the resource constraints mentioned earlier means that the platform can minimize the old application's use of precious resources (mainly CPU and memory) and reclaim these for use in the new application. To ensure a seamless, well-performing user interface, the foreground application must be allocated the maximum amount of resources that the platform can offer (after reserving required resources for indispensable system processes and services). On a phone—which inherently has limited physical resources—the best way to ensure that this happens is to reclaim resources from applications that are not currently the foreground application.

When users navigate backward, even when the platform starts a process for a new application instance, it appears to the users that they are merely navigating back to a running instance. This is because the restart is often faster than the initial launch of the application, and also because the platform keeps track of things such as which page the user was on in the application before she navigated away. The application developer is offered the opportunity to take part in this behavior and to make the task-switching or reactivation/restart even more seamless by responding to system lifecycle and navigation events, bringing the application to the appropriate state at those points.

From an application developer's perspective, there are really only two primary lifetime scenarios:

- The *Closing* case, in which an application terminates and receives the *Closing* event. This is unambiguous and simple to handle. This happens when a user presses the hardware Back button from the first page in the application, which kills the process and application instance. This is also the case with XNA games, using the XNA Exit functionality.
- The *Deactivated* case, in which an application is deactivated. In this case, *sometimes* the process is terminated, and sometimes it is not. This also varies according to the OS version, the amount of free memory, whether you launched a particular type of chooser, and so on. The key point to remember is that you never know which situation (deactivated or terminated) will occur, so your applications must be prepared for the worst possible case (process terminated). If the application is reactivated, you can opportunistically take advantage of the case wherein the process wasn't terminated.

The unhandled exception case—an application terminates by crashing—is arguably a third lifetime scenario, but by definition, there is nothing the application can do to recover from this, so that scenario is not germane for this discussion. For the cases that do matter, all the complexity arises in the deactivated case. From a developer's perspective, the details are almost irrelevant; you just need to allow for the possibility that your application can be terminated after deactivation. That said, the following sections explore the internal behavior so that you can understand the reasoning behind this. First, you can further divide the *Deactivated* case into two scenarios:

- *Tombstoning*, in which the user navigates forward away from the application, the application is deactivated, and the process is killed, but the application instance is maintained.
- *Fast reactivation* (also known as the non-tombstone case), in which the application is deactivated and then immediately reactivated, without being tombstoned in the interim.

For marketplace applications, it is not possible to have more than one instance of an application running at the same time. The system will detect an attempt to launch a second instance of an application that is already running and will remove the first instance from the backstack. This restriction applies to all Microsoft Silverlight and XNA applications, but it does not to certain built-in applications such as the email client.

It is simple enough to confirm the order in which lifecycle events are raised and under what circumstances; just put *Debug.WriteLine* statements in each of the interesting methods of the *App* and *Page* classes, as shown in the following (this is the *TestActivation* application in the sample code):

```
public partial class App : Application
{
    public App()
    {
        Debug.WriteLine("App.ctor");
        ... irrelevant code omitted for brevity.
    }

    // Code to execute when the application is launching (eg, from Start).
    // This code will not execute when the application is reactivated.
    private void Application_Launching(object sender, LaunchingEventArgs e)
    {
        Debug.WriteLine("Application_Launching");
    }

    // Code to execute when the application is activated (brought to foreground).
    // This code will not execute when the application is first launched.
    private void Application_Activated(object sender, ActivatedEventArgs e)
    {
        Debug.WriteLine("Application_Activated");
    }

    // Code to execute when the application is deactivated (sent to background).
    // This code will not execute when the application is closing.
    private void Application_Deactivated(object sender, DeactivatedEventArgs e)
    {
        Debug.WriteLine("Application_Deactivated");
    }

    // Code to execute when the application is closing (eg, user hit Back).
    // This code will not execute when the application is deactivated.
    private void Application_Closing(object sender, ClosingEventArgs e)
    {
        Debug.WriteLine("Application_Closing");    }
}

public partial class MainPage : PhoneApplicationPage
{
    public MainPage()
    {
        Debug.WriteLine("MainPage.ctor");
    }

    protected override void OnNavigatedFrom(NavigationEventArgs e)
    {
        Debug.WriteLine("MainPage.OnNavigatedFrom");
    }
}
```

```

        protected override void OnNavigatedTo(NavigationEventArgs e)
        {
            Debug.WriteLine("MainPage.OnNavigatedTo");
        }
    }
}

```

In this example, the application has two pages: one named *MainPage*, and the other named *SecondPage*. Upon application start, you see the following debug output, illustrating the sequence of events, as the *App* object is constructed and the *Launching* event is raised. This is followed by the *MainPage* construction and the *OnNavigatedTo* called for the *MainPage*.

```

App.Ctor
Application_Launching
MainPage.Ctor
MainPage.OnNavigatedTo

```

Then, when the user navigates to the *SecondPage*, you will see the output shown below, as the application constructs the *SecondPage*, navigates away from the *MainPage*, and navigates to the *SecondPage*.

```

SecondPage.Ctor
MainPage.OnNavigatedFrom
SecondPage.OnNavigatedTo

```

When the user navigates back from the *SecondPage* to the *MainPage*, you see the debug output that follows. Notice that the *MainPage* constructor is not called, because the page still exists.

```

SecondPage.OnNavigatedFrom
MainPage.OnNavigatedTo

```

If the user navigates from the *MainPage* of the application forward to another application via the Start button, and then navigates back again through the application backstack to the first application, you'll see the debug output that follows. Note that the *Activated* event is raised in this case, not the *Launching* event.

```

App.Ctor
Application_Activated
MainPage.Ctor
MainPage.OnNavigatedTo

```

If the user presses the Back key from the *MainPage* of the application, the application will terminate. Just before the process is torn down, the *Closing* event is raised.



Note In version 7, it is not possible to determine when your process is terminated after it was previously deactivated unless you have the debugger attached and look for the “process TaskHost exited” debug output. This changed in version 7.1, in which you can check the new *IsAppInstancePreserved* property.

Be aware that the precise behavior of the underlying application platform when managing the backstack varies slightly between version 7 and 7.1. You can find further details in Chapter 15, “Multi-Tasking and Fast App Switching.”

Application Closing

This section explores the termination scenario in detail. The first scenario is normal termination. The only formal, direct way for users to terminate an application is to press the Back button from the first page of an application. When they do this, a *Closing* event is raised on the application, and its hosting process—and the logical application instance—is then terminated. The normal termination sequence is illustrated in Figure 6-1.

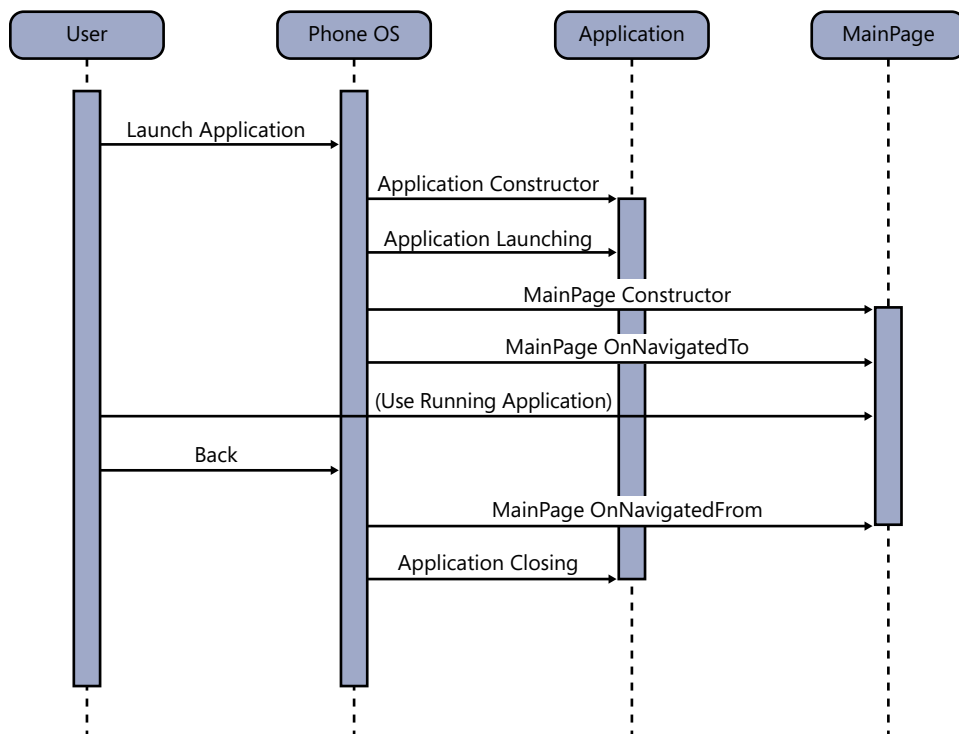


FIGURE 6-1 A diagram of the closing (normal termination) sequence.

Note that “Launch Application” in these diagrams is an abbreviated term that means launch application by any means, including from the Start screen, the installed application list, via a toast notification, and so on.

It is also important to understand why the application platform surfaces lifecycle events to the application. The idea is to allow the application to take part in its own lifetime management, specifically so it can save or load persistable state, and so it can make informed decisions about when to

perform certain operations. For example, it is a marketplace certification requirement that the application must show its first screen within 5 seconds of launch, and be responsive to user input within 20 seconds. Application initialization can be done in the *App* class constructor, but you should limit this to only critical operations and defer as much as possible to a later point to provide the best user experience (UX).

The system sends a *Launching* event, which you can handle to perform additional initialization. However, given the startup time performance requirements, you should be careful to avoid doing any lengthy operations when the *Launching* event is received. By the same token, you must also complete activation, deactivation, and navigation within 10 seconds, or you will be terminated. So, the various lifecycle events give you an opportunity to take action, but you should design your application so that whatever you do in these events happens quickly. The time to do lengthy operations is during normal running, not when you're handling a lifecycle event. Due to the time-sensitive nature of these events, techniques such as incrementally saving your data are critically important on the phone. Even if your application's data usage starts out small, if you design your application around a single load at startup and a single close at shutdown, you will hit a brick wall when the complexity or data usage of your application grows over time.

Application Deactivated

The second scenario is when the user brings some other application into the foreground. Typically, this is done either by pressing the Start button or by accepting a notification or other alert that launches the new application, or when the phone locks. This is illustrated in Figure 6-2. Regardless of how it's triggered, when the user switches to a new application, a *Deactivated* event is raised on the old application. The process is suspended and is a candidate for termination, but the application instance is still valid as the platform keeps any saved state information such that it can quickly reactivate the application (and send it an *Activated* event) if the user subsequently presses Back to go back to the application. In the normal case in version 7, the platform will subsequently terminate the process.

When an application is terminated in this manner, all the resources (CPU and memory) that it was consuming are taken away from it and made available to other processes. The platform retains the barest minimum it needs to be able to reactivate the application at a later point, should it be called upon to do so. There will be an entry for the application in the application backstack, including a note of the page within the application that the user was last viewing, the intra-application page backstack, and some limited transient state stored by the application itself (such as the state of UI controls).

If the tombstoned application is later reactivated, the lifecycle events are almost identical to its first-time launch. The only difference is that after the *App* constructor is invoked, the platform sends an *Activated* event, not a *Launching* event.

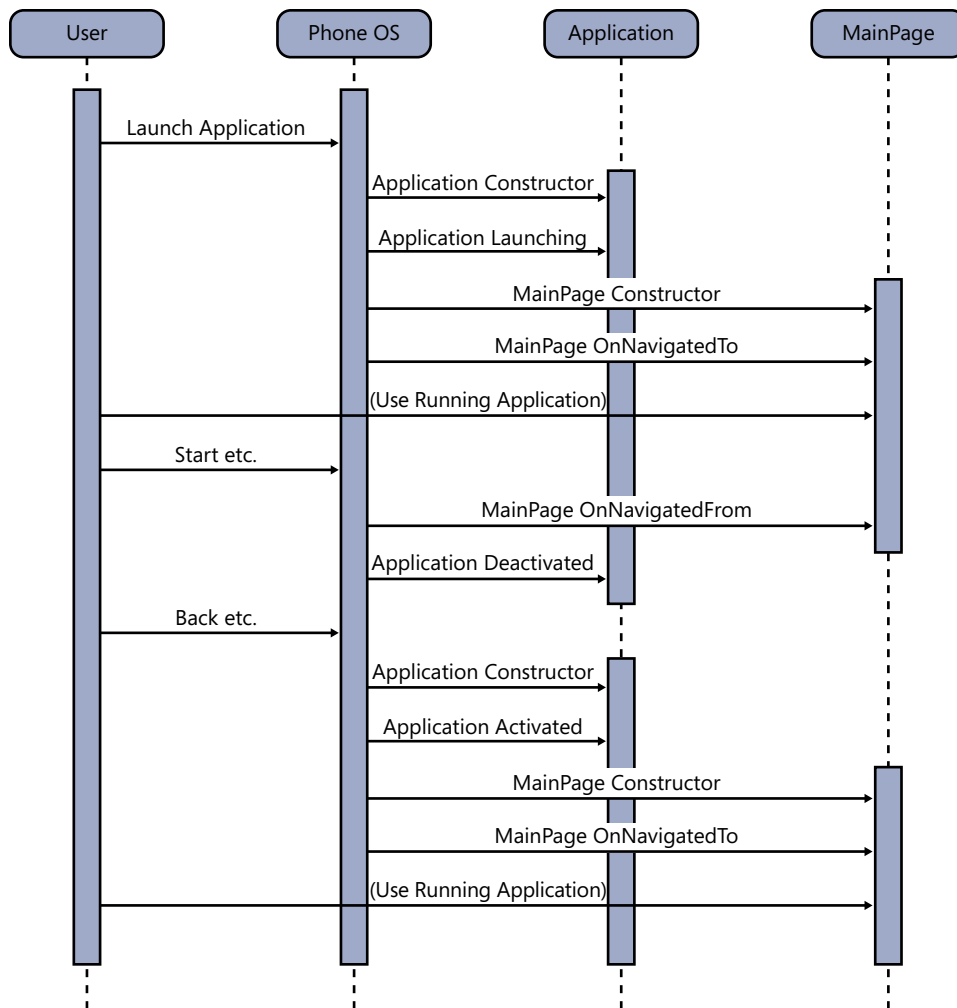


FIGURE 6-2 A typical tombstone sequence.

In this sequence, the appropriate actions that your application should take for each lifecycle event (or virtual method override) are summarized in Table 6-1.

TABLE 6-1 Expected Behavior During Lifecycle Events

Class	Event/Override	Suitable Actions
Any Page	<i>OnNavigatedFrom</i>	Save transient page state in <i>PhoneApplicationPage.State</i> .
App	<i>Deactivated</i>	The app is being deactivated and is a candidate for tombstoning: save transient application state in <i>PhoneApplicationService.State</i> ; save persistent state to isolated storage.
App	<i>Activated</i>	Load application state from <i>PhoneApplicationService.State</i> . Avoid lengthy operations.
Any Page	<i>OnNavigatedTo</i>	Load page state from <i>PhoneApplicationPage.State</i> .

The diagram makes the simplification that the user was on a page called *MainPage* when he navigated away from the application. In fact, the behavior holds true regardless of which page he was on. The platform keeps track of the page he was on, and then upon reactivation, constructs the page, if necessary (that is, if the application was tombstoned), and invokes the *OnNavigatedTo* method.

In the interest of resource management, the platform retains only five applications on the backstack, including the currently active one. As soon the user launches the sixth application, the application at the beginning of the backstack is discarded completely. In this situation, the discarded application will have received a *Deactivated* event just as the user navigated away from it. It does not receive any further events, and there is no indication when it is discarded from the backstack.

It should be clear from the foregoing that there is often asymmetry in the lifecycle event sequence. An application does not always receive a *Closing* event. Also, an application receives only a *Launching* event when a fresh instance is launched. From a user's perspective, she can "start," "stop," and "pause" an application multiple times, but the application only ever receives the *Launching* event once. So, in many cases, the application will not have symmetry in *Launching* and *Closing*, in *Launching* and *Activated*, or in *Deactivated* and *Closing*. In fact, there are only two guarantees for a given application instance: *Launching* always happens exactly once, and *Activated* is always preceded by *Deactivated*. For a developer, this means that any code that performs lifecycle housekeeping should be done at the appropriate time. For loading content, it should be done just in time. For saving content, it should be done as soon as possible. That typically means loading only enough content for the current page that you are on and saving that content when you navigate away. So, per-page lifecycle housekeeping should be done in the *OnNavigatedTo* and *OnNavigatedFrom* overrides.

Application Deactivated (the Non-Tombstone Case)

The third lifetime scenario, which is shown in Figure 6-3, is when the user runs an application and then immediately after presses Start and then Back. This is known as the "fast reactivation" or "non-tombstone" case. If this particular sequence of button presses happens fast enough, the application is not tombstoned, the hosting process is not torn down, and the *Deactivated* and *Activated* events follow in quick succession. This scenario also happens when certain Launchers or Choosers are invoked while the application is running. In Windows Phone 7.1, this also happens in the Fast Application Switching (FAS) feature (discussed in Chapter 15).

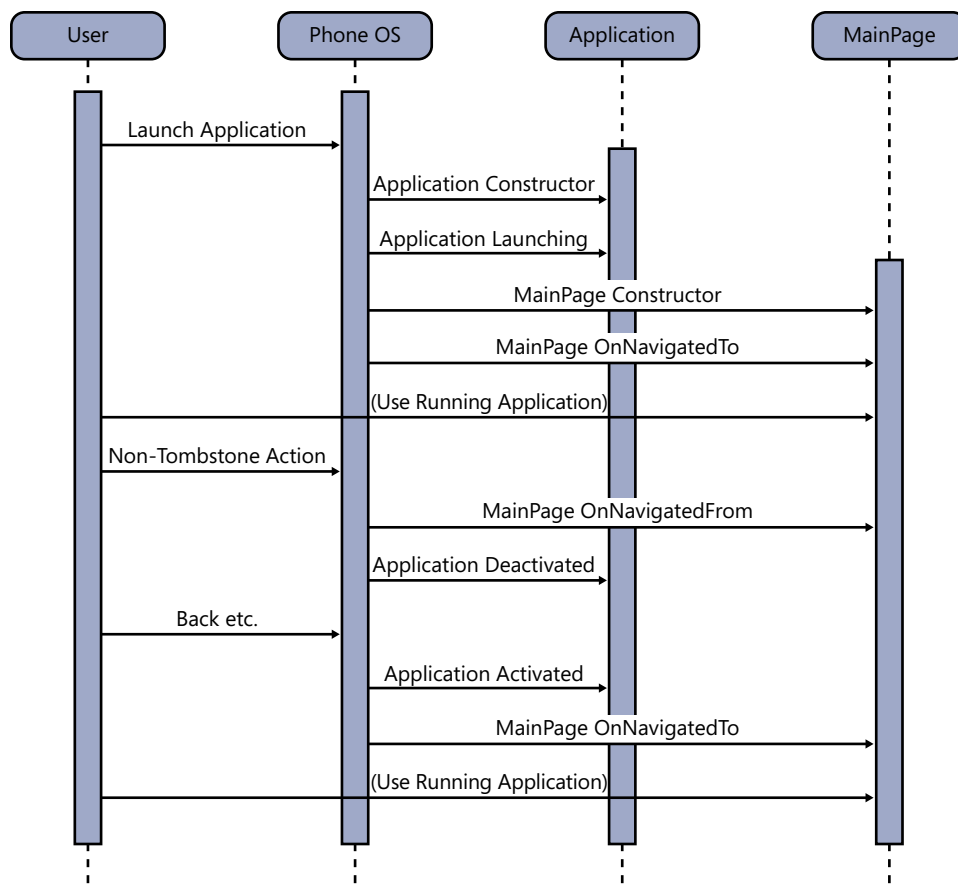


FIGURE 6-3 This diagram portrays the non-tombstone sequence.

Why does this scenario receive special treatment? The answer is simple: usability tests show that the sequence of Start | Back button presses almost always results from the user accidentally pressing Start while working in an application. The team at Microsoft responsible for building the Windows Phone application platform paid careful attention to this and many other user scenarios to ensure that the user has the best possible overall experience. Specifically, the critical factor is performance—it is not efficient for the application platform to terminate the process and bring up a new one if the old one is still viable.



Note In this scenario, none of the constructors are called the second time (neither *App* nor *Page* constructors), because the *App* and *Page* objects were never destroyed. Given this, it should be clear that the developer should avoid any asymmetry in work done during the various lifecycle events. Specifically, if the application destroys things in the *Deactivated* handler, it should be careful to recreate them in the *Activated* handler, not in the constructors.

Unhandled Exceptions

For circumstances in which the application throws an exception that is not handled, no lifecycle events are raised at all. If there is an *UnhandledException* handler in place—which is something the Microsoft Visual Studio templates insert by default—then code execution will jump there. The default handler put in place by Visual Studio merely checks to see if the application is being debugged, and if so, it invokes *Debugger.Break* to break into the debugger. If the application is not being debugged, the exception is then handled in the application platform itself, and the application is immediately terminated and removed from the backstack. Recall that the only legitimate way for an application to terminate is when the user presses the Back button from the application's first page. Developers must not be tempted to use unhandled exceptions as a means of exiting the application. It is important that the user feels that the phone experience is predictable and consistent. Users know that they can always leave an application and go somewhere else on the phone by pressing Start. They also know that they can always exit an application by pressing Back from the application's initial page. Exiting via unhandled exceptions breaks consistency and leads to user confusion.

Worse, for those times when the user has agreed to send usage data back to Microsoft, it will trigger a stack dump (which takes up CPU and battery power) and upload that dump to Microsoft (taking up yet more CPU, more battery, and network bandwidth). In addition, an engineer at Microsoft will have to examine the report to ensure that it is not a bug in the application platform. If all this is not sufficient to dissuade a developer from throwing unhandled exceptions, then consider also that an application that does this will likely fail marketplace certification.

Of course, it is possible to implement an *UnhandledException* handler and handle the exception. If the handler sets the *Handled* property to true, the application will continue. You can experiment with the runtime behavior by using the *FatalError* solution in the sample code. Even so, in practice, exceptions should be handled locally, and the model of a catch-all final exception handler is a universally bad technique. If an exception has propagated out of your code to such a handler, it is extremely unlikely that you can actually do anything about it, especially as you will have lost some or all of the context of where the exception was originally raised. It's better to have the exception crash the application during development so that the bug can be identified and fixed prior to publication. If your testing is not thorough enough to unearth all exceptions (and no testing is ever 100 percent foolproof), and you publish your application, only for it to throw an unhandled exception in production, it's not the end of the world, because the marketplace will provide crash information to you. That is your opportunity to fix any unexpected bugs and publish an update. However, there is one exception to this rule (no pun intended), and that is for the rare exceptions that can be thrown as a result of system-initiated navigations, such as the "cannot navigate when already navigating" case—such exceptions can safely be ignored.

Marketplace certification includes requirements about handling exceptions and memory consumption. Specifically, an application must handle all managed exceptions. If it does not, and if this is detected during marketplace ingestion, then the application will be rejected for publication. On top of this, if you have an exception from which the application cannot recover and continue to make forward progress, you should alert the user to this fact with a non-technical message. It is not acceptable to fail silently, and it is not acceptable to display a cryptic error message, *HRESULT*, stack trace, or other non-user-friendly message.

Why Is There No App.Exit?

There is no Exit method for Silverlight applications for three main reasons:

- There is very little need for this from a UX perspective. The user can always press Start at any time to leave an application. He can also always press Back from an application's initial page to leave the application. The user is rightly not concerned with the implementation detail of how this works. He doesn't care whether the application is still technically running, whether some underlying tombstoning mechanism is kicking in, or what the state of the hosting process is.
- There is very little need for this from an application platform perspective. As soon as the user navigates away (either forward or backward), most of the application's resources are reclaimed for use elsewhere. The application platform is extremely efficient at doing this. This ensures that the new foreground application always offers the best possible UX.
- Introducing application Exit functionality would increase the complexity of the UX. Instead of two well-understood mechanisms for leaving an application—neither of which burden the user with the need to know or decide whether to terminate a process or suspend it—there would now be three. Moreover, the third mechanism would not be supported by a well-known hardware button; instead, it would be invoked by application-specific logic, which might or might not even be visible to the user.

Given that, why does XNA provide Exit functionality? It is possible to build a non-game XNA application, just as it is possible to build a non-XNA game. However, XNA is designed very specifically for games, and not just on the Windows Phone platform, but also for Xbox and desktop Windows. Therefore, there is a requirement for XNA to support mechanisms that make sense on Xbox and Windows, even if those mechanisms make less sense on Phone. Plus, of course, XNA was ported to Phone long after it was available on Xbox and Windows, and many XNA games, game frameworks, and middleware were built to include the notion of a *Game.Exit*. It is generally a bad strategy to forcibly fork software for different platforms unless there is a good case for it. XNA games for all three platforms have the same *Game.Exit* functionality. Removing this just for Phone offers no real benefit and entails considerable cost and risk. Games do not have the same page-based user navigation model as non-game Phone applications, so users have different expectations for games than for non-game applications. On top of that, XNA game developers are strongly encouraged to invoke the *Game.Exit* method only when the user presses the Back button. Most (if not all) of the games in the marketplace have taken this advice, and do not have an explicit Exit button. Thus, you should do the same.

Obscured and Unobscured

Activation/Deactivation happens when the user navigates away from the app and when the application invokes Launchers and Choosers. On the other hand, some external operations merely result in the application becoming temporarily obscured. In this scenario, there is no *NavigatedFrom* or *Deactivated* event. Instead, the system raises an *Obscured* event. A common example of such an external operation is when a notification for an incoming call or a reminder is received.



Note An incoming SMS toast does not raise the *Obscured* event.

Obscuring does not cause navigation away from the application—the application continues to run in the foreground—it's just that some higher-priority UI is obscuring the application's UI. This does not cause a frozen application display; the application does actually continue running, executing whatever operations it was performing when it was interrupted.

You can test this with some simple code in the *App* class, as shown in the code that follows (and demonstrated in the *TestObscured* application in the sample code). In this example, the application sets up a timer to print debug output, simply to confirm that the application continues running while obscured. Note that you cannot test this on the emulator; you will have to run this on a physical device, and then call that phone (or set up a reminder to fire while the application is running).

```
private DispatcherTimer timer;
private int count;

public App()
{
    UnhandledException += Application_UnhandledException;
    InitializeComponent();
    InitializePhoneApplication();

    timer = new DispatcherTimer();
    timer.Interval = TimeSpan.FromSeconds(1);
    timer.Tick += delegate(object sender, EventArgs e)
    {
        Debug.WriteLine(String.Format("count={0}", count++));
    };
    timer.Start();

    RootFrame.Obscured +=
        new System.EventHandler<ObscuredEventArgs>(RootFrame_Obscured);
    RootFrame.Unobscured +=
        new System.EventHandler(RootFrame_Unobscured);
}

private void RootFrame_Obscured(object sender, ObscuredEventArgs e)
{
    Debug.WriteLine("RootFrame_Obscured");
    if (e.IsLocked)
    {
        Debug.WriteLine("IsLocked == true");
    }
}

private void RootFrame_Unobscured(object sender, System.EventArgs e)
{
    Debug.WriteLine("RootFrame_Unobscured");
}
```

The *Obscured* event does not imply that the entire application UI is obscured. In many cases, including for an incoming phone call, the UI is only partially obscured (at least until the call is accepted). Another scenario in which this event is raised occurs when the phone lock screen is engaged. An application can determine whether this is the cause of the obscuring by testing the *IsLocked* property on the *ObscuredEventArgs* object passed in as a parameter to the *Obscured* event handler, as shown in the preceding example.

Note that the application will not always receive a matching *Unobscured* event for every *Obscured* event. For example, that's true for the scenario in which a user navigates away from the application by pressing the Start button. It's also true in the case for which the *Obscured* event is the result of the lock screen engaging. When the user later unlocks the screen, the application is not sent an *Unobscured* event. So, if you get an *Obscured* event and then the lock screen engages, your application will be deactivated (sent the *Deactivated* event) and then later reactivated.

If you disable the lock screen, you obviously won't get any *Obscured* events for this case because the screen will not lock. You can disable the lock screen by setting *UserIdleDetectionMode* to *Disabled*. This statement is generated in the *App* constructor (and commented out) by the standard Visual Studio project templates. The Visual Studio code generation is intended only for debugging scenarios. In general, you should use this setting only after very careful consideration; it is legitimate only for an application that absolutely must continue running, even when the user is not interacting with the phone. An example of this is a run-tracker type application: the user starts the application before carrying out some other activity that the application would track in some way.

```
PhoneApplicationService.Current.UserIdleDetectionMode = IdleDetectionMode.Disabled;
```

If you actually need to use the feature in normal situations, you should not set it globally at startup. Instead, you should turn it on only when the user is actively using the feature that requires non-locking, and then turn it off again as soon as she is done with that activity. For example, in a game, you should not disable lock while the user is on a menu screen or has already paused the game; you would turn it on only while she is actively playing the game.

A related setting is *ApplicationIdleDetectionMode*. The platform's normal assumption is that if an application is running and the lock screen engages, it is reasonable to deactivate the application. By disabling *ApplicationIdleDetectionMode*, the application can continue to run under screen lock. If you do disable *ApplicationIdleDetectionMode*, then the system does not deactivate idle applications. In this case, when the user eventually unlocks the screen again, the application will receive the *Unobscured* event.

```
PhoneApplicationService.Current.ApplicationIdleDetectionMode = IdleDetectionMode.Disabled;
```

Although there are legitimate cases for using this, in version 7 it was unfortunately used as a hack to improve the application resume performance. The legitimate cases include when you initiate large downloads, which should be allowed to continue while the screen is off, for media to continue playing in the background, or for location-based applications. In Windows Phone 7.1, this is largely redundant because of the FAS and the support for background transfers and background audio. If you do

disable *ApplicationIdleDetectionMode*, you should also do as much as possible to minimize battery consumption. Specifically, you should stop all active timers, animations, use of the accelerometer, GPS, FM radio, isolated storage, and network. You would then use the *Unobserved* event to reinstate them, as appropriate.

Launchers/Choosers and Tombstoning

Invoking most Launchers/Choosers will tombstone the application. However, in version 7 the following do not trigger an automatic tombstone in the calling application (although the phone might still tombstone if it runs low on resources):

- PhotoChooserTask
- CameraCaptureTask
- MediaPlayerLauncher
- EmailAddressChooserTask
- PhoneNumberChooserTask
- Multiplayer Game Invite
- Gamer You Card

With Choosers, the expectation is that the Chooser will return some value back to the application. This presents an interesting conundrum, given that the application will be deactivated in the interim. The solution is to ensure that you connect to the Chooser in the application constructor. You should be aware that the behavior of Launchers and Choosers changes with version 7.1, as is discussed in Chapter 16, “Enhanced Phone Services.”

User Expectations

There is no requirement for an application to handle lifecycle events, but you must ensure that your app behaves in the manner that the user expects, especially when returning to the application via the backstack. For simple applications, this happens “for free.” The lifecycle events are raised specifically to allow an application to load and save state at key points as the user is navigating. So, it is generally in the application’s best interest to pay attention to those events and to understand the event sequences that apply in different scenarios. Also, as an example, when the phone receives notification of an incoming phone call, the active application receives an *Obscured* event. It is a certification requirement that the application must not stop responding at this time, must not terminate as a result of this event, and must not do anything to prevent the user seeing the notification or responding to it.

The application must also start up and render its first screen within 5 seconds of launch, must be responsive to the user within 20 seconds, and must maintain responsiveness such that the user does not experience lack of response for more than 3 seconds at any time. These constraints govern the kinds of operations the application can legitimately perform. For instance, developers need to pay careful attention when loading large amounts of data, especially upon startup. In fact, any lengthy

operation should be avoided during startup, and the application should use a judicious combination of idle-time loading and on-demand (or *lazy loading*) of data, so as to maintain a fluid UX.

The developer should take advantage of the lifecycle and navigation events exposed by the platform as positive opportunities to ensure that her application is resilient to the extreme resource constraints on the phone as well as the dynamic resource allocation that arises from normal phone usage.

Page Model

Windows Phone applications employ a page-based model, offering a UX that is similar in many respects to the browser page model. The user typically starts the application at an initial landing page and then navigates through other pages in the application. Each page typically displays different data along with different visual elements. As the user navigates forward, each page is added to the in-application page backstack (sometimes called the *journal*) so that they can always navigate backward through the stack, eventually ending up back at the initial page. Although the inter-application backstack of application instances is limited to five applications, there is no hard limit to the number of intra-application pages that can be kept in the page backstack. However, in practice it is uncommon to have more than six or so pages in the backstack; any more than that degrades the UX. Usability studies show that the optimal number is somewhere between 4 and 10. That doesn't mean that an application can't have dozens or even scores of pages—it just means that the navigation tree should keep each branch relatively short. You should also remember to clean up unused resources on pages in the backstack (such as images or large data context items), because they continue to consume memory and can often be recreated cheaply.

The application can support forward navigation in a wide variety of ways: through *HyperlinkButton* controls, regular *Button* controls, or indeed any other suitable trigger. An application can use the *NavigationService* to navigate explicitly to a relative or absolute URL. Relative URLs are used for navigation to another page within the application. Absolute URLs can be used to navigate to external web pages via the web browser (Internet Explorer). Backward navigation should typically be done via the hardware Back button on the device. If the user navigates back from the initial page, this must terminate the application, as depicted in Figure 6-4.

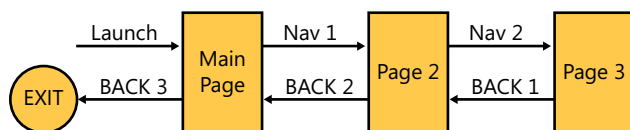


FIGURE 6-4 Intra-application page navigation.

Users can also navigate forward out of an application and into another application. This can be achieved both from within the application via links with external URLs (which use the web browser) or by directly invoking Launchers and Choosers. At any time, the user can navigate away from the application by pressing the hardware Start button on the device. Whenever the user navigates forward out of an application, that application is added to the system's application backstack. As the user navigates backward from within an application, he would move backward to that application's initial page, and then back out of the application to the previous application in the backstack. Eventually, he would navigate back to the beginning of the backstack. Navigating backward from there takes him to the Start screen.

The application platform—strictly, the shell in this context—maintains both the in-application page backstack and the overall application backstack. Here's a simple example. Suppose that a user starts the phone, and launches two applications. In the second application, he navigates forward to page three, and then presses Start, launching a third application. Pressing the Back button from the initial page of application 3 will take him back to page 3 in application 2. Pressing Back again will take him back to page 2 in application 2. He would continue to navigate backward within the in-application page backstack until he gets to the initial page in application 2. After that, pressing Back again takes him back to whatever page he was on in application 1, and so on. This behavior is illustrated in Figure 6-5.

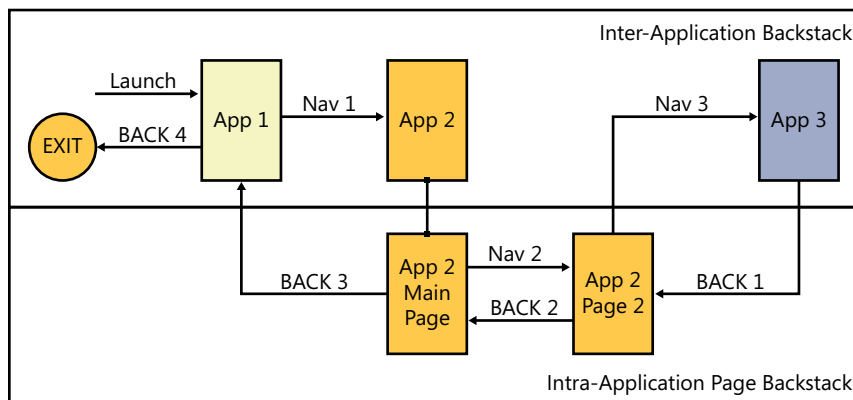


FIGURE 6-5 The inter and intra-application navigation model.

How does this model work in a more realistic scenario? What guidelines can an application developer use in the face of pivots or panoramas, with lists of items and subitems, with ancillary visuals such as splash screens, logon screens, context-sensitive help, settings, feedback and support forms, search pages, and so on? Just such a scenario is illustrated in Figure 6-6.

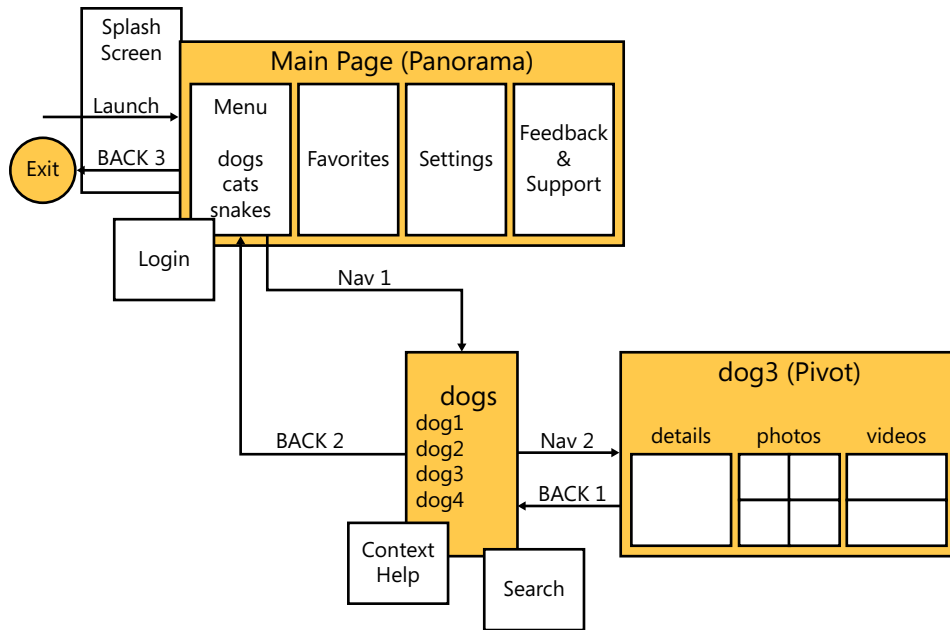


FIGURE 6-6 The model becomes more complex when you consider pages and non-page visuals.

It turns out this is not as difficult a problem as it might seem. There are a couple of principles to keep in mind that make the choices very simple. Let's take each of these visual items in turn. First, a splash screen is a transient visual; it is not a page, so it does not feature in the page navigation backstack. Indeed, the application platform controls the window that displays the splash screen. The developer's only input here is to provide the image for this window. A logon screen is common, but it has the potential for introducing complexity and confusion if not managed appropriately. This is especially true because logon pages are not always appropriate to present on the initial page, and it might be appropriate to present it in more than one place in the application. It would be a mistake to implement a logon screen as a page. Instead, it should be a *Panel* or *UserControl* on whatever page or pages that require its services. Alternatively, you could put it inside the *Frame*, but that would require you to retemplate the *Frame*.

You can use exactly the same model for context-sensitive help. In both cases, the appropriate design is to use a *Popup* or a *UserControl*, and to toggle its visibility when needed. If the application provides extremely comprehensive help, this could conceivably be implemented as its own hierarchy of pages, but that presupposes a very complex application that might not be appropriate for the phone, at all.

In some cases, this design would also be appropriate for a search window; in other cases, the search functionality might warrant a panel or control on a regular page. This approach reduces the number of pages that the user has to deal with. You can provide a very simple, easy-to-understand UX if you collect such ancillary UI elements together in one place. This leaves the application (and the user) to focus on the more important domain data.

In this example, the domain data relates to animals. The main page offers a menu of different types of animals. When users select one type of animal—for example, dogs—the application navigates to a list of all dogs. From that list, they can drill down to an individual dog. When they reach an individual dog, you can supply rich information, grouped into meaningful categories, perhaps using a *Pivot* control. Back in the main menu, if the user selects cats, does this imply that you should navigate to a listing page for cats that is different from the listing page for dogs? Almost certainly it does not. The various listing pages offer the same behavior, and they might even have the same or very similar visual elements. This scenario is best served by using data binding. You would use one listing page, and the display switches between dogs, cats, and snakes by simply switching the data binding (probably by switching the *DataContext*). Similarly, the individual item page switches between each individual dog as well as between an individual cat or snake via data binding.

Items such as the favorites, settings, and feedback (represented in the preceding example as *PanoramaItem* controls) as well as details, photos, and videos (represented as *PivotItem* controls) could all equally be implemented as individual pages, which would all typically be “dead-end” pages; that is, pages from which there is no forward navigation option. Such dead-end pages would appear in the page backstack; however, only one such page could be in the stack at a time, because by definition, users *must* navigate back from a dead-end page before they can navigate anywhere else. Figure 6-7 illustrates an alternative approach to designing pages and non-page visuals. In this model, even though there are many pages in the application, the backstack is never more than three pages deep (plus the current page), which represents significant savings in memory.

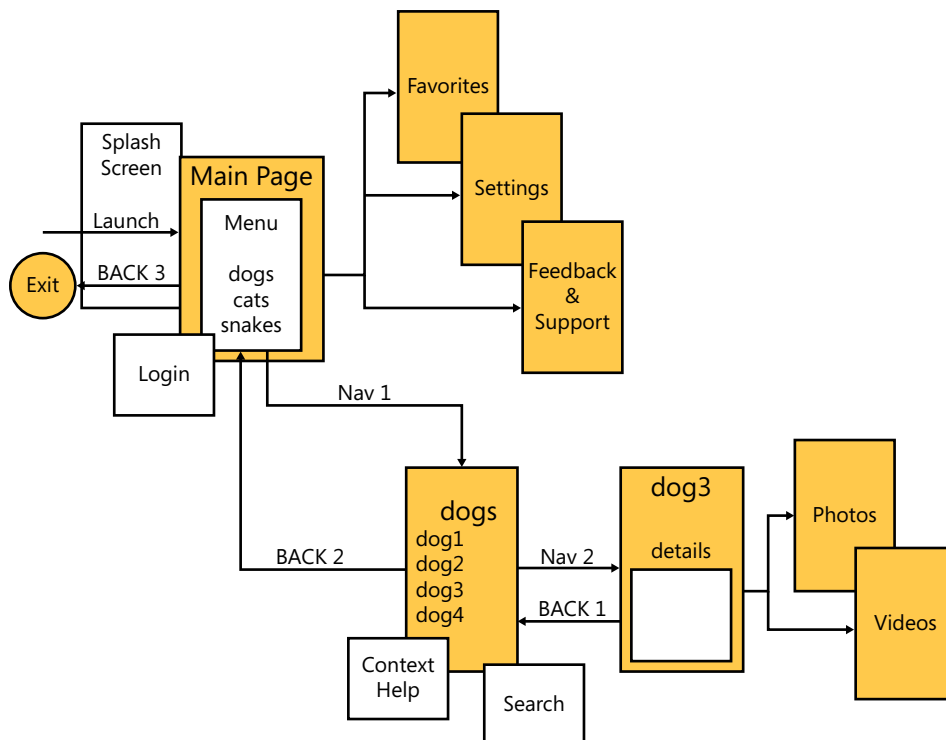


FIGURE 6-7 An alternative approach to dealing with pages and non-page visuals.

Note that *Panorama* and *Pivot* controls are child elements on a page. There are typically multiple *PanoramaItem* or *PivotItem* controls, among which the user moves about, but this is not navigation; the user does not change to a different page. The corollary of this is that even though a page that contains a *Panorama/Pivot* is in the backstack, the individual items are not. Suppose that the user is on *PanoramaItem* 3 and then presses Start. When she later presses Back, she will be returned to the page in the application that she last visited, which in this case is the page that hosts the *Panorama*. The question is to which *PanoramaItem* within the *Panorama* will she be returned? The behavior changed between version 7 and 7.1. In version 7, the user is returned to the first item of the *Panorama*. In version 7.1, she is returned to the specific item she last visited, and the title of the *Panorama* is updated so that it is fully visible. This behavior is enforced in the implementation of the *Panorama* and *Pivot* controls. The rationale for this is that it makes for a simpler UX: it is easier for the user to recognize where she is if the start of the *Panorama* title is always visible while navigating back.

If this guidance is to be more than a question of subjective style preference, you need to distill some defining characteristics of the visuals that you want to categorize in different ways. So, what is the defining difference between a logon screen and a data listing page or a data detail page, or between a context help dialog and a favorites page? Is it a question of whether these screens are data-bound to underlying data? No, because you could data-bind almost anything (including settings and context help in this example). Data binding is an implementation detail used to support good engineering practices; it is not relevant (or visible) to the user. Even in this simple example, a number of the screens could validly be implemented as either pages or pivot/panorama items, or as *Popup* or *UserControl* objects.

What would make a developer want to implement a visual as a page that persists in the page backstack? The question probably answers itself: you should promote a visual to a page if you want it to appear in the backstack—both in-application, and therefore, potentially across applications. That is, it is a meaningful location that the user might want to return to, that he might expect to return to when navigating back through the application backstack, and that he might be confused about if it doesn't appear where he expects it to be in the backstack. From this example, which of the visuals meets the requirements of a meaningful location? One interpretation is offered in Table 6-2.

TABLE 6-2 Pages, Controls, and Transient Panels

Visual	Meaningful Location	Valid Implementation Options
Splash screen	No	An application developer has almost no control over the splash screen, beyond providing an image.
Logon screen	No	<i>Popup</i> or transient panel.
Context help	No	<i>Popup</i> or transient panel.
Error message	No	<i>Popup</i> or transient panel.
Main page	Yes	<i>Page</i> : an application will always have at least one page where the application starts.
Favorites	Yes	<i>Page</i> or <i>Pivot/Panorama</i> item.
Data listing	Yes	<i>Page</i> or <i>Pivot/Panorama</i> item.

Visual	Meaningful Location	Valid Implementation Options
Data item	Yes	<i>Page</i> or <i>Pivot/Panorama</i> item.
Search	Sometimes	(Dead-end) <i>page</i> or <i>UserControl</i> .
Settings	Sometimes	(Dead-end) <i>page</i> or <i>UserControl</i> .
Feedback	Sometimes	(Dead-end) <i>page</i> or <i>UserControl</i> .
Main help	Sometimes	(Dead-end) <i>page</i> or <i>UserControl</i> .

Note that this table provides some suggested guidelines. In many cases, the best choice will depend on the context. For example, it makes sense for Search to be an explicit page in an online shopping application because it is a core feature, but it might not make sense if your Search feature is merely a filter on a set of data.

From this, the guiding principle is to ask the question: is the visual a meaningful location that should appear in the backstack? If the answer is yes, then it must be implemented as a *Page*. If the answer is no, then it should be implemented as some kind of popup, child window, dialog, or other transient panel. If the answer is sometimes, then it depends on the specific behavior of this item in the application; it might be best implemented as a page or as a control within a page, but it would never be implemented as a popup or transient panel. Of course, these guidelines are merely advisory. Each application developer is free to structure the application in whatever way is appropriate. That said, remember that your application will fail marketplace certification if the navigation patterns are not obvious to the marketplace testers. Much like the Metro guidelines, developers are encouraged to use a consistent, user-friendly model, but are free to deviate from this wherever it makes sense. Buying into an established navigation pattern is similar to buying in to Metro style guides. Keep in mind that Windows Phone is perceived a user-focused experience rather than a wide-open platform for experimentation.

Page Creation Order

As part of its backstack management, the system keeps track of which page (in a multi-page application) the user was on when she navigated away. So, if the user is on the second page when she navigates away from the application, and then goes back, she will end up going back to the second page. In Windows Phone 7, this causes the page to be recreated. If she subsequently navigates back to the main page from there, then at that point the main page will be recreated. So, the order of page creation in the application can change according to circumstances. The main or initial page in an application is not always constructed first. In fact, if the user has navigated forward through the in-application page hierarchy, and then forward to another application that uses a lot of memory (causing the original application to be tombstoned), and then navigates back to the first application, then the pages will be constructed in reverse order. This is shown in Figure 6-8 for the tombstone behavior (the normal case in version 7), and Figure 6-9 for the non-tombstone behavior (the normal case in version 7.1). You can verify this behavior by using the *PageCreationOrder* application in the sample code.

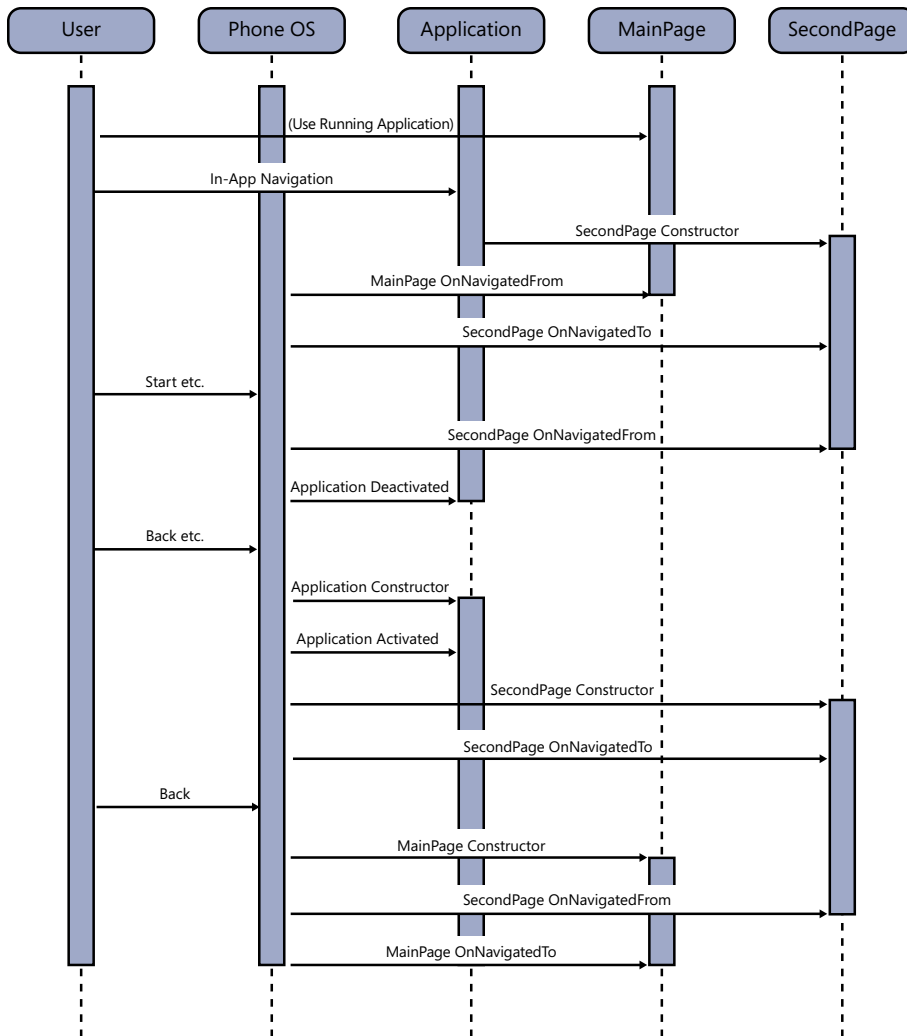


FIGURE 6-8 Unexpected page creation ordering (tombstone case). Here, *SecondPage* is constructed before *MainPage*.

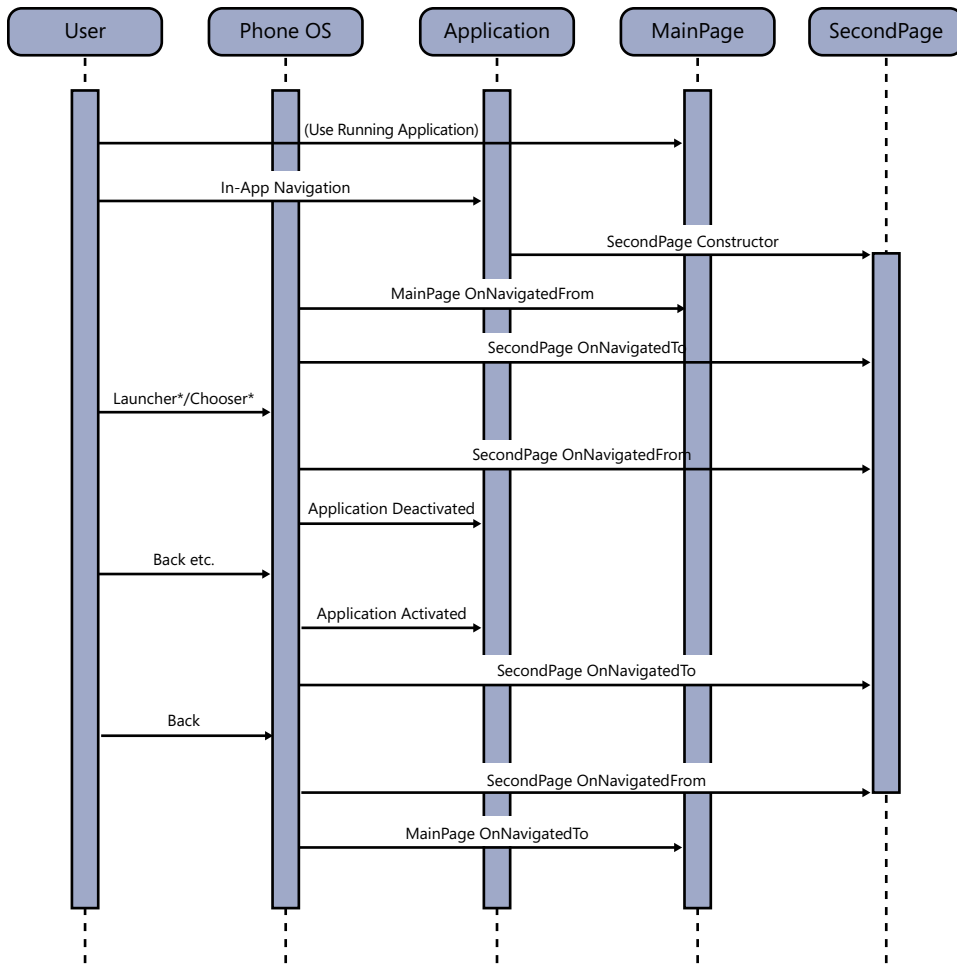


FIGURE 6-9 Page creation ordering (non-tombstone case) with no fresh page construction.

One consequence of this is that the application should not rely on a hierarchical relationship between pages, in terms of object lifetime. That is, don't construct objects in Page X that are required in Page Y. Instead, all pages should be responsible for maintaining their own private state, and any state that is used across multiple pages should be held in the viewmodel (see Chapter 4, "Data Binding and Layer Decoupling," for details on viewmodels). Furthermore, the viewmodel should be accessible to all pages at all times, with predictable finite lifetime characteristics, which pretty much means it should be held in the *App* class.

To ensure consistent state in the face of navigation requires that the developer understands the navigation sequences, and to do work to persist state where necessary.

Summary

This chapter examined the application model, and in particular, the application lifecycle and related events. The tight resource constraints inherent in all mobile devices offer challenges for application developers, particularly with regard to CPU, memory, and disk space. The Windows Phone platform presents a seamless UX with reasonably fast switching between applications to provide an environment that appears to users as if multiple applications are running at the same time. More important, the platform exposes just the right number and type of events so that an application developer can hook into the system and use the opportunities presented to make the most of the phone's limited resources. If you pay attention to these events and take the recommended actions in your event handlers, your application takes part in the overall phone ecosystem, gives users a great experience, and cooperates with the platform to maintain system health.

Navigation State and Storage

The application model presents a user experience (UX) of multiple applications running concurrently, and the navigation model supports this by providing a framework for the user to navigate between pages within an application as well as between applications. At both the page level and the application level, the platform raises navigation events to which you can listen to maintain your application's state. As the user navigates away from one of your pages, or from your application altogether, you can persist any state you might need. Later, as the user navigates back to that page, or to your application, you can restore that state from persisted storage. All of this helps to support the UX of seamless switching between pages and between applications.

It is important to have a good understanding of the navigation model, so that your application can integrate seamlessly with the phone's ecosystem and behave in a manner that is consistent with other applications and with users' expectations. This chapter examines the navigation model as well as the events and methods with which you can take part in the model to provide the best possible UX.

Navigation and State

In the context of application navigation (both intra-application and inter-application), application state can be divided into three categories: transient page state, transient application state, and persistent state, which are summarized in Table 7-1.

In all forward navigation, *OnNavigatedFrom* is called for the current page, and then the new (destination) page is constructed from scratch and *OnNavigatedTo* is called for the new page. In backward navigation, *OnNavigatedFrom* is called for the current page, and then *OnNavigatedTo* for the destination page to which the user is returning (possibly with a page constructor call in between, for situations in which you were tombstoned). So, *OnNavigatedTo* is always called for the new page, just as the user navigates to it; *OnNavigatedFrom* is always called for the old page, just as the user navigates away from it. This holds true both for page navigation within an application and for navigation from one application to another, except, of course, that one application doesn't receive the events targeted for the other application. The situation with page constructors is more complicated. In *intra*-application navigation, a page constructor is only called in forward navigation, just before the *OnNavigatedTo*. In *inter*-application navigation, this is still technically true. The subtle distinction is that, as the user navigates back to a previous application through the backstack, this might cause fresh instances of a given page to be constructed. In reality, the previously tombstoned or terminated application is recreated, and the appropriate page constructor is called. From the user's perspective,

she is navigating back to a previously visited page; however, this is only logically the same page she saw before. In reality, it's a fresh instance. This behavior can be summarized as follows:

- Navigating forward (through a hyperlink or a call to *NavigationService.Navigate*) always constructs the target page. Even if an existing instance of the page exists on the backstack, a new one will be created (this differs from desktop Microsoft Silverlight, in which you can configure it to reuse instances).
- Navigating backward (via the Back button or *NavigationService.GoBack*) will not construct the target page if it already exists (for instance, the process has not been tombstoned since you last visited that page). If the application has been tombstoned and the page instance does not exist, it will be constructed.

It should be clear from this that the critical times to consider state management are in the *OnNavigatedTo* and *OnNavigatedFrom* handlers, not in the page constructor. Furthermore, it is sometimes useful to handle the *Loaded* event for a page, or even the *LayoutUpdated* event, but neither of these are suitable places to perform state management. Both of these are called more often than you might expect and are not intended for state management operations.

TABLE 7-1 Categories of Application and Page State

Type of State	Description	Guidelines
Transient page state	State specific to a page that does not need to persist between runs of the application; for example, the value of uncommitted text changes or the visual state of a page.	Store this in the <i>PhoneApplicationPage.State</i> property in the <i>NavigatedFrom</i> event, and retrieve it in the <i>NavigatedTo</i> event.
Transient application state	State that applies across the application that does not need to persist between runs of the application.	Store this in the <i>PhoneApplicationService.State</i> property when the application handles the <i>Deactivated</i> event, and retrieve it in the <i>Activated</i> event. Alternatively, store this in fields or properties of the <i>App</i> object (or of the <i>ViewModel</i> object, which is itself a property of the <i>App</i> object), and then serialize these.
Persistent state	State of any kind that needs to persist across runs of the application—essentially, anything that would upset the user if you didn't save it.	Store this to isolated storage during both the <i>Deactivated</i> and <i>Closing</i> events (the application might not return from <i>Deactivated</i> , and will not return from <i>Closing</i>). Also persist this during the <i>OnNavigatedFrom</i> call (for any state modified inside a page) or even periodically. Applications should not defer all saving until the user switches from the application; instead, they should incrementally save throughout the application's lifetime.

Application and Page State

The *PhoneApplicationPage.State* and *PhoneApplicationService.State* properties are *IDictionary* objects that are saved by the OS in a tombstoning scenario. All you have to do is write to them (or read from them) in the appropriate event handler. These dictionaries are not persisted across separate instances of the application; that is, across fresh launches from the Start page, and so on.

You can use the phone's *NavigationService* to navigate to another page. When the user navigates back from the second page, however, the second page is destroyed. If he navigates to the second page again, it will be recreated. The main page (that is, the entry point page) is not destroyed/recreated

during in-application navigation; however, it is destroyed/recreated if the user navigates away from the application (always in version 7, sometimes in 7.1). The sequence of creation and destruction across two pages in an application is shown in Figure 7-1.

If you were to implement a destructor in a page class, you could see more clearly that this would be invoked some time after the *OnNavigatedFrom* for that page. However, there is no useful work that you can do after the *OnNavigatedFrom*, and you cannot rely on the state of any objects after this call, so it is not useful to implement a destructor in a real application. Equally, there is nothing useful you can do in your *App* class after handling the *Closing* or *Deactivated* events. Finally, destructors can be invoked in an indeterministic manner, and might not be invoked at all.

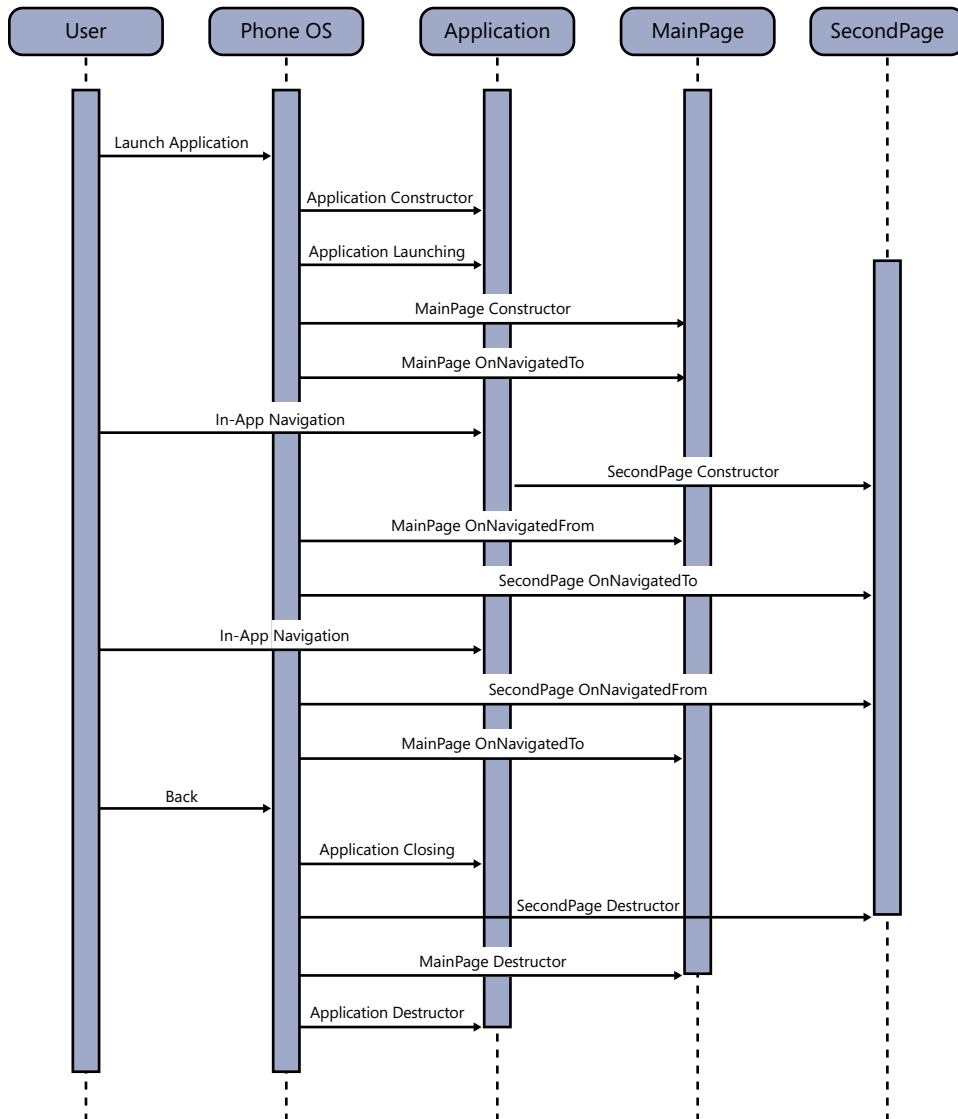


FIGURE 7-1 The sequence of page construction and destruction, and navigation events.

Because the application pages can be dynamically destroyed/recreated, the application should handle maintaining the state of each page itself. The following example illustrates this (this is the *Test Navigation* solution in the sample code). The application has two pages: *MainPage* and *Second Page*. Each page offers a *TextBlock* that displays the date and time at which the page was created. Each page also displays a *Button*, with which the user can jump from one page to the other, and back again.

```
<StackPanel>
  <TextBlock
    Name="txtCreated"
    Style="{StaticResource PhoneTextLargeStyle}" />
  <Button
    Content="go to second page" Click="Button_Click"
    HorizontalAlignment="Left"
    FontSize="{StaticResource PhoneFontSizeLarge}" />
</StackPanel>
```

The code-behind sets the *TextBlock* to the current *DateTime* at the time of construction. When the user taps the *Button*, the application navigates to the second page.

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    NavigationService.Navigate(new Uri("/SecondPage.xaml", UriKind.Relative));
}
```

The second page behaves in a similar way, with one subtle difference: the code in the *Button Click* handler navigates backward to the *MainPage*, not forward. It will become apparent later why this distinction is important.

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    NavigationService.GoBack();
}
```

The sequence of screenshots in Figures 7-2, 7-3, 7-4, and 7-5 shows the user experience and highlight the page creation behavior. The main page is created at 9:09:25; the second page is created at 9:09:55. When you navigate back to the main page, it still displays 9:09:25, as expected. However, when you navigate to the second page a second time, it now displays 9:10:33; the second page is not retaining its state because it's being recreated each time. Similarly, if the user navigates away from the application (by using the Start button, for example), and then goes back to the application, the "created" time of the main page will also change, because a new instance is created.

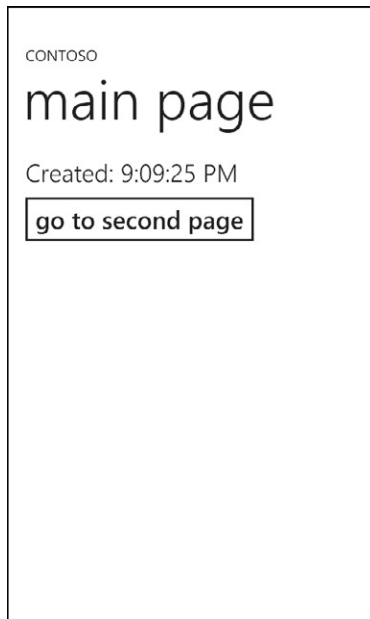


FIGURE 7-2 The main page is created. Note the “Created” time.

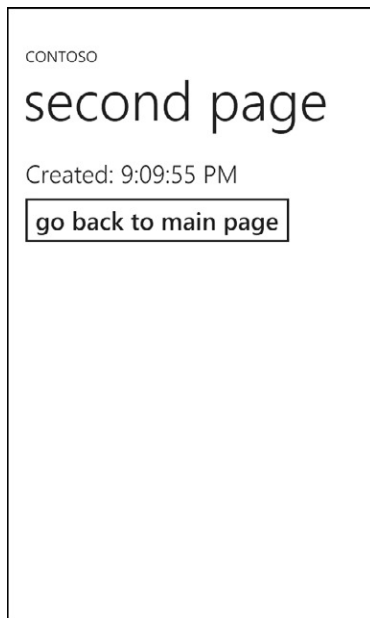


FIGURE 7-3 The user goes to the second page, which is created for the first time.

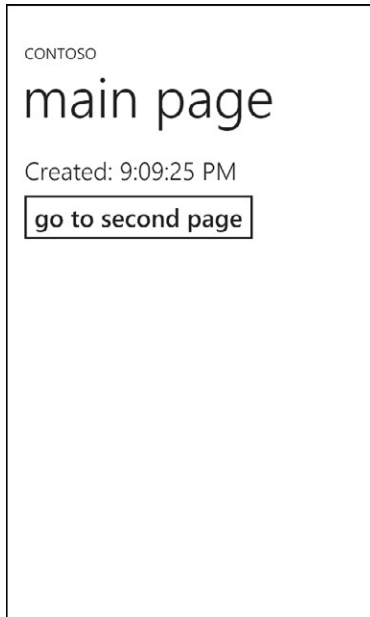


FIGURE 7-4 The user revisits the main page; the time hasn't changed.

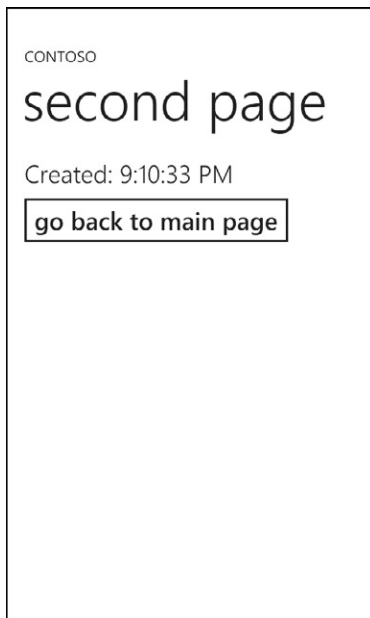


FIGURE 7-5 A second visit to the second page, but notice that the "Created" time has changed.

It might be that this is exactly the behavior you want, but there is some inconsistency here. The displayed time of the main page is the creation time, whereas the displayed time of the second page is the "last visited" time. If the intention is to have the second page always show the same value (that

is, the time of first creation, set once and never subsequently reset), then you need to do some work to establish that behavior. To put this in a more realistic context, imagine that it takes some time to compute and/or download the information needed for the second page. In this case, rather than compute it each time the page is visited, you want to compute it once and then cache the value. The next time the page is visited, you don't need to perform the calculation again (saving battery and time).

Here is how to fix this: in the `SecondPage.xaml.cs`, store the state of the page just before navigating away from it, and then restore it when the user navigates back to it. To do this, you need to override the virtual `OnNavigatedFrom` and `OnNavigatedTo` event handlers. In these methods, you save and restore the state that you care about in the `State` collection for the application. The `State` property is defined as an *IDictionary*, which is a very simple collection type. It includes a string indexer, which is used both for writing data to the collection and as the key to retrieve it subsequently. The string names used as the collection key for each state variable are completely arbitrary, although they obviously have to be unique within the application. Note that we're explicitly using the *application* state dictionary, not the *page* state dictionary. Using the page state dictionary would not work in this scenario, because it would be deleted when the user navigates back from the page.

```
public partial class SecondPage : PhoneApplicationPage
{
    private const String SECOND_CREATION_STATE = "SecondPageCreationTime";

    ...

    protected override void OnNavigatedFrom(NavigationEventArgs e)
    {
        PhoneApplicationService.Current.State[SECOND_CREATION_STATE] = txtCreated.Text;
    }

    protected override void OnNavigatedTo(NavigationEventArgs e)
    {
        if (PhoneApplicationService.Current.State.ContainsKey(SECOND_CREATION_STATE))
        {
            txtCreated.Text = (string)
                PhoneApplicationService.Current.State[SECOND_CREATION_STATE];
        }
    }

    private void Button_Click(object sender, RoutedEventArgs e)
    {
        NavigationService.GoBack();
    }
}
```

Do exactly the same for the main page (using a different dictionary key, of course).

```
public partial class MainPage : PhoneApplicationPage
{
    private const String MAIN_CREATION_STATE = "MainPageCreationTime";
```

```

...
protected override void OnNavigatedFrom(NavigationEventArgs e)
{
    PhoneApplicationService.Current.State[MAIN_CREATION_STATE] = txtCreated.Text;
}

protected override void OnNavigatedTo(NavigationEventArgs e)
{
    if (PhoneApplicationService.Current.State.ContainsKey(MAIN_CREATION_STATE))
    {
        txtCreated.Text = (string)
            PhoneApplicationService.Current.State[MAIN_CREATION_STATE];
    }
}

private void Button_Click(object sender, RoutedEventArgs e)
{
    NavigationService.Navigate(new Uri("/SecondPage.xaml", UriKind.Relative));
}
}

```

For each individual item of state that you persist, you should test *State.ContainsKey* to ensure that it is in the collection before attempting to access it. Also, if you have a large number of individual items, you can test to ensure that there is some persisted state in the collection before testing for each one individually, such as in the following example:

```
if (State.Keys.Count > 0) { // we have some state, now check for each item. }
```

You might wonder why this example uses *PhoneApplicationService.State* rather than *PhoneApplicationPage.State*. The answer is that in this example, the main page stays alive so long as the application is alive; however, the second page is destroyed and recreated each time a user navigates to or from it. So, for situations in which you want to cache data to be reused on subsequent forward navigations between pages (as in this example), you should use *PhoneApplicationPage.State* for the *MainPage* (that is, the first page in your application). For all other pages, you would use *PhoneApplicationService.State*, and it is therefore reasonable to use *PhoneApplicationService.State* for all pages. In other scenarios, it is common to use *PhoneApplicationPage.State* for maintaining all transient state that is specific to a page instance.



Note The application platform enforces limits on storing state. No single page is allowed to store more than 2 MB of data, and the application overall is not allowed to store more than 4 MB. However, these values are *far* larger than anything you should ever use. If you persist large amounts of state data, not only are you consuming system memory, but the time taken to serialize and deserialize large amounts of data is going to affect your application's pause/resume time. The basic guidance should be to store only simple things in the state dictionaries, and not, for example, the entire cache of a database.

Also, any object that you attempt to store in these dictionaries must be serializable. If the application attempts to store an object that cannot be serialized, this will throw an exception during debugging and fail silently during production runtime. Finally, note that there will be issues if you have serialized a type in an assembly that isn't loaded at startup. In this case, you need to load that assembly artificially in your *App* constructor; otherwise, you get a deserialization error.

Detecting Resurrection

A consistent UX is very important, and there are a few guidelines to keep in mind when managing state across the various application lifecycle events:

- Whenever the user returns to an application after navigating away (including using a Launcher or Chooser), it should appear to be in the same state as when she left it, not as a fresh instance.
- Conversely, whenever the user launches an application from Start, it should behave like a fresh instance.
- Applications must complete all actions in the *Deactivated* event within 10 seconds. If the application exceeds this time, it will be force-terminated and will not be available on the backstack. You must also complete activation *and* navigation within 10 seconds or you will be terminated. It is very hard to determine reliably how much work an application can get done in 10 seconds, as there will be many factors in play (how fast is the permanent storage, how much CPU is available for the application at this time, how much data it wants to save, how complex is it to serialize, and so on). To mitigate these factors, the application should save state incrementally during normal running before it gets a *Deactivated* event, and aim to do as little work as possible in the *Deactivated* event handler.

If an application wants to detect whether it's being resurrected after tombstoning, one option is to use simple flags, set at appropriate times. This technique can also be used to detect circumstances in which the user pressed Start and immediately pressed Back. Note that Windows Phone 7.1 introduces additional support for this scenario, which is discussed in Chapter 15, "Multi-Tasking and Fast App Switching." What follows is a technique that you can use for version 7.0 applications (demonstrated in the *Tombstoning* solution in the sample code). In this simple approach, you first define an enum type, as shown here:

```
public enum StartState { FreshStart, TombstoneResurrected, NonTombstoneReactivated }
```

In the *App* class, declare a field of this *enum* type, and also a simple *bool*.

```
public static StartState StartState;  
private static bool wasTerminated = true;
```

Then, set the values for these fields in the appropriate event handlers. First, the *Launching* event—you will get this only in the event of a fresh start, not in either the tombstone or reactivation cases.

```
private void Application_Launching(object sender, LaunchingEventArgs e)
{
    // This is the fresh start case:
    // [Closing-->]Ctor-->Launching
    StartState = StartState.FreshStart;
}
```

Next, in the *Deactivated* event handler, set the *bool* flag to *false*.

```
private void Application_Deactivated(object sender, DeactivatedEventArgs e)
{
    wasTerminated = false;
}
```

In the *Activated* event handler, test the value of this flag. You get the *Activated* event only in the case of tombstone resurrection or non-tombstone reactivation, not in the fresh start case. So, if this handler is invoked and the flag is set to *true*, then this must be the tombstone resurrection case. If the handler is invoked but the flag is *false*, then this must be the non-tombstone reactivation case (because the flag is still set from the previous *Deactivated* call, which means the constructor was not called), as demonstrated in the following:

```
private void Application_Activated(object sender, ActivatedEventArgs e)
{
    if (wasTerminated)
    {
        // This is the tombstone/resurrection case:
        // Deactivated-->Ctor-->Activated
        StartState = StartState.TombstoneResurrected;
    }
    else
    {
        // The non-tombstone case:
        // Deactivated-->Activated
        StartState = StartState.NonTombstoneReactivated;
    }
}
```

Finally, you can make use of the application's state over in your UI classes; for example, you can use it in the *MainPage OnNavigatedTo* override. This allows you to determine whether to try to retrieve dictionary state.

```
protected override void OnNavigatedTo(NavigationEventArgs e)
{
    switch (App.StartState)
    {
        case StartState.FreshStart :
            Debug.WriteLine("FreshStart - no state to retrieve");
            break;
        case StartState.TombstoneResurrected :
            Debug.WriteLine("Resurrected - we can retrieve state");
    }
}
```

```

        break;
    case StartState.NonTombstoneReactivated :
        Debug.WriteLine("Reactivated - no need to retrieve state");
        break;
    }
}

```

To make it easier to test/debug the non-tombstone reactivation scenario, you can simply pause during deactivation. This provides a chance to press Start and then Back quickly enough to go through the non-tombstone reactivation path in the code, as shown in the example that follows. Observe how *Sleep* is surrounded with conditional directives, just to make sure that even if you forget to remove this statement, it is never built in to the released application. Also note that this is not required in version 7.1 projects, where there is a Microsoft Visual Studio project setting that you can use to control tombstoning behavior under debugging.

```

private void Application_Deactivated(object sender, DeactivatedEventArgs e)
{
    wasTerminated = false;

    #if DEBUG
        // Pause on deactivation, so we can test the non-tombstone case.
        System.Threading.Thread.Sleep(5000);
    #endif
}

```

Navigation Options

The Windows Phone application platform offers a number of options for navigation, both within an application and externally. In addition to the *Navigate* and *GoBack* methods on the *Navigation Service* class, an application can use the *NavigateUri* property of a *HyperlinkButton* and can reroute navigation by either runtime interception or static URI mapping. There are also issues to consider with regard to navigation between pages across multiple assemblies, and options for passing data between pages during navigation.

One question that developers often ask is, “When would you ever need to invoke the *base.OnNavigatedTo/OnNavigatedFrom*?” When you use auto-complete to create an override for these virtual methods, Visual Studio generates a call to the base class version, which seems to imply that calling the base version is useful sometimes or always. In fact, this is simply an artifact of how auto-complete works; Visual Studio will always generate a base class call. With respect to *Page.OnNavigatedTo* and *OnNavigatedFrom*, the developer never needs to invoke the base version, because the base version is empty, so you can safely delete these calls.

Using *NavigateUri*

Instead of using a *Button* for navigation, it is common to use a *HyperlinkButton*. The *HyperlinkButton* class exposes a *Click* event, which you can handle in exactly the same way as you would the *Click* event on a regular *Button*. Alternatively—and more elegantly—you can specify the target URI in

XAML instead of in code. This removes the need for a *Click* handler in code. However, this technique only supports forward navigation; there is no way to specify the equivalent of *NavigationService.GoBack*. This means that you could use this technique for navigation from the *MainPage* forward to the *SecondPage* but not for navigating back from *SecondPage* to *MainPage*. In fact, if you did use this technique for both pages, navigation would appear to work (the user could successfully move from *MainPage* to *SecondPage* and back again), but in fact it creates new instances. This is by design: if a hyperlink could navigate backward, it would confuse the user, because the normal expectation is that hyperlinks navigate forward.

Also, assuming that you have the creation time state persisted for both pages in both sets of *OnNavigatedTo* and *OnNavigatedFrom*, then the *TextBlock* data displayed would also be correct. However, under the covers, what is really happening here is that each time the user navigates back to *MainPage*, a fresh instance of *MainPage* is created. This would be obvious if you remove the creation time state persistence for *MainPage* (or if you include a *Debug.WriteLine* statement in the *MainPage* constructor). The normal way to use a *HyperlinkButton* is to specify the *NavigateUri* in XAML, and not to specify a *Click* event handler.

```
<HyperlinkButton
    Content="go to SecondPage"
    HorizontalAlignment="Left"
    FontSize="{StaticResource PhoneFontSizeLarge}"
    Click="HyperlinkButton_Click"
    NavigateUri="/SecondPage.xaml"/>
```

Calling *NavigationService.GoBack* from any page but the first has exactly the same effect as the user pressing the hardware Back button. From the first page, pressing the hardware Back button exits the application, whereas calling *NavigationService.GoBack* will throw an exception. You can use the *CanGoBack* property to ensure that you don't attempt to navigate back when there's nothing left in the backstack.

Pages in Separate Assemblies

From an engineering perspective, it is perfectly reasonable to divide a solution into multiple assemblies—possibly with different developers working on different assemblies. This model also works with Phone pages; that is, one or more pages for an application could be built in separate assemblies. This technique can also help with application startup performance, because the code for the second and subsequent pages does not need to be loaded on startup, and the assembly load cost is deferred until the point when the user actually navigates to the second and subsequent pages, if ever. How does this affect navigation? Navigating back is always the same; *NavigationService.GoBack* takes no parameters. However, navigating forward requires a URI (either in the *NavigateUri* property or in the call to *Navigate*), and this URI must be able to be resolved. The simplest case is where the URI is relative to the application root, indicated with a leading slash. If the URI identifies a page that is in another assembly, the string format is as follows:

```
"/[assembly short name];component/[page pathname]"
```

So, for example, if you have *Page2* in a separate class library project named *PageLibrary*, then to navigate to *Page2*, you would use this syntax:

```
NavigationService.Navigate(new Uri(
    "/PageLibrary;component/Page2.xaml", UriKind.Relative));
```

You can see this at work in the *NavigatingAssemblies* solution in the sample code.

Fragment and QueryString

To pass state values between pages on navigation, an application can use *PhoneApplicationService.State*, isolated storage, or state fields/properties on the *App* object. In addition, you can use *Fragment* or *QueryString*. Note that you cannot use *Fragment* to navigate within a page; it can only be used when navigating to another page. Here is an example (the *NavigationQueryString* solution in the sample code) in which the main page offers a choice between Whisky and Gin. The user's choice is passed down to *Page2* either via a *Fragment* or via a *QueryString*, depending on which button the user clicks. This is illustrated in Figure 7-6.

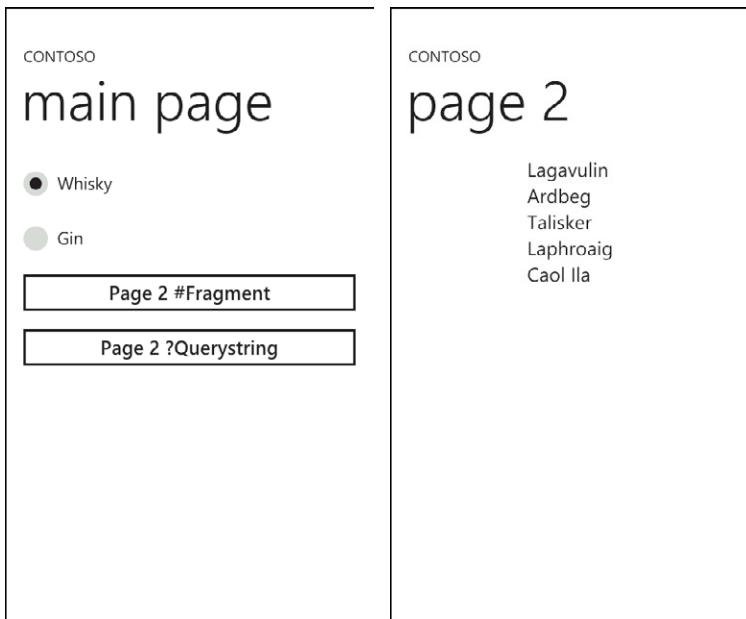


FIGURE 7-6 The *NavigationQueryString* sample tests navigation by using *Fragment* and *QueryString*.

Page2 has a *ListBox* that is data-bound to one of two *ObservableCollection* objects.

```
private ObservableCollection<string> whiskies = new ObservableCollection<string>();
private ObservableCollection<string> gins = new ObservableCollection<string>();

public Page2()
{
    InitializeComponent();
```

```

        whiskies.Add("Lagavulin");
        whiskies.Add("Ardbeg");
        whiskies.Add("Talisker");
        whiskies.Add("Laphroaig");
        whiskies.Add("Caol Ila");

        gins.Add("Gordons");
        gins.Add("Plymouth");
        gins.Add("Beefeater");
        gins.Add("Tanqueray");
        gins.Add("Sipsmith");
    }

```

To use a *Fragment*, you simply append a “#” (hash) character to the target URI, followed by the fragment value.

```

private void buttonPage2Fragment_Click(object sender, RoutedEventArgs e)
{
    if ((bool)radioWhisky.IsChecked)
    {
        NavigationService.Navigate(new Uri(
            "/Page2.xaml#whisky", UriKind.Relative));
    }
    else
    {
        NavigationService.Navigate(new Uri(
            "/Page2.xaml#gin", UriKind.Relative));
    }
}

```

On the navigation destination page, you could override *OnNavigatedTo*, but the *Fragment* is not accessible in that method. After *OnNavigatedTo* is called, the system calls *OnFragmentNavigation*, and it is here that you can get the *Fragment*.

```

protected override void OnFragmentNavigation(FragmentNavigationEventArgs e)
{
    base.OnFragmentNavigation(e);
    switch (e.Fragment)
    {
        case "whisky" :
            listDrinks.ItemsSource = whiskies;
            break;
        case "gin" :
            listDrinks.ItemsSource = gins;
            break;
    }
}

```

Conversely, you can provide a conventional query string, by appending a “?” (question mark) to the end of the URI, and then appending “key=value” pairs. Unlike *Fragment*, this allows you to pass more than one value, in the format:

```
"/[pagename].xaml?[param1=value1]&[param2=value2]&[param3=value3]"
```

For example:

```
private void buttonPage2Querystring_Click(object sender, RoutedEventArgs e)
{
    if ((bool)radioWhisky.IsChecked)
    {
        NavigationService.Navigate(new Uri(
            "/Page2.xaml?drink=whisky", UriKind.Relative));
    }
    else
    {
        NavigationService.Navigate(new Uri(
            "/Page2.xaml?drink=gin", UriKind.Relative));
    }
}
```

If you use a query string, then in the receiving page, this is provided as an *IDictionary* property on the *NavigationContext*, which itself is a property of the *Page* object. You can use both a query string and a fragment in the same URL, but this is not likely to be useful: it would require you to handle both, and to parse the URL in both cases to extract either one.

```
protected override void OnNavigatedTo(NavigationEventArgs e)
{
    if (NavigationContext.QueryString.ContainsKey("drink"))
    {
        try
        {
            switch (NavigationContext.QueryString["drink"])
            {
                case "whisky":
                    listDrinks.ItemsSource = whiskies;
                    break;
                case "gin":
                    listDrinks.ItemsSource = gins;
                    break;
            }
        }
        catch (Exception ex)
        {
            MessageBox.Show(ex.ToString());
        }
    }
}
```

Be aware that although the *QueryString* property will not be null, it might be empty, so you should check for this before attempting to access its collection. Note also that, as with any *Dictionary* object, if you attempt to use an indexer that does not exist in the collection, this will throw an exception. So, rather than using the indexer, you can use the *TryGetValue* method, instead. That is, change this:

```
switch (NavigationContext.QueryString["drink"])
{
    ...
}
```

to this:

```
string drinkType;
if (NavigationContext.QueryString.TryGetValue("drink", out drinkType))
{
    switch (drinkType)
    {
        ...
    }
}
```

The *NavigationMode* Property

Overrides of *OnNavigatedTo* and *OnNavigatedFrom* are passed a *NavigationEventArgs* object. This exposes two useful properties: *Content*, set to the instance of the destination page (always to, never from); and *Uri*, which will be the URI of the destination page.

Windows Phone 7.1 exposes a *NavigationMode* property. Interestingly, if you have the version 7.1 tools (including the 7.1 Visual Studio components), then you would see *NavigationMode* in the debugger even for a version 7 application where it is not available. The value of *NavigationMode* will be either *New* or *Back*, which identifies the direction of navigation. That is, forward equals *New*, backward equals *Back*. Forward equals *New* because navigating forward always creates a new instance of the target page. Forward does not equal *Forward* (as it sometimes does in desktop Silverlight) because Windows Phone has no notion of a forward stack, only a backward stack.

Without this, the developer needs to know the page hierarchy. That is, when you override *OnNavigatedTo*, are you arriving at the current page as the result of forward navigation or backward navigation? Similarly, in *OnNavigatedFrom*, are you going forward to a new page or backward?

It would certainly be useful to be able to evaluate this property. It is not exposed in Windows Phone 7, but it is exposed in version 7.1 for this very reason. If an application targets version 7, there is a suitable workaround. The code generated by Visual Studio for the *App* class includes a handler for the *Navigated* event on the *RootFrame*. This is a one-time handler that is implemented to assign the *RootVisual*, and then is unhooked. This is not interesting in this discussion, except to point out that there is *another* event that you can hook up in the *App* class for the *RootFrame*: the *Navigating* event. The point here is that this event handler is passed a *NavigatingCancelEventArgs* object, and this type does expose the *NavigationMode* property. It should be apparent from the name that this event is raised before the *Navigated* event, and therefore, before the calls to the virtual *OnNavigatedTo*/*OnNavigatedFrom* methods.

```
public App()
{
    UnhandledException += Application_UnhandledException;
    InitializeComponent();
    InitializePhoneApplication();

    RootFrame.Navigating +=
        new NavigatingCancelEventHandler(RootFrame_Navigating);
}
```



```
private void RootFrame_Navigating(object sender, NavigatingCancelEventArgs e)
{
    switch (e.NavigationMode)
    {
        case NavigationMode.Back :
            // Do something here.
            break;
        case NavigationMode.New :
            // Do something else here.
            break;
    }
}
```

It would be even more useful to declare a *NavigationMode* as a static field or property on the *App* class and simply set the value of this in the event handler. Then, this would be accessible from anywhere in the application.

```
private static NavigationMode navMode;
public static NavigationMode NavMode
{
    get { return navMode; }
}

public App()
{
    ...
    RootFrame.Navigating += new NavigatingCancelEventHandler(RootFrame_Navigating);
}

private void RootFrame_Navigating(object sender, NavigatingCancelEventArgs e)
{
    navMode = e.NavigationMode;
}
```

You can see this at work in the *TestNavigationMode* solution in the sample code.

Rerouting Navigation and URI Mappers

Another case for which it is useful to handle the *Navigating* event is when an application wants to cancel a navigation. This is a rare scenario—the page to which the user should navigate, given any known conditions, is normally predictable, and can be hard-wired (or dynamically determined) *before* navigation happens. One situation for which this is used is in the Microsoft Outlook application, when the user hits the Back button while composing a message. When this happens, Outlook asks the user if he wants to save or cancel his message. If he responds that he wants to save, Outlook cancels the navigation. For these rare scenarios, the platform allows an application to cancel a navigation in flight. Not only that, but it is possible to reroute a navigation from one target page to another at runtime, and there are two distinct ways to achieve this.

The first approach is navigation rerouting, as demonstrated in the *ReRouting* solution in the sample code. Suppose that you have a requirement by which most days of the week, when the user asks to navigate to *Page2*, you sent him to a default *Page2a*, but on Tuesdays, you send him instead to *Page2b*. In the *MainPage* code-behind, on the UI trigger to go to *Page2*, you navigate to *Page2*:

```
private void GotoPage2_Click(object sender, RoutedEventArgs e)
{
    NavigationService.Navigate(new Uri("/Page2.xaml", UriKind.Relative));
}
```

However, the application does not in fact contain a *Page2.xaml*. Instead, it contains a *Page2a.xaml* and a *Page2b.xaml*. In the *App* class, you hook the *Navigating* event on the *RootFrame* and perform some navigation rerouting whenever you detect that the user is attempting to go to *Page2*.

```
private void RootFrame_Navigating(object sender, NavigatingCancelEventArgs e)
{
    if (e.Uri.ToString() == "/Page2.xaml")
    {
        Uri newUri = null;
        if (DateTime.Now.DayOfWeek == DayOfWeek.Tuesday)
        {
            newUri = new Uri("/Page2a.xaml", UriKind.Relative);
        }
        else
        {
            newUri = new Uri("/Page2b.xaml", UriKind.Relative);
        }
        RootFrame.Dispatcher.BeginInvoke(() => RootFrame.Navigate(newUri));
        e.Cancel = true;
    }
}
```

Note that the handler uses *Dispatcher.BeginInvoke*. This is because the *Navigating* event is being handled during a navigation, and the platform does not allow overlapping navigations. Therefore, you must ensure that the second navigation is queued up. Meanwhile, you terminate the current navigation by setting *NavigatingCancelEventArgs.Cancel* to *true*.

The second technique you can use is URI Mapping (demonstrated in the *TestUriMapping* solution in the sample code). With this approach, instead of handling the *Navigating* event and manually cancelling and rerouting the navigation, you can simply provide a mapping from the original URI to a new URI. This can be either statically declared in XAML or dynamically determined in code. For example, here is a static mapping from *Page2* to *Page2b*:

```
<Application.Resources>
    <nav:UriMapper x:Name="mapper">
        <nav:UriMapping
            Uri="/Page2.xaml"
            MappedUri="/Page2b.xaml"/>
    </nav:UriMapper>
</Application.Resources>
```

This assumes that you have declared a *nav* namespace in the App.xaml.

```
xmlns:nav="clr-namespace:System.Windows.Navigation;assembly=Microsoft.Phone"
```

As this namespace indicates, the *UriMapper* and *UriMapping* types are declared in the *System.Windows.Navigation* namespace in the *Microsoft.Phone.dll*. Having declared the mapper and at least one mapping entry, you can use it in the application—typically after the standard initialization code.

```
UriMapper mapper = (UriMapper)Resources["mapper"];
RootFrame.UriMapper = mapper;
```

That would suffice for a static mapping. But if you need a dynamic mapping—as you do in this example—then you need to modify the *MappedUri* property before using it.

```
Uri newUri = null;
if (DateTime.Now.DayOfWeek == DayOfWeek.Tuesday)
{
    newUri = new Uri("/Page2a.xaml", UriKind.Relative);
}
else
{
    newUri = new Uri("/Page2b.xaml", UriKind.Relative);
}
UriMapper mapper = (UriMapper)Resources["mapper"];

// dynamic mapping, overwrites any MappedUri set statically in XAML.
mapper.UriMappings[0].MappedUri = newUri;

RootFrame.UriMapper = mapper;
```

Nonlinear Navigation Service

Navigation in Windows Phone is strictly linear. That is, the user navigates from one page to the next, with no loops, forks, skips, or hierarchy. An application can have a logical page hierarchy, but ultimately all navigation is strictly either forward or back. Every page the user visits is stacked up in a line, and if she wants to go back to a specific page, she must step back through every page in the stack between where she is and where she wants to go. The same model is used for both pages within an application and for navigating between applications.

The Nonlinear Navigation Service (NLNS) is an unsupported library built by the Windows Phone team to mitigate the problem of navigation loops. (You can access the library at <http://create.msdn.com/en-us/education/catalog/article/nln-serv-wp7>.) Note that this solves a problem for Windows Phone 7 applications, and that problem has been solved in the application platform itself for Windows Phone 7.1 applications. This is discussed further in Chapter 19, “Framework Enhancements.” So, this technique is strictly useful for version 7 applications only.

Suppose that the application's main page acts like a "home" page. Suppose also that one or more of the application's subpages has a "go back to main page" button. A good example for why you want to do this is a check-out experience on a shopping application. In this scenario, you don't want all the pages of the check-out (billing, address, confirmation, and so on) lying around on the backstack when you're done; you just want to be back at the beginning. Figure 7-7 distills the scenario with a simple three-page application. In this example, if the user navigates from the main page to page 2, and from there to a subpage of page 2, and then presses the "go back to main page" button, then what he (probably) expects to happen is a navigation loop.

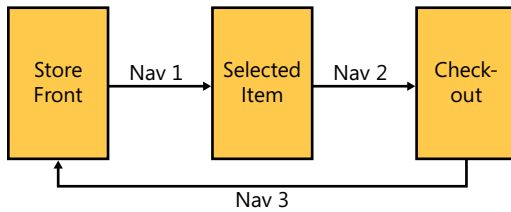


FIGURE 7-7 An application can give the impression of a logical navigation loop.

Figure 7-8 shows what actually happens; there is no loop, navigation is always linear.

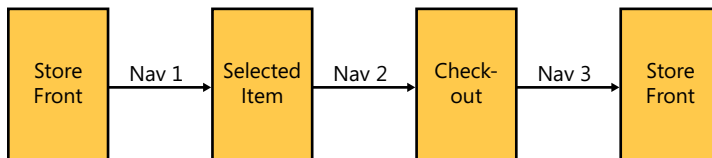


FIGURE 7-8 A navigation sequence perceived as looping is actually linear.

In other words, a new instance of *MainPage* is constructed. Furthermore, it is added to the application's backstack. So, if the user then presses the Back button from this second instance of *MainPage*, he would have to go back multiple times to exit from the app, as shown in Figure 7-9.

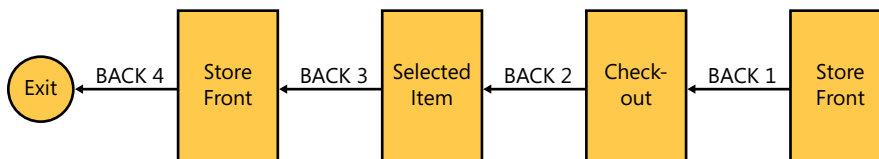


FIGURE 7-9 Introducing a logical navigation loop complicates the backstack.

The normal expected behavior when the user presses Back from the application's main page is to exit immediately. The NLNS prevents the application developer from adding perceived loops such as this one (that is, perhaps adding another main page into the navigation backstack). Every time you do a forward navigation, it determines if that page exists in the backstack, and if so, navigates back until it finds it. To the user, the strictly linear navigation stack appears to be a non-linear loop or fork. So, this is what the NLNS achieves, as shown in Figure 7-10.

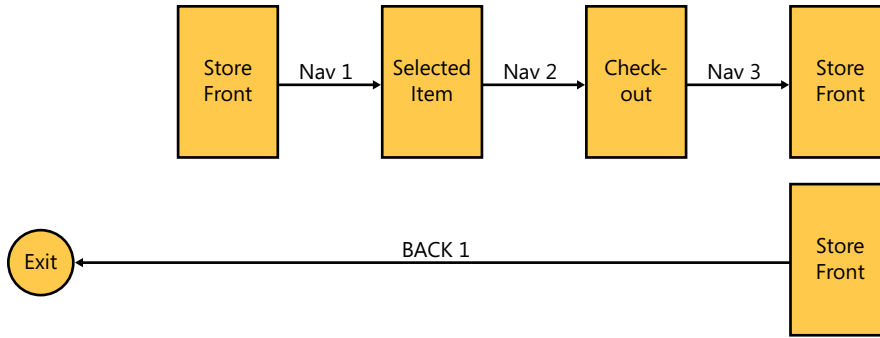


FIGURE 7-10 With the NLNS, your application can skip duplicate pages on navigation.

Of course, this is not restricted to the main page; the same behavior applies to any page. Regardless of any perceived forks or loops in page navigation, going back from a page always goes back to the same place. The NLNS achieves this by keeping a history list of all navigated pages. When the user navigates, the service checks to see if it needs to skip any intermediate pages, and if so, it recursively issues a *PhoneApplicationFrame.GoBack*. While it's doing this, it also makes the frame transparent so that there is no flicker as it navigates back.

This is demonstrated in the *TestNlns* application in the sample code. In this sample, the *MainPage* provides a *Button*, and the *Click* handler is implemented to navigate to *Page2*.

```

public partial class MainPage : PhoneApplicationPage
{
    private void gotoPage2_Click(object sender, RoutedEventArgs e)
    {
        NavigationService.Navigate(new Uri(
            "/Page2.xaml", UriKind.Relative));
    }
}

```

Page2 has a similar *Button*, with which the user can navigate to *Page2.1*.

```

public partial class Page2 : PhoneApplicationPage
{
    private void gotoPage2_1_Click(object sender, RoutedEventArgs e)
    {
        NavigationService.Navigate(new Uri(
            "/Page2_1.xaml", UriKind.Relative));
    }
}

```

Page2.1 has a *Button* with which users can navigate explicitly to the *MainPage*.

```

public partial class Page2_1 : PhoneApplicationPage
{
    private void gotoMainPage_Click(object sender, RoutedEventArgs e)
    {

```

```

        NavigationService.Navigate(new Uri(
            "/MainPage.xaml", UriKind.Relative));
    }
}

```

This one line is added to the end of the *App* constructor, to set up the NLNS (add a reference to the NLNS *WindowsPhoneStateService.dll*):

```
NonLinearNavigationService.Instance.Initialize(RootFrame);
```



More Info In Windows Phone 7, there were few scenarios for which a “home” button made sense. However, the introduction of new features in version 7.1 has increased the number of valid home scenarios. In particular, the use of secondary tiles can often make a home experience very useful. This is discussed more in Chapter 19.

Isolated Storage

It is important for an application to maintain consistent state in the face of user navigation. This inevitably means persisting state in some way. In Windows Phone 7, the only out-of-the-box form of persistent storage available on the phone itself is isolated storage. Windows Phone 7.1 introduces support for local databases (this is discussed in depth in Chapter 18, “Data Support”). Each application can only access its own isolated storage. There are two ways an application can use isolated storage:

- For application/user settings—typically, small pieces of data—via the *IsolatedStorageSettings* dictionary class.
- For anything else (as much data as you like, up to the limits of disk space on the device), via the *IsolatedStorageFile* APIs.

The isolated storage for all applications is in a subfolder of `\Applications\Data`. Below that path, there is a subfolder for each application, and each application’s isolated storage is in a further subfolder of `\Data\IsolatedStore`. The general path format is as follows:

```
\\Applications\\Data\\[Application ProductID]\\Data\\IsolatedStore\\[Arbitrary FileName]
```

For example:

```
\\Applications\\Data\\D48FD8D4-8B8A-45A2-81EC-17C35E7F4887\\Data\\IsolatedStore\\MyData.xml
```

Although this path will be visible in the debugger, all of these locations are strictly controlled from a security perspective, and there is no way for an application to get any access (or even any visibility) of any other application’s isolated storage. Indeed, an application cannot see anything outside of its own isolated storage subfolder.

Simple Persistence

The following example (*SimplePersistence* in the sample code) illustrates the simple use of isolated storage, both for *IsolatedStorageSettings* and for the *IsolatedStorageFile* APIs. In this example, you offer the user two *TextBox* controls: one for a small piece of data (his name), and another for a bigger piece of data (some arbitrary text). Although the second piece of data is still fairly small, the idea here is to distinguish between relatively small and large pieces of data that you want to persist. You persist the small data by using *IsolatedStorageSettings*; you persist the big data by using *IsolatedStorageFile* APIs. The application UI is shown in Figure 7-11.

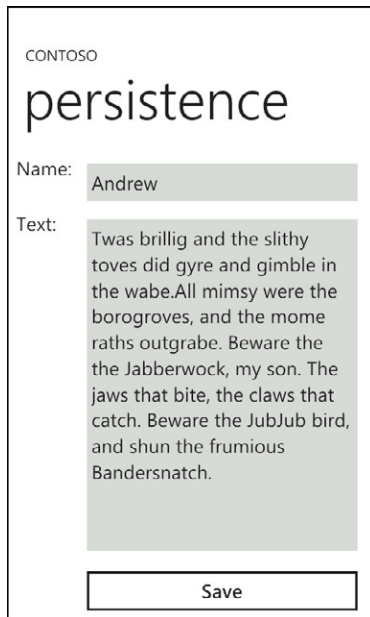


FIGURE 7-11 You can persist application data by using isolated storage.

When the user taps the Save button, you persist the data to a file in the application's isolated storage. Upon saving, you open the file, or create it if it doesn't already exist, and then use a simple *StreamWriter* to write out the text. In this example, you overwrite the contents each time. Note that you should be careful to call *Save* on the *IsolatedStorageSettings* object, because writes to physical storage are deferred by default—typically until the application closes. If you want to ensure that data is persisted to disk at the point where you logically write to the dictionary, and for it to be robust in the face of application exceptions, you should call *Save* proactively.

```
private const string fileName = "UserText.txt";

private void saveButton_Click(object sender, RoutedEventArgs e)
{
    IsolatedStorageSettings.ApplicationSettings["UserName"] = this.userName.Text;
    IsolatedStorageSettings.ApplicationSettings.Save();
}
```

```

        using (IsolatedStorageFile isoFile =
            IsolatedStorageFile.GetUserStoreForApplication())
        {
            using (IsolatedStorageFileStream isoStream =
                isoFile.OpenFile(fileName, FileMode.Create))
            {
                using (StreamWriter writer = new StreamWriter(isoStream))
                {
                    {
                        writer.WriteLine(this.userText.Text);
                    }
                }
            }
        }
    }
}

```

Conversely, upon startup, in the main page's constructor, you attempt to retrieve both sets of data, fetching the username from application settings, if it exists, and reading isolated storage to retrieve the user's text file. Note that you would generally want to avoid doing I/O during the constructor because I/O is a relatively slow operation, and you don't want to risk an unacceptable UX. If you do find yourself loading a large amount of data at startup, and if you cannot divide it and defer it a later point, then you should probably do this on a background thread, while showing a progress bar or similar "loading..." indicator.

```

public MainPage()
{
    InitializeComponent();

    string userName;
    if (IsolatedStorageSettings.ApplicationSettings.TryGetValue<String>
        ("UserName", out userName))
    {
        this.userName.Text = userName;
    }

    using (IsolatedStorageFile isoFile =
        IsolatedStorageFile.GetUserStoreForApplication())
    {
        if (isoFile.FileExists(fileName))
        {
            using (IsolatedStorageFileStream isoStream =
                isoFile.OpenFile(fileName, FileMode.Open))
            {
                using (StreamReader reader = new StreamReader(isoStream))
                {
                    {
                        this.userText.Text = reader.ReadToEnd();
                    }
                }
            }
        }
    }
}

```


An application can create its own directory structure within its private isolated storage, if required. For example, you could create a folder named “Docs,” and save your text file within that folder quite simply by using *CreateDirectory* and specifying the appropriate path. This is demonstrated in the *SimplePersistence_directory* solution in the sample code.

```
private const string storageDir = "Docs";
private const string fileName = @"Docs\UserText.txt";

private void saveButton_Click(object sender, RoutedEventArgs e)
{
    IsolatedStorageSettings.ApplicationSettings["UserName"] = this.userName.Text;
    IsolatedStorageSettings.ApplicationSettings.Save();

    using (IsolatedStorageFile isoFile =
        IsolatedStorageFile.GetUserStoreForApplication())
    {
        if (!isoFile.DirectoryExists(storageDir))
        {
            isoFile.CreateDirectory(storageDir);
        }

        using (IsolatedStorageFileStream isoStream =
            isoFile.OpenFile(fileName, FileMode.Create))
        {
            using (StreamWriter writer = new StreamWriter(isoStream))
            {
                writer.WriteLine(this.userText.Text);
            }
        }
    }
}
```

Obviously, this example is very simple. Maintaining state across page navigation on the basis of individual data items rapidly becomes unmanageable. A better approach is to encapsulate all of the persistable items into a class, and then just save/load an instance of that class. This approach allows the application to manage all of its state persistence entities and behavior in one place. This will reduce the scope for bugs and for any increased complexity that might arise due to dependencies between variables. It will also often provide performance benefits, especially if the application typically uses a high proportion of the individual items. Disk I/O is a relatively expensive operation in terms of time on the phone, and therefore, potentially in terms of the UX. Reducing disk read/write calls to a more chunky, less chatty model is in general good practice, but you should balance this with the need to avoid a high volume of read/writes at critical points in the application lifecycle. Specifically, it is important to save state relatively often, so that you don’t have to do it all during deactivation, because you only have a limited amount of time to complete deactivating before the application process is terminated.

In some cases, it is also appropriate to persist the entire viewmodel—not so much for intra-application page navigation, but certainly for inter-application navigation. Recall that the application and page *State* dictionaries have a 2 MB per-page and 4 MB per-application limit. However, there is no such limit on an application’s use of isolated storage. Although parts of the phone’s physical storage will be reserved for the system, there is a single logical quota allocated for application isolated

storage. The individual isolated storage for each application uses this same logical quota. When the available storage space is reduced to 10 percent free, the system sends the user a notification. This is his opportunity to delete unwanted data, such as photos, videos, and music. This is also his opportunity to uninstall applications that he suspects are consuming a lot of storage space. However, this is speculative at best, because there is no indication on a per-application basis on the phone as to how much storage any application is consuming. Indeed, it is possible for any one application to consume the entire storage capacity, thus starving all other applications of storage.

This goes back to the notion discussed in Chapter 1, “Vision and Architecture,” of the dynamic tension between an application platform that maintains overall device health, and a development platform that gives developers the freedom—and responsibility—to self-police. At its simplest, an application that consumes all available isolated storage space will not compromise system health to the extent that it could prevent the user from making an emergency call. On the other hand, it is very likely to prevent the user from installing more applications or even running some applications. And it will almost certainly lead to the user removing the badly behaved application and giving it a low rating on the marketplace. Unfortunately, there is no good way for a user to determine which application(s) are behaving badly, beyond simple observation.

Here are a few issues to keep in mind when working with isolated storage:

- During development, working with either the emulator or a physical device, when doing an incremental build, the application’s isolated storage is left untouched by the build system. Any usage of storage will be retained between executions and builds. This includes a rebuild if this is not preceded by cleaning the project or solution. Note that the exact behavior of Visual Studio’s rebuild command depends on the build targets used in the current projects and solution. For most project types, rebuild performs a clean and then a build, but for Windows Phone projects, rebuild only builds if any of the source files have changed since the last build. Cleaning the project itself also does not affect the application installed on the emulator/device, nor its storage. However, cleaning and then building (or rebuilding) and then executing the application will uninstall and then reinstall the application. Uninstall always completely removes all isolated storage used by an application.
- Something similar happens if an application is published to marketplace and later upgraded or updated. If the application’s *ProductId* does not change, then an update will remove and replace the application in the application’s install folder, but will not touch the application’s isolated storage. If an application update includes changes to the stored data schema, or if the update requires cleaning the storage for any other reason, then it would be the developer’s responsibility to take action.

- Wherever possible, for data that the application uses internally and never surfaces to the user, the application should delete this data when it is no longer needed. The compromise here is when you compute data dynamically and then want to cache it for performance reasons. You can make a judgement call as to where is the optimal place to draw the line between consuming disk space and consuming processor cycles at runtime.
- If it is too difficult to make a good judgement call, developers are encouraged to follow the same model that Internet Explorer uses: offer the user a mechanism by which she can delete these files whenever she wants to free up space on the phone. It is “good citizen” behavior for an application to allow the user to delete unnecessary data, especially if there is a lot of it.
- There is another reason why an application should provide this capability. Any data that an application saves on behalf of the user that might be considered to be personal data should also be able to be deleted by the user. The user should be in control of her data, and it is not always obvious what kind of data any given user might consider to be personal. Most users would consider personal data to include any personally identifiable information, but also items such as photos, run logs of a “run-tracker” type application, to-do lists, any notes the user has created, lists of favorites, and so on.
- The documentation states that *IsolatedStorageSettings* for an application are saved when the application exits. However, this is not always guaranteed. It depends on what work the application does as it is terminated. If it tries to do too much work, the *Save* might not be called. Internally, the *Save* is actually invoked in a finalizer, and finalizers might not be invoked in some scenarios. Certainly, data is not actually written to the disk when the application writes to *IsolatedStorageSettings*. For these reasons, the application can consider invoking *Save* explicitly at a suitable point. As always, there is a trade-off. If *Save* is called too frequently, this implies frequent disk I/O operations, which are time-consuming and costly in terms of battery consumption. A suitable compromise would be to batch the work, and invoke *Save* during *OnNavigatedFrom*.

Persisting the ViewModel

The following example shows the standard Visual Studio data-bound application project that is enhanced to persist its viewmodel to isolated storage. Figure 7-12 shows the UI (this is the *IsoData Bound* solution in the sample code). Three *AppBar* buttons have been added to load the data from storage, save the data to storage, and clear the data from the viewmodel in memory.

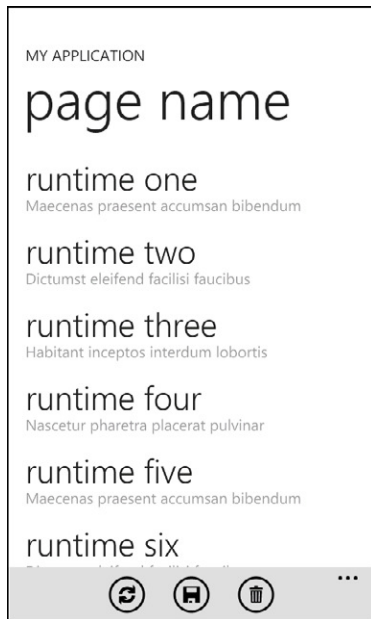


FIGURE 7-12 The *IsoDatabound* sample adds UI for saving, clearing, and restoring a persisted viewmodel.

When the application starts, it has no data in the viewmodel. To ensure this, you comment out the Visual Studio–generated code in *Activated* event handler, which would otherwise load the data upon startup.

```
private void Application_Activated(object sender, ActivatedEventArgs e)
{
    // Ensure that application state is restored appropriately
    //if (!App.ViewModel.IsDataLoaded)
    //{
    //    App.ViewModel.LoadData();
    //}
}
```

For the same reason, you comment out the same code that Visual Studio provided in the *Loaded* event handler in the *MainPage*. Conversely, you add that same code into the *Click* handler for the “Load” *ApplicationBar* button. The other two buttons invoke new *SaveData* and *ClearData* methods, which you will add to the viewmodel. As a result of these changes, the viewmodel (and therefore the data-bound view) will only be populated with data on user control.

```
private void MainPage_Loaded(object sender, RoutedEventArgs e)
{
    //if (!App.ViewModel.IsDataLoaded)
    //{
    //    App.ViewModel.LoadData();
    //}
}
```

```

private void Load_Click(object sender, EventArgs e)
{
    if (!App.ViewModel.IsDataLoaded)
    {
        App.ViewModel.LoadData();
    }
}

private void Save_Click(object sender, EventArgs e)
{
    App.ViewModel.SaveData();
}

private void Clear_Click(object sender, EventArgs e)
{
    App.ViewModel.ClearData();
}

```

In the viewmodel, the *ClearData* and *SaveData* methods are quite simple. *ClearData* removes all the items from the collection. *SaveData* saves the in-memory collection to an arbitrary file in isolated storage by using the standard *XmlSerializer*.

```

private const string storageFile = "ViewModel.xml";

internal void ClearData()
{
    this.Items.Clear();
    this.IsDataLoaded = false;
}

internal void SaveData()
{
    using (IsolatedStorageFile isoFile =
        IsolatedStorageFile.GetUserStoreForApplication())
    {
        using (IsolatedStorageFileStream isoStream =
            isoFile.OpenFile(storageFile, FileMode.Create))
        {
            XmlSerializer xs = new XmlSerializer(
                typeof(ObservableCollection<ItemViewModel>));
            xs.Serialize(isoStream, Items);
        }
    }
}

```

Because you're using XML serialization, the file will end up with the format shown in the code example that follows. Note that the Windows Phone 7 SDK did not provide any tools for examining isolated storage files. However, the Windows Phone 7.1 SDK does include the Isolated Storage Explorer Tool, which works for both versions 7 and 7.1 isolated storage. Further details on this are available in Chapter 18.

```

<?xml version="1.0"?>
<ArrayOfItemViewModel xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">

```

```

<ItemViewModel>
  <LineOne>runtime one</LineOne>
  <LineTwo>Maecenas praesent accumsan bibendum</LineTwo>
  <LineThree>Facilisi faucibus habitant inceptos interdum lobortis nascetur pharetra
lacerat pulvinar sagittis senectus sociosqu</LineThree>
</ItemViewModel>
...
</ArrayOfItemViewModel>

```

Conversely, the *LoadData* method first tries to load data from isolated storage. When it successfully opens the data file, it deserializes it into the *Items* collection. If, after this, there are still no items in the collection, then you create them in code, as in the original Visual Studio-generated version.

```

public void LoadData()
{
    using (IsolatedStorageFile isoFile =
        IsolatedStorageFile.GetUserStoreForApplication())
    {
        if (isoFile.FileExists(storageFile))
        {
            using (IsolatedStorageFileStream isoStream =
                isoFile.OpenFile(storageFile, FileMode.Open))
            {
                XmlSerializer xs = new XmlSerializer(
                    typeof(ObservableCollection<ItemViewModel>));
                items = (ObservableCollection<ItemViewModel>)
                    xs.Deserialize(isoStream);
            }
        }

        if (this.Items.Count == 0)
        {
            this.Items.Add(new ItemViewModel() { LineOne = "runtime one", LineTwo = "Maecenas
raesent accumsan bibendum", LineThree = "Facilisi faucibus habitant inceptos interdum
obortis nascetur pharetra placerat pulvinar sagittis senectus sociosqu" });
            ...
        }
        this.IsDataLoaded = true;
    }
}

```

In its current form, the application has a bug. The *SaveData* and *ClearData* methods work correctly, and the viewmodel items collection will be saved to isolated storage. The *LoadData* method will open and read the saved file and will read the data into the viewmodel collection. Unfortunately, the fact that the collection now has new data is not communicated to the data-binding framework, so the data will not be displayed in the view. There are two options for fixing this. The first approach is to read the data into a temporary collection, and then walk the collection manually, adding each item to the *Items* collection. The consequence of adding items to the collection will raise the *NotifyCollectionChanged* event for each one, thereby alerting data-binding to fetch new data for the view.

```

using (IsolatedStorageFileStream isoStream = isoFile.OpenFile(storageFile, FileMode.Open))
{
    XmlSerializer xs = new XmlSerializer(
        typeof(ObservableCollection<ItemViewModel>));

    //items = (ObservableCollection<ItemViewModel>)
    var v = (ObservableCollection<ItemViewModel>) xs.Deserialize(isoStream);
    foreach (var i in v)
    {
        Items.Add(i);
    }
}

```

A more elegant (and faster) approach is to update the definition of the *Items* property so that its property setter raises a *NotifyPropertyChanged* event when the collection itself is changed.

```

//public ObservableCollection<ItemViewModel> Items { get; private set; }
private ObservableCollection<ItemViewModel> items;
public ObservableCollection<ItemViewModel> Items
{
    get { return items; }
    private set
    {
        if (items != value)
        {
            items = value;
            NotifyPropertyChanged("Items");
        }
    }
}

```

Note that this approach relies on assigning the file data to the *Items* property, not directly to the *items* field that backs the property.

```
Items = (ObservableCollection<ItemViewModel>) xs.Deserialize(isoStream);
```

Serialization Options

There are three standard choices for serializing data in Windows Phone applications: *XmlSerializer*, *DataContractSerializer*, and *DataContractJsonSerializer*. Along with these types, there are a set of supporting *DataContract* and *DataMember* attribute types. There are pros and cons to each approach, and this section summarizes the differences between them.

First, just to get this out of the way, the *Serializable* attribute and *ISerializable* interface are desktop Microsoft .NET features, not available for Silverlight or Windows Phone.

XmlSerializer:

- The *XmlSerializer* does not require declaring any attributes on public members of the type to be serialized. This is effectively an “opt-out” model; every public property will be serialized unless explicitly excluded.
- The type to be serialized must have an explicit or implicit default constructor, and all members to be serialized must be public properties with both *get* and *set* methods.
- The *XmlSerializer* does not use (nor understand) the *DataContract* attribute, which makes it problematic for schema versioning.
- It will generate default XML namespace references, and there is no option to change these.
- It will include fields and properties with null values.
- The *XmlSerializer* affords the developer considerable control over how a property is serialized, including whether it is represented as a node or an attribute. This level of control means that it is possible to devise a serialization format that results in significantly smaller output than when using *DataContractSerializer*. The smaller data size with *XmlSerializer* can make it a better choice, even though *DataContractSerializer* might be faster.
- It requires adding a reference to *System.Xml.Serialization.dll*.

DataContractSerializer:

- The *DataContractSerializer* is the default mechanism in Windows Communication Foundation (WCF) and is therefore supported directly by various tools and proxy-generators, including Visual Studio.
- The *XmlSerializer* has been part of the base class library since .NET 1.0, and its implementation has not changed much in that time. By comparison, *DataContractSerializer* was introduced in .NET 3.0, and considerable work went into optimizing its performance so that it will generally be marginally faster than *XmlSerializer*. On the other hand, this optimization also means that the developer has slightly fewer options for controlling the operation.
- If no attributes are applied to the type being serialized, then all public members will be serialized. In other words, if you don’t specify the *DataContract* attribute, this is an opt-out model. The *DataContract* attribute can be applied to the type, and *DataMember* attribute can be applied to members of that type to be serialized. If the *DataContract* attribute is applied, then by default no members will be serialized unless explicitly attributed with *DataMember*—that is, if you do specify the *DataContract* attribute, then this is an opt-in model.
- With *DataContract*, the developer can specify the name and namespace of each member, giving the developer more flexibility in conforming to contracts and resolving name clashes.

- The *IgnoreDataMember* can be used to exclude a member from serialization for circumstances in which the type itself is not attributed with *DataContract*.
- The *DataContractSerializer* can serialize any data members, including fields, properties, and internal members if you have the *InternalsVisibleTo* attribute on the assembly. A property to be serialized does not need to have a set method, although if it doesn't, then while the property can be serialized, it cannot be deserialized. All serialized members will be serialized as elements, and there is no option to represent them as attributes.
- It requires adding a reference to *System.Runtime.Serialization.dll*.

DataContractJsonSerializer:

- A simple variation on the *DataContractSerializer*, with all the same characteristics. The difference is that the data format will be JavaScript Object Notation (JSON) instead of XML. JSON format data will be considerably smaller than the equivalent XML data, because it avoids the overhead of opening and closing XML tags.
- The *DataContractJsonSerializer* requires adding a reference to *System.ServiceModel.Web.dll*.

Figure 7-13 depicts an example application that can be used to exercise these options and compare the performance and data size. This is the *SerializeOptions* solution in the sample code.

CONTOSO

serialization

id

name

salary

☒ XmlSerializer
☐ DataContractSerializer
☐ DataContractJsonSerializer

FIGURE 7-13 Applications have three main serialization options from which to choose.

The three *TextBox* controls are two-way data-bound to a simple *Employee* class. The user can employ the four *ApplicationBar* buttons to save the data to isolated storage, clear the in-memory data (and therefore the UI), load previously saved data from isolated storage, and delete any existing isolated storage data file for this data. For both load and save operations, the application uses the *XmlSerializer*, *DataContractSerializer*, or *DataContractJsonSerializer*, depending on which *RadioButton* is clicked. In each case, the application accesses its isolated storage and creates or opens the same data file. For example, the method invoked by the UI to save by using *XmlSerializer* is shown in the following:

```
private void SaveXml()
{
    using (IsolatedStorageFile isoFile =
        IsolatedStorageFile.GetUserStoreForApplication())
    {
        using (IsolatedStorageFileStream isoStream =
            isoFile.OpenFile(storageFile, FileMode.Create))
        {
            XmlSerializer xs = new XmlSerializer(typeof(Employee));
            xs.Serialize(isoStream, emp);
        }
    }
}
```

The methods to invoke *DataContractSerializer* and *DataContractJsonSerializer* differ only in the final serialization calls. That is, the use of *XmlSerializer* and the *Serialize* method:

```
XmlSerializer xs = new XmlSerializer(typeof(Employee));
xs.Serialize(isoStream, emp);
```

is replaced with either *DataContractSerializer* and *WriteObject*:

```
DataContractSerializer dcs =
    new DataContractSerializer(typeof(Employee));
dcs.WriteObject(isoStream, emp);
```

or *DataContractJsonSerializer* and *WriteObject*:

```
DataContractJsonSerializer dcjs =
    new DataContractJsonSerializer(typeof(Employee));
dcjs.WriteObject(isoStream, emp);
```

The save operations are also broadly similar.

```
private void LoadXml()
{
    using (IsolatedStorageFile isoFile =
        IsolatedStorageFile.GetUserStoreForApplication())
    {
        if (isoFile.FileExists(storageFile))
        {
            using (IsolatedStorageFileStream isoStream =
                isoFile.OpenFile(storageFile, FileMode.Open))
```

```

        {
            XmlSerializer xs = new XmlSerializer(typeof(Employee));
            Employee e = (Employee)xs.Deserialize(isoStream);
            emp.ID = e.ID;
            emp.Name = e.Name;
            emp.Salary = e.Salary;
        }
    }
}

private void LoadDataContract()
{
    ...
        DataContractSerializer dcs =
            new DataContractSerializer(typeof(Employee));
        Employee e = (Employee)dcs.ReadObject(isoStream);
    ...
}

private void LoadJson()
{
    ...
        DataContractJsonSerializer dcjs =
            new DataContractJsonSerializer(typeof(Employee));
        Employee e = (Employee)dcjs.ReadObject(isoStream);
    ...
}

```

Examples of the resulting data file that is generated with each approach are listed in the example that follows. In terms of size, it should be clear that the JSON format is always going to be smaller, and the larger the data set (or strictly, the higher the number of elements), the bigger the size difference.

XmlSerializer:

```

<?xml version="1.0"?>
<Employee
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <ID>876</ID>
  <Name>Andrew</Name>
  <Salary>123.45</Salary>
</Employee>

```

DataContractSerializer:

```

<Employee
  xmlns:i="http://www.w3.org/2001/XMLSchema-instance"
  xmlns="http://schemas.datacontract.org/2004/07/SerializeOptions">
  <ID>876</ID>
  <Name>Andrew</Name>
  <Salary>123.45</Salary>
</Employee>

```

DataContractJsonSerializer:

```
{"ID":876,"Name":"Andrew","Salary":123.45}
```

The serialization metadata is managed by the use of the *DataContract* attribute on the *Employee* class, and the *DataMember* attribute on those members of the class that you want to serialize/deserialize. For example, by default, the *DataContract Name* would be the same name as the class type, and the *Namespace* would be the standard XML schema namespace: <http://www.w3.org/2001/XMLSchema>. By the same token, the *Name* of each *DataMember* would be the name of the property itself. You can change these values to suit your own requirements. Perhaps you want the *Employee* to be persisted as the *Person* element, and the *ID* property to be persisted as the *Code* element, and so on.

```
[DataContract(Name="Person", Namespace="http://www.contoso.com")]
public class Employee : INotifyPropertyChanged
{
    private int id;
    [DataMember(Name="Code")]
    public int ID
    {
        ...
    }
}
```

The listing that follows shows the file as generated by the *DataContractSerializer* and the *DataContractJsonSerializer*, when the *DataContract* and *DataMember* attributes on the *Employee* type specify additional parameters, and the *Salary* member is not attributed with *DataMember* (or is attributed with *IgnoreDataMember*):

```
<Person
  xmlns:i="http://www.w3.org/2001/XMLSchema-instance"
  xmlns="http://www.contoso.com">
  <Code>876</Code>
  <Name>Andrew</Name>
</Person>

{"Code":876,"Name":"Andrew"}
```

From a functional perspective, the choices can be summarized as follows:

- Use *XmlSerializer* if the application needs precise control over the data format.
- Use *DataContractSerializer* for ease of development, especially when interoperating with WCF web services. Even if you intend to use JSON for compactness, it is much easier to develop and debug against XML.
- Use *DataContractJsonSerializer* if compact data is the critical decision point.

Apart from differences in functionality, the different techniques also exhibit differences in performance. This is a lot harder to pin down; the exact performance characteristics depend not only on differences in functionality, but also on the data itself. For example, serializing a type that has a large

amount of small elements will exhibit a different performance profile from serializing a type that has a small amount of large elements. Metrics for serialization will be different from deserialization for the same payload, and so on. Figure 7-14 shows a variation on the previous application (*Serialize Options_Perf* in the sample code), in which the user can serialize/deserialize multiple copies of a data set, specify how many copies to work with, and report on the save and load times.

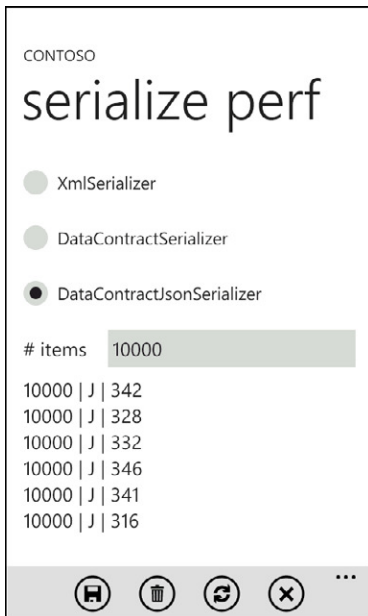


FIGURE 7-14 It's useful to compare performance among the different serialization options.

As before, the user can choose one of the three serialization techniques. He can type in the number of items to serialize (deserialization, obviously, will simply read back however many items were written). For each save/load operation, you surround the critical work with a simple timestamp, compose a string from the time difference, and add that string to a list. This list is used as the *ItemsSource* for a *ListBox*. For example, the serialization code for the *XmlSerializer* is shown in the following:

```
private long itemCount;
private ObservableCollection<string> results = new ObservableCollection<string>();

private void SaveXml()
{
    using (IsolatedStorageFile isoFile =
        IsolatedStorageFile.GetUserStoreForApplication())
    {
        using (IsolatedStorageFileStream isoStream =
            isoFile.OpenFile(storageFile, FileMode.Create))
        {
            DateTime start = DateTime.Now;
            XmlSerializer xs = new XmlSerializer(typeof(List<Employee>));
            xs.Serialize(isoStream, emps);
            DateTime end = DateTime.Now;
```

```

        TimeSpan diff = end - start;
        results.Insert(0,String.Format("{0} | {1} | {2}",
            itemCount, "X", diff.TotalMilliseconds));
    }
}

```

The *emps* variable in the preceding code is a *List<Employee>*. When the user changes the number of items to process, you fetch the text, parse it to a long integer, and add that number of *Employees* to the list.

```

private List<Employee> emps = new List<Employee>();

private void numItems_LostFocus(object sender, TextChangedEventArgs e)
{
    itemCount = Int64.Parse(numItems.Text);
    if (emps != null)
    {
        emps.Clear();
    }
    else
    {
        emps = new List<Employee>();
    }
    for (int i = 0; i < itemCount; i++)
    {
        emps.Add(emp);
    }
}

```

With this very simple example and this particular data payload (working with 10,000 Employees for each save/load operation), the results provided the averages shown in Table 7-2 over long runs. The interesting anomaly is that *XmlSerializer* was unexpectedly the fastest to serialize this particular payload. Apart from that, the figures follow commonly observed patterns.

TABLE 7-2 Anecdotal Serialization Performance Comparisons

Type	Save MSec	Load MSec	File Size Kb
<i>XmlSerializer</i>	280	525	987
<i>DataContractSerializer</i>	351	420	508
<i>DataContractJsonSerializer</i>	319	220	284

Of course, this is by no means an exhaustive test, nor will it be representative of all payloads. That said, it is reasonably common to find that *DataContractJsonSerializer* involves not only a smaller payload, but it also operates faster than the other techniques, in many cases. The real point is that you can carry out similar tests yourself, with your own payloads and constraints, to determine the optimal solution for your application.

Isolated Storage Helpers

Working with isolated storage is a sufficiently common activity that many developers take the opportunity to develop helper classes to streamline the operations and abstract them to a common library, which can then be used within multiple applications. This becomes particularly useful when the data that the application wants to read and write consists of more complex types rather than simple strings. The application shown in Figure 7-15 is little more than a test harness for such a storage helper utility class (demonstrated in the *TestIsoStorage* application in the sample code).



CONTOSO

storage helper

Name: Andrew

favorite things:

Cereal: Cheerios

Number: 16

Color:

Save

FIGURE 7-15 The *TestIsoStorage* application is a test harness for a storage helper utility.

This application presents three *TextBox* controls and a *ListBox*. The *ListBox* items are data-bound to a list of *Brush* objects, initialized in the *MainPage* to the list of standard Phone accent colors. The idea is that the user can type in values for their favorite cereal and number, and then select their favorite color from a list. You use a *Color* field in this application just to emphasize that serialization works with complex types, not just simple strings.

```
public List<SolidColorBrush> BrushColors = new List<SolidColorBrush>();
public Favorites favs = new Favorites();
private int selectedColorIndex;

public MainPage()
{
    InitializeComponent();
}
```

```

BrushColors.Add(new SolidColorBrush(Color.FromArgb(255, 255, 000, 151)));
BrushColors.Add(new SolidColorBrush(Color.FromArgb(255, 162, 000, 255)));
BrushColors.Add(new SolidColorBrush(Color.FromArgb(255, 000, 171, 169)));
BrushColors.Add(new SolidColorBrush(Color.FromArgb(255, 140, 191, 038)));
BrushColors.Add(new SolidColorBrush(Color.FromArgb(255, 160, 008, 000)));
BrushColors.Add(new SolidColorBrush(Color.FromArgb(255, 230, 113, 184)));
BrushColors.Add(new SolidColorBrush(Color.FromArgb(255, 240, 150, 009)));
BrushColors.Add(new SolidColorBrush(Color.FromArgb(255, 027, 161, 226)));
BrushColors.Add(new SolidColorBrush(Color.FromArgb(255, 229, 200, 000)));
BrushColors.Add(new SolidColorBrush(Color.FromArgb(255, 051, 153, 051)));

FavoritesPanel.DataContext = favs;
ColorList.ItemsSource = BrushColors;
ColorList.SelectedIndex = selectedColorIndex;
}

```

The user's *Name* value is a simple string that is saved to, and subsequently read back from, *Isolated StorageSettings*. The "favorite things" are represented in code by a simple *Favorites* data model class. An instance of this model class serves as the *DataContext* for the view.

```

public class Favorites : INotifyPropertyChanged
{
    public event PropertyChangedEventHandler PropertyChanged;

    private string cereal;
    public string Cereal
    {
        get { return cereal; }
        set
        {
            cereal = value;
            NotifyPropertyChanged("Cereal");
        }
    }

    private int number;
    public int Number
    {
        get { return number; }
        set
        {
            number = value;
            NotifyPropertyChanged("Number");
        }
    }

    private Color color;
    public Color Color
    {
        get { return color; }
        set
        {
            color = value;
            NotifyPropertyChanged("Color");
        }
    }
}

```



```

private void NotifyPropertyChanged(string propertyName)
{
    PropertyChangedEventHandler handler = PropertyChanged;
    if (null != handler)
    {
        handler(this, new PropertyChangedEventArgs(propertyName));
    }
}
}

```

The application uses a helper library for reading and writing isolated storage. There are four methods: two for working with *IsolatedStorageSettings*, and two for working with arbitrary files by using the *IsolatedStorage* APIs. The methods are generic, so they can be used with any data type. They are also fairly simple, and the class could be extended to provide more functionality, such as generating temporary file names, reading and writing directory structures, enumerating storage contents, and so on.

```

public class StorageHelper
{
    public static void SaveToSettings<T>(T setting, string name)
    {
        IsolatedStorageSettings.ApplicationSettings[name] = setting;
        IsolatedStorageSettings.ApplicationSettings.Save();
    }

    public static T ReadFromSettings<T>(string name)
    {
        T setting = default(T);
        IsolatedStorageSettings.ApplicationSettings.TryGetValue<T>(
            name, out setting);
        return setting;
    }

    public static void SaveToStorage<T>(T data, string fileName)
    {
        using (IsolatedStorageFile isoFile =
            IsolatedStorageFile.GetUserStoreForApplication())
        {
            using (IsolatedStorageFileStream isoStream =
                isoFile.OpenFile(fileName, FileMode.Create))
            {
                XmlSerializer xs = new XmlSerializer(typeof(T));
                xs.Serialize(isoStream, data);
            }
        }
    }

    public static T ReadFromStorage<T>(string fileName)
    {
        T data = default(T);
        using (IsolatedStorageFile isoFile =
            IsolatedStorageFile.GetUserStoreForApplication())
        {
            if (isoFile.FileExists(fileName))

```

```

        {
            using (IsolatedStorageFileStream isoStream =
                isoFile.OpenFile(fileName, FileMode.Open))
            {
                XmlSerializer xs = new XmlSerializer(typeof(T));
                data = (T)xs.Deserialize(isoStream);
            }
        }
    }

    return data;
}
}

```

Using the storage helper class, you can trivially save data to either settings or storage.

```

private void saveButton_Click(object sender, RoutedEventArgs e)
{
    StorageHelper.SaveToSettings<string>(userName.Text, "UserName");
    StorageHelper.SaveToStorage<Favorites>(favs, "Favorites");
}

```

Loading data from settings and storage is also very simple with the helper class.

```

string tmp1 = StorageHelper.ReadFromSettings<string>("UserName");
if (tmp1 != null)
{
    this.userName.Text = tmp1;
}

Favorites tmp2 = StorageHelper.ReadFromStorage<Favorites>("Favorites");
if (tmp2 != null)
{
    favs = tmp2;
    for (int i = 0; i < BrushColors.Count; i++)
    {
        Color c = BrushColors[i].Color;
        if (c == favs.Color)
        {
            selectedColorIndex = i;
            break;
        }
    }
}
}

```



Note There is limited support for formal databases in version 7. The Windows Phone 7 OS uses SQL-CE, but this is not exposed to applications. A number of third parties have built database support for Windows Phone 7 on top of isolated storage. However, Windows Phone 7.1 includes full-platform support for local databases, including LINQ-to-SQL. This is examined in detail in Chapter 18.

Summary

In this chapter, you looked in depth at the navigation model, including inter and intra-application page navigation, and page/application state and storage. Although it is possible for you to provide a navigation experience that is different from other applications on the phone, this would be a mistake. Instead, you are strongly encouraged to take a proactive part in the standard navigation model, to respond to the standard navigation events, and to maintain navigation behavior that is consistent with users' expectations. All but the simplest applications will have some state on both a page basis and an application-wide basis that needs to be persisted across navigation. The application platform on the phone provides targeted support for persisting limited volumes of page and application state, and unlimited volumes of arbitrary application data. Although the core behavior in Windows Phone 7 is maintained in Windows Phone 7.1, the later version also introduces some refinements, which you can learn about in Chapter 15.

Diagnostics and Debugging

Debugging is a critical part of development, and the Windows Phone SDK offers a rich set of tools to support this effort, including the Microsoft Visual Studio debugger itself, and the emulator. The application platform also provides some supporting metrics that can be useful in debugging. There are also a few external tools that can be used to complement Visual Studio during development. Run-time diagnostics and trace logging (whether during debugging or after release) present particular problems in phone development. This chapter examines the landscape for debugging tool support and how you can implement re-usable runtime diagnostics capabilities with your phone applications.

Visual Studio Debugging

Visual Studio includes very rich debugging capabilities, most (but not all) of which are available for Windows Phone development. You can debug Windows Phone applications the same way that you debug any other project type in Visual Studio. When you press F5 on your keyboard, Visual Studio starts the application in either the emulator or attached device, simultaneously starting the debugger. You can then perform common tasks, such as setting breakpoints, stepping through code, and examining the various debug windows, including the output, locals, watch, immediate, and call stack windows.

Behind the scenes, considerable custom work is going on to establish the connection between Visual Studio and the device or emulator. Figure 8-1 summarizes the key components in a Visual Studio debugging session for Windows Phone development. Note that the key connectivity component is a set of DLLs that make up the Smart Device Core Connectivity feature, known as “CoreCon.” Some of the functionality of these components is exposed through an API for automation purposes, as you will see later in this chapter.

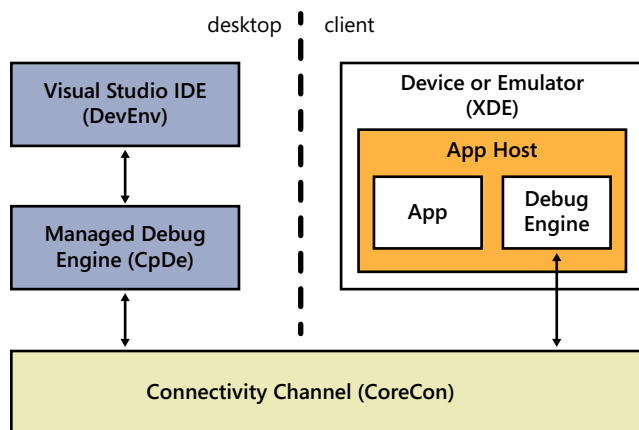


FIGURE 8-1 The CoreCon components connect Visual Studio with the device or emulator.

Simple Diagnostics

For diagnostic output, you can use the standard *System.Diagnostics.Debug.WriteLine* API for sending diagnostics information to the debug output window in Visual Studio. You could also implement some simple “printf-style” diagnostics by outputting to the screen. The classic reason for adopting this approach is to provide diagnostic output in a release build, where the *Debug.WriteLine* output does not apply.

Setting Up a Diagnostics Pop-Up Window

One option that you can use for printf-style diagnostics output is to overlay a transparent or partially transparent pop-up window on your main user interface (UI). Obviously, this would be too intrusive for normal use in the released application, but it is a reasonable and simple approach during development to supplement debug output in Visual Studio. It can also be reasonable to turn on this capability even in a released application, for those scenarios in which you need to walk your user through some operation in real time.

The screenshot in Figure 8-2 shows just such an application (*FloatingDiagnostics* in the sample code). The diagnostics window is composed of a *UserControl*, which contains a *ListBox*, which itself is hosted in a *Popup* window. The diagnostics output strings are pushed to the top of the list to ensure that the most recent output is always immediately visible.

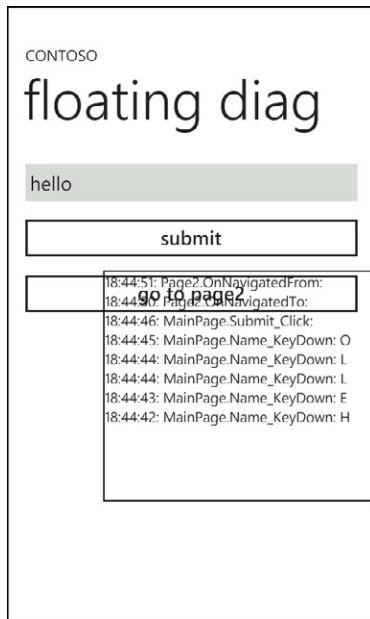


FIGURE 8-2 You can overlay a floating pop-up window for diagnostics reporting.

To increase reusability, the diagnostics control itself is built in to a separate library assembly, which can then be consumed in any phone application. In the *UserControl* XAML, there is a *Border* containing a *ListBox*. The *Border* represents the bounding box for the control. This is positioned at an arbitrary offset. The *ListBox* contains *TextBlock* items, which will be data-bound to a collection of strings that represent the diagnostics output. The *TranslateTransform* on the *Border* element will be used to enable drag operations on the control, triggered by the *ManipulationDelta* events. By doing this, the user can move the window around in case she wants to reveal some part of the page behind the control.

```
<Border x:Name="DiagnosticsBox"
    BorderBrush="{StaticResource PhoneForegroundBrush}"
    BorderThickness="2" Height="300" Width="350" Margin="80,400"
    ManipulationDelta="DiagnosticsBox_ManipulationDelta">

    <Border.RenderTransform>
        <TransformGroup>
            <TranslateTransform x:Name="DragTransform" />
        </TransformGroup>
    </Border.RenderTransform>

    <ListBox x:Name="historyList" >
        <ListBox.ItemTemplate>
```

```

        <DataTemplate>
            <TextBlock
                Text="{Binding}" TextWrapping="Wrap"
                FontSize="{StaticResource PhoneFontSizeSmall}"/>
        </DataTemplate>
    </ListBox.ItemTemplate>
</ListBox>
</Border>

```

In the code-behind, set up an *ObservableCollection* of strings: when you want to write out some diagnostic output, add a string to this collection. You then use data binding by assigning the collection to the *ItemsSource* of the *ListBox*.

```

public partial class DiagnosticsControl : UserControl
{
    private static ObservableCollection<String> debugHistory;

    public DiagnosticsControl()
    {
        InitializeComponent();

        debugHistory = new ObservableCollection<String>();
        historyList.ItemsSource = debugHistory;
    }

    public void WriteLog(object payload)
    {
        StackTrace stackTrace = new StackTrace();
        MethodBase method = stackTrace.GetFrame(1).GetMethod();
        String status = String.Format("{0}: {1}.{2}: {3}",
            DateTime.Now.ToString("H:mm:ss"),
            method.DeclaringType.Name,
            method.Name,
            payload);
        debugHistory.Insert(0, status);
    }

    private void DiagnosticsBox_ManipulationDelta(
        object sender, ManipulationDeltaEventArgs e)
    {
        DragTransform.X += e.DeltaManipulation.Translation.X;
        DragTransform.Y += e.DeltaManipulation.Translation.Y;
    }
}

```

Observe the use of the *StackTrace* and *MethodBase* classes to provide information about the currently executing method as part of the diagnostics output. You need to specify *GetFrame(1)* to skip the current frame (which would be the frame for the *WriteLog* method itself). In the *ManipulationDelta* handler, use the *TranslateTransform* that you declared in XAML to move the X and Y values for the control. This affords the user the ability to drag the *Popup* around the screen.



Note As an alternative, you could use the Blend behaviors feature instead of the *ManipulationDelta* event handling, as discussed in Chapter 5, “Touch UI.” You can see this at work in the *FloatingDiagnostics_Behavior* solution in the sample code. The added wrinkle for this reusable library control is that you would have to add references to System.Windows.Interactivity.dll and Microsoft.Expression.Interactions.dll in both the library project and the consuming application project. Then, in the *DiagnosticsControl.xaml*, you would add XML namespaces for these assemblies and declare the *MouseDragElementBehavior* within the *Border* element, as shown in the following:

```
<interact:Interaction.Behaviors>
    <ilayout:MouseDragElementBehavior />
</interact:Interaction.Behaviors>
```

In the host application, the diagnostics control is instantiated and set as the *Child* of a *Popup* window. In this application, you test the diagnostics feature in a simple way by handling the *KeyDown* and *Click* event handlers to write to the log.

```
public partial class MainPage : PhoneApplicationPage
{
    private DiagnosticsControl diagnostics = new DiagnosticsControl();
    private Popup diagnosticsPopup = new Popup();

    public MainPage()
    {
        InitializeComponent();
        diagnosticsPopup.Child = diagnostics;
        diagnosticsPopup.IsOpen = true;
    }

    private void Name_KeyDown(object sender, KeyEventArgs e)
    {
        diagnostics.WriteLog(e.Key);
    }

    private void Submit_Click(object sender, RoutedEventArgs e)
    {
        diagnostics.WriteLog(null);
    }
}
```

One of the most common reasons why you’ll be glad you implemented logging is for scenarios in which the app throws an unhandled exception. It is simple enough to enhance the control with a handler for the *UnhandledException* event for the application, and you would probably want to hook this up in the control’s constructor. You can see this technique at work in the *SimpleDiagnostics_UEH* solution in the sample code.

```

public DiagnosticsControl()
{
    InitializeComponent();

    debugHistory = new ObservableCollection<String>();
    historyList.ItemsSource = debugHistory;

    Application.Current.UnhandledException +=
        new EventHandler<ApplicationUnhandledExceptionEventArgs>(
            application_UnhandledException);
}

private void application_UnhandledException(
    object sender, ApplicationUnhandledExceptionEventArgs e)
{
    if (debugHistory != null)
    {
        WriteLog(e.ExceptionObject.ToString());
    }
}

```

You might want to make the exception reporting a little more sophisticated; for example, try iterating through any inner exceptions, and reporting specifics for each one.

```

private void application_UnhandledException(
    object sender, ApplicationUnhandledExceptionEventArgs e)
{
    try
    {
        if (debugHistory != null)
        {
            //WriteLog(e.ExceptionObject.ToString());
            WriteLog("Caught Unhanded Exception");
            Exception ex = e.ExceptionObject;
            while (ex != null)
            {
                WriteLog(String.Format("\type      = {0}", ex.GetType()));
                WriteLog(String.Format("\tmessage = {0}", ex.Message));
                WriteLog(String.Format(ex.StackTrace));
                ex = ex.InnerException;
            }
        }
    }
    catch (Exception ex)
    {
        Debug.WriteLine("Diagnostics exception: " + ex.ToString());
    }
}

```

This approach illustrates how you can start to build your own diagnostics feature, but it has some limitations. One obvious problem is that simplistically incorporating the control within a page means that the collection of diagnostic strings is also scoped to just that one page. It is far more likely that you will want to make the diagnostics log accessible to all pages, throughout the application.

To implement this, you can move the control to the *App* class, and expose it as a static property. You can see this variation at work in the *FloatingDiagnostics_AppScope* solution in the sample code.

```
public partial class App : Application
{
    ...
    private static DiagnosticsControl diagnostics;
    public static DiagnosticsControl Diagnostics
    {
        get
        {
            if (diagnostics == null)
            {
                diagnostics = new DiagnosticsControl();
            }
            return diagnostics;
        }
    }
}
```

The additional problem this introduces is that the control should only ever have one parent. If you set it as the child of a *Popup* in the *MainPage*, then you need to detach it before you navigate to another page. And you need to do the same when coming back from that page to the *MainPage*. This implies moving the *Popup* hosting behavior from the page constructor to the *OnNavigatedTo* override and doing the detaching in the *OnNavigatedFrom* override in each page, as shown in the example that follows. A suitable alternative to this approach would be to dedicate a single page to the diagnostics control.

```
protected override void OnNavigatedFrom(NavigationEventArgs e)
{
    diagnosticsPopup.IsOpen = false;
    diagnosticsPopup.Child = null;
}

protected override void OnNavigatedTo(NavigationEventArgs e)
{
    diagnosticsPopup.Child = App.Diagnostics;
    diagnosticsPopup.IsOpen = true;
}
```

Fixed Diagnostics Control

It might not always be feasible to overlay diagnostics output on top of other visual elements in your application. So, an alternative option would be to keep a separate page or other visual element dedicated to your diagnostic output. If your application uses a *Pivot* or *Panorama*, you could dedicate one of the subitems to diagnostics. The next example (the *SimpleDiagnostics* solution in the sample code) illustrates this. As with the floating *Popup* example, there is a custom *UserControl* with a data-bound *ListBox*.

```

<Border
    BorderBrush="{StaticResource PhoneForegroundBrush}"
    BorderThickness="2" Height="800">
    <ListBox x:Name="historyList" >
        <ListBox.ItemTemplate>
            <DataTemplate>
                <TextBlock
                    Text="{Binding}" TextWrapping="Wrap"
                    FontSize="{StaticResource PhoneFontSizeSmall}"/>
            </DataTemplate>
        </ListBox.ItemTemplate>
    </ListBox>
</Border>

```

The code-behind is almost identical to the previous version, with an *ObservableCollection* of strings, an *UnhandledException* handler, and a public *WriteLog* method. The only difference is that you don't need the *ManipulationDelta* handler because this version of the control will be fixed in position. Figure 8-3 presents screenshots of how this looks at runtime, as used in a *Panorama*-based test application. The first *PanoramaItem* has a *TextBox* and a *Button*. Event handlers for these controls call into the diagnostics *UserControl* to log diagnostics information, placed on the second *PanoramaItem*.

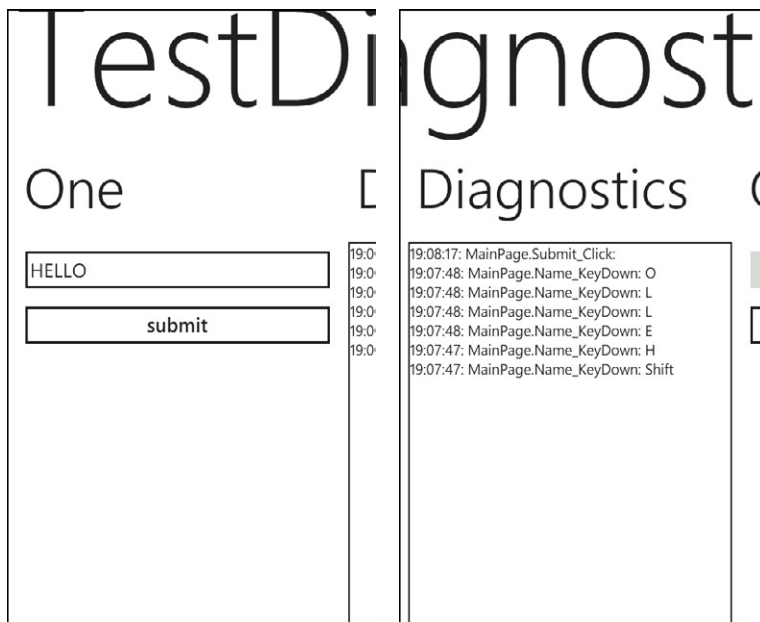


FIGURE 8-3 You can implement diagnostics reporting in a *UserControl*.

Unlike the previous example, this application uses the diagnostics control as a fixed element on the page. The XAML for this test application is shown in the code that follows, with the control on the second *PanoramaItem*. As before, the *KeyDown* event handler on the *TextBox* and the *Click* handler for the *Button* do nothing in this example except write to the diagnostics log.

```

<controls:Panorama Title="TestDiagnostics">
  <controls:PanoramaItem Header="One">
    <StackPanel>
      <TextBox x:Name="Name" KeyDown="Name_KeyDown"/>
      <Button x:Name="Submit" Content="submit" Click="Submit_Click"/>
    </StackPanel>
  </controls:PanoramaItem>

  <controls:PanoramaItem Header="Diagnostics">
    <StackPanel Height="800">
      <utils:DiagnosticsControl x:Name="diagnostics"/>
    </StackPanel>
  </controls:PanoramaItem>
</controls:Panorama>

```

This works in a simple application, but again, there's a scoping problem. As previously done, this can be solved by moving the declaration to the *App* class and exposing it as a property. The page XAML then needs to be updated to remove the declarative instantiation of the *Diagnostics* control, and instead set up a host control to which it can be added programmatically. That is, change this:

```

<controls:PanoramaItem Header="Diagnostics">
  <StackPanel Height="800">
    <utils:DiagnosticsControl x:Name="diagnostics"/>
  </StackPanel>
</controls:PanoramaItem>

```

to this (in all pages where you want to use the control):

```

<controls:PanoramaItem Header="Diagnostics">
  <StackPanel Height="800" x:Name="DiagnosticsHost"/>
</controls:PanoramaItem>

```

The corresponding *OnNavigatedTo* and *OnNavigatedFrom* overrides on each page need to attach and detach the diagnostics control exposed from the *App* class to the hosting control on the page.

```

protected override void OnNavigatedFrom(NavigationEventArgs e)
{
    DiagnosticsHost.Children.Remove(App.Diagnostics);
}

protected override void OnNavigatedTo(NavigationEventArgs e)
{
    DiagnosticsHost.Children.Add(App.Diagnostics);
}

```

Post-Release Diagnostics

Viewing diagnostic information in real time on the device is useful during development. After release, it is more useful to log this information, and then send it to the development team, typically by email. It is reasonable to add this functionality to your diagnostics control. This approach is shown in Figure 8-4 (*Diagnostics_Email* in the sample code). But, you will find that as soon as you start putting input controls on your diagnostics control, you introduce a dilemma: it becomes more challenging

to host your control. Specifically, you generally don't want to have too many (or any) input controls on *Panorama* or *Pivot*-based pages because of the "duelling inputs" problem. That is, the *Panorama* and *Pivot* controls make extensive use of touch gestures to manipulate the UI, and introducing other touch-input controls like *TextBox*, *Button*, *ToggleSwitch*, and so on makes the user touch interface more complex.

So, if your diagnostics control does include input controls, it is generally better to dedicate a separate (non-*Panorama*, non-*Pivot*) page to host it. This has the added benefit of reduced complexity because it eliminates the need to attach and detach the diagnostics control for multiple pages.

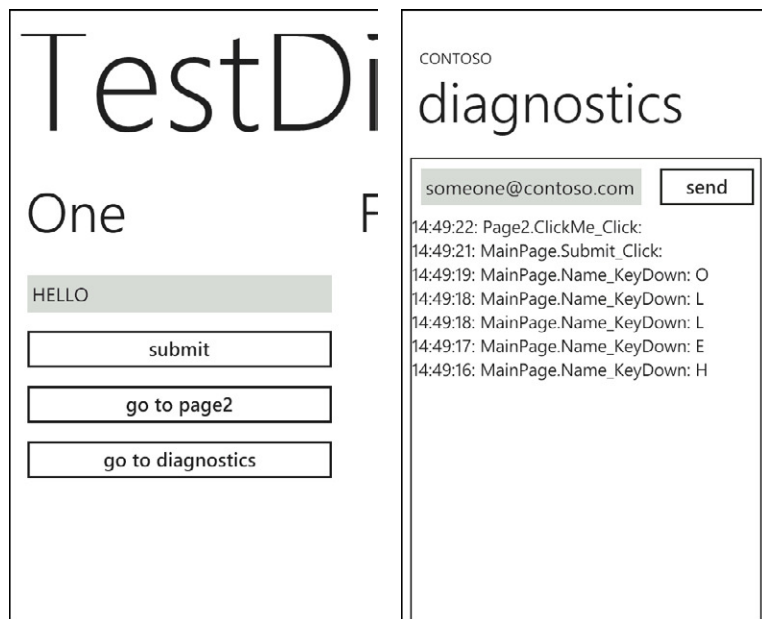


FIGURE 8-4 It is often useful to have a separate page dedicated for diagnostics output.

To add the email feature, you first extend the simple diagnostics control with a *TextBox* and *Button*. This is so that the user can enter an email address (or accept the default address provided in code) and then send the message:

```
<Border BorderBrush="{StaticResource PhoneForegroundBrush}"
    BorderThickness="2" Height="800">
    <StackPanel>
        <StackPanel Orientation="Horizontal">
            <TextBox
                x:Name="emailTextBox" Text="someone@contoso.com" InputScope="EmailUserName"/>
            <Button x:Name="sendEmail" Content="send" Click="sendEmail_Click"/>
        </StackPanel>
        <ListBox x:Name="historyList" >
            <ListBox.ItemTemplate>
                <DataTemplate>
                    <TextBlock Text="{Binding}" TextWrapping="Wrap" />
                </DataTemplate>
            </ListBox.ItemTemplate>
        </ListBox>
    </StackPanel>
</Border>
```

```

        </ListBox.ItemTemplate>
    </ListBox>
</StackPanel>
</Border>

```

The code-behind simply uses the supplied email address and creates a suitable *EmailComposeTask*, filling in the body of the email from the collection of diagnostic strings. Note that you're limited to 64 KB of text in emails, so you should check the length of the log before attempting to send it, and perhaps filter it or send multiple emails if necessary.

```

private void sendEmail_Click(object sender, RoutedEventArgs e)
{
    if (debugHistory.Count > 0)
    {
        EmailComposeTask task = new EmailComposeTask();
        StringBuilder builder = new StringBuilder();
        foreach (String s in debugHistory)
        {
            builder.AppendLine(s);
        }

        task.To = emailTextBox.Text;
        task.Subject = "Diagnostics";
        task.Body = builder.ToString();
        task.Show();
    }
}

```

Persisting Logs

Recall that executing a Launcher such as the *EmailComposeTask* will navigate away from the application, which will therefore be deactivated and might be tombstoned. If you want the diagnostic information to persist in such a scenario, you need to save it to isolated storage and subsequently retrieve it upon initialization. Persisting diagnostic information in this way will also persist it across runs of the application, which might also be useful in some scenarios.

You can see this at work in the *Diagnostics_Persist* solution in the sample code. First, you modify the *WriteLog* method to persist the diagnostic data in addition to adding it to the debug history collection, as shown in the following:

```

public void WriteLog(object payload)
{
    StackTrace stackTrace = new StackTrace();
    MethodBase method = stackTrace.GetFrame(1).GetMethod();
    String status = String.Format("{0}: {1}.{2}: {3}",
        DateTime.Now.ToString("H:mm:ss"),
        method.DeclaringType.Name,
        method.Name,
        payload);
    debugHistory.Insert(0, status);
    SaveLog(status);
}

```

All the real persistence work is done in the *SaveLog* method. This uses the application's isolated storage via the standard *IsolatedStorageFile* and *IsolatedStorageFileStream* classes.

```
private string logFileName = "DiagnosticsLog.txt";

public void SaveLog(String logLine)
{
    using (IsolatedStorageFile isoFile =
        IsolatedStorageFile.GetUserStoreForApplication())
    {
        using (IsolatedStorageFileStream isoStream =
            new IsolatedStorageFileStream(
                logFileName, FileMode.Append, isoFile))
        {
            using (StreamWriter writer = new StreamWriter(isoStream))
            {
                writer.WriteLine(logLine);
            }
        }
    }
}
```

You also need a way to retrieve existing log information from isolated storage. This is simply a matter of reading the *IsolatedStorageFileStream* with a *StreamReader* and then adding the resulting strings to the data-bound collection.

```
public ObservableCollection<String> GetLog()
{
    ObservableCollection<String> data = new ObservableCollection<String>();
    object lockObject = new object();
    lock (lockObject)
    {
        using (IsolatedStorageFile isoFile =
            IsolatedStorageFile.GetUserStoreForApplication())
        {
            if (isoFile.FileExists(logFileName))
            {
                using (IsolatedStorageFileStream isoStream =
                    new IsolatedStorageFileStream(
                        logFileName, FileMode.Open, isoFile))
                {
                    using (StreamReader reader = new StreamReader(isoStream))
                    {
                        String logLine;
                        while ((logLine = reader.ReadLine()) != null)
                        {
                            data.Insert(0, logLine);
                        }
                    }
                }
            }
        }
    }
    return data;
}
```


Call this new *GetLog* method on initialization; specifically, update the *Loaded* event handler to replace this line:

```
debugHistory = new ObservableCollection<String>();
```

with this one:

```
debugHistory = GetLog();
```

If you're persisting all diagnostics to isolated storage, the log will persist across runs of the application, so it's probably useful to offer the option to clear the log. If you're adding a timestamp to each diagnostic entry, this reduces the risk of confusion between multiple entries in the log, but the problem here is to mitigate the chance of an ever-increasing log, which could eventually become unmanageable and even risks running the phone out of disk space.

```
private void clearLog_Click(object sender, RoutedEventArgs e)
{
    debugHistory.Clear();
    using (IsolatedStorageFile isoFile =
        IsolatedStorageFile.GetUserStoreForApplication())
    {
        isoFile.DeleteFile(logFileName);
    }
}
```

Configurable Diagnostics

Abstracting the diagnostics functionality to a *UserControl* makes it more reusable, but the downside is that if you want something to be reusable, it needs to be flexible. The more features you build into the control, the more you'll need to provide ways for the consumer and/or the user to configure them. In the following evolution of our diagnostics control, there are additional options to turn logging on or off, and to turn the diagnostics display on or off, as shown in Figure 8-5 (the *Diagnostics_Settings* solution in the sample code). These are really just representative examples of the kind of configuration you could implement. Other configurations might include log output formatting options, severity levels, event logging triggers, log file size, log record purging behavior, and so on. The application uses *ToggleSwitch* controls, provided in the Silverlight Toolkit.

```
<Border BorderBrush="{StaticResource PhoneForegroundBrush}"
        BorderThickness="2" Height="800">
    <StackPanel Name="diagnosticsPanel" >
        <toolkit:ToggleSwitch
            x:Name="toggleLogging" Content="Logging is OFF" IsChecked="False"
            Checked="toggleLogging_Checked"
            Unchecked="toggleLogging_Unchecked"/>
        <toolkit:ToggleSwitch
            x:Name="toggleDisplay" Content="Display is OFF" IsChecked="False"
            Checked="toggleDisplay_Checked"
            Unchecked="toggleDisplay_Unchecked"/>
    </StackPanel>
</Border>
```

```

<Grid>
  <Grid.ColumnDefinitions>
    <ColumnDefinition/>
    <ColumnDefinition/>
  </Grid.ColumnDefinitions>
  <Button Grid.Column="0" x:Name="clearLog" Content="Clear Log"
    Click="clearLog_Click"/>
  <Button Grid.Column="1" x:Name="emailLog" Content="Email Log"
    Click="emailLog_Click"/>
</Grid>
<StackPanel Orientation="Horizontal">
  <TextBlock Text="Email To: " VerticalAlignment="Center"
    Margin="10,0,0,0"/>
  <TextBox
    x:Name="emailTextBox" LostFocus="emailTextBox_LostFocus"
    InputScope="EmailUserName"/>
</StackPanel>
<ListBox x:Name="historyList" Visibility="Collapsed">
  <ListBox.ItemTemplate>
    <DataTemplate>
      <StackPanel>
        <TextBlock Text="{Binding}" TextWrapping="Wrap" />
      </StackPanel>
    </DataTemplate>
  </ListBox.ItemTemplate>
</ListBox>
</StackPanel>
</Border>

```

When the consuming application declares an instance of the control, the developer can supply initial values for the interesting properties (*IsLogging*, *DeleteLogOnStart*, and so on), as shown in the following:

```

public partial class App : Application
{
  ...
  private static DiagnosticsControl diagnostics;
  public static DiagnosticsControl Diagnostics
  {
    get
    {
      if (diagnostics == null)
      {
        diagnostics = new DiagnosticsControl (
          "TestDiagnostics_Log", true, false, false, "someone@contoso.com");
      }
      return diagnostics;
    }
  }
}

```

This means that you need to supply a constructor in the control class to take in the various parameters and assign them internally to fields and properties.

```

public DiagnosticsControl(
    string logname, bool islogging, bool deletelog, bool isVisible, string emailTo)
{
    LogName = logname;
    IsLogging = islogging;
    DeleteLogOnStart = deletelog;
    IsHistoryVisible = isVisible;
    EmailTo = emailTo;

    InitializeComponent();

    logFileName = LogName + ".txt";
    debugHistory = GetLog();
    historyList.ItemsSource = debugHistory;
    toggleLogging.IsChecked = IsLogging;
    toggleDisplay.IsChecked = IsHistoryVisible;
    emailTextBox.Text = EmailTo;

    if (DeleteLogOnStart)
    {
        DeleteLogFile();
    }

    Application.Current.UnhandledException +=
        new EventHandler<ApplicationUnhandledExceptionEventArgs>(
            application_UnhandledException);
}

```

In this implementation example, if the consumer sets *DeleteLogOnStart=true*, then you'll always launch with a fresh log file. This includes when returning from navigating away, which includes when returning from sending an email of the diagnostics. It's up to the developer to decide the exact behavior that is appropriate in each of these scenarios.

The various handlers for the UI events set the internal properties, as you would expect.

```

private void toggleLogging_Checked(object sender, RoutedEventArgs e)
{
    IsLogging = true;
    this.toggleLogging.Content = loggingIsOn;
}

private void toggleLogging_Unchecked(object sender, RoutedEventArgs e)
{
    IsLogging = false;
    this.toggleLogging.Content = loggingIsOff;
}

private void toggleDisplay_Checked(object sender, RoutedEventArgs e)
{
    historyList.Visibility = Visibility.Visible;
    this.toggleDisplay.Content = displayIsOn;
}

```

```
private void toggleDisplay_Unchecked(object sender, RoutedEventArgs e)
{
    historyList.Visibility = Visibility.Collapsed;
    this.toggleDisplay.Content = displayIsOff;
}
```

Some of the properties govern conditional behavior. For example, you now write only fresh strings to the collection (and to the log) if the *IsLogging* property is set to true.

```
public void WriteLog(object payload)
{
    if (IsLogging)
    {
        StackTrace stackTrace = new StackTrace();
        MethodBase method = stackTrace.GetFrame(1).GetMethod();
        String status = String.Format("{0} - {1}.{2}: {3}",
            DateTime.Now.ToString("H:mm:ss"),
            method.DeclaringType.Name,
            method.Name,
            payload);
        debugHistory.Insert(0, status);
        SaveLog(status);
    }
}
```

As a further enhancement, given is the constraints on real estate, you could provide a *Button* to show/hide the diagnostic control's settings UI. This leaves more space for the diagnostics output itself, as shown in Figure 8-5. You can see this at work in the *Diagnostics_SettingsExpando* solution in the sample code.

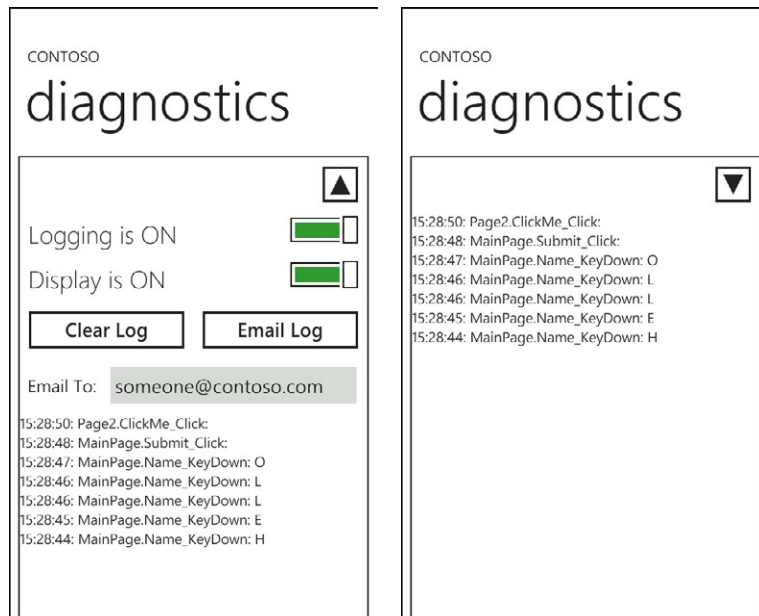


FIGURE 8-5 It might be useful to allow the user to expand or collapse your custom diagnostics chrome.

This is achieved with two *Button* controls, only one of which is visible at a time (they occupy the same space).

```
<Button x:Name="openButton" Width="60" Height="60"
        Padding="0"
        HorizontalAlignment="Right"
        Click="openButton_Click"
        Visibility="Collapsed">
    <Button.Content>
        <Path
            Width="20" Height="20" Stretch="Fill"
            Fill="{StaticResource PhoneForegroundBrush}"
            Data="M0,0 L1,0 0.5,1Z" />
        </Button.Content>
    </Button>
<Button x:Name="closeButton" Width="60" Height="60"
        Padding="0"
        HorizontalAlignment="Right"
        Click="closeButton_Click">
    <Button.Content>
        <Path
            Width="20" Height="20" Stretch="Fill"
            Fill="{StaticResource PhoneForegroundBrush}"
            Data="M0,1 L1,1 0.5,0Z" />
        </Button.Content>
    </Button>
```

The *Click* handlers show or hide the entire settings panel as well as toggling the visibility of the *Button* controls themselves.

```
private void openButton_Click(object sender, RoutedEventArgs e)
{
    settingsPanel.Visibility = Visibility.Visible;
    openButton.Visibility = Visibility.Collapsed;
    closeButton.Visibility = Visibility.Visible;
}

private void closeButton_Click(object sender, RoutedEventArgs e)
{
    settingsPanel.Visibility = Visibility.Collapsed;
    closeButton.Visibility = Visibility.Collapsed;
    openButton.Visibility = Visibility.Visible;
}
```

Screen Capture

Sometimes it's useful to see what the user sees on the screen. Fortunately, it's simple enough to capture the screen (of your own application) and store the image in the phone's media library. The following application provides an App Bar button that when tapped by the user renders the current page into a *WriteableBitmap* and then saves that bitmap to the media library. You'll also display the bitmap in an *Image* control on the screen, set to half the screen size. Clearly, you could take a screenshot of this screen also. Figure 8-6 illustrates such recursive screenshots (using the *ScreenCapture* application in the sample code).

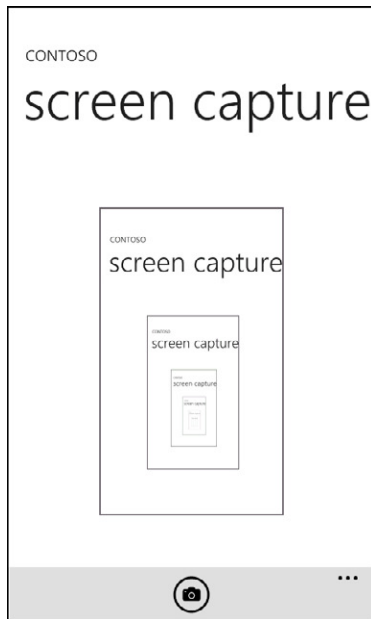


FIGURE 8-6 Capturing your application's screen can be a useful diagnostics support technique.

Here's the XAML for the *Image* control, set to half the width and height of the screen, with a 2-pixel border:

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <Border
        Width="240" Height="400"
        BorderThickness="2" BorderBrush="{StaticResource PhoneBorderBrush}">
        <Image x:Name="capturedImage" Stretch="Uniform" />
    </Border>
</Grid>
```

The example that follows shows the code-behind for the App Bar button. All the hard work is done by the *WriteableBitmap.Render* method, which renders any given *UIElement* to the bitmap object. Notice how you're capturing the entire page but the image doesn't include the *SystemTray* or the App Bar region, and also won't include the contents of *WebBrowser* or *MediaElement* controls. Having captured the screen, you then also set the bitmap as the *Source* for your *Image* control. Finally, save the image as a JPEG file to the phone's media library.

```
private void appBarCapture_Click(object sender, System.EventArgs e)
{
    WriteableBitmap wb = new WriteableBitmap(
        (int)this.ActualWidth, (int)this.ActualHeight);

    wb.Render(this, null);
    wb.Invalidate();
}
```

```

// Display the captured image.
this.capturedImage.Source = wb;

// Save the image to the media library.
MemoryStream stream = new MemoryStream();
wb.SaveJpeg(stream, wb.PixelWidth, wb.PixelHeight, 0, 100);
stream.Seek(0, SeekOrigin.Begin);
MediaLibrary lib = new MediaLibrary();
lib.SavePicture("ScreenCapture", stream);
}

```

Although you can programmatically send emails via the *EmailComposeTask*, there's currently no way to add attachments to one programmatically. Also, although you can programmatically invoke the *PhotoChooser* task, you can't programmatically get to the feature that sends a photo via email. So, if you need your user to send you a screen capture image, you can provide runtime diagnostics features to take the screen captures, but you'll then have to ask him to select the images from the media library and email them manually.

Note also that this feature uses the media APIs, so you can't test it on a physical device while the Zune software is running (Zune locks the local media database). Therefore, to test this, you need to use the Connect tool supplied with the Windows Phone Developer Tools.

Emulator Console Output

Debug.WriteLine of course works only in debug builds. The strategy of using a custom logging mechanism will work for either debug or release builds. Suppose that you don't want to go to the extent of custom logging, but you do want simple diagnostics output in a release build. If you don't mind using regedit, you direct the emulator to show its associated console window. As always, you should ensure that you have a current backup of the registry before you make any changes so that you can restore its previous state, if need be. You should also be aware that editing the registry is normally considered an advanced technique; you should not undertake this unless you fully understand the risks involved.

If you want to go ahead and show the emulator console window, run regedit, and then go to HKLM\Software\Microsoft\XDE. Create a new *DWORD* value named *EnableConsole*, and then set the value to 1, as shown in Figure 8-7. After setting (or changing) the registry key, you must restart the emulator for the change to take effect.

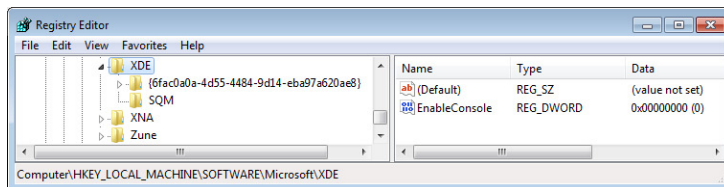


FIGURE 8-7 If you're an advanced user, you can enable the emulator's console output in the registry.

After this, when you run the emulator, it will display its console window. You can take advantage of this by sending console output to this window as part of your release build application. Here's a simple wrapper class to a static method that will provide release diagnostics output:

```
public class ConsoleOutput
{
    public static void WriteLine()
    {
        WriteLine(null);
    }

    public static void WriteLine(object payload)
    {
        StackTrace stackTrace = new StackTrace();
        MethodBase method = stackTrace.GetFrame(2).GetMethod();
        String status = String.Format("{0} - {1}.{2}: {3}",
            DateTime.Now.ToString("H:mm:ss"),
            method.DeclaringType.Name,
            method.Name,
            payload != null ? payload : "(null)");
        Console.WriteLine(status);
    }
}
```

Here's how you might use this in your application code:

```
public MainPage()
{
    ConsoleOutput.WriteLine();
    InitializeComponent();
}

private void gotoPage2_Click(object sender, RoutedEventArgs e)
{
    Uri page2Uri = new Uri("/Page2.xaml", UriKind.Relative);
    ConsoleOutput.WriteLine(page2Uri);
    NavigationService.Navigate(page2Uri);
}
```

Figure 8-8 shows this application using console output. This is the *ConsoleDiagnostics* solution in the sample code.



FIGURE 8-8 Tracking emulator console output can be useful in some scenarios.

Debugging Tombstoning and Lock-Screen

In Chapter 6, “Application Model,” you took an in-depth look at application lifecycle events and the various ways that an application behaves when the user navigates between applications. As part of developing your application, you need to be able to check the behavior when the user navigates away. However, when the user navigates away from your application, it will be terminated. Depending on exactly how the user navigates away, the application might be deactivated, and the current process might be terminated. If the user presses Back to back out of the first page in your application, the application is closed, and any debug session running at the time is also terminated. However, if the user presses Start instead of Back or launches a Launcher or Chooser from your application, the application is deactivated and moved to the backstack. If she later returns back to your application, it will be reactivated. We’ve seen that tombstoning an application actually terminates its process but flags it for potential reactivation at a later point. When deactivated and then tombstoned, your application’s process no longer exists. Given that, how can you possibly test the reactivation code path?

Fortunately, the debugger in Visual Studio has special code to deal specifically with this scenario. For the scenario in which the application is deactivated and placed on the backstack, the debugging session remains alive, even though the target process is no longer running. Obviously, none of the debugging operations are functional at this point; the debug session is in a frozen state, and the only action that is supported is to stop debugging. However, if you leave the debugging session alive, and the user eventually returns back to the application, the debugger will be attached to the new process, and debugging can then continue.

Also, as you saw in Chapter 5, the phone pays attention to its activities and is smart about turning off the screen and other peripherals when the user has been idle for a configurable period of time. During debugging, it is often useful to disable this screen-lock behavior so that it doesn't interfere with debug sessions. The standard Visual Studio project templates all include a statement in the `App.xaml.cs` to turn off idle detection if a debugger is attached.

```
PhoneApplicationService.Current.UserIdleDetectionMode = IdleDetectionMode.Disabled;
```

If you do use this technique, you should remove this statement toward the final stages of development, when you want to test your application under the most realistic conditions. Before releasing the application to marketplace, you will certainly test in release mode, without the debugger attached, in which case this won't be a problem. When the screen-lock is allowed to engage, the phone also deactivates the foreground application at that time. You should test your application's behavior under this condition as well.

Debugging MediaPlayer

When you use Visual Studio to debug an application on a physical device, the device is attached to the PC. When the device is attached, the Zune software is generally running. The problem is that you cannot debug an application that uses the Zune media library, including the photo chooser and camera launcher tasks, because Zune locks the local media database. The fix for this is to use the Windows Phone Connect Tool.

The Connect tool is installed with the SDK, and you can typically find it at `%ProgramFiles%\Microsoft SDKs\Windows Phone\v7.1\Tools\WPConnect\x86\WPConnect.exe`. You can make this slightly more convenient to use if you add it as an external tool in the Visual Studio IDE. To do this, in Visual Studio, click **Tools | External Tools**. Add a new tool and specify the path to `WpConnect.exe`, as shown in Figure 8-9. Note that this only works with the full version of Visual Studio, not with Visual Studio Express. To use the Connect tool, you first connect your phone device and run the Zune software to ensure that the device is recognized. Then, close the Zune software and run `WPConnect`; this is a console application, which should produce a confirmation something like this:

```
Connecting for device 'HTC HD7'.....
Connection established.
```



Note The Connect tool is shipped in both a 32-bit version and a 64-bit version. However, Visual Studio 2010 is a 32-bit process, so even if you're running it on a 64-bit computer, you should not link in the 64-bit version of the Connect tool. In this scenario, you should use the 64-bit Connect from a command prompt, not from Visual Studio.

You need to run this only once, so long as the device remains connected. The connection will persist across runs of Visual Studio.

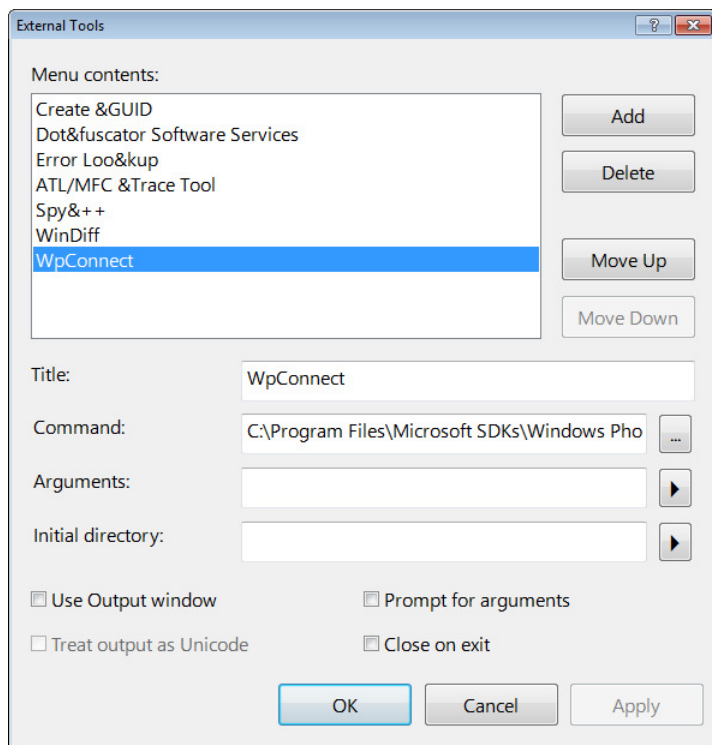


FIGURE 8-9 You can add the WpConnect tool to the Visual Studio Tools Menu.

You can use the same technique for another SDK tool, the Windows Phone Capability Detection Tool. You can run this tool to scan your application, and detect the capabilities that your application actually requires. This scanning operation is equivalent to what happens when you submit your application to the marketplace. You'll find the tool here:

`%ProgramFiles%\Microsoft SDKs\Windows Phone\v7.0\Tools\CapDetect\CapabilityDetection.exe.`

The tool requires two command-line arguments: the path to a rules file that controls the scanning operation, and the path to the output folder for your project. The default rules.xml is provided with the tool; it's a list of types and assemblies that govern which capabilities your app uses, and matches the marketplace rules. To set this up as an external tool in Visual Studio, In the Arguments field of

the External Tools dialog box, select “Rules.xml” and “\$(TargetDir)”, and then set Initial Directory to the directory that contains the rules.xml file. Also select the Use Output Window option, as shown in Figure 8-10.

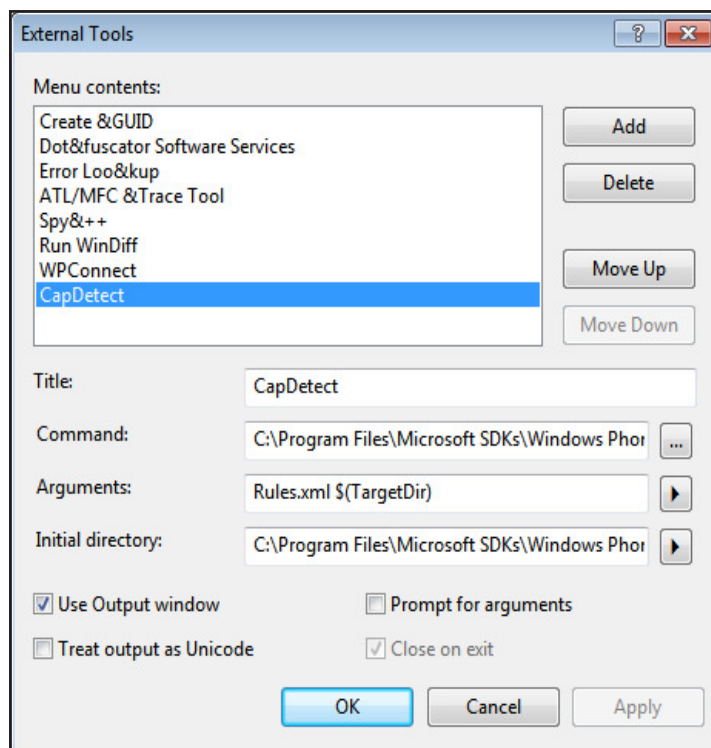


FIGURE 8-10 You can add the Capability Detection tool to Visual Studio.

This will produce a simple list of the capabilities used in your application. If your application doesn’t use any phone capabilities, the list will be empty. Here’s an example output list:

```
ID_CAP_NETWORKING
ID_CAP_PUSH_NOTIFICATION
ID_CAP_SENSORS
```



Note If you don’t use any phone capabilities, and you correctly edit the WMAppmanifest.xml file to remove all the auto-generated ones, the next time you open the solution in Visual Studio you’ll get a spurious complaint about missing capabilities. You can safely ignore this. Note also that this tool is largely superseded in the version 7.1 SDK by the Marketplace Test Kit; however, that tool only works on version 7.1 projects, so you would still need the Capability Detection tool if you’re building version 7 projects.

Device and User Information

Additional information about the application's runtime environment that could be useful in debugging or diagnostics can be garnered from the platform itself. There are various classes in the platform that you can use to obtain information about the current device, user, and status. Specifically, they are the *DeviceExtendedProperties*, *UserExtendedProperties*, *NetworkInterface*, *PhoneApplicationService*, *Microsoft.Devices.Environment*, and *System.Environment* classes. In this example (the *DeviceInfo* solution in the sample code), you gather the information into multiple collections of strings, and then data-bind each one to a *ListBox*, as shown in Figure 8-11. The data is refreshed when the user taps the App Bar button.

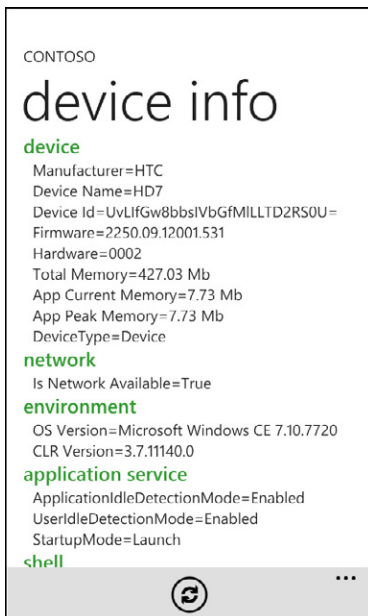


FIGURE 8-11 Device and user information can be helpful in diagnosing application problems.

First, fetching information by using the *DeviceExtendedProperties* API is very straightforward. This will yield the device manufacturer, model name, unique ID, version numbers, and so on. It also provides data on total memory available, and the current application's memory usage. You use *TryGetValue* as a normal due diligence technique, although in this context—assuming we've got the spelling of each property correct—you can be sure that these specific keys will be in the collection. Note that *DeviceUniqueId* is a *byte[20]*, and it is unlikely to be useful to represent this as a character string, although it is included here for completeness. Also note that this API is superseded in version 7.1 by the *DeviceStatus* API (more on this in Chapter 16, "Enhanced Phone Services").

```
object propertyValue;
if (DeviceExtendedProperties.TryGetValue("DeviceManufacturer", out propertyValue))
{
    deviceItems.Add(String.Format("Manufacturer={0}", propertyValue));
}
```

```

if (DeviceExtendedProperties.TryGetValue("DeviceName", out propertyValue))
{
    deviceItems.Add(String.Format("Device Name={0}", propertyValue));
}
if (DeviceExtendedProperties.TryGetValue("DeviceUniqueId", out propertyValue))
{
    byte[] bytes = (byte[])propertyValue;
    String idString = Convert.ToBase64String(bytes);
    deviceItems.Add(String.Format("Device Id={0}", idString));
}
if (DeviceExtendedProperties.TryGetValue("DeviceFirmwareVersion", out propertyValue))
{
    deviceItems.Add(String.Format("Firmware={0}", propertyValue));
}
if (DeviceExtendedProperties.TryGetValue("DeviceHardwareVersion", out propertyValue))
{
    deviceItems.Add(String.Format("Hardware={0}", propertyValue));
}
if (DeviceExtendedProperties.TryGetValue("DeviceTotalMemory", out propertyValue))
{
    deviceItems.Add(String.Format(
        "Total Memory={0:N} Mb", (double)(long)propertyValue / 1024 / 1024));
}
if (DeviceExtendedProperties.TryGetValue(
    "ApplicationCurrentMemoryUsage", out propertyValue))
{
    deviceItems.Add(String.Format(
        "App Current Memory={0:N} Mb", (double)(long)propertyValue / 1024 / 1024));
}
if (DeviceExtendedProperties.TryGetValue(
    "ApplicationPeakMemoryUsage", out propertyValue))
{
    deviceItems.Add(String.Format(
        "App Peak Memory={0:N} Mb", (double)(long)propertyValue / 1024 / 1024));
}

```

You can use the *Microsoft.Devices.Environment* class to determine whether the application is currently running on a device or emulator, as demonstrated in the following:

```

String deviceType = Microsoft.Devices.Environment.DeviceType.ToString();
deviceItems.Add(String.Format("DeviceType={0}", deviceType));

```

You can use the *NetworkInterface* class to get information about the available network, as shown here.

```

bool isNetworkAvailable = NetworkInterface.GetIsNetworkAvailable();
networkItems.Add(String.Format(
    "Is Network Available={0}", isNetworkAvailable));

```

For OS and CLR version, you can use the standard Microsoft .NET *System.Environment* class. Note, however, that not all features of this type are applicable on Windows Phone; specifically, if you attempt to access the *CurrentDirectory* property, this will throw an exception. There is no programmatic access to the filesystem on the phone, except for the very constrained use of the application's isolated storage.

```
environmentItems.Add(String.Format(
    "OS Version={0}", System.Environment.OSVersion));

environmentItems.Add(String.Format(
    "CLR Version={0}", System.Environment.Version));

// Throws a NotSupportedException.
//environmentItems.Add(String.Format(
//    "CurrentDirectory={0}", Environment.CurrentDirectory));
```

The *PhoneApplicationService* class will indicate whether application and/or user idle detection are turned on as well as specify the current startup mode. Applications can be started by the user launching the application explicitly (typically by selecting the application from the *Start* experience) or implicitly (by the user returning to the application after having previously navigated away by using the *Back* button or the task switcher).

```
applicationItems.Add(String.Format(
    "ApplicationIdleDetectionMode={0}",
    PhoneApplicationService.Current.ApplicationIdleDetectionMode));

applicationItems.Add(String.Format(
    "UserIdleDetectionMode={0}",
    PhoneApplicationService.Current.UserIdleDetectionMode));

applicationItems.Add(String.Format(
    "StartupMode={0}",
    PhoneApplicationService.Current.StartupMode));
```

You can get the status of the App Bar and System Tray from the corresponding classes in the platform:

```
shellItems.Add(String.Format(
    "ApplicationBar.IsVisible={0}", ApplicationBar.IsVisible));

shellItems.Add(String.Format(
    "SystemTray.IsVisible={0}", SystemTray.IsVisible));
```

Although there is no support for accessing the filesystem directly, you can retrieve information about isolated storage. In fact, you can get the total amount of free disk space available to all applications. Unlike desktop isolated storage, there is no quota imposed on an application, so it is possible for any application to consume all available storage.

```
using (IsolatedStorageFile storage = IsolatedStorageFile.GetUserStoreForApplication())
{
    storageItems.Add(String.Format(
        "IsoStorage Free Space={0:N} Mb",
        (double)(long)storage.AvailableFreeSpace / 1024 / 1024));
}
```

It might also be useful to extract information from the application's marketplace manifest, that is, `WMAAppManifest.xml`. You can do this fairly easily by loading the manifest into an *XDocument* and parsing it, as shown in the code that follows. The manifest will be at the root of the folder from which the current application is running.

```
manifestItems.Add(String.Format(
    "ProductID={0}", GetApplicationAttribute("App", "ProductID")));

manifestItems.Add(String.Format(
    "AppPlatformVersion={0}", GetApplicationAttribute(
        "Deployment", "AppPlatformVersion")));

private string GetApplicationAttribute(String elementName, String attributeName)
{
    String attributeValue = String.Empty;
    XDocument appManifest = XDocument.Load("WMAAppManifest.xml");
    if (appManifest != null)
    {
        using (XmlReader reader = appManifest.CreateReader(ReaderOptions.None))
        {
            if (reader.ReadToDescendant(elementName))
            {
                attributeValue = reader.GetAttribute(attributeName);
            }
        }
    }
    return attributeValue;
}
```

The final piece of device/user information that you can retrieve is the Anonymous User ID (ANID). For this, you can use the *UserExtendedProperties* class. This class exposes just one piece of information: the ANID. The ANID will be returned in a 44-character string. In fact, the ANID itself is only a 32-character subset of this string; the remainder of the string is used internally by the phone application platform; it is opaque to your application.

```
object propertyValue;
if (UserExtendedProperties.TryGetValue("ANID", out propertyValue))
{
    string anid = String.Empty;
    if (propertyValue != null)
    {
        anid = propertyValue.ToString().Substring(2, 32);
    }
    userItems.Add(String.Format("User Id={0}",
        (String.IsNullOrEmpty(anid) ? "(null)" : anid)));
}
```


Using *DeviceExtendedProperties* will flag your application as using phone identity information, which will require *ID_CAP_IDENTITY_DEVICE*, which in turn will require you to notify the user that your application wants to access identity information. When warned that your application requires access to personal identity information, many users will err on the side of caution and might well change their mind about installing your application. The *DeviceStatus* API introduced in version 7.1 supersedes *DeviceExtendedProperties* and does not require this capability. You are strongly discouraged from using the ANID, and there's every chance that the warning will become more urgent in future releases. Using the *UserExtendedProperties* API also requires the *ID_CAP_IDENTITY_USER* capability.

Windows Phone Performance Counters

The standard applications that the Visual Studio templates generate all include some profiling code in the *App* class constructor, as shown in the following:

```
if (System.Diagnostics.Debugger.IsAttached)
{
    Application.Current.Host.Settings.EnableFrameRateCounter = true;
    Application.Current.Host.Settings.EnableRedrawRegions = true;
    Application.Current.Host.Settings.EnableCacheVisualization = true;
}
```

By default, the generated code applies these only if a debugger is attached, and only actually enables the frame-rate counters. These three performance counters track your application's performance in terms of how many frames are rendered per second (FPS), which areas of the screen are being redrawn, and so on, as summarized in Table 8-1.

TABLE 8-1 Performance Counters

Profiling	Description
Frame-rate counter	Show six counters, including how many frames are being rendered per second, count of surfaces and textures, and screen fill rate.
Redraw regions	Show the areas of the application that are being redrawn in each frame.
Cache visualization	Show the areas of an application that are being passed to the GPU for acceleration (the inverse of the Silverlight browser behavior).

It's generally not useful to have both *EnableRedrawRegions* and *EnableCacheVisualization* turned on at the same time. Also, if you're examining caching, with *EnableCacheVisualization* turned on, the frame rate counter values will be largely meaningless during that time.

The frame-rate counters are described in Table 8-2, with an indication of an optimal range for each one (where that makes sense).

TABLE 8-2 Frame-Rate Counters

Counter	Description	Warning Level	Optimal Range
Render Thread FPS	The number of frames per second that the rendering (composition) thread is using. Values <30 are presented in red.	≤30	45–60
User Interface Thread FPS	The number of FPS that the primary UI thread is using. Presented in red if the count is <15.	≤15	30–60
Texture Memory Usage	The video memory used for storing application textures/surfaces.		
Surface Counter	The number of explicit surfaces that are being passed to the GPU for processing.		
Intermediate Surface Counter	The number of implicit surfaces generated when the compositor thread optimizes/combines cached surfaces.		
Screen Fill Rate	The number of total pixel draw operations as a multiple of the number of pixels on the screen. A value of 1 == 480x800 pixels. Presented in red if the counter is >3.0.	>3.0	1.0–2.0

The render thread FPS counter is shown only if you set the *SystemTray* to invisible (or transparent in version 7.1). The *SystemTray* is an area normally reserved at the top of the screen (in portrait mode) for signal strength and battery life indicators. This would normally obscure the render thread FPS counter. You can turn it off in your page XAML, as shown in the following:

```
<phone:PhoneApplicationPage
...
    shell:SystemTray.IsVisible="False">
```

The frame counters are depicted in Figure 8-12 (full-size on the left; magnified on the right).

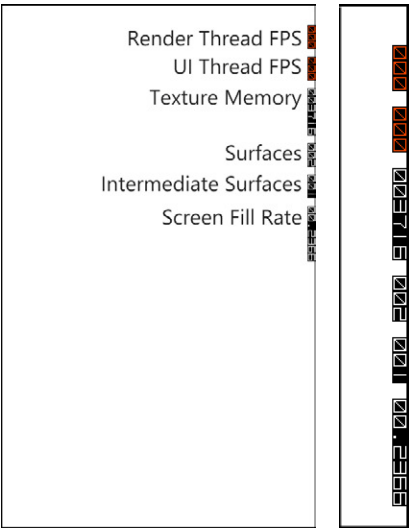


FIGURE 8-12 The standard Windows Phone performance counters track your application’s performance.

The application shown in Figure 8-13 (the *TestPerfCounters* solution in the sample code) illustrates how some of the performance counters are used. The UI offers a *ListBox* and three *Button* controls. When the user taps the Add Item button, you add a new string item to the *ListBox*, when he taps Clear List, you clear the list of strings. The Toggle Redraw *Button* toggles the *EnableRedrawRegions* setting on the *Application.Host* object. This application is designed to work in *Landscape* orientation to facilitate reading the performance counters while working with the *Button* controls and the *ListBox*.

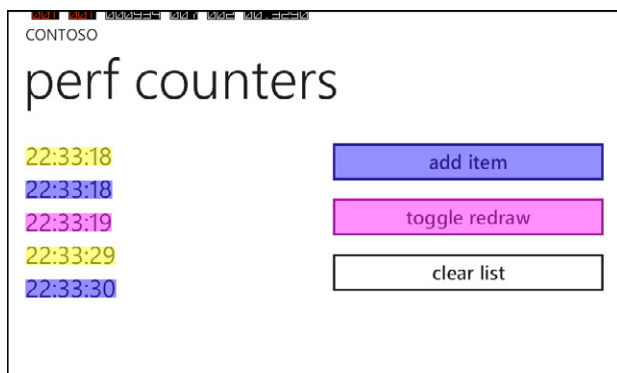


FIGURE 8-13 Visualizing redraw regions is useful for diagnosing and optimizing performance.

If you enable redraw regions and then start adding items, you'll see that each item is drawn when it is added. Each *Button* is redrawn every time it is tapped. Also, the surface counter increments by one with each item you add, as does the texture/surface memory consumption, of course. Observe also that the surface counter continues to increment with each item you add, even though the new items are initially invisible, off the end of the list. The count of surfaces is reset when you clear the list. The fill rate for this application is consistently <1.0 ; this is because you're only drawing relatively small elements, a relatively small number of pixels for each frame.

You should obviously test your application and analyze the performance counters on a real device, which will be significantly more resource-constrained than the emulator. Chapter 14, "Go to Market," explores best practices for Windows Phone development in general, including how to optimize your application's rendering behavior.

Memory Diagnostics

The Windows Phone application platform cooperates with the underlying operating system to manage system resources, including memory and CPU time. This resource management ensures that the phone maintains the optimal balance of resource allocation/deallocation, resolving contention according to established priorities, while also not degrading battery consumption.

The guiding principles are that the phone is a consumer device; the end-user expects all exposed features to work equally well. This does not mean that all applications and system processes are granted equal resources. For instance, phone calls are generally considered more important to the user because they are interruptive and happen in real-time. Most assuredly, the user doesn't want to miss a call and doesn't want to be prevented from making a call at any time. The application platform

enforces behavior that maintains an optimal balance between foreground and background applications and their resource consumption. This makes it easy for developers to build applications that are consistent with these guiding principles. Resources are allocated according to the type of task. For instance, the currently active application will be granted a lot of resources, whereas a dormant application on the backstack will be allocated almost no resources. The operating system, device drivers, the application platform, and the Silverlight and XNA runtimes obviously carve out resources, too.

One of the most constrained resources on any mobile device is memory. Although technically, the minimum requirement is only 256 MB, every one of the first generation of Windows Phone 7 devices has at least 512 MB of system memory. Of this, on Windows Phone 7, 90 MB is carved out for the current foreground application. This is the amount of memory that an application running in the foreground can always assume is available. There is also a certification requirement that applications restrict themselves to running within 90 MB, because you are never guaranteed to have more than that. But, in practice on version 7 devices, you often have more than 90 MB available.

The behavior changes slightly in version 7.1, but for both versions, applications should restrict themselves such that they always run in 90 MB or less. Over time, a wider range of devices will most likely be supported, including devices with significantly less than 512 MB of total memory. Therefore it makes sense to tune your application so that as much as possible it minimizes its use of memory. In many scenarios, you might have a choice of alternative techniques to use. Monitoring memory consumption with each choice will help you to make an informed decision regarding the optimal approach.

The *DeviceExtendedProperties* class in the platform provides three memory-related statistics (all values are reported in bytes):

- **DeviceTotalMemory** This is, for all intents and purposes, the total memory on the device. This is not necessarily the same as the total physical memory, but the difference is irrelevant, because any physical memory above the reported total memory is not accessible to an application anyway. Note that from version 7.1, applications should use the new *ApplicationMemoryUsageLimit* property instead.
- **ApplicationCurrentMemoryUsage** This is the current memory consumption of the current application.
- **ApplicationPeakMemoryUsage** This is the peak memory consumption of the current application, during this session. This peak is not persisted across tombstoning.

The next application (the *TestMemory* application in the sample code), shown in Figure 8-14, uses a *UserControl* named *MemoryDiagnostics* to monitor these three values. The *UserControl* displays memory starts at the upper-right corner of the screen.

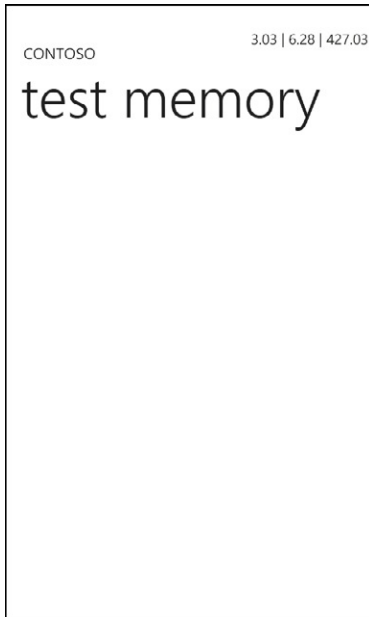


FIGURE 8-14 You can use a custom control to monitor memory consumption in your application.

In the test application, the *App* class declares and initializes an instance of the *MemoryDiagnostics UserControl* as a field. In the *Deactivated* event, you stop the control, although in this case, this is redundant because the application is about to be removed from memory at that point anyway. This is really just to make the point that if you used this *MemoryControl* object with a smaller lifetime scope, you would then want to ensure that the underlying timer is stopped properly when you're finished with it. Note the use of conditional statements to ensure that you only use this code for debug builds.

```
public partial class App : Application
{
    #if DEBUG
        public static MemoryDiagnostics MemoryControl = new MemoryDiagnostics();
    #endif

    private void Application_Deactivated(object sender, DeactivatedEventArgs e)
    {
        #if DEBUG
            MemoryControl.Stop();
        #endif
    }
    ... irrelevant code omitted for brevity
}
```

Then, for any page on which you want to display the memory statistics that this control is monitoring, you create a *Popup* window for it. The example that follows shows the code in the *MainPage* constructor. Another way to achieve the same result is to put all the *MemoryControl* code into a new method and add the *[Conditional("DEBUG")]* attribute to it.

```
public MainPage()
{
    InitializeComponent();

#if DEBUG
    Popup p = new Popup();
    p.Child = App.MemoryControl;
    p.IsOpen = true;
#endif
}
```

So, using the control from an application perspective is trivial. The control itself is also quite simple internally. You set its size and layout so that the text contents is displayed at the upper-right corner of the page, allowing for the 32 vertical pixels that the *SystemTray* occupies. The control contains just one *TextBlock*.

```
<UserControl x:Class="Utilities.MemoryDiagnostics"
...
    Margin="320, 32, 0, 0" Width="160" Height="24" IsHitTestVisible="False">

    <TextBlock
        x:Name="MemoryDisplay" FontSize="{StaticResource PhoneFontSizeSmall}"/>
</UserControl>
```

The code-behind for the control sets up a one-second *DispatcherTimer*, and starts it on construction. For each one-second tick, you fetch the memory values from *DeviceExtendedProperties*, convert them from bytes to megabytes, and then format them into a string for the *TextBlock*.

```
public partial class MemoryDiagnostics : UserControl, IDisposable
{
    private DispatcherTimer timer;

    public MemoryDiagnostics()
    {
        InitializeComponent();

        timer = new DispatcherTimer { Interval = TimeSpan.FromMilliseconds(1000) };
        timer.Tick += (s, e) =>
        {
            long currentMemory =
                (long)DeviceExtendedProperties.GetValue(
                    "ApplicationCurrentMemoryUsage");
            long peakMemory =
                (long)DeviceExtendedProperties.GetValue(
                    "ApplicationPeakMemoryUsage");
```

```

        long totalMemory =
            (long)DeviceExtendedProperties.GetValue(
                "DeviceTotalMemory");
        MemoryDisplay.Text = String.Format(
            "{0:N} | {1:N} | {2:N}",
            (double)currentMemory / 1024 / 1024,
            (double)peakMemory / 1024 / 1024,
            (double)totalMemory / 1024 / 1024);
    };
    timer.Start();

}

public void Stop()
{
    if (timer != null)
    {
        timer.Stop();
        timer = null;
    }
}
}

```

You could take this further and track when your application is approaching the 90 MB limit—or, better yet, when it is approaching some lower threshold that you want to target for your application. For an example, see Peter Torr’s excellent memory helper utility, shown in Figure 8-15 and available at <http://blogs.msdn.com/b/ptorr/archive/2010/10/30/that-memory-thing-i-promised-you.aspx>.

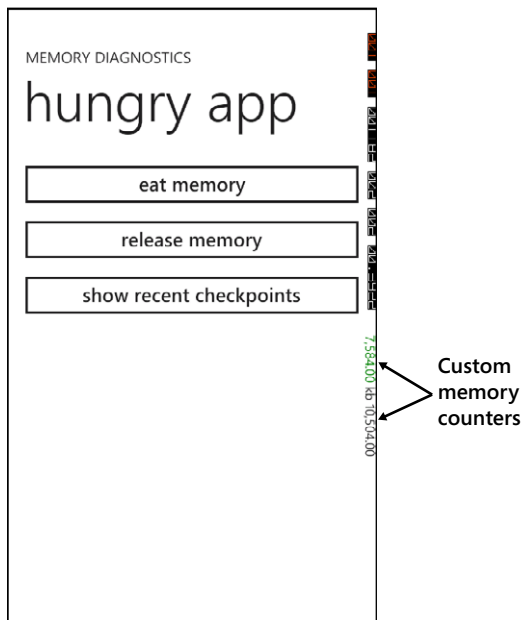


FIGURE 8-15 Custom memory counters extend the standard performance counters.

In essence, this uses a *Popup* window, rotated and scaled, and positioned in line with the standard performance counters. The *Popup* contains *TextBlock* controls that are updated on a *Timer* tick with the results of these calls:

```
DeviceExtendedProperties.GetValue("ApplicationCurrentMemoryUsage");  
DeviceExtendedProperties.GetValue("ApplicationPeakMemoryUsage");
```

The utility provides methods for you to take snapshots of memory usage, as required, and also warns if you exceed the marketplace maximum of 90 MB. It even calculates a “safety band,” within which available memory is at some configured percentage of the maximum. As the application’s memory consumption approaches the limits, it changes the color of the text.

The Device Emulator

The Device Emulator (XDE) is a desktop application that emulates the behavior of a Windows Phone device. The XDE only emulates hardware; to complete the picture, you need to have the XDE load an OS image file. The XDE combined with the OS image together provide a virtual machine that emulates both the phone hardware and the phone OS. On top of that, you need a visual representation of the phone. This is provided by a “skin” image file, which is a simple PNG. These three combined are known as the Windows Phone Emulator. With the emulator, you can run, test, and debug a run-time image without the need for a physical device. Whereas a physical device is still required for final testing and sign-off before publishing the application to the marketplace, it was a design goal of the emulator that it should be an acceptable surrogate for a physical phone for at least 80 percent of the end-user experience of the application.

The emulator also mimics some—but not all—of the phone peripherals. For a list of the differences, see Table 8-3. Apart from peripherals, the critical difference between the emulator and a real device is performance. The emulator uses the CPU and memory resources of the host computer; these will be significantly faster and bigger than on a physical phone.

If you prefer, you can start the emulator from outside Visual Studio by using a command line such as the following:

```
XDE.exe <path to OS image bin file> /skin <path to skin file> /vmid <valid GUID>
```

For example:

```
"%ProgramFiles%\Microsoft XDE\1.0\XDE.exe" "%ProgramFiles%\Microsoft SDKs\Windows Phone\v7.x\  
Emulation\Images\WM70C1.en-US.bin" /skin "%ProgramFiles%\Microsoft XDE\1.0\WM7_Skin_Up.png" /  
vmid {FE530891-F2BC-4D95-A1A9-FA17DD2FA012}
```

Alternatively, you could use the XdeLauncher application, like so:

```
"%ProgramFiles%\Microsoft SDKs\Windows Phone\v7.x\Tools\XDE Launcher\XdeLauncher.exe" "Windows  
Phone 7" "Windows Phone Emulator"
```


Keep in mind that even though you can run the emulator from the command line and configure it in various ways, there's no way to specify a XAP to load, so it's not really useful for Windows Phone development. However, the ability to run from the command line is useful to troubleshoot scenarios in which the emulator fails to run from Visual Studio. If the emulator runs from the command line but not from Visual Studio, the most likely explanation is low memory. If this happens, simply terminate other running processes that you don't need, and then try again. Your desktop computer should have at least 2 GB of memory, and preferably a lot more.

Your development computer should also support the following:

- Hardware-assisted virtualization. Remember that the emulator is a virtual machine, and its performance is significantly improved if it can avail itself of specific virtualization support from the computer hardware itself. If this option is available on your computer, you will see it listed in the BIOS setup screen. If it is available, it should be enabled. If you're unsure whether your computer offers hardware-assisted virtualization, you can download and run the Hardware-Assisted Virtualization Detection Tool, which is available at <http://www.microsoft.com/download/en/details.aspx?id=592>.
- A high performance DirectX 10–capable graphics card with a WDDM 1.1 driver that can take advantage of GPU acceleration on the emulator. You can check your computer's DirectX capabilities with the DxDiag.exe tool. This is typically installed in %SystemRoot%\System32.

Emulator vs. Device

A number of features that are available on the phone itself are either not supported or are significantly restricted on the emulator. These are listed in Table 8-3.

TABLE 8-3 Differences Between Emulator and Device

Feature	Description
Disk storage	Restricted storage. The emulator simulates a flash drive of only 2 GB, whereas most phones have at least 8 or 16 GB. There are two consequences of this: first, an application that uses isolated storage will have a lower maximum storage limit, and second, you cannot install as many applications on the emulator as on a retail physical phone. Note, however, that you can only deploy up to 10 applications on a developer-unlocked physical phone via developer side-loading (that is, not via the marketplace). There is no hard restriction on deploying applications to the emulator, apart from physical storage.
Persistence between runs	Restarting the emulator always restarts the hosted OS image from scratch; nothing is persisted in that image between runs. This means that any applications installed in one run are not persisted to a subsequent run. It also means that any data stored by an application to isolated storage during one run will not persist to any subsequent run.
Multi-touch	Multi-touch capabilities require a host computer that supports multi-touch input: specifically, 4-point multi-touch. Most computers that support multi-touch only support 2 points. Simulation of multi-touch by using the mouse is not supported in the emulator.
Peripherals and sensors	There is no real camera on the emulator; a simple representation of camera behavior is available that always returns to the application a simple image of a rectangle at different positions on the screen. The version 7 release of the emulator does not support the phone accelerometer or GPS capabilities.

Feature	Description
Email	You cannot set up an email account on the emulator; therefore the <i>EmailComposeTask</i> has restricted behavior.
Windows Live ID	The emulator does not include support for Windows Live ID.
MediaPlayer	An application can start the media player and the user can play music, but video is not rendered.
Phone calls and SMS	The emulator uses a non-functional GSM capability and a false SIM card. Placed phone calls always appear to connect, but there is no real cellular connectivity. Phone calls cannot be received. SMS messages always appear to be sent, but cannot be received.

XDE Automation

Recall that the key connectivity component that enables the Visual Studio debugger to connect to an application running on the emulator or device is CoreCon, the Smart Device Connectivity component. Visual Studio ships an API for Smart Device Connectivity. You'll find the managed assemblies that wrap the native COM components at %ProgramFiles%\Common Files\Microsoft Shared\PhoneTools\CoreCon\10.0\Bin\. You can use these to automate the Windows Phone emulator and the physical device. The basic steps are as follows:

1. Add a reference to %ProgramFiles%\Common Files\Microsoft Shared\CoreCon\1.0\Bin\Microsoft.Smartdevice.Connectivity.dll.
2. Instantiate a *DatastoreManager* and use it to enumerate a collection of *Platform* objects. Each *Platform* object represents a platform such as Windows Phone 7 that is installed in your local PC's *Datastore*. A platform is really little more than a registered capability, or category of devices.
3. Select a *Platform* and use it to enumerate its collection of *Device* objects. Each *Device* object represents one of the devices or emulators, such as Windows Phone 7 Emulator, that belongs to the platform. This does not imply that any of the devices are actually present on the system, only that they are registered in the platform's capabilities.
4. Select a *Device* object to access information about the device, provision it, and connect to it. At this point, you can discover whether the device actually exists on the system and is available for connection.
5. After connecting to the device, you can use the *RemoteApplication* type to install/uninstall and launch/terminate an application on that device.

Here's a simple example console application (the *AutomateEmulator* solution in the sample code) that finds either a physical phone or the emulator, installs an application, and launches it.

You're ultimately calling *Device.InstallApplication*, which requires you to pass in the application's GUID (the *ProductID* in the WMAAppManifest.xml), application icon path, and application XAP path. In this example, you provide these values on the command line (where "AutomateEmulator" is the name of this console application):

```
AutomateEmulator true "{70d7ea03-f1a3-4c85-8900-779ac50d4a80}" "C:\Temp\SampleXap\
ApplicationIcon.png" "C:\Temp\SampleXap\BouncingBall.xap"
```

The first step is to access the *DataStoreManager*: this component manages the registry-based data store that contains information on the computer about platforms and devices registered for smart device automation. From the *DataStoreManager*, you can retrieve the “Windows Phone 7” platform object.

```
public static void Main(string[] args)
{
    bool useEmulator = bool.Parse(args[0]);
    String appGuid = args[1];
    String appIconPath = args[2];
    String appXapPath = args[3];

    DatastoreManager manager =
        new DatastoreManager(CultureInfo.CurrentCulture.LCID);
    Collection<Platform> platforms = manager.GetPlatforms();
    Platform platform = platforms.Single(p => p.Name == "Windows Phone 7");
```

Then, you make a decision as to whether you want to connect to the emulator or physical device so that you can query the platform for the device in the collection that matches this name. The string identifiers for the emulator and device changed between Windows Phone 7 and 7.1, but you can deal with that via conditional directives.

```
#if WINDOWS_PHONE_71
    String deviceName = useEmulator ?
        "Windows Phone Emulator" : "Windows Phone Device";
#else
    String deviceName = useEmulator ?
        "Windows Phone 7 Emulator" : "Windows Phone 7 Device";
#endif
```

After you’ve decided whether to connect to the emulator or device, you can use the corresponding string to search in the collection of “devices” known to the platform to get a corresponding *Device* object to which you can attempt to connect.

```
Collection<Device> devices = platform.GetDevices();
Device device = devices.Single(d => d.Name == deviceName);
device.Connect();
```

Once you’ve established a connection to the device (emulator or physical device), terminate any running instance of the application, uninstall it, install a fresh version, and then finally, launch it.

```
RemoteApplication app;
Guid appID = new Guid(appGuid);
if (device.IsApplicationInstalled(appID))
{
    app = device.GetApplication(appID);
    app.TerminateRunningInstances();
    app.Uninstall();
}

app = device.InstallApplication(
    appID, appID, "NormalApp", appIconPath, appXapPath);
app.Launch();
}
```

This technique can be useful in setting up automated build/test scripts in a fairly simplistic manner. A more sophisticated use of this would include functionality to unzip a target application XAP file so that it could extract the *WMAAppManifest.xml*. From that manifest, the solution could further extract the application icon name and the *ProductID*. Be aware that several of the methods and properties on the *RemoteApplication* type are not implemented, including *GetInstalledFileInfo*, *GetIsolatedStore*, *IsRunning*, *Genre*, and *Title*.

If you want to retrieve baseline metrics for a device—perhaps for benchmarking your application's behavior across multiple devices—you can also use the CoreCon APIs to return system information about any device (or the emulator). Specifically, you can call the *GetSystemInfo* method on the *Device* object, as shown in the following:

```
SystemInfo si = device.GetSystemInfo();
Console.WriteLine("OSMajor = {0}", si.OSMajor);
Console.WriteLine("OSMinor = {0}", si.OSMinor);
Console.WriteLine("OSBuildNo = {0}", si.OSBuildNo);
Console.WriteLine("ProcessorArchitecture = {0}", si.ProcessorArchitecture);
Console.WriteLine("InstructionSet = {0}", si.InstructionSet);
Console.WriteLine("NumberOfProcessors = {0}", si.NumberOfProcessors);
Console.WriteLine("TotalPhys = {0:N} Mb", si.TotalPhys / 1024 / 1024);
Console.WriteLine("AvailPhys = {0:N} Mb", si.AvailPhys / 1024 / 1024);
Console.WriteLine("TotalPageFile = {0}", si.TotalPageFile);
Console.WriteLine("TotalVirtual = {0:N} Mb", si.TotalVirtual / 1024 / 1024);
Console.WriteLine("AvailVirtual = {0:N} Mb", si.AvailVirtual / 1024 / 1024);
Console.WriteLine("AvailPageFile = {0}", si.AvailPageFile);
Console.WriteLine("PageSize = {0}", si.PageSize);
Console.WriteLine("SystemDefaultLocaleId = {0}", si.SystemDefaultLocaleId);
Console.WriteLine("CurrentTime = {0}", si.CurrentTime);
```

When run on an HTC HD7 phone, *GetSystemInfo* provided the following data:

```
OSMajor = 7
OSMinor = 0
OSBuildNo = 7390
ProcessorArchitecture = Arm
InstructionSet = Armv4ifp
NumberOfProcessors = 1
TotalPhys = 474.00 Mb
AvailPhys = 346.00 Mb
TotalPageFile = 0
TotalVirtual = 1,024.00 Mb
AvailVirtual = 1,020.00 Mb
AvailPageFile = 0
PageSize = 4096
SystemDefaultLocaleId = 1033
CurrentTime = 9/23/2011 5:14:40 PM
```

You can get information about (and update, install, uninstall, launch, and terminate) installed developer applications (that is, unsigned XAPs deployed to a developer-unlocked device or to the emulator). These are represented by the *RemoteApplication* class. However, you cannot obtain information about installed marketplace applications on the device or emulator.

```
Collection<RemoteApplication> apps = device.GetInstalledApplications();
foreach (RemoteApplication app in apps)
{
    Console.WriteLine("{0}", app.ProductID);
}
```

Using the Microsoft Network Monitor

Microsoft Network Monitor (NetMon) is a protocol analyzer. With it, you can capture network traffic, view it, and analyze it. You can download it from <http://www.microsoft.com/en-us/download/details.aspx?id=4865>. There's also a command-line version called NMCap, which can be used in a less resource-intensive manner.

When you start a capture session, NetMon stores frames in a sequence of capture files in the \Temp folder. By default, each capture file is 20 MB, and NetMon will continue to capture files until your disk space drops to less than 2 percent. This is configurable.

In addition to capturing data, NetMon also assigns properties to frames, and then uses the properties to group the frames into conversations, which are displayed in a tree view in the NetMon UI. Keep in mind that this conversations feature significantly increases both CPU utilization and memory use and can cause the computer to become unresponsive.

To use NetMon in Windows Phone development, start your application in the emulator (or the device), run NetMon, and then create a new capture. When you're ready to start exercising network operations, in the NetMon menu, click Start. The emulator will be listed in the conversations tree (XDE.EXE). If you're using NetMon with a physical device, you need to look at the WMZuneComm.EXE entry, instead. The NetMon UI provides a rich set of features for viewing and analyzing network frames, as shown in Figure 8-16.

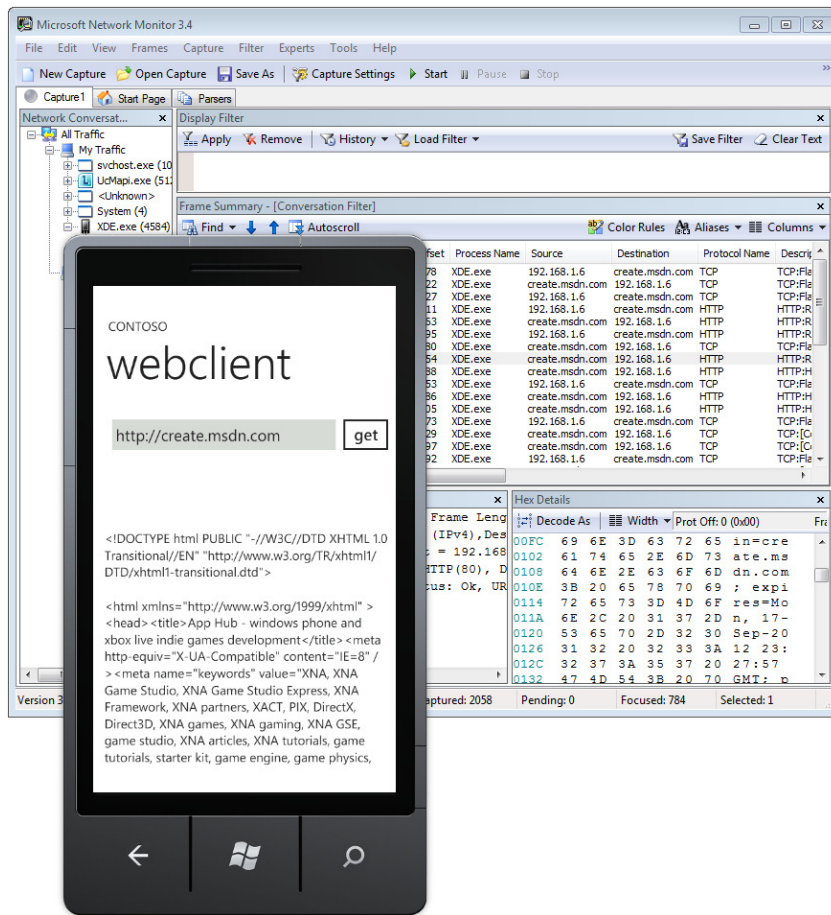


FIGURE 8-16 NetMon is a useful tool for analyzing network traffic.

NetMon supports plug-ins, called “experts.” One such expert is the TCP Analyzer provided by Microsoft Research, which you can download from <http://research.microsoft.com/en-us/downloads/05136260-202d-4a01-bb29-33454d0c30c2/>. This tool analyzes NetMon trace captures, providing a range of performance statistics and visualizations for the captured TCP connection, including the time-sequence graph, round-trip time measurements, and the like, as shown in Figure 8-17. The tool also includes an analysis engine that attempts to explain what the limiting performance factor of a particular connection was, such as limited physical bandwidth, network congestion, or a receiver or sender window size that is too small. This can help you to understand what a connection is doing, and why it might be slow.

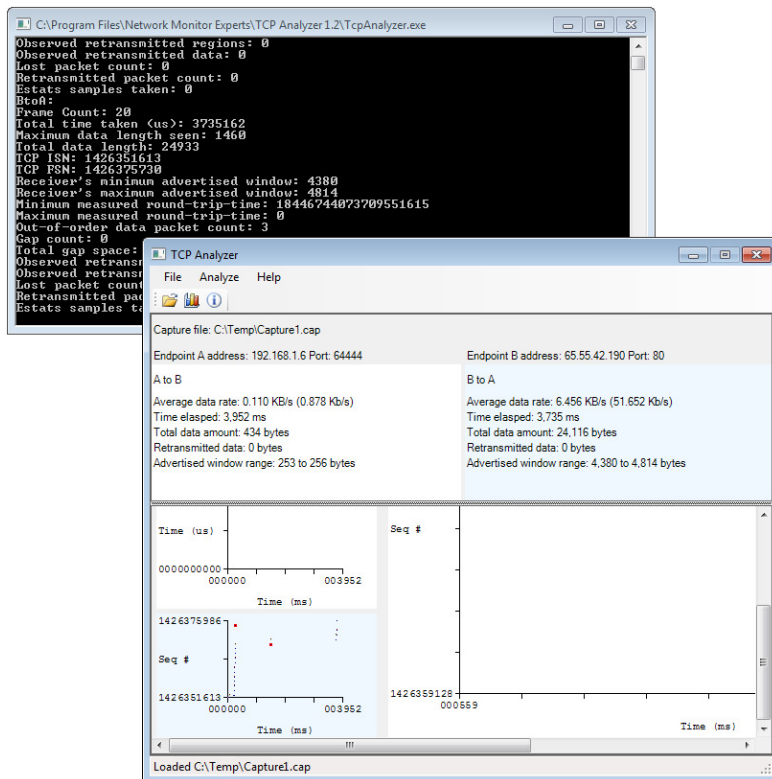


FIGURE 8-17 NetMon supports “experts” (plug-ins) such as the TCP Analyzer.

Fiddler

Fiddler is an internet debugging proxy that logs all HTTP(S) traffic between your computer and the Internet. You can download it from <http://www.fiddler2.com/fiddler2/>. Using Fiddler, you can also inspect the traffic, set breakpoints, and “fiddle” with incoming or outgoing data. To configure it for use with the emulator, you need to take these steps:

1. In Fiddler, click Tools | Fiddler Options | Connections, and then select the Allow Remote Computers To Connect option.
2. In the tiny QuickExec window at the bottom of the session list, type the following:
prefs set fiddler.network.proxy.registrationhostname *YourComputerName*
 (where *YourComputerName* is the name of your computer).
3. Close and restart Fiddler.
4. Start (or restart) the emulator.

With these steps completed, you can test any application that uses HTTP(S) and see traffic logging in Fiddler, as shown in Figure 8-18.

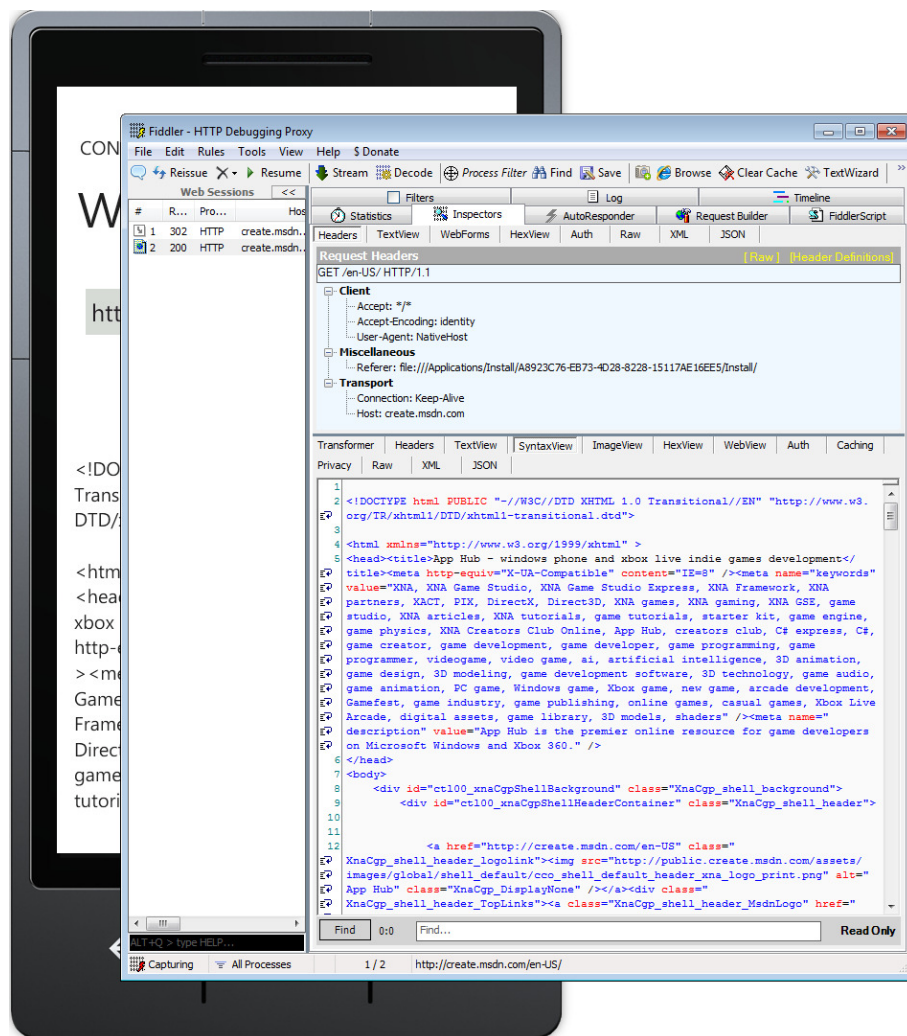


FIGURE 8-18 The Fiddler tool is very useful for debugging Windows Phone applications.



Note There are additional issues if you want to use Fiddler with a physical device. For example, if the computer on which you're running Fiddler is on a corporate network with IPsec enabled, you won't be able to exchange traffic between your phone and your computer. Also, for HTTPS traffic, Fiddler resigns the traffic with its own certificate, which would typically not be installed on your phone.

Silverlight Spy

Silverlight Spy is a third-party runtime inspector tool for examining Silverlight XAPs, including in-browser, out-of-browser, and Windows Phone 7 applications. Using this tool, you can explore the UI element tree, examine element properties, monitor events, extract XAML, interactively execute DLR code, and view runtime statistics. The tool hooks into the Windows Phone 7 emulator to monitor events and performance, as shown in Figure 8-19.

You can also point it to Reflector, and it will integrate static code disassembly into its UI. This is a commercial tool; you can obtain a downloadable trial at <http://firstfloorsoftware.com/silverlightspy>.

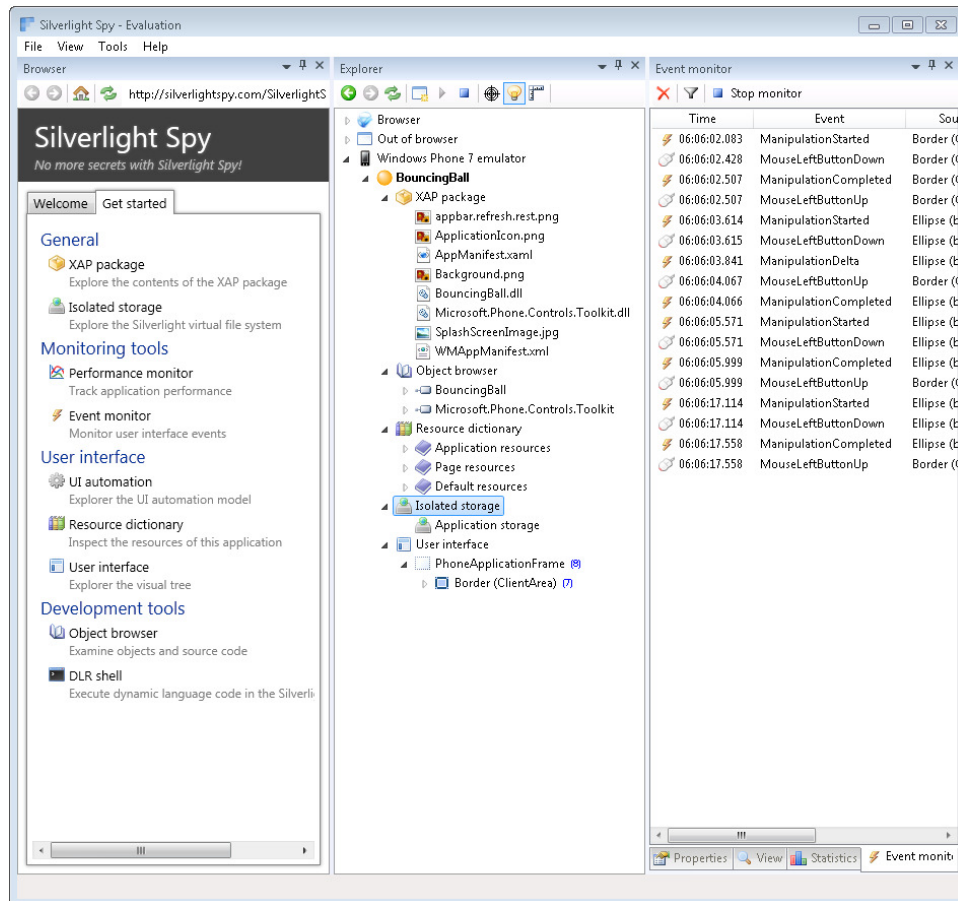


FIGURE 8-19 Silverlight Spy is a runtime inspector tool for working with Silverlight XAPs.

Summary

In this chapter, you examined the range of tools—both within the Windows Phone SDK, and external tools—that support debugging for phone development. You also looked at the supporting metrics that the application platform exposes and the specific issues of debugging tombstoning and media player scenarios. Finally, you explored the various ways that you can implement reusable runtime diagnostics capabilities with your phone applications.

The Windows Phone 7.1 SDK includes a very rich profiler, which brings an additional dimension to development diagnostics and debugging. This will be examined in detail in Chapter 20, “Tooling Enhancements.”

PART III

Extended Services

CHAPTER 9	Phone Services	291
CHAPTER 10	Media Services	319
CHAPTER 11	Web and Cloud.	349
CHAPTER 12	Push Notifications	409
CHAPTER 13	Security	445
CHAPTER 14	Go to Market.	499

A key aspect of the Windows Phone vision is that applications take part in an integrated ecosystem. This part explores how your application can take advantage of functionality provided by built-in features of the phone, including photos, email, audio and video media, the accelerometer sensor, and web connectivity. It also includes crucial chapters on security and what you must do in order to optimize performance, and publish your application to the marketplace.

Phone Services

The most important feature of Windows Phone 7 devices is the cellular phone functionality. Beyond that, the phone offers a range of additional hardware features, from the high-resolution, true multi-touch screen to the accelerometer. Unlike historical versions of Windows Mobile (the precursor to Windows Phone), there is a very constrained, finite set of possible variations on the standard hardware specification. This ensures an extremely high degree of consistency across different phone models and individual devices. On top of that, the application platform encapsulates and exposes a set of features with a number of sensor-related classes. And on top of that, there is a set of launchers and choosers, which is the way that the application platform repurposes built-in applications, such as the camera, photo library, contacts, and so on. You can take advantage of these features within your own applications. In this chapter, you'll explore the various levels of application support and your choices for using sensor streams and integration with standard system features.

Phone Hardware

All Windows Phone devices must conform to a set of minimum standards specified by Microsoft. These hardware requirements were determined as a result of extensive market research, usability studies, and evaluation with hardware suppliers. All Windows Phone devices include a large WVGA format display, a true multi-touch screen, three sets of sensors, and Assisted-GPS. The large screen is capable of rendering most web content in full-page width and allows users to view movies in an aspect ratio of 15:9, which is close to the high-definition cinematic ratio of 16:9. The multi-touch support provides an intuitive and compelling user experience (UX). The accelerometer provides additional interface controls for games and other applications. The light sensor improves power consumption by adjusting screen brightness according to environmental conditions, and the proximity sensors turn off the touch screen when the device is held close to the head during phone calls or when it is in a pocket or handbag. Table 9-1 summarizes the minimum hardware requirements. All devices support these features, and most devices support additional, optional features.

TABLE 9-1 Minimum Hardware Requirements for Windows Phone

Component	Required Feature	Description
Applications processor	Type	Qualcomm QSD8x50.
Graphics processor	Direct3D	Direct3D 10 Level 9, hardware acceleration, driver-level support for GDI and DirectDraw (although GDI is not exposed to applications).
Memory	RAM	256 MB LPDDR1 (low-power double data-rate). In practice, all Windows Phone 7 devices ship with at least 512 MB.
	Flash system partition	Raw NAND.
	Flash user partition	e-MMC or microSD or Raw NAND.
Power	Usability	Monitor/gauge.
Screen	Type	LCD or OLED.
	Screen size	3.5" to 4.4" WVGA (800x480 pixels = 15:9 aspect ratio).
	Bit depth	At least 16 bits of color per pixel (5 red, 6 green, 5 blue).
	Screen surface	Glass or polyethylene terphthalate (PET).
	Touch support	At least 4-point true multi-touch.
Digital camera	Still capture	At least 5 megapixel.
	Video capture	At least VGA.
	Viewfinder	VGA.
	Automatic control	Auto-exposure, auto white balance, auto-focus.
	Electronic flash	Xenon or LED.
	Zoom	Either optical or digital or both. Digital zoom is limited to 4x maximum to reduce image quality degradation.
	Aspect ratio	4:3.
	Photo imaging	JPEG encoding.
Wireless	Cellular radio	UMTS/GSM/GPRS/EDGE, HSPA/HSPA+ and/or CDMA2000 3xEVDO, Rev B.
	Bluetooth	2.1.
	Wi-Fi	802.11b/g/n.
	FM radio	Worldwide (76 MHz to 108 MHz) band support.
Sensors	A-GPS	Assisted GPS (helps to obtain a faster time to fix location, especially when GPS signals are weak or not available).
	Accelerometer	Three-axis, with hardware sampling rate up to 100 Hz.
	Magnetometer (Compass)	Three-axis, with sampling rate of 60 Hz. Note that Windows Phone SDK 7.0 did not provide API support for the Compass. This was introduced in Windows Phone SDK 7.1 (see Chapter 16, "Enhanced Phone Services," for details).
	Ambient light sensor	Dynamic range 0 lux to $\geq 4,000$ lux.
	Proximity sensor	Detects the presence of nearby objects without physical contact. This is used mainly to conserve battery power by turning off the screen when the phone senses that the user is on a call (phone is close to the ear), or that it is in a pocket/handbag.
	Vibration motor	Used to vibrate the phone as a quieter alternative to the phone's ringer.

Launchers and Choosers

A Windows Phone “task” is a generic concept that provides for consistent behavior in a set of standard features. Tasks break down into Launchers (which launch a feature, and don’t return a value) and Choosers (which launch a feature, and do return a value). Table 9-2 summarizes these tasks. Behind the scenes, system services and built-in applications like the phone dialer or the web browser expose a range of application programming interfaces (APIs) through Component Object Model (COM), Remote Procedure Call (RPC) or simple dynamic-link library (DLL) exports. However, none of these protocols is available to marketplace application developers. Instead, the application platform on the phone provides suitable wrappers to managed applications that expose a set of these features in a consistent, rapid application development (RAD)–friendly manner.

Invoking a Launcher or Chooser causes the invoking application to be sent to the backstack. It can additionally cause the application to be tombstoned; this depends on which Launcher/Chooser is used and the version of the phone OS. Either way, as you would expect, the inter-application navigation behavior is consistent with all other inter-application navigations. Users can return back to the original application upon completing the task inside the launcher/chooser, or they can use the Back key. If the launcher/chooser has multiple pages, then the Back key will navigate the user through the previous pages and, finally, back to the calling application. In the same manner, if the user navigates forward through multiple applications, this can result in the original calling application falling off the backstack, as normal. Also, the chooser is auto-dismissed if the user forward navigates away from it.

TABLE 9-2 Launchers and Choosers

Type	Task	Description	Defers Tombstoning in Version 7.0
Launchers	<i>EmailComposeTask</i>	Composes a new email.	No
	<i>PhoneCallTask</i>	Initiates a phone call to a specified number.	No
	<i>SmsComposeTask</i>	Composes a new text message.	No
	<i>SearchTask</i>	Launches Microsoft Bing Search with a specified search term.	No
	<i>WebBrowserTask</i>	Launches Microsoft Internet Explorer and browses to a specific URL.	No
	<i>MarketplaceXXXTask</i>	Launches Marketplace.	No
	<i>MediaPlayerLauncher</i>	Launches Media Player.	Yes
Choosers	<i>CameraCaptureTask</i>	Opens the camera application to take a photo.	Yes
	<i>PhotoChooserTask</i>	The user can select an image from his Picture Gallery or take a photo.	Yes
	<i>EmailAddressChooserTask</i>	The user can select an email address from his Contacts List.	Yes
	<i>PhoneNumberChooserTask</i>	The user can select a phone number from his Contacts List.	Yes
	<i>SaveEmailAddressTask</i>	Saves an email address to an existing or new contact.	No
	<i>SavePhoneNumberTask</i>	Saves a phone number to an existing or new contact.	No

In the following example, buttons are available to invoke a *CameraCaptureTask*, *PhoneCallTask*, *WebBrowserTask*, and *SearchTask*, as shown in Figure 9-1. This is the *SimpleTasks* application in the sample code.



FIGURE 9-1 Many standard tasks are exposed programmatically as Launchers and Choosers.

In general, the application code for invoking Launchers and Choosers is very simple. The application platform provides easy-to-use wrappers for all the sensors and application-accessible system tasks on the device. The basic steps for using a Launcher are as follows:

1. Create an instance of the type that represents the specific Launcher feature that you want to use.
2. Set properties on the object, as appropriate.
3. Invoke the *Show* method.

In this example, when the user clicks the phone button, you instantiate a *PhoneCallTask* object and call *Show*, which prompts the user to confirm the outgoing call.

```
PhoneCallTask phone = new PhoneCallTask { PhoneNumber = phoneNumber.Text };  
phone.Show();
```

The search button invokes *SearchTask.Show*; it is equally simple to use.

```
SearchTask search = new SearchTask { SearchQuery = searchText.Text };  
search.Show();
```


The first time you do this, the system displays a prompt, indicating that the Search feature can make use of your location, as shown in Figure 9-2.

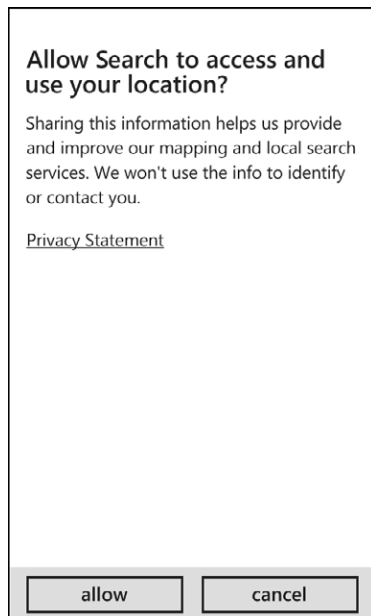


FIGURE 9-2 Invoking a search task triggers a permission request to the user.

When you proceed with the search, the task displays a scrolling list with the Bing search results for the specified term, as illustrated in Figure 9-3.



FIGURE 9-3 When you invoke a search task, the Bing search results page appears.

Using the *WebBrowserTask* follows the same pattern.

```
WebBrowserTask browser = new WebBrowserTask { URL = browseText.Text };  
browser.Show();
```

Using the *CameraCaptureTask* is only marginally more involved. So far, you've seen how to use Launchers, but the *CameraCaptureTask* is a Chooser, which means it returns a value to the calling application. The basic steps for using a Chooser are as follows:

1. Create an instance of the type that represents the specific Chooser feature that you want to use. You would typically do this in your page constructor.
2. Hook up the *Completed* event on the object, which will call back on your event handler when the Chooser task completes. You should do this very early, typically in your page constructor.
3. Set properties on the object as appropriate.
4. Invoke the *Show* method.
5. In your *Completed* event handler, process the return value from the Chooser.

So, to use this, you need to hook up the *Completed* event, and implement your event handler to retrieve the return value.

```
CameraCaptureTask camera = new CameraCaptureTask();  
  
public MainPage()  
{  
    InitializeComponent();  
    camera.Completed += new EventHandler<PhotoResult>(camera_Completed);  
}  
  
private void cameraButton_Click(object sender, RoutedEventArgs e)  
{  
    camera.Show();  
}  
  
private void camera_Completed(object sender, PhotoResult e)  
{  
    if (e.TaskResult == TaskResult.OK && e.ChosenPhoto != null)  
    {  
        BitmapImage bitmap = new BitmapImage();  
        bitmap.SetSource(e.ChosenPhoto);  
        cameraImage.Source = bitmap;  
    }  
}
```

Under the covers, many of the exposed UI features on the phone are implemented internally as applications—this includes the features behind the Launchers and Choosers. So, for example, if you open Reflector (see the Note that follows) and examine the *Show* method on the *PhoneNumber ChooserTask*, you'll see code similar to the following listing:

```
public override void Show()
{
    if (ChooserHelper.NavigationInProgressGuard(() => this.Show()))
    {
        ParameterPropertyBag ppb = new ParameterPropertyBag();
        byte[] buffer = ChooserHelper.Serialize(ppb);
        Uri appUri = new Uri(
            "app://5B04B775-356B-4AA0-AAF8-6491FFE5615/ChoosePhonePropertyOfExistingPerson",
            UriKind.Absolute);
        base.Show();
        ChooserHelper.Invoke(appUri, buffer, base._genericChooser);
    }
}
```



More Info Reflector is a tool for browsing, analyzing, decompiling, and debugging Microsoft .NET assemblies. It is available at <http://www.reflector.net/>.

The relevant code to call out here is that internally this is invoking the *ChoosePhonePropertyOfExistingPerson* task within the contacts application, specified by its URI, which always includes the *ProductID* of the application. The same pattern is followed by almost all the launchers and choosers.

Photo Extras



Note The following information applies to Windows Phone SDK 7. The behavior changed somewhat in the version 7.1 release (see Chapter 16 for details).

The standard photo/picture library in version 7 includes a menu, and that menu offers an “extras” item, as shown in Figure 9-4. This is a phone extensibility point. You can write an application that will be listed in the Extras menu. The idea is that the user selects a picture from the gallery, and then selects your application to perform some operation on that picture.

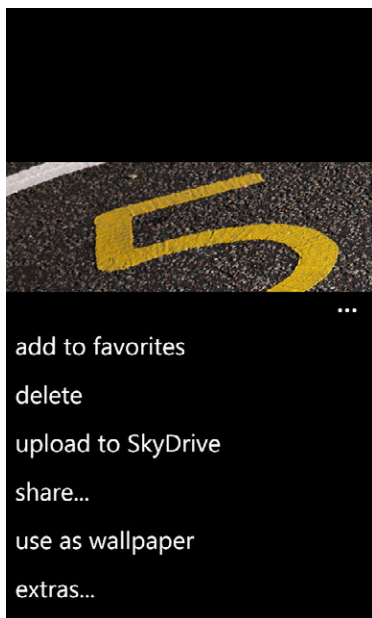


FIGURE 9-4 You can add your application to the photo library Extras menu.

To create a photo extras application with Windows Phone SDK 7, you need to add an `extras.xml` file to your project and implement `OnNavigatedTo` for your main page to perform some operation on the specified picture. The `extras.xml` is always the same (see the listing that follows), and its build properties must be set to *Content*. You can see this at work in the *MyPhotoExtra* solution in the sample code.

```
<Extras>
  <PhotosExtrasApplication>
    <Enabled>true</Enabled>
  </PhotosExtrasApplication>
</Extras>
```

In this example, you'll take the selected picture and place it into an *Image* control on your main page.

```
<StackPanel x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
  <Image Height="350" HorizontalAlignment="Left" Name="selectedPicture"
    VerticalAlignment="Top" Width="450" />
</StackPanel>
```

You need to override `OnNavigatedTo`, because if the user has navigated to this application from the Pictures application, you'll be passed the key named "token" in the *QueryString*. You need to extract the value that corresponds to this key from the navigation context; this will be the picture that the user selected. Having retrieved this value from the media library, you would then go ahead and do some work with it. In this example, you take the picture and place it into the empty *Image* control on the page. Keep in mind that if you use the *MediaLibrary* type, you must add a reference to *Microsoft.Xna.Framework.dll*.

```
protected override void OnNavigatedTo(NavigationEventArgs e)
{
    if (this.NavigationContext.QueryString.ContainsKey("token"))
    {
        MediaLibrary library = new MediaLibrary();
        Picture picture = library.GetPictureFromToken(
            this.NavigationContext.QueryString["token"]);

        DoSomethingWithSelectedImage(picture.GetImage());
    }
}

private void DoSomethingWithSelectedImage(Stream imageStream)
{
    BitmapImage bitmap = new BitmapImage();
    bitmap.SetSource(imageStream);
    selectedPicture.Source = bitmap;
}
```

The emulator doesn't have a picture library, so you need to test this on the device directly. If your application uses the *MediaLibrary*, you can't test while the Zune software is running. For this reason, Microsoft provides the WPConnect tool as part of the Windows Phone SDK, installed to %Program Files%\Microsoft SDKs\Windows Phone\7.x\Tools\WPConnect.

You should also allow for the possibility that the user launches your application not from the photo gallery, but from the Start menu. This is a marketplace certification requirement. One way to handle this is to provide a closely mirrored UX and launch the *PhotoChooser* explicitly, upon startup. The obvious place to do this is in your override of *OnNavigatedTo*. If you take this approach, you could then handle the *Completed* event and duplicate the desired operation there.

The following code illustrates this approach. In it, a flag is set in the main page constructor, and then in the *OnNavigatedTo* override, you check to see if you are being invoked as a Photos extra, as before. On the other hand, if there's no token in the *NavigationContext*, then the user must have launched the application from the Start menu, in which case, you launch the *PhotoChooser*. As you've seen from an earlier example, you need to hook up the *Completed* event on the *PhotoChooserTask* and implement it to mirror the behavior that the user sees when she goes through the Photo extras path.

```
public partial class MainPage : PhoneApplicationPage
{
    private bool launchedFromStart;
    private PhotoChooserTask chooser;

    public MainPage()
    {
        launchedFromStart = true;
        InitializeComponent();
        chooser = new PhotoChooserTask();
        chooser.Completed +=
            new System.EventHandler<PhotoResult>(chooser_Completed);
    }

    void chooser_Completed(object sender, PhotoResult e)
```

```

{
    DoSomethingWithSelectedImage(e.ChosenPhoto);
}

protected override void OnNavigatedTo(NavigationEventArgs e)
{
    if (NavigationContext.QueryString.ContainsKey("token"))
    {
        launchedFromStart = false;

        MediaLibrary library = new MediaLibrary();
        Picture picture = library.GetPictureFromToken(
            NavigationContext.QueryString["token"]);

        DoSomethingWithSelectedImage(picture.GetImage());
    }
    else
    {
        if (launchedFromStart)
        {
            launchedFromStart = false;
            chooser.Show();
        }
    }
}
}
}

```

For this to work in the face of tombstoning, you would want to persist the value of the *launched FromStart* flag in page state. You cannot normally step through a photo extras application in the debugger, because it must be invoked from the photo gallery application. However, there is a way to simulate this code path. What you need to do is to have your application launched with the token for a picture as part of the launch parameters. To do this, change the *DefaultTask* entry in your *WMAppManifest.xml*, to provide the *"token="* parameter. The value must be a valid token GUID for a photo on the phone.

```

<!--<DefaultTask Name="_default" NavigationPage="MainPage.xaml"/>-->
<DefaultTask Name="_default" NavigationPage="MainPage.xaml?token={3074581d-5422-b770-9570-92a2e7958035}"/>

```

If you do use this approach during development and testing, you must remember to remove the additional parameters prior to submitting your application to the marketplace, or it will fail certification.

Accelerometer

The accelerometer is a sensor on the phone that detects changes in X,Y,Z orientation. When it detects a change, it raises an event. There are two basic ways with which the accelerometer can be used in your application:

- To determine the phone's orientation at any point in time.
- To detect movement of the phone relative to an initial point.

Note that the accuracy of the accelerometer at any point depends on the Earth's gravity being a constant everywhere on the planet—which it isn't—and on the user's ability to keep the phone completely steady, which is very difficult, even when laid flat on a table. The accelerometer assumes a baseline gravity of 1.0, when in fact the Earth's gravity varies from 9.78 m/s^2 to 9.83 m/s^2 . The effect of centrifugal force due to the Earth's rotation means that gravity increases with latitude (it's higher at the poles than at the equator). Gravity also decreases with altitude, and the density and composition of the local rocks can also affect it.

You can use the *Accelerometer* class in the *Microsoft.Devices.Sensors* namespace to start and stop the accelerometer and to respond to the events. The *AccelerometerReading* event arguments include the values of the force applied to the phone in the X,Y,Z planes, in the range -1.0 to $+1.0$, as illustrated in Figure 9-5.

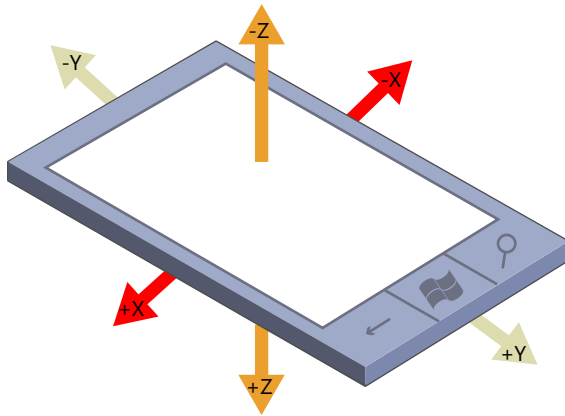

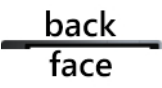


FIGURE 9-5 Accelerometer events specify the force applied in the X,Y,Z planes.

Table 9-3 lists the minimum and maximum values for each plane.

TABLE 9-3 Minima and Maxima for X,Y,Z Planes

Orientation	X	Y	Z
	0	-1	0
	+1	0	0
	0	+1	0
	-1	0	0
	0	0	-1
	0	0	+1

The next example, which is illustrated in Figure 9-6, shows the basics of how to program the accelerometer. This is the *TestAccelerometer* application in the sample code. The UI offers a *TextBlock* and an App Bar button. The button acts as a toggle with which the user can start or stop the accelerometer. It is important to give the user a way to control the accelerometer, because running the accelerometer constantly will eventually drain the battery. When the user toggles the accelerometer on, you instantiate a new *Accelerometer* object and hook up the *ReadingChanged* event. When the user toggles the accelerometer off, you unhook the event handler, stop the accelerometer, and set the object reference to null so that it becomes available for garbage collection.


```

private Accelerometer accelerometer;

private void appBarStartStop_Click(object sender, EventArgs e)
{
    if (accelerometer == null)
    {
        accelerometer = new Accelerometer();
        accelerometer.ReadingChanged +=
            new EventHandler<AccelerometerReadingEventArgs>(
                accelerometer_ReadingChanged);
        try
        {
            accelerometer.Start();
        }
        catch (AccelerometerFailedException ex)
        {
            statusText.Text = ex.ToString();
        }
    }
    else
    {
        accelerometer.ReadingChanged -=
            new EventHandler<AccelerometerReadingEventArgs>(
                accelerometer_ReadingChanged);
        try
        {
            accelerometer.Stop();
            accelerometer = null;
            statusText.Text = "Accelerometer stopped";
        }
        catch (AccelerometerFailedException ex)
        {
            statusText.Text = ex.ToString();
        }
    }
}

```

When you receive accelerometer *ReadingChanged* events, you extract the X,Y,Z axis values to compose a string for the *TextBlock*. Note that these events come in on a different thread from the UI thread, so you need to dispatch the action to the UI thread via the *Dispatcher* property on that page.

```

private void accelerometer_ReadingChanged(object sender, AccelerometerReadingEventArgs e)
{
    this.Dispatcher.BeginInvoke(() =>
    {
        statusText.Text = String.Format("X={0}, Y={1}, Z={2}",
            e.X.ToString("0.00"), e.Y.ToString("0.00"), e.Z.ToString("0.00"));
    });
}

```

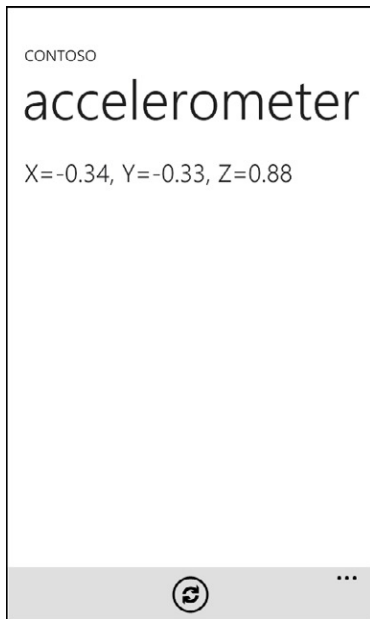


FIGURE 9-6 You can extract the X,Y,Z axis values from accelerometer events.

This illustrates the basic code requirements for working with the accelerometer, although a realistic application would do something more interesting with the X,Y,Z data. Note that you cannot test the accelerometer on the emulator in Windows Phone SDK 7; however, the version 7.1 release does include a sophisticated accelerometer emulation tool, as described in Chapter 20, “Tooling Enhancements.”

Reactive Extensions for .NET

With the Reactive Extensions for .NET (Rx .NET), implemented in System.Observable.dll and Microsoft.Phone.Reactive.dll, you can treat sequences of events as collections instead of as independent events. Thus, you can perform such operations as filtering and aggregating. You can perform Language-Integrated Query (LINQ) operations on the sequence, or compose multiple events into a higher-level virtual event. This capability is particularly useful in high-volume scenarios, such as sensor events, geo-location events, or low-level touch manipulation events.



More Info Rx .NET ships as part of .NET Framework 4.0 and also with the Windows Phone SDK. You can read more about it at <http://msdn.microsoft.com/en-us/data/gg577609>.

In this enhanced version of the accelerometer test application (the *FilteredAccelerometer* application in the sample code), you add a flag field to determine whether to apply filtering, and an additional App Bar button with which the user can toggle the flag.

```
private bool isFiltered = false;

private void appBarFilter_Click(object sender, EventArgs e)
{
    isFiltered = !isFiltered;
}
```

Instead of simply hooking the *ReadingChanged* events with a traditional *EventHandler<>*, you want to maintain a collection of events on which you can filter. Specifically, you'll get an *IObservable<IEvent<>>*. You then subscribe to it. This means that you hook up your event handler so that it's invoked when the observable collection changes—in other words, whenever a *ReadingChanged* event is added to the sequence. The *Subscribe* method of the *IObservable<IEvent<>>* type returns an *IDisposable*. This is so that you can unhook the event handler and clean up when you need to by disposing of the object. Therefore, you need to add a couple of *IDisposable* fields to your page class.

```
private IDisposable eventsSubscription;
private IDisposable filteredEventsSubscription;
```

Observe how the *ReadingChanged* events are hooked (and unhooked) to the collection.

```
var rawEvents =
    Observable.FromEvent<AccelerometerReadingEventArgs>
        (ev => accelerometer.ReadingChanged += ev,
         ev => accelerometer.ReadingChanged -= ev);
```

This seems a little cumbersome. If you look at the class reference documentation, you'll see that the definition of *FromEvent* is even more cumbersome.

```
public static IObservable<IEvent<TEventArgs>> FromEvent<TEventArgs>(
    Action<EventHandler<TEventArgs>> addHandler,
    Action<EventHandler<TEventArgs>> removeHandler
)
where TEventArgs : EventArgs
```

This all arises unfortunately because of certain constraints on events in the .NET framework that restrict the extent to which they can be used in type inference with generics. This slight developer awkwardness aside, this is actually a very efficient mechanism for feeding events into the collection.

The only other change in our application is in the Start/Stop button *Click* handler. Now, when you start the accelerometer, you also create an *IObservable<IEvent<>>* collection based on the *ReadingChanged* events. If the filter flag is off, you simply subscribe to the full collection of events. On the other hand, if the filter flag is on, you first sample the sequence before you subscribe to it. In this example, you're sampling every second. If the user is asking to stop the accelerometer, you first dispose of the subscription objects (which will unhook the events), before cleaning up the *Accelerometer* object, as before.

```
private void appBarStartStop_Click(object sender, EventArgs e)
{
    if (accelerometer == null)
    {
        accelerometer = new Accelerometer();
```

```

var rawEvents =
    Observable.FromEvent<AccelerometerReadingEventArgs>
        (ev => accelerometer.ReadingChanged += ev,
         ev => accelerometer.ReadingChanged -= ev);

if (!isFiltered)
{
    eventsSubscription = rawEvents.Subscribe(
        args => accelerometer_ReadingChanged(
            args.Sender, args.EventArgs));
}
else
{
    var sampledEvents =
        rawEvents.Sample<
            IEvent<AccelerometerReadingEventArgs>>(
                TimeSpan.FromMilliseconds(1000));
    filteredEventsSubscription = sampledEvents.Subscribe(
        args => accelerometer_ReadingChanged(
            args.Sender, args.EventArgs));
}

try
{
    accelerometer.Start();
}
catch (AccelerometerFailedException ex)
{
    statusText.Text = ex.ToString();
}
}
else
{
    if (eventsSubscription != null)
        eventsSubscription.Dispose();
    if (filteredEventsSubscription != null)
        filteredEventsSubscription.Dispose();
    try
    {
        accelerometer.Stop();
        accelerometer = null;
        statusText.Text = "Accelerometer stopped";
    }
    catch (AccelerometerFailedException ex)
    {
        statusText.Text = ex.ToString();
    }
}
}
}

```

The net result of this from the user's perspective is that the accelerometer text value will only change every second, regardless of how fast she moves the device.

Level Starter Kit

Microsoft provides additional code to support the use of the accelerometer in your application. One in particular, the Level Starter Kit for Windows Phone, you can obtain at http://create.msdn.com/en-US/education/catalog/sample/level_starter_kit.

This kit provides a complete level application that you can adapt for your own purposes. Within this, the kit includes a couple of very useful wrapper classes: the *AccelerationHelper* and the *OrientationHelper*. The *AccelerationHelper* provides methods to calibrate the accelerometer as well as to smooth out the raw accelerometer data stream. Strictly speaking, the code does not calibrate the accelerometer. Of course, this is because your application cannot write to the system's accelerometer driver. Rather, the code calibrates itself; it computes the practical values for the X and Y axes when the phone is at rest, face up on a horizontal surface. All other things being equal, this should provide X,Y values of 0,0; the calibration allows for variations in local gravity, device shake, and so on.

Smoothing out the raw data is useful because the raw data stream comes in at 50 Hz (50 data points per second). In the earlier example, you smoothed this out by sampling the data only every second. Sampling the data is often useful, but is a fairly crude approach and risks losing data that might otherwise be of interest.

Exactly how you use the accelerometer depends on the specifics of the application you're building. It might be that you need to see every data point in the incoming stream. It might be that an *n*-second sample is appropriate. In other scenarios, you might want to smooth the data by ignoring changes that fall below some threshold. This approach reveals a "bigger picture" trend in the data, rather than focusing on the individual data points.

The *AccelerometerHelper* class includes some signal processing functionality to apply smoothing to the data stream in a number of different ways:

- **Averaging** This averages the data over time, using an arithmetic mean of the last 25 samples (the last 500 ms of data). This provides a very stable reading, but there is an obvious delay introduced by waiting for the 25 samples before computing the reading. Obviously, you can adjust the number of samples, increasing the value to improve the average at the cost of increased latency, or reducing to make the averaging less smooth but faster.
- **Low-Pass Filtering** This smooths the data stream to eliminate the main sensor noise. Essentially, the current value is adjusted to make it closer to the previous one.
- **Optimal Filtering** This combines the low-pass filtering with a threshold-based high-pass filter, to eliminate most of the low amplitude noise, while trending very quickly to large offsets, and with very low latency.

These are the basic steps for reusing the *AccelerometerHelper* from the Level Starter Kit:

1. Create a Phone Application project named, for example, TestAccelerometerHelper.
2. Add a Class Library project to this solution. Name it *AccelerometerHelper*.

3. Add existing items to the class library: the *AccelerometerHelper.cs*, *ApplicationSettingHelper.cs*, and *Simple3DVector.cs*.
4. Add a reference to *Microsoft.Devices.Sensors.dll*.
5. Build the *AccelerometerHelper* class library.
6. In the *TestAccelerometerHelper* project, add a reference to the *AccelerometerHelper* class library.
7. In the application, hook up the *ReadingChanged* event on the singleton *AccelerometerHelper* instance, and then start the accelerometer by setting the *IsActive* property to *true*.
8. In your *ReadingChanged* event handler, retrieve the *AccelerometerHelperReadingEventArgs* values, and then do something interesting with them.

Figure 9-7 shows an example that presents the solution as it appears in Solution Explorer on the left, and the runtime behavior on the right. In this application, you fetch the X,Y,Z values for all four of the data sets—the raw data, plus the three smoothed streams. This is the *TestAccelerometerHelper* solution in the sample code.

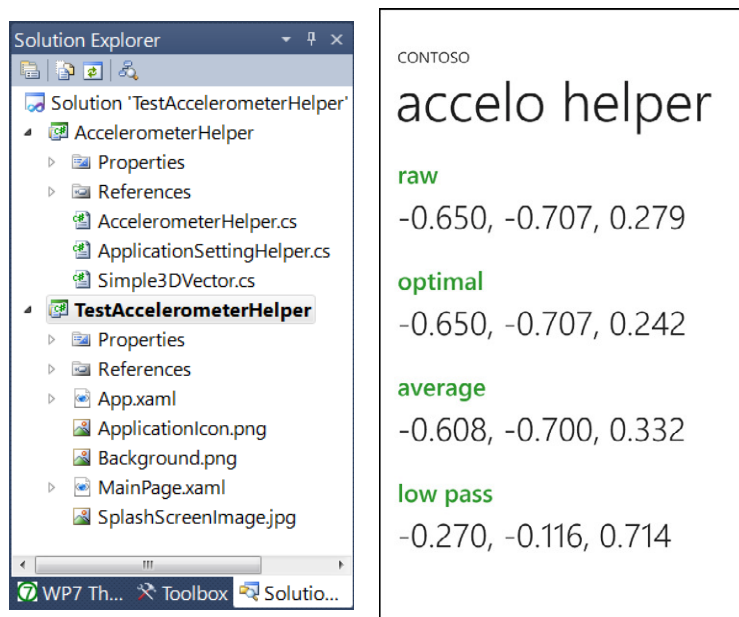


FIGURE 9-7 You can use the *AccelerometerHelper* class to apply smoothing to the raw readings.

The XAML in this application defines a *Grid* comprising four rows of label *TextBlock* controls (“raw,” “optimal,” “average,” and “low pass”) and four rows of data *TextBlock* controls. The only custom code in the application is in the *MainPage*, in which you hook up and handle the *ReadingChanged* event.

```

public MainPage()
{
    InitializeComponent();
    AccelerometerHelper.Instance.ReadingChanged +=
        new EventHandler<AccelerometerHelperReadingEventArgs>
            (OnAccelerometerHelperReadingChanged);
    AccelerometerHelper.Instance.IsActive = true;
}

private void OnAccelerometerHelperReadingChanged
(object sender, AccelerometerHelperReadingEventArgs e)
{
    Dispatcher.BeginInvoke(() =>
    {
        raw.Text = String.Format("{0:0.000}, {1:0.000}, {2:0.000}",
            e.RawAcceleration.X,
            e.RawAcceleration.Y,
            e.RawAcceleration.Z);
        optimal.Text = String.Format("{0:0.000}, {1:0.000}, {2:0.000}",
            e.OptimallyFilteredAcceleration.X,
            e.OptimallyFilteredAcceleration.Y,
            e.OptimallyFilteredAcceleration.Z);
        average.Text = String.Format("{0:0.000}, {1:0.000}, {2:0.000}",
            e.AverageAcceleration.X,
            e.AverageAcceleration.Y,
            e.AverageAcceleration.Z);
        lowPass.Text = String.Format("{0:0.000}, {1:0.000}, {2:0.000}",
            e.LowPassFilteredAcceleration.X,
            e.LowPassFilteredAcceleration.Y,
            e.LowPassFilteredAcceleration.Z);
    }
    );
}

```

The data sets are listed in order of latency, given the current settings for things such as the number of data points to average, and so on. The raw data stream is unprocessed, so there's effectively zero latency in rendering the data to the UI. At the other end, the processing for the low-pass data takes the longest time. Increasing the number of data points to average will generally add latency to the averaging data set. When this application runs, the raw data values stabilize rapidly, whereas the higher latency data sets take longer to stabilize.

Figure 9-8 shows another example, this one focusing on the *OrientationHelper* in the Level Starter Kit. Again, the solution is shown on the left; the runtime behavior is on the right. This is the *Test OrientationHelper* solution in the sample code.

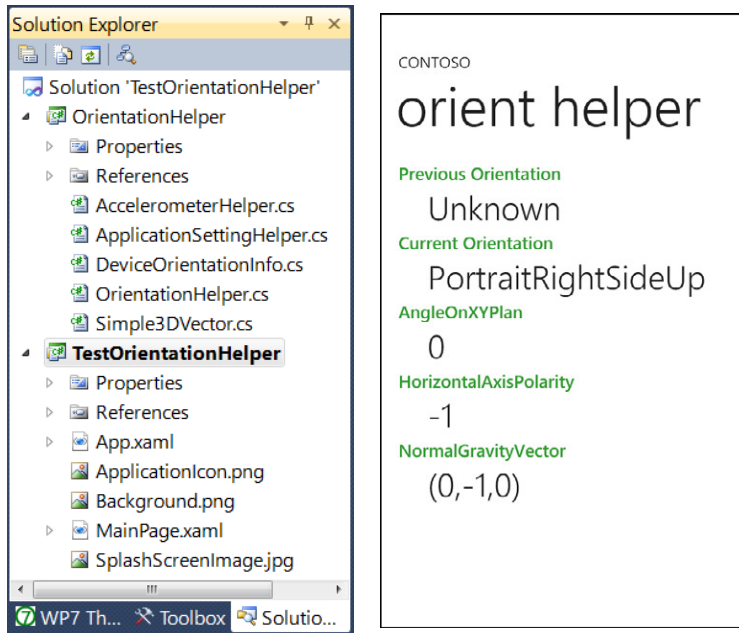


FIGURE 9-8 You can use the *OrientationHelper* class to smooth orientation readings.

The *OrientationHelper* uses the *AccelerometerHelper* underneath, because it also processes accelerometer reading events, taking the processed signal from the *AccelerometerHelper*.

These are the basic steps for reusing the *OrientationHelper* from the Level Starter Kit:

1. Create a Phone Application project named, for example, *TestOrientationHelper*.
2. Add a Class Library project to this solution. Name it *OrientationHelper*.
3. Add existing items to the class library: the *AccelerometerHelper.cs*, *ApplicationSettingHelper.cs*, and *Simple3DVector.cs*, as before, plus the *DeviceOrientationInfo.cs* and *OrientationHelper.cs*.
4. Add a reference to *Microsoft.Devices.Sensors.dll*.
5. Build the *OrientationHelper* class library.
6. In the *TestOrientationHelper* project, add a reference to the *OrientationHelper* class library.
7. In the application, hook up the *OrientationChanged* event on the singleton *OrientationHelper* instance. As before, start the accelerometer by setting the *IsActive* property on the *AccelerometerHelper* singleton instance to *true*.
8. In your *OrientationChanged* event handler, retrieve the *OrientationHelperReadingEventArgs* values, and do something interesting with them.

From this event object, you can retrieve the current and previous orientation (an enumeration with the values: *Unknown*, *ScreenSideUp*, *ScreenSideDown*, *PortraitRightSideUp*,

LandscapeRight, *LandscapeLeft*, *PortraitUpSideDown*), the *AngleOnXYPlan* (which will be 0° for vertical, or ±90° for horizontal), the *HorizontalAxisPolarity* (0 for vertical, or ±1 for horizontal), and the *NormalGravityVector* (X,Y,Z values).

The following listing shows the simplest use of this, rendering the orientation values as strings:

```
public MainPage()
{
    InitializeComponent();

    AccelerometerHelper.Instance.IsActive = true;
    DeviceOrientationHelper.Instance.OrientationChanged +=
        new EventHandler<DeviceOrientationChangedEventArgs>(
            orientationHelper_OrientationChanged);
}

private void orientationHelper_OrientationChanged(
    object sender, DeviceOrientationChangedEventArgs e)
{
    Dispatcher.BeginInvoke(() =>
    {
        previous.Text = e.PreviousOrientation.ToString();
        current.Text = e.CurrentOrientation.ToString();

        DeviceOrientationInfo doi =
            DeviceOrientationHelper.GetDeviceOrientationInfo(e.CurrentOrientation);
        angle.Text = doi.AngleOnXYPlan.ToString();
        polarity.Text = doi.HorizontalAxisPolarity.ToString();
        vector.Text = doi.NormalGravityVector.ToString();
    });
}
```

Shake

Just as low-level touch events can be modeled as more complex user-centric gestures such as pinch/stretch or flick, so too can raw accelerometer events be modeled as more complex user-centric gestures such as rotate or shake. If you analyze a shake gesture, it can be modeled as a series of accelerometer changes (typically on one axis) that oscillates between maxima in the two opposite directions of that dimension. In other words, it swings back and forth (or up and down, or left and right), with roughly the same extremes of value in both directions, and changes in the other two axes are effectively noise that should be ignored.

Microsoft provides a Shake Gesture Library on the AppHub, which is available at http://create.msdn.com/en-us/education/catalog/article/Recipe_Shake_Gesture_Library. You can use this to model shake gestures in your applications. This library also makes use of the *AccelerometerHelper* class available in the Level Starter Kit, primarily to smooth out the raw data stream before processing it. The core class in the Shake Gesture Library is the *ShakeGesturesHelper*, which takes the smoothed data and categorizes it into “shake” and “still” segments, and then determines the signal boundaries to determine where the stream includes shake gestures.

Not surprisingly, shake gestures are individualized: different people shake differently, and different applications have different shake requirements. For this reason, the library includes several configuration options, with which you can establish what constitutes a valid shake gesture for your application.

Figure 9-9 illustrates an example application that uses the Shake Gesture Library (the *TestShake* solution in the sample code).

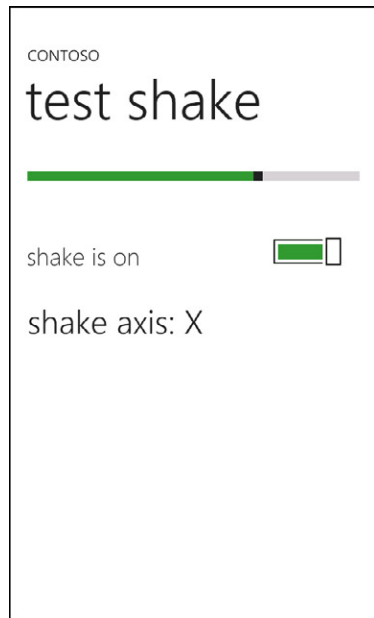


FIGURE 9-9 You can use the Shake Gesture Library to model shake gestures in your application.

In this example, you use the *Slider* to govern the sensitivity of the shake readings, and the *Toggle Switch* to turn the accelerometer on and off. The *TextBlock* displays the shake axis. To use the Shake Gesture Library, you can download the sample code from AppHub and build the project. Then, add a reference to the *ShakeGestures.dll* to your application project. In this example, you need to set up the *ShakeGesturesHelper* singleton object in the *MainPage* constructor. At a minimum, you need to hook up the *ShakeGesture* event. You can optionally also set properties, and in this example, you should set the *MinimumRequiredMovesForShake* to 4, which means that the user will have to shake the phone four times (twice in each direction) on the same axis before you start processing readings.

```
public MainPage()
{
    InitializeComponent();

    ShakeGesturesHelper.Instance.ShakeGesture +=
        new EventHandler<ShakeGestureEventArgs>(Instance_ShakeGesture);
    ShakeGesturesHelper.Instance.MinimumRequiredMovesForShake = 4;
}
```

In the handler for the *ShakeGesture* event, you extract the *ShakeType* value from the *ShakeGestureEventArgs* and use it to update the *TextBlock*. Note that this is the only property exposed from the event, which is minimally useful. If you need to get more values from the underlying accelerometer readings, you can always modify the library source code to suit your own purposes.

```
private void Instance_ShakeGesture(object sender, ShakeGestureEventArgs e)
{
    this.Dispatcher.BeginInvoke(() =>
    {
        this.ShakeTypeText.Text = String.Format("shake axis = {0}", e.ShakeType);
    });
}
```

You handle the *ToggleSwitch* events to toggle the *ShakeGestureHelper* (and therefore, the underlying accelerometer) on and off. The *Slider* value is used to set the *ShakeMagnitudeWithoutGravitationThreshold* property—any readings below this value will be ignored for the purposes of computing the shake gesture. Higher values indicate more vigorous shaking.

```
private void ToggleSwitch_Checked(object sender, RoutedEventArgs e)
{
    ShakeGestureHelper.Instance.Active = true;
    this.ShakeToggle.Content = "shake is on";
}

private void ToggleSwitch_Unchecked(object sender, RoutedEventArgs e)
{
    ShakeGestureHelper.Instance.Active = false;
    this.ShakeToggle.Content = "shake is off";
}

private void SensitivitySlider_ValueChanged(
    object sender, RoutedPropertyChangedEventArgs<double> e)
{
    ShakeGestureHelper.Instance.ShakeMagnitudeWithoutGravitationThreshold =
        SensitivitySlider.Value;
}
```

In the XAML, you should set the *Slider Maximum* to 1.5, which is about the biggest threshold value that you can use realistically, and even this will generally require that the user makes shake gestures of a foot or two on each shake.

```
<Slider
    x:Name="SensitivitySlider"
    ValueChanged="SensitivitySlider_ValueChanged"
    Maximum="1.5"
    LargeChange="0.15" />
```

Geo-Location

The application platform exposes a *GeoCoordinateWatcher* class, which, like the *Accelerometer* class, provides a managed representation of the geolocation services provided by the underlying phone system. Under the covers, location information is gathered from a variety of sources, including both on-device hardware and drivers and via web service calls to a cloud-based Microsoft location service. The architecture is illustrated in Figure 9-10.

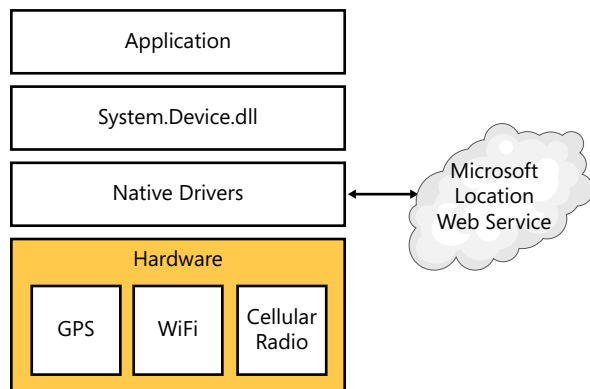


FIGURE 9-10 The geolocation architecture provides a software stack that's based on the underlying hardware.

The location data surfaced by the *GeoCoordinateWatcher* can be sourced from any of the underlying sensors (GPS, WiFi, cellular radio) or from the Microsoft location web service. The code in *System.Device.dll* wraps the underlying native OS feature that determines the best data source to feed to the *GeoCoordinateWatcher*, depending on which source(s) are available and whether the application requires default or high accuracy. Figure 9-11 illustrates an application (the *SimpleGeoWatcher* application in the sample code) that uses location data, reporting each location event as it occurs. You can see how this could easily form the basis of a run or trail-tracking application.

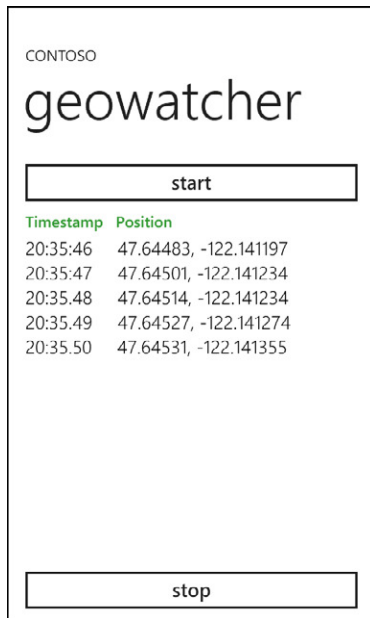


FIGURE 9-11 The *GeoCoordinateWatcher* provides longitude and latitude readings.

To use location functionality, the application declares a *GeoCoordinateWatcher* object and instantiates it in the *OnNavigatedTo* override. The constructor is your only opportunity to set the required accuracy. You can subsequently retrieve this value from the *DesiredAccuracy* property, which is read-only. You also declare a collection of *GeoPosition* objects, which will be stored each time you get a location event. This is enabled by hooking up the *PositionChanged* event.

```
private GeoCoordinateWatcher geoWatcher;
public ObservableCollection<GeoPosition<GeoCoordinate>> Positions;

protected override void OnNavigatedTo(NavigationEventArgs e)
{
    if (geoWatcher == null)
    {
        geoWatcher = new GeoCoordinateWatcher(GeoPositionAccuracy.High);
    }

    geoWatcher.PositionChanged +=
        new EventHandler<GeoPositionChangedEventArgs<GeoCoordinate>>
            (gcw_PositionChanged);
}
```

In the *PositionChanged* event handler, extract the *Position* value and store it in the collection. The *Position* property is of type *GeoPosition<T>*, which includes both a *DateTimeOffset* and a position of type *<T>* (in this case, a *GeoCoordinate* value). The *GeoCoordinate* type represents a geographical location with latitude and longitude coordinates.

```
private void gcw_PositionChanged(object sender, GeoPositionChangedEventArgs<GeoCoordinate> e)
{
    Positions.Add(e.Position);
}
```

As with the *Accelerometer* class, the only other thing you need to do is to *Start* and *Stop* the *GeoCoordinateWatcher* at appropriate times. You only start it under user control, in the *Click* handler for the corresponding button. However, you stop it either when the user asks to stop, or—as a good housekeeping technique—in the *OnNavigatedFrom* override.

```
protected override void OnNavigatedFrom(NavigationEventArgs e)
{
    geoWatcher.Stop();
    geoWatcher = null;
}
```

The *GeoCoordinate* type provides additional values, including *Speed* and *Altitude*, as shown in Figure 9-12. This is the *TestGeoCoordinates* application in the sample code. The interesting code for this application is in the *PositionChanged* event handler.

```
private void gcw_PositionChanged(object sender, GeoPositionChangedEventArgs<GeoCoordinate> e)
{
    timestamp.Text = e.Position.Timestamp.ToString();
    latitude.Text = String.Format("{0:0.00}", e.Position.Location.Latitude);
    longitude.Text = String.Format("{0:0.00}", e.Position.Location.Longitude);
    speed.Text = String.Format("{0:0.00}", e.Position.Location.Speed);
    altitude.Text = String.Format("{0:0.00}", e.Position.Location.Altitude);
}
```

You don't need to bother using *Dispatcher.BeginInvoke* to update the UI in the event handler. This is because the *PositionChanged* events will come in on the UI thread anyway. This seems a little unexpected, but it's actually a deliberate technique to make it easier for developers to consume.

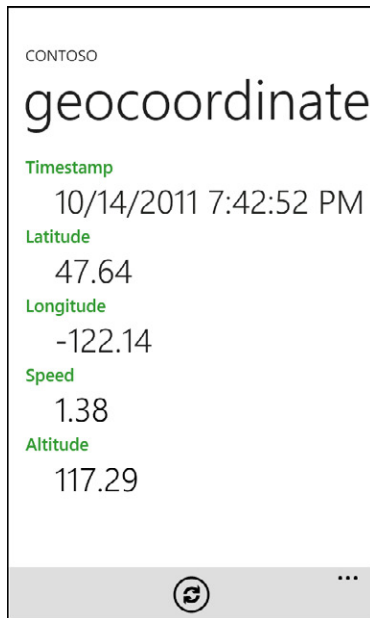


FIGURE 9-12 *GeoCoordinate* values include speed and altitude readings.

Keep in mind that sensors use power. And sensors such as the accelerometer and location can consume a lot of power (reducing battery life) because of the way they fetch and report data very frequently. You should therefore consider very carefully how you want to use classes such as the *Accelerometer* and *GeoCoordinateWatcher*. You should avoid running them without notifying the user, and always give the user control over starting and stopping them. In some applications, the use of these features is implicit and obvious from the nature of the application, but it's still a good idea to inform the user that you're doing this, and preferably to ask his permission at least once, in advance. You should also turn them off when not needed. Consider using the default accuracy setting, which is less accurate but consumes less power. For example, use the default setting if you only need a less granular location, such as the general city area for weather forecasts or basic personalization. Note that accelerometer readings are not sent while the screen lock is engaged.

You are also encouraged to set the *MovementThreshold* property on the *GeoCoordinateWatcher* class. This is the distance in meters that the phone must move (relative to the last *PositionChanged* event) before the location provider raises another *PositionChanged* event. To optimize battery life, the recommended setting is above 20. An appropriate setting depends on the nature of your application: if you're tracking position changes for someone walking, then a smaller setting might be more appropriate. On the other hand, if you're tracking movements of a car, then a much higher value will probably be more useful. Setting *MovementThreshold* to a higher value will save some CPU time as the application platform does not report all values from the sensor, but it does not prevent the sensors from retrieving the data, so battery savings are minimal.

You can use the Reactive Extensions for *GeoCoordinateWatcher* data in exactly the same way as for *Accelerometer* data. This way, you can sample the data stream and apply a filter of some kind. However, this also does not prevent the underlying system from sourcing the sensor data, nor does it prevent the application platform from propagating the data changed events to your application, so again, there are no battery savings with this approach.

Summary

In this chapter, you examined the different levels and types of application platform support for integrating your application with standard features and services on the phone. Built on top of a baseline represented by a consistent hardware specification, the application platform exposes sensor and device functionality through a solid set of classes. Built-in applications such as photo gallery, contacts, and connection to the Windows Phone marketplace are all thoughtfully exposed by wrapper classes that take care of the complex internal behavior and cross-application hook-ups, while providing a developer-friendly API surface with which to work. Note that Windows Phone 7.1 introduces additional sensors and enhanced sensor application support, as described in Chapter 16.

Media Services

As detailed in Chapter 9, “Phone Services,” the Windows Phone platform exposes a range of features, mostly via Launchers and Choosers. In addition, the platform also provides extensive support for media services, including audio and video playback and an FM radio tuner. You can choose from several different sets of media-related application programming interfaces (APIs), depending on the specific requirements of your application. If you have a simple requirement for media playback, you can use the *MediaPlayerLauncher*. If you need more flexibility, the *MediaElement* type might suit your needs better. If you need more finely-grained control over the media file content, you can use the *MediaStreamSource* API, and so on. In this chapter, you’ll explore these various levels of application support and your choices for working with audio and video.

Audio and Video Hardware

The audio hardware specifications include:

- A microphone capable of handling input in the 150 Hz to 7 kHz range.
- Audio codecs at 8 kHz and 16 kHz mono input/output sample rates, plus one or more stereo output sample rates of ≥ 44.1 kHz.

The Windows Phone is designed from the ground up to excel at entertainment, with the explicit aim of giving customers access to media on their terms—including the ability to stream music and video, and to watch full-length movies on a large, wide, bright, high-resolution screen. As a result, the minimum hardware requirements for Windows Phone devices set the bar high for video encode and decode standards, including the following:

- The video must be encoded in H.264 Baseline Profile, with the audio stream encoded in AAC-LC.
- Video must be packaged as an MP4 media file.
- The encoders and the MP4 file writer/multiplexor must be provided in the form of *DirectShow* filters.
- If H.264 encoding is not supported at an HD resolution, then MPEG4 Layer 2 encoding can be used, instead.

- Video encoding at VGA (640x480) or higher resolution must be supported at a minimum frame rate of 30 frames per second (FPS) and a minimum bit rate of 2.5 megabits per second (Mbps).

The number one reason for these encoding requirements is to preclude the need for proprietary software to play back Windows Phone videos on your PC or other device.

Audio and Video APIs

Windows Phone includes a range of techniques for working with media, both audio and video, in four broad categories, as described in Table 10-1.

TABLE 10-1 Media Techniques for Windows Phone

Category	Technique	Description
Media Playback	<i>MediaPlayerLauncher</i>	A launcher for playing audio or video with the built-in player experience. Primarily used for XNA videos.
	<i>MediaElement</i>	The primary wrapper class for audio and/or video files in Microsoft Silverlight applications.
	<i>MediaStreamSource</i>	Allows you to work directly with the media pipeline, and is most often used to enable the <i>MediaElement</i> to use a container format not natively supported by Silverlight.
Audio Input and Manipulation	<i>SoundEffect</i> , <i>SoundEffectInstance</i> , <i>DynamicSoundEffect</i>	XNA classes for working with audio content, in both Silverlight and XNA applications.
	<i>Microphone</i>	The only API for the microphone on the phone is the XNA <i>Microphone</i> class. It is used for both XNA and Silverlight applications.
Platform Integration	<i>MediaHistory</i>	Allows you to integrate your application with the Music and Videos hub.
Radio	<i>FMRadio</i>	Simple wrapper class for interacting with the FM radio on the phone.

Note that DirectX and XNA developers are accustomed to using the Microsoft Cross-Platform Audio Creation Tool (XACT). This is a GUI tool that helps you to create and edit audio content, manage audio file banks, and implement audio playback triggers in games. However, this tool is not used in Windows Phone 7 development.

Media Playback

The platform provides three main types for playing audio and video: the *MediaPlayerLauncher*, *MediaElement*, and *MediaStreamSource*. Each type offers varying levels of flexibility and control, which are described in the following sections.

The *MediaPlayerLauncher* Class

As with all Launchers and Choosers provided by the application platform, the *MediaPlayerLauncher* is very easy to use. It is a simple wrapper that provides access to the underlying media player application, without exposing any of the complexity. To use this in your application, you follow the same pattern as for other launchers. The listing that follows shows how to invoke the *MediaPlayerLauncher* with a media file that is deployed as *Content* within the application's XAP. You can see this at work in the *TestMediaPlayer* application in the sample code.

```
MediaPlayerLauncher player = new MediaPlayerLauncher();
player.Media = new Uri(@"Media/campus_20111017.wmv", UriKind.Relative);
player.Controls = MediaPlayerControls.Pause | MediaPlayerControls.Stop;
player.Location = MediaLocationType.Install;
player.Show();
```

You can assign the *Controls* property from a flags enum of possible controls. The preceding listing specifies only the *Pause* and *Stop* controls, whereas the listing that follows specifies all available controls (including *Pause*, *Stop*, *Fast Forward*, *Rewind*). This listing also shows how to specify a remote URL for the media file, together with the *MediaLocationType.Data*:

```
MediaPlayerLauncher player = new MediaPlayerLauncher();
player.Media = new Uri(
    @"http://ecn.channel9.msdn.com/o9/ch9/0882/570882/WelcomeToInsideWPShow_ch9.mp4",
    UriKind.Absolute);
player.Controls = MediaPlayerControls.All;
player.Location = MediaLocationType.Data;
player.Show();
```

The *MediaElement* Class

Figure 10-1 shows a simple application (*TestMediaElement* in the sample code) that uses the *MediaElement* control. The *MediaElement* class is a *FrameworkElement* type that you can use in your application—superficially, at least—in a similar way as the *Image* type.

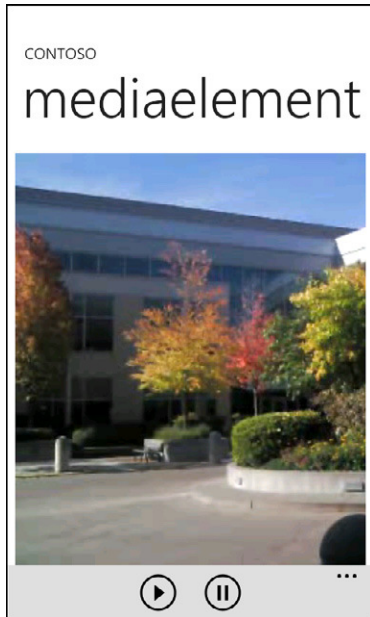


FIGURE 10-1 You can use the *MediaElement* control for simple audio and video playback.

You can set up a *MediaElement* in code or in XAML. The *MediaElement* class exposes a set of media-specific properties, including:

- **AutoPlay** This property defines whether to start playing the content automatically. The default is true, but in many cases, you probably want to set it to false because of application lifecycle/tombstoning issues.
- **IsMuted** This defines whether sound is on (which is the default).
- **Volume** This property is set in the range 0 to 1, where 1 (the default) is full volume.
- **Stretch** This is the same property used by an *Image* control to govern how the content fills the control (the default is *Fill*).

The following code shows how to set up a *MediaElement* in XAML:

```
<MediaElement
  x:Name="myVideo" Source="/Media/campus_20111017.wmv" AutoPlay="False"
  IsMuted="False" Volume="0.5" Stretch="UniformToFill"/>
```

It then just remains to invoke methods such as *Play* and *Pause*; in this example, these are triggered in App Bar button *Click* handlers.

```
private void appBarPlay_Click(object sender, EventArgs e)
{
    myVideo.Play();
}
```

```
private void appBarPause_Click(object sender, EventArgs e)
{
    myVideo.Pause();
}
```

You might be thinking, “This seems to be too simple to be true”—and you’d be right. In the application’s current state, there is a very small chance that the media file might not be fully opened at the point when the user taps the play button, and this would raise an exception. To make the application more robust and preclude this scenario, it would be better to have the App Bar buttons initially disabled and to handle the *MediaOpened* event on the *MediaElement* object to enable them.

```
private void myVideo_MediaOpened(object sender, System.Windows.RoutedEventArgs e)
{
    ((ApplicationBarIconButton)AppBar.Buttons[0]).IsEnabled = true;
    ((ApplicationBarIconButton)AppBar.Buttons[1]).IsEnabled = true;
}
```



More Info If you need sample media files for testing your application, you can use the collection that Microsoft provides on AppHub, which is available at <http://create.msdn.com/en-US/education/catalog/utility/soundlab>. These are provided under the Microsoft Public License (Ms-PL). You can also use the audio and video files provided by default with Windows, which are typically installed to C:\Users\Public\Music\Sample Music and C:\Users\Public\Videos\Sample Videos.

The *MediaStreamSource* and *ManagedMediaHelpers* Classes

Using the *MediaPlayerLauncher* provides the simplest approach for playing media in your application. For more flexibility, the *MediaElement* class offers a good set of functionality and is suitable for most phone applications. However, if you really need lower-level access to the media file contents, you can use the *MediaStreamSource* class. This class offers more control over the delivery of content to the media pipeline, and is particularly useful if you want to use media files in a format that are not natively supported by *MediaElement*, or for scenarios that are simply not yet supported in Silverlight, such as RTSP protocol support, SHOUTcast protocol support, seamless audio looping, ID3 v1/v2 meta-data support, adaptive streaming, or multi-bitrate support.

Unfortunately, the *MediaStreamSource* class is only minimally documented. Fortunately, Microsoft has made available a set of helper classes, which you can obtain at <https://github.com/loarabia/ManagedMediaHelpers>. These are provided in source-code format and include library projects and demonstration apps for Silverlight Desktop and Windows Phone. Note that the library source code is all in the Desktop projects—the phone projects merely reference the Desktop source files. The phone demonstration application is, of course, independent.

Here’s how you can use these. First, create a phone application solution, as normal. Then, add the *ManagedMediaHelpers* library projects (either take copies, so that you have all the sources available, or build the library assemblies, and then use *CopyLocal=true* to reference them in your

solution). If you add the library phone projects to your solution, you will then need to copy across the source files from the Desktop projects. You need two library projects: the *MediaParsers.Phone* and *Mp3MediaStreamSource.Phone* projects. Between them, these projects provide wrapper classes for the MP3 file format. The *Mp3MediaStreamSource.Phone* project has a reference to the *MediaParsers.Phone* project. Your application needs to have a reference to the *Mp3MediaStreamSource.Phone* project. Figure 10-2 shows a solution with this setup. This is the *TestMediaHelpers* solution in the sample code.

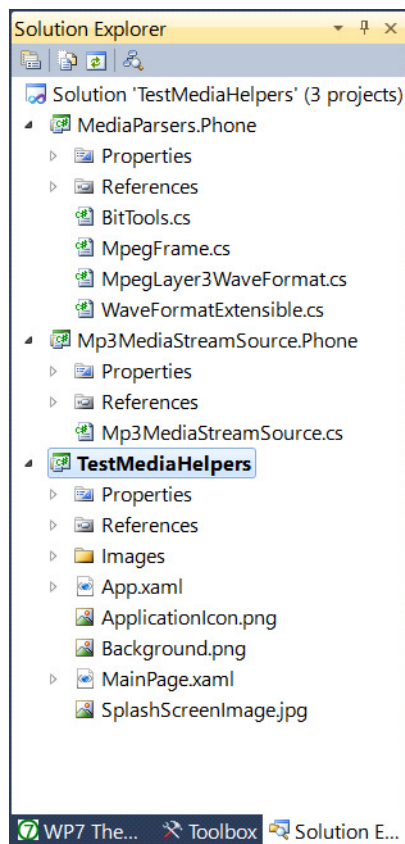


FIGURE 10-2 You can use *ManagedMediaHelpers* for low-level control of media playback.

Having set up the projects, you can then declare an *Mp3MediaStreamSource* object. In this example, you fetch a remote MP3 file by using an *HttpWebRequest*. When you get data back, you use it to initialize your *Mp3MediaStreamSource* and set that as the source for a *MediaElement* object declared in XAML.

```

private HttpWebRequest request;
private Mp3MediaStreamSource mss;
private string mediaFileLocation =
    @"http://ecn.channel9.msdn.com/o9/ch9/0882/570882/WelcomeToInsideWPShow_ch9.mp3";

public MainPage()
{
    InitializeComponent();
}

private void Get_Click(object sender, EventArgs e)
{
    request = WebRequest.CreateHttp(mediaFileLocation);
    request.AllowReadStreamBuffering = true;
    IAsyncResult result =
        request.BeginGetResponse(new AsyncCallback(this.RequestCallback), null);
}

private void RequestCallback(IAsyncResult asyncResult)
{
    HttpWebResponse response =
        request.EndGetResponse(asyncResult) as HttpWebResponse;
    Stream s = response.GetResponseStream();
    mss = new Mp3MediaStreamSource(s, response.ContentLength);
    Dispatcher.BeginInvoke(() =>
    {
        MyMp3.SetSource(mss);
    });
}

```

Notice how this code sets the *AllowReadStreamBuffering* property to *true*. If you enable buffering like this, then it becomes easier to work with the stream source, because all the data is downloaded first. On the other hand, you can't start processing the data until the entire file is downloaded—plus, it uses more memory. The alternative is to use the asynchronous methods and read the stream in the background. This simple example shows how you can easily use the *MediaStreamSource* type via the *ManagedMediaHelpers*, although it doesn't really show the power of these APIs—by definition, these are advanced scenarios.

MediaElement Controls

When you point a *MediaElement* to a remote media source and start playing, the content is downloaded to the device and playing commences as soon as there is enough in the buffer to play. Download and buffering continues in the background while content that was already buffered is playing. The *MediaElement* class exposes *BufferingChanged* and *DownloadChanged* events, which you can handle if you're interested in the progress of these operations. The standard media player application on the device, invoked via *MediaPlayerLauncher*, offers a good set of UI controls for starting, stopping, and pausing, as well as a timeline progress bar that tracks the current position in the playback, and a countdown from the total duration of the content. *MediaElement* does not provide such UI controls, but you can emulate these features by using the properties that it exposes, notably the *Position* and *NaturalDuration* values.

Figure 10-3 shows an application (the *TestVideo* solution in the sample code) that uses *MediaElement* with a *Slider* and *TextBlock* controls to mirror some of the UI features of the standard media player.

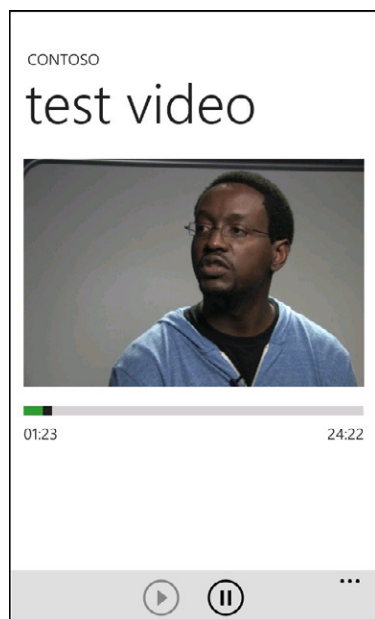


FIGURE 10-3 You can report media playback progress by using custom UI.

In the XAML, you declare a *MediaElement*, a *Slider*, and a couple of *TextBlock* controls (to represent the playback timer count-up and count-down values).

```
<StackPanel x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <MediaElement
        x:Name="Player" Height="297" Width="443" AutoPlay="False" Stretch="UniformToFill"
        Source="http://ecn.channel9.msdn.com/o9/ch9/f076/8a77bfbcb-5e80-4bcb-aa44-9e1a0116f076/
IWPS08TakingALookInToArch_ch9.wmv"/>
    <Slider
        x:Name="MediaProgress" Height="90" Margin="-5,0"
        Maximum="1" LargeChange="0.1" ValueChanged="MediaProgress_ValueChanged"/>
</StackPanel>
<TextBlock
    Grid.Row="1" x:Name="ElapsedTime" Text="00:00" IsHitTestVisible="False"
    Width="60" Height="30" Margin="19,180,0,0" HorizontalAlignment="Left" />
<TextBlock
    Grid.Row="1" x:Name="RemainingTime" Text="00:00" IsHitTestVisible="False"
    Width="60" Height="30" Margin="0,180,6,0" HorizontalAlignment="Right"/>
```

The *MediaElement* points to a video file on AppHub—as it happens, this example is an actual interview with the Windows Phone architect, Abolade Gbadegesin. For the *Slider*, the important piece is to sink the *ValueChanged* event. Note that the two *TextBlock* controls are not part of the same *StackPanel*, thus you can specify *Margin* values that effectively overlay them on top of the *Slider*. Because

of this, you need to ensure that the *TextBlock* controls are not hit-testable so that they don't pick up touch gestures intended for the *Slider*.

In the *MainPage* code-behind, you declare a *TimeSpan* field for the total duration of the video file, and a *bool* to track whether or not you are updating the *Slider* based on the current playback position. The *Slider* performs a dual role: the first aspect is a passive role, in which you update it programmatically to synchronize it with the current playback position; the second aspect is an active role, in which the user can click or drag the *Slider* position—you respond to this in the application by setting the *MediaElement.Position* value.

```
private bool isUpdatingSliderFromMedia;
private TimeSpan totalTime;
private DispatcherTimer timer;

public MainPage()
{
    InitializeComponent();
    timer = new DispatcherTimer();
    timer.Interval = new TimeSpan(0, 0, 0, 0, 500);
    timer.Tick += new EventHandler(timer_Tick);
    timer.Start();

    appBarPlay = ApplicationBar.Buttons[0] as ApplicationBarIconButton;
    appBarPause = ApplicationBar.Buttons[1] as ApplicationBarIconButton;
}
```

Here's how to implement the first role. You implement a *DispatcherTimer* with a half-second interval, updating the *Slider* on each tick. The first thing you do is cache the total duration of the video file—this is a one-off operation. Next, you calculate the time remaining and render this in the corresponding *TextBlock*. Assuming that the playback has actually started (even if it is now paused), you then calculate how much of the video playback is complete and use the resulting value to update the position of the *Slider*. You also need to update the current "elapsed time" value to match the playback position. Throughout this operation, you toggle the *isUpdatingSliderFromMedia* flag—this will be used in another method.

```
private void timer_Tick (object sender, EventArgs e)
{
    if (totalTime == TimeSpan.Zero)
    {
        totalTime = Player.NaturalDuration.TimeSpan;
    }

    TimeSpan remainingTime = totalTime - Player.Position;
    String remainingTimeText = String.Format("{0:00}:{1:00}",
        (remainingTime.Hours * 60) + remainingTime.Minutes, remainingTime.Seconds);
    RemainingTime.Text = remainingTimeText;

    isUpdatingSliderFromMedia = true;
    if (Player.Position.TotalSeconds > 0)
    {
        double fractionComplete = Player.Position.TotalSeconds / totalTime.TotalSeconds;
        MediaProgress.Value = fractionComplete;
        TimeSpan elapsedTime = Player.Position;
```

```

        String elapsedTimeText = String.Format("{0:00}:{1:00}",
            (elapsedTime.Hours * 60) + elapsedTime.Minutes, elapsedTime.Seconds);
        ElapsedTime.Text = elapsedTimeText;
        isUpdatingSliderFromMedia = false;
    }
}

```

In the handler for the *ValueChanged* event on the *Slider*, check first that you're not in this handler as a result of what you did in the previous method. That is, you need to verify that you're not here because you're updating the *Slider* from the media position. The other scenario for which you'd be in this handler is if the user is clicking or dragging the *Slider* position. In this case, assuming that the media content can actually be repositioned (*CanSeek* is *true*), you reset its position based on the *Slider* position. This is the inverse of the normal behavior, for which you set the *Slider* position based on the media position.

```

private void MediaProgress_ValueChanged(
    object sender, RoutedEventArgs<double> e)
{
    if (!isUpdatingSliderFromMedia && Player.CanSeek)
    {
        TimeSpan duration = Player.NaturalDuration.TimeSpan;
        int newPosition = (int)(duration.TotalSeconds * MediaProgress.Value);
        Player.Position = new TimeSpan(0, 0, newPosition);
    }
}

```

The App Bar buttons invoke the *MediaElement Play* and *Pause* methods, each of which is very simple. In the case of *Pause*, you need to first establish that this media content can actually be paused. Note that if you don't check *CanSeek* or *CanPause*, and just go ahead and attempt to set *Position* or call *Pause*, in neither case is an exception thrown. Rather, the method simply does nothing. So, these checks are arguably redundant, except that you should use them to avoid executing unnecessary code.

```

private void appBarPause_Click(object sender, EventArgs e)
{
    if (Player.CanPause)
    {
        Player.Pause();
    }
}

```

Audio Input and Manipulation

Both *MediaElement* and *MediaStreamSource* give you some ability to manipulate media during playback. For even greater flexibility, you can use the *SoundXXX* classes. You can also use the *Dynamic SoundEffectInstance* class in combination with the *Microphone* class to work with audio input.

The *SoundEffect* and *SoundEffectInstance* Classes

As an alternative to using *MediaElement*, you can use the XNA *SoundEffect* classes, instead. The disadvantage of this is that it is an XNA type, so your Silverlight application needs to pull in XNA libraries and manage the different expectations of the XNA runtime. One of the advantages is that you cannot play multiple *MediaElements* at the same time, whereas you can play multiple *SoundEffects* at the same time. Another advantage is that the *SoundEffect* class offers better performance than *MediaElement*. This is because the *MediaElement* carries with it a lot of UI baggage, relevant for a *Control* type. On the other hand, the *SoundEffect* class is focused purely on audio and has no UI features.

The code that follows shows how to use *SoundEffect*. It also illustrates the *SoundEffectInstance* class, which offers greater flexibility than the *SoundEffect* class. This application is in the *TestSoundEffect* solution in the sample code. The primary difference is that *SoundEffect* has no *Pause* method—the playback is essentially “fire and forget.” You can create a *SoundEffectInstance* object from a *SoundEffect*, and this does have a *Pause* method. Also, you can create multiple *SoundEffectInstances* from the same *SoundEffect*; they’ll all share the same content, but you can control them independently.

This application has two sound files, built as *Content* into the XAP (but not into the DLL). These are available in the SoundLab download from AppHub, mentioned earlier. In the application, you first need to add a reference to the Microsoft.Xna.Framework.dll. Then, declare *SoundEffect* and *SoundEffectInstance* fields. Early in the life of the application, you load the two sound files from the install folder of the application by using *Application.GetResourceStream*. This can be slightly confusing, because you need to explicitly build the files as *Content* not *Resource*. However, *GetResourceStream* can retrieve a stream for either *Content* or *Resource*. If the sound file is a valid PCM wave file, you can use the *FromStream* method to initialize a *SoundEffect* object. For one of these *SoundEffect* objects, you create a *SoundEffectInstance*.

```
private SoundEffect sound;
private SoundEffectInstance soundInstance;

public MainPage()
{
    InitializeComponent();

    sound = LoadSound("Media/AfternoonAmbienceSimple_01.wav");
    SoundEffect tmp = LoadSound("Media/NightAmbienceSimple_02.wav");
    if (tmp != null)
    {
        soundInstance = tmp.CreateInstance();
    }
    InitializeXna();
}

private SoundEffect LoadSound(String streamPath)
{
    SoundEffect s = null;
    try
    {
        StreamResourceInfo streamInfo =
```

```

        App.GetResourceStream(new Uri(streamPath, UriKind.Relative));
        s = SoundEffect.FromStream(streamInfo.Stream);
    }
    catch (Exception ex)
    {
        Debug.WriteLine(ex.ToString());
    }
    return s;
}

```

Not only must the file be a valid WAV file, it must also be in the RIFF bitstream format, mono or stereo, 8 or 16 bit, with a sample rate between 8,000 Hz and 48,000 Hz.



More Info The full set of media codecs supported on Windows Phone is documented at [http://msdn.microsoft.com/en-us/library/ff462087\(VS.92\).aspx](http://msdn.microsoft.com/en-us/library/ff462087(VS.92).aspx).

If the sound file was created on the phone with the same microphone device and saved as a raw audio stream (no file format headers), you could instead work with the stream directly and assume the same sample rate and *AudioChannels* values. This alternative approach is shown in the following code:

```

try
{
    StreamResourceInfo streamInfo =
        App.GetResourceStream(new Uri(streamPath, UriKind.Relative));
    byte[] buffer = new byte[streamInfo.Stream.Length];
    streamInfo.Stream.Read(buffer, 0, (int)streamInfo.Stream.Length);
    s = new SoundEffect(buffer, Microphone.Default.SampleRate, AudioChannels.Mono);
}

```

Also, very early in the life of the application, you must do some housekeeping to ensure that any XNA types work correctly. The basic requirement is to simulate the XNA game loop—this is the core architectural model in XNA, and most significant XNA types depend on this. XNA Framework event messages are placed in a queue that is processed by the XNA *FrameworkDispatcher*. In an XNA application, the XNA *Game* class calls the *FrameworkDispatcher.Update* method automatically whenever *Game.Update* is processed. This *FrameworkDispatcher.Update* method causes the XNA Framework to process the message queue. Conversely, if you use the XNA Framework from an application that does not implement the *Game* class, you must call the *FrameworkDispatcher.Update* method yourself to process the XNA Framework message queue.

There are various ways to achieve this. The simplest approach here is to set up a *DispatcherTimer* to call *FrameworkDispatcher.Update*. The typical tick rate for processing XNA events is 33 ms. The XNA game loop updates and redraws at 30 FPS; that is, one frame every 33 ms.

```

private void InitializeXna()
{
    DispatcherTimer timer = new DispatcherTimer();

```

```

        timer.Interval = TimeSpan.FromMilliseconds(33);
        timer.Tick += delegate { try { FrameworkDispatcher.Update(); } catch { } };
        timer.Start();
    }
}

```

The application provides three App Bar buttons. The *Click* handler for the first one simply plays the *SoundEffect* by invoking the “fire-and-forget” *Play* method. The other two are used to *Start* (that is, *Play*) or *Pause* the *SoundEffectInstance*.

```

private void appBarPlay_Click(object sender, EventArgs e)
{
    if (sound != null)
    {
        sound.Play();
    }
}

private void appBarStart_Click(object sender, EventArgs e)
{
    if (soundInstance != null)
    {
        soundInstance.Play();
    }
}

private void appBarPause_Click(object sender, EventArgs e)
{
    if (soundInstance != null)
    {
        soundInstance.Pause();
    }
}

```

Audio Input and the Microphone

The only way to work with audio input in a Windows Phone application is to use the XNA *Microphone* class. This provides access to the microphone (or microphones) available on the system. Although you can get the collection of microphones, the collection always contains exactly one microphone, so you would end up working with the default microphone anyway. The standard hardware specification requires that all microphones on the device conform to the same basic audio format, and return 16-bit PCM mono audio data, with a sample rate between 8,000 Hz and 48,000 Hz. The low-level audio stack uses an internal circular buffer to collect the input audio from the microphone device. You can configure the size of this buffer by setting the *Microphone.BufferDuration* property. This buffer is actually double-buffered, so the real size of the buffer is twice the size you specify in *BufferDuration*. *BufferDuration* is of type *TimeSpan*, so setting a buffer size of 300 ms will result in a buffer of $2 * 16 * 300 = 9,600$ bytes. *BufferDuration* must be between 100 ms and 1000 ms, in 10 ms increments, giving an underlying buffer size of 3,200–64,000 bytes. The size of the buffer is returned by *GetSampleSizeInBytes*.

There are two different methods for retrieving audio input data:

- Handle the *BufferReady* event and process data when there is a *BufferDuration*'s-worth of data received in the buffer. This has a minimum latency of 100 ms.
- Pull the data independently of *BufferReady* events, at whatever time interval you choose, including more frequently than 100 ms.

For a game, it can often be more useful to pull the data so that you can synchronize sound and action in a flexible manner. For a non-game it is more common to respond to *BufferReady* events. With this approach, the basic steps for working with the microphone are as follows:

1. For convenience, cache a local reference to the default microphone.
2. Specify how large a buffer you want to maintain for audio input, and declare a byte array for this data.
3. Hook up the *BufferReady* event, which is raised whenever a buffer's-worth of audio data is ready.
4. In your *BufferReady* event handler, retrieve the audio input data and do something interesting with it.
5. At suitable points, start and stop the microphone to start and stop the buffering of audio input data.

You might wonder what happens if your application is using the microphone to record sound, and then a phone call comes in and the user answers it. Is the phone call recorded? The answer is no, specifically because this is a privacy issue. So, what happens is that your application keeps recording, but it records silence until the call is finished.

Figure 10-4 shows a simple decibel meter that illustrates a simple use of the microphone. This is the *DecibelMeter* application in the sample code. The application takes audio input data, converts it to decibels, and then displays a graphical representation of the decibel level, using both a rectangle and a text value.

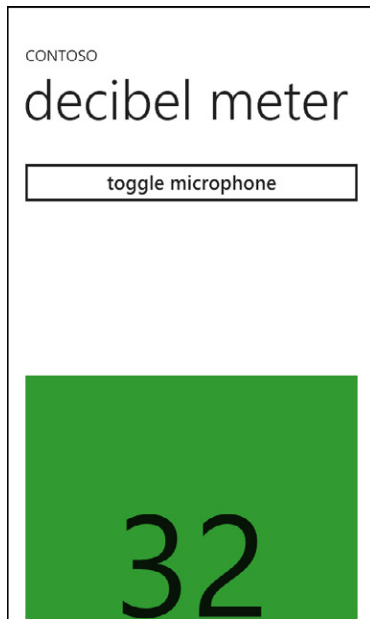


FIGURE 10-4 You can build a simple decibel meter to exercise the microphone.

In the XAML, the application defines a *StackPanel* that contains a *Button* and an inner *Grid*. Inside the *Grid*, you have a *Rectangle* and a *TextBlock*. These are both bottom-aligned and overlapping (the control declared last is overlaid on top of the previous one).

```
<StackPanel x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <Button x:Name="ToggleMicrophone" Content="toggle microphone"
        Click="ToggleMicrophone_Click"/>
    <Grid Height="535">
        <Rectangle x:Name="LevelRect"
            Height="0" Width="432" VerticalAlignment="Bottom"
            Margin="{StaticResource PhoneHorizontalMargin}" />
        <TextBlock
            Text="0" x:Name="SoundLevel" TextAlignment="Center" Width="432"
            FontSize="{StaticResource PhoneFontSizeHuge}"
            VerticalAlignment="Bottom"/>
    </Grid>
</StackPanel>
```

First, you declare a byte array for the audio data and a local reference to the default microphone, and then initialize these in the *MainPage* constructor. You specify that you want to maintain a 300 ms buffer for audio input. Whenever the buffer is filled, you'll get a *BufferReady* event. Retrieve the size of the byte array required to hold the specified duration of audio for this microphone object by using *GetSampleSizeInBytes* (this is how you know what size buffer to allocate). The following code also retrieves the current accent brush and sets this as the *Brush* object with which to fill the rectangle:

```
private byte[] soundBuffer;
private Microphone mic;
```

```

public MainPage()
{
    InitializeComponent();

    Brush accent = (Brush)Resources["PhoneAccentBrush"];
    LevelRect.Fill = accent;

    mic = Microphone.Default;
    mic.BufferDuration = TimeSpan.FromMilliseconds(300);
    mic.BufferReady += Microphone_BufferReady;

    int bufferSize = mic.GetSampleSizeInBytes(mic.BufferDuration);
    soundBuffer = new byte[bufferSize];
}

```

Whenever a buffer's-worth of audio input data is received, you pull that data from the *Microphone* object and copy it into your private buffer in order to work on it. You process this data by determining the average sound level in decibels and rendering text and graphics to represent that level. The rectangle height and position are constrained by the height of the containing grid (534 pixels).

```

void Microphone_BufferReady(object sender, EventArgs e)
{
    int soundDataSize = mic.GetData(soundBuffer);
    if (soundDataSize > 0)
    {
        SoundLevel.Dispatcher.BeginInvoke(() =>
        {
            int decibels = GetSoundLevel();
            SoundLevel.Text = decibels.ToString();
            LevelRect.Height = Math.Max(0, Math.Min(534, decibels * 10));
        });
    }
}

```

The sound pressure level ratio in decibels is given by $20 \cdot \log(\text{actual value} / \text{reference value})$, where the logarithm is to base 10. Realistically, the <reference value> would be determined by calibration. In this example, you use an arbitrary hard-coded calibration value (300), instead. First, you must convert the array of bytes into an array of shorts. Then convert these shorts into decibels.

```

private int GetSoundLevel()
{
    short[] audioData = new short[soundBuffer.Length / 2];
    Buffer.BlockCopy(soundBuffer, 0, audioData, 0, soundBuffer.Length);

    double calibrationZero = 300;
    double waveHeight = Math.Abs(audioData.Max<short>() - audioData.Min<short>());
    double decibels = 20 * Math.Log10(waveHeight / calibrationZero);

    return (int)decibels;
}

```

Finally, you provide a button in the UI so that the user can toggle the microphone on or off.

```

private void ToggleMicrophone_Click(object sender, RoutedEventArgs e)

```



```

{
    if (mic.State == MicrophoneState.Started)
    {
        mic.Stop();
    }
    else
    {
        mic.Start();
    }
}

```

As before, you need to ensure that the XNA types work correctly in a Silverlight application. Previously, you took the approach of a *DispatcherTimer* to provide a tick upon which you could invoke *FrameworkDispatcher.Update* in a simple fashion. A variation on this approach is to implement *IApplicationService* and put the *DispatcherTimer* functionality in that implementation. *IApplicationService* represents an extensibility mechanism in Silverlight. The idea is that where you have a need for some global “service” that must work across your application, you can register it with the runtime. This interface declares two methods: *StartService* and *StopService*. The Silverlight runtime will call *StartService* during application initialization, and it will call *StopService* just before the application terminates. Effectively, you’re taking the *InitializeXna* custom method from the previous example, and reshaping it as an implementation of *IApplicationService*. Then, instead of invoking the method directly, you register the class and leave it to Silverlight to invoke the methods.

The following is the class implementation. As before, you simply set up a *DispatcherTimer* and invoke *FrameworkDispatcher.Update* on each tick.

```

public class XnaFrameworkDispatcherService : IApplicationService
{
    private DispatcherTimer timer;

    public XnaFrameworkDispatcherService()
    {
        timer = new DispatcherTimer();
        timer.Interval = TimeSpan.FromTicks(333333);
        timer.Tick += OnTimerTick;
        FrameworkDispatcher.Update();
    }

    private void OnTimerTick(object sender, EventArgs args)
    {
        FrameworkDispatcher.Update();
    }

    private void IApplicationService.StartService(ApplicationServiceContext context)
    {
        timer.Start();
    }

    private void IApplicationService.StopService()
    {
        timer.Stop();
    }
}

```

Registration is a simple matter of updating the App.xaml to include the custom class in the *ApplicationLifetimeObjects* section.

```
<Application
...standard declarations omitted for brevity.
  xmlns:local="clr-namespace:DecibelMeter">

  <Application.ApplicationLifetimeObjects>

    <local:XnaFrameworkDispatcherService />

    <shell:PhoneApplicationService
      Launching="Application_Launching" Closing="Application_Closing"
      Activated="Application_Activated" Deactivated="Application_Deactivated"/>
  </Application.ApplicationLifetimeObjects>
</Application>
```

Figure 10-5 shows an application that uses the microphone to record sound, and then plays back the sound. This application (the *SoundFx* solution in the sample code) uses a slider to control the sound pitch.



FIGURE 10-5 It's very simple to build sound recording and playback features.

In the *MainPage* constructor, you need to set up the XNA message queue processing, initialize the default microphone (with a 300 ms buffer), and create a private byte array for the audio data, as before. Set the *SoundEffect.MasterVolume* to 1. This is relative to the volume on the device/emulator itself. You can set the volume in a range of 0–1, where 0 approximates silence, and 1 equates to the device volume. You cannot set the volume higher than the volume on the device. Each time the audio

input buffer is filled, you get the data in the private byte array, and then copy it to a *MemoryStream* for processing. Note that you should protect the buffer with a *lock* object. This addresses the issue of the user pressing *Stop* while you're writing to the buffer (this would reset the buffer position to zero).

```
private byte[] soundBuffer;
private Microphone mic;
private MemoryStream ms;
private SoundEffectInstance sei;
private DispatcherTimer timer;

public MainPage()
{
    InitializeComponent();
    timer = new DispatcherTimer();
    timer.Interval = TimeSpan.FromMilliseconds(33);
    timer.Tick += delegate { try { FrameworkDispatcher.Update(); } catch { } };
    timer.Start();

    mic = Microphone.Default;
    mic.BufferDuration = TimeSpan.FromMilliseconds(300);
    mic.BufferReady += Microphone_BufferReady;
    int bufferSize = mic.GetSampleSizeInBytes(mic.BufferDuration);
    soundBuffer = new byte[bufferSize];

    SoundEffect.MasterVolume = 1.0f;
}

private void Microphone_BufferReady(object sender, EventArgs e)
{
    object lockObject = new object();
    lock (lockObject)
    {
        mic.GetData(soundBuffer);
        ms.Write(soundBuffer, 0, soundBuffer.Length);
    }
}
```

The user can tap the App Bar buttons to start and stop the recording. You handle these actions by calling *Microphone.Start* and *Microphone.Stop*. When the user chooses to start a new recording, you close any existing stream and set up a fresh one, and then start the microphone. Conversely, when the user asks to stop recording, you stop the microphone and reset the stream pointer to the beginning.

```
private void StartRecording()
{
    if (ms != null)
    {
        ms.Close();
    }
    ms = new MemoryStream();
    mic.Start();
    isRecording = true;
}

private void StopRecording()
{
    mic.Stop();
}
```

```

        ms.Position = 0;
        isRecording = false;
    }

```

The only other interesting code is starting and stopping playback of the recorded sound. To start playback, you first create a new *SoundEffect* object from the buffer of microphone data. Then, you create a new *SoundEffectInstance* from the *SoundEffect* object, varying the pitch to match the slider value. You also set the *Volume* to 1.0, relative to the *SoundEffect.MasterVolume*; the net effect is to retain the same volume as the device itself. To stop playback, simply call *SoundEffectInstance.Stop*, as before.

```

private void StartPlayback()
{
    SoundEffect se = new SoundEffect(ms.ToArray(), mic.SampleRate, AudioChannels.Mono);
    sei = se.CreateInstance();
    sei.Volume = 1.0f;
    sei.Pitch = (float)Frequency.Value;
    sei.Play();
}

private void StopPlayback()
{
    if (sei != null)
    {
        sei.Stop();
    }
}

```

You can take this one step further by persisting the recorded sound to a file in isolated storage. You can see this at work in the *SoundFx_Persist* solution in the sample code. To persist the sound, you can add a couple of extra App Bar buttons for *Save* and *Load*. To save the data, simply write out the raw audio data by using the isolated storage APIs. This example uses a .wav file extension because the data is in fact PCM wave data. However, this is not a WAV file in the normal sense, because it is missing the header information that describes the file format, sample rate, channels, and so on.

```

private const string soundFile = "SoundFx.wav";

private void appBarSave_Click(object sender, EventArgs e)
{
    using (IsolatedStorageFile storage =
        IsolatedStorageFile.GetUserStoreForApplication())
    {
        using (IsolatedStorageFileStream stream =
            storage.OpenFile(soundFile, FileMode.Create, FileAccess.Write))
        {
            stream.Write(soundBuffer, 0, soundBuffer.Length);
        }
    }
}

```

To load the persisted file from disk, load the raw stream data back into your in-memory byte array, ready for playback.

```
private void appBarLoad_Click(object sender, EventArgs e)
{
    using (IsolatedStorageFile storage =
        IsolatedStorageFile.GetUserStoreForApplication())
    {
        using (IsolatedStorageFileStream stream =
            storage.OpenFile(soundFile, FileMode.Open, FileAccess.Read))
        {
            byte[] buffer = new byte[stream.Length];
            stream.Read(buffer, 0, buffer.Length);

            if (ms != null)
            {
                ms.Close();
            }
            ms = new MemoryStream();
            ms.Write(buffer, 0, buffer.Length);
        }
    }
}
```

You've seen already that you can use the *SoundEffect* class to load a conventional WAV file (including header) from disk. There's no support in *SoundEffect*—or indeed any other Silverlight or XNA classes—for saving WAV files with header information. This is not generally a problem on Windows Phone, because if the same application is both recording the data and playing it back, then it can precisely control the file contents without the need for a descriptive header. On the other hand, if you need to record audio on the phone, and then transmit it externally (for example, via a web service) to a consuming user or application that is using a different device (perhaps a PC, not a phone at all), then you need to save a descriptive header in the file along with the audio data.



More Info One solution to this is the NAudio library. NAudio is an open-source Microsoft .NET audio and MIDI library that contains a wide range of useful audio-related classes intended to speed development of audio-based managed applications. NAudio is licensed under the Ms-PL, which means that you can use it in whatever project you like, including commercial projects. It is available at <http://naudio.codeplex.com/>.

The *DynamicSoundEffectInstance* Class

So far, we've used the *SoundEffect* and *SoundEffectInstance* classes to play back audio streams, either from static audio content or from dynamic microphone input. The *DynamicSoundEffectInstance* is derived from *SoundEffectInstance*. The critical difference is that it exposes a *BufferNeeded* event. This is raised when it needs more audio data to play back. You can provide the audio data from static files or from dynamic microphone input; however, the main strength of this feature is that you can manipulate or compute the audio data before you provide it. Typically, you would modify source data, or even compute the data entirely from scratch.

The following example does just that—it provides a simple sound based on a sine wave. But before you get started, here’s a little background information that might be helpful. Sound is the result of a vibrating object that creates pressure oscillations—variations in pressure over time—in the air. A variation over time is modeled in mathematical terms as a wave. A wave can be represented by a formula that governs how the amplitude (or height) of the signal varies over time and the frequency of the oscillations. Given two otherwise identical waves, if one has higher amplitude it will be louder; if one has greater frequency it will have a higher pitch. A wave is continuous, but you need to end up with a buffer full of discrete items of audio data, where each datapoint is a value that represents a sample along the wave. This behavior is explored in the *TestDynamicSounds* solution in the sample code.

First, you need to declare fields for the *DynamicSoundEffectInstance*, a sample rate set to the maximum achievable on the device (48,000), and a buffer to hold the sound data. The sample count is computed as the maximum sample rate divided by the sample size (16 bits). From this, you can calculate the required buffer size as twice this value. For the purposes of this example, set the frequency to an arbitrary value of 100.

```
private DynamicSoundEffectInstance dynamicSound;
private const int sampleRate = 48000;
private const int sampleCount = sampleRate / 16;
private byte[] soundBuffer = new byte[sampleCount * 2];
private int totalTime = 0;
private double frequency = 100;
```

At a suitable early point—for example, in the *MainPage* constructor—you would set up your preferred method for pumping the XNA message queue. You want to initialize the *DynamicSoundEffectInstance* early on, but the catch is that the constructor is too early, because you won’t yet have started pumping the XNA message queue. One solution is to hook up the *Loaded* event on the page and do your initialization of the XNA types there, but there is a possible race condition with that approach. The simplest approach is to just pump the XNA message queue first, before performing initialization. Apart from the timing aspect, the key functional requirement is to hook up the *Buffer Needed* event. This will be raised every time the audio pipeline needs input data.

```
public MainPage()
{
    InitializeComponent();

    timer = new DispatcherTimer();
    timer.Interval = TimeSpan.FromMilliseconds(33);
    timer.Tick += delegate { try { FrameworkDispatcher.Update(); } catch { } };
    timer.Start();
    FrameworkDispatcher.Update();

    dynamicSound = new DynamicSoundEffectInstance(sampleRate, AudioChannels.Mono);
    dynamicSound.BufferNeeded += dynamicSound_BufferNeeded;
    dynamicSound.Play();
}
```

In the handler for the *BufferNeeded* event, the task is to fill in the byte array of sound data. In this example, you fill it with a simple sine wave. The basic formula for a sine wave as a function of time is as follows:

$$y(t) = A \cdot \sin(\omega t + \varphi)$$

Where:

- **A = amplitude** The peak deviation of the function from its center position (loudness).
- **ω = frequency** How many oscillations occur per unit of time (pitch)
- **φ = phase** Where in the cycle the oscillation begins

In this example, for the sake of simplicity, you can default the amplitude to 1 (parity with the volume on the device), and the phase to be zero (oscillation starts at the beginning of the cycle). You loop through the whole buffer, 2 bytes (that is, 16 bits: one sample) at a time. For each sample, compute the floating-point value of the sine wave and convert it to a short (16 bits). The double value computed from the sine wave formula is in the range -1 to 1, so you multiply by the *MaxValue* for a short in order to get a short equivalent of this.

Then, you need to store the short as 2 bytes. The low-order byte of the short is stored as an element in the sample array, and then the high-order byte is stored in the next element. Fill the second byte with the low-order byte of the short by bit-shifting 8 bits to the right. Finally, you submit the newly filled buffer to the *DynamicSoundEffectInstance* so that it can play it back.

```
private void dynamicSound_BufferNeeded(object sender, EventArgs e)
{
    for (int i = 0; i < sampleCount - 1; i += 2)
    {
        double time = (double)totalTime / (double)sampleRate;
        short sample =
            (short)(Math.Sin(2 * Math.PI * frequency * time) * (double)short.MaxValue);

        soundBuffer[i] = (byte)sample;
        soundBuffer[i + 1] = (byte)(sample >> 8);
        totalTime += 2;
    }
    dynamicSound.SubmitBuffer(soundBuffer);
}
```

The result will be a continuously oscillating tone. Figure 10-6 shows a variation on this application (*TestDynamicSounds_Controls* in the sample code), which includes an App Bar *Button* to start/stop the playback, and a *Slider* to control the frequency of the wave.

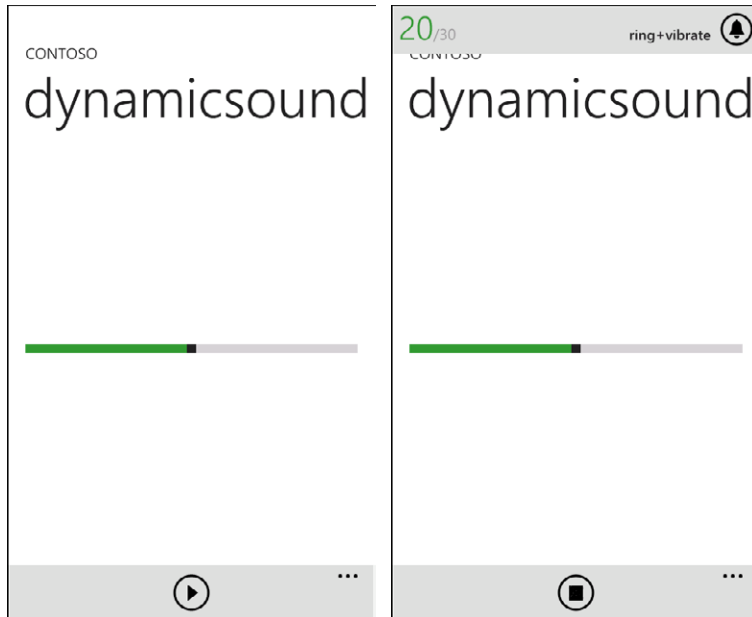


FIGURE 10-6 You can use *DynamicSoundEffectInstance* to manipulate audio data before playback.

In the XAML, you need to define a *Slider*. Give it a range from 1.0 to 1000.0, and set the initial position at halfway along the range, as demonstrated in the following:

```
<Slider
    Grid.Row="1" Margin="12,0,12,0"
    x:Name="Frequency" Minimum="1.0" Maximum="1000.0" Value="500.0" />
```

The implementation of the *BufferNeeded* event handler is changed slightly to use the *Slider* value instead of the fixed frequency value.

```
short sample =
    (short)(Math.Sin(2 * Math.PI * Frequency.Value * time)
        * (double)short.MaxValue);
```

The only other work is to respond to button *Click* events to start and stop the playback.

```
private void appBarPlay_Click(object sender, EventArgs e)
{
    if (isPlaying)
    {
        dynamicSound.Stop();
        appBarPlay.IconUri = new Uri("/Images/play.png", UriKind.Relative);
        appBarPlay.Text = "play";
        isPlaying = false;
    }
    else
    {
        dynamicSound.Play();
        appBarPlay.IconUri = new Uri("/Images/stop.png", UriKind.Relative);
    }
}
```



```

        appBarPlay.Text = "stop";
        isPlaying = true;
    }
}

```

When this application runs, the user can manipulate the slider to control the data that's fed into the playback buffer. Because you've tied the amplitude to the volume on the device, the user can change the volume of the playback by invoking the universal volume control (UVC) (see Figure 10-6). On the emulator, this is invoked by pressing either F9 or F10 while audio playback is ongoing; press F9 to increase the volume and F10 to decrease it. On the device, this is invoked by the hardware volume controls.

Music and Videos Hub

The Music and Videos Hub on Windows Phone is a centralized location for accessing the phone's music and videos library. Figure 10-7 shows an application that integrates with the Music and Videos Hub to fetch a list of all songs in the library and render them in a *ListBox*. This is the *TestMediaHub* solution in the sample code. When the user selects an item from the *ListBox*, the application fetches the selected song's album art and presents buttons with which the user can play/pause the selected song.

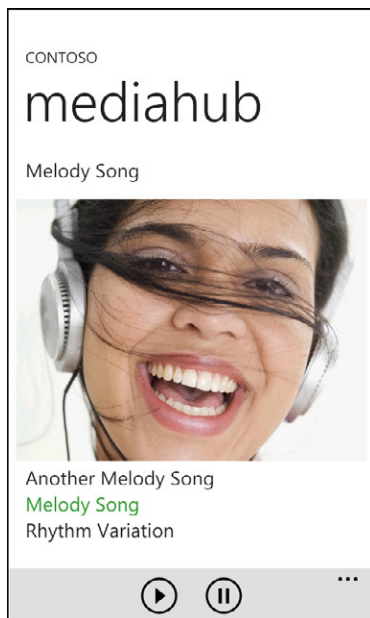


FIGURE 10-7 Your application can integrate with the Music and Videos Hub on the phone.

In the *MainPage* class, declare fields for the *MediaLibrary* itself and for the current *Song*. As always, you need to ensure that you pump the XNA message queue. Next, initialize the *MediaLibrary* field and set the collection of *Songs* to be the *ItemsSource* on your *ListBox*. In the XAML, data bind the *Text* property on the *ListBox* items to the *Name* property of each *Song*.

```
private MediaLibrary library;
private Song currentSong;

public MainPage()
{
    InitializeComponent();

    Play = ApplicationBar.Buttons[0] as ApplicationBarIconButton;
    Pause = ApplicationBar.Buttons[1] as ApplicationBarIconButton;

    DispatcherTimer dt = new DispatcherTimer();
    dt.Interval = TimeSpan.FromMilliseconds(33);
    dt.Tick += delegate { FrameworkDispatcher.Update(); };
    dt.Start();

    library = new MediaLibrary();
    HistoryList.ItemsSource = library.Songs;
}
```

You can test this on the emulator; however, it has only three songs, which won't yield very thorough results. Conversely, if you test on a physical device, you would want one that has a representative number of songs. If there are very many songs on the device, then initializing the list could be slow—in this case you could restrict the test to perhaps the first album by using this syntax:

```
HistoryList.ItemsSource = library.Albums.First().Songs;
```

The application's play and pause operations are more or less self-explanatory, invoking the *MediaPlayer Play* or *Pause* methods.

```
private void Play_Click(object sender, EventArgs e)
{
    MediaPlayer.Play(currentSong);
    Pause.IsEnabled = true;
    Play.IsEnabled = false;
}

private void Pause_Click(object sender, EventArgs e)
{
    MediaPlayer.Pause();
    Play.IsEnabled = true;
}
```

The interesting work is done in the *SelectionChanged* handler for the *ListBox*. Here, you fetch the currently selected item and search for it in the *MediaLibrary* collection. When you find the matching *Song*, fetch its album art to render in an *Image* control in the application's UI. If you can't find any corresponding album art, use a default image built in to the application.

```
private void HistoryList_SelectionChanged(object sender, SelectionChangedEventArgs e)
```

```

{    if (HistoryList.SelectedIndex == -1)
    {
        return;
    }

    currentSong = HistoryList.SelectedItem as Song;
    if (currentSong != null)
    {
        Play.IsEnabled = true;

        Stream albumArtStream = currentSong.Album.GetAlbumArt();
        if (albumArtStream == null)
        {
            StreamResourceInfo albumArtPlaceholder =
                App.GetResourceStream(new Uri(
                    "Images/AlbumArtPlaceholder.jpg", UriKind.Relative));
            albumArtStream = albumArtPlaceholder.Stream;
        }
        BitmapImage albumArtImage = new BitmapImage();
        albumArtImage.SetSource(albumArtStream);
        MediaImage.Source = albumArtImage;    }
}

```

The FM Tuner

All Windows Phone 7 devices ship with an FM radio, covering worldwide FM bands between 76 MHz and 108 MHz. The valid ranges depend on the region selected. The *FMRadio* class in the *Microsoft.Devices* namespace exposes properties for the region, frequency, power mode (on or off), and signal strength. Figure 10-8 shows an application that exercises the *FMRadio* class (the *TestRadio* solution in the sample code).

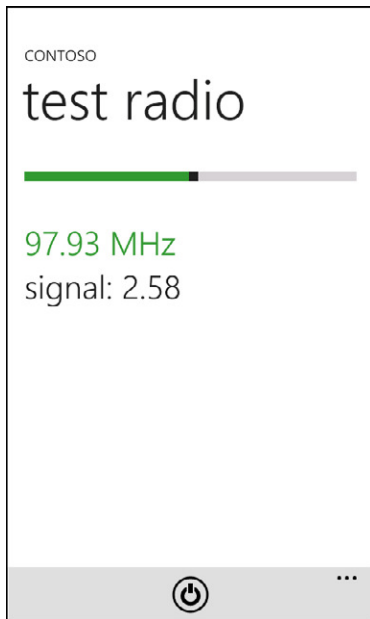


FIGURE 10-8 The *FMRadio* class provides programmatic access to the radio tuner on the phone.

In the *MainPage* class, cache the global radio instance and set the *CurrentRegion*. There are only three supported regions: United States, Europe, and Japan—which is slightly misleading, as the Europe setting matches all countries that are not the US or Japan. In a real application, you might want to use location services to match the current region that the user is actually in. Note also that the default is United States. You also need to set up a *DispatcherTimer*. Unlike other examples in this chapter, *FMRadio* does not use any XNA types, so this *DispatcherTimer* is not for the XNA message queue; instead, you use it to keep the signal strength display updated.

```
private FMRadio radio;

public MainPage()
{
    InitializeComponent();

    Power = ApplicationBar.Buttons[0] as ApplicationBarIconButton;

    DispatcherTimer timer = new DispatcherTimer();
    timer.Interval = TimeSpan.FromMilliseconds(1000);
    timer.Tick += new EventHandler(timer_Tick);
    timer.Start();

    radio = FMRadio.Instance;
    radio.CurrentRegion = RadioRegion.UnitedStates;
}
```

In the *Tick* event handler for the timer, update the signal strength display from the corresponding *FMRadio* property.

```
void timer_Tick(object sender, EventArgs e)
{
    if (radio != null && radio.PowerMode == RadioPowerMode.On)
    {
        Signal.Text = String.Format("signal: {0:F2}", radio.SignalStrength);
    }
}
```

In the *Click* event handler for the App Bar Button, toggle the *PowerMode* on/off, set the App Bar *Button Text* to match, and set a default string into the signal strength display if you're turning the radio off.

```
private void Power_Click(object sender, EventArgs e)
{
    if (radio.PowerMode == RadioPowerMode.On)
    {
        radio.PowerMode = RadioPowerMode.Off;
        Power.Text = "off";
        Signal.Text = "(no signal)";
    }
    else
    {
        radio.PowerMode = RadioPowerMode.On;
        Power.Text = "on";
    }
}
```

Finally, track the *ValueChanged* event on the *Slider* to update the *Frequency* property of the radio as well as the frequency display text.

```
private void Frequency_ValueChanged(
    object sender, RoutedPropertyChangedEventArgs<double> e)
{
    if (radio != null)
    {
        radio.Frequency = Frequency.Value;
        FrequencyValue.Text = String.Format("{0:F2} MHz", Frequency.Value);
    }
}
```

Note that although most of the *FMRadio* class functionality is operational on the emulator, there will be no signal. Also, on the physical device, you should plug in the headphones because the radio uses these as an antenna. Both tuning and powering up the radio can take up to a second before the *FMRadio* class is responsive to other calls. So, for example, setting *PowerMode=RadioPowerMode.On* and then immediately checking *SignalStrength* will generally result in a zero value return.

Summary

In this chapter, you saw how the application platform provides four broad categories of API support for building audio and video features into your application. There are three main classes for media playback at varying levels of flexibility. Audio input via the microphone and low-level manipulation of audio data is enabled through a second set of classes. Integration with the phone's media hub is enabled through the *XNA MediaLibrary* class. And finally, radio tuner support is provided through the *FMRadio* class. With judicious use of these APIs, you can easily build a very compelling media-focused user experience into your application.

Web and Cloud

The Windows Phone application platform provides some basic support for connecting to the web via two core classes and their supporting types: *WebClient* and *HttpWebRequest*. It also provides a modified version of the Microsoft Silverlight *WebBrowser* control for rendering HTML web pages. Beyond these, many of the standard techniques for connecting to web services, including Windows Communication Foundation (WCF) data services, are also directly supported on the phone. This chapter will examine each of these techniques, and also look at additional support that is available for connecting to Microsoft Bing Maps, Microsoft Windows Azure, bitly, Microsoft Windows Live, Microsoft SkyDrive, and Facebook. You should read this chapter in conjunction with Chapter 13, “Security,” which discusses the security aspects of web connectivity.

The *WebClient* Class

The *WebClient* class is part of the standard Microsoft .NET base class library (BCL). With it, you can download or upload web content. The version that is available to you in a Silverlight application is more constrained than the standard desktop Common Language Runtime (CLR) version, mainly by restricting you to asynchronous calls. In general, all web calls on the phone are asynchronous. To use the Silverlight *WebClient* class, you need to:

- Create an instance of *WebClient*, typically as a class-level field (because the asynchronous model means that you’ll most likely need to use it across more than one method). Alternatively, you can use inline lambdas.
- Wire up either the *DownloadStringCompleted* event or the *OpenReadCompleted* event (or both), implementing the event handlers to process the *Result* data returned from the service call.
- Call the *DownloadStringAsync* or *OpenReadAsync* method, to start fetching data from the web.

WebClient: The *DownloadStringAsync* Method

If you know that the target URL specifies content that can be represented as a string, you can use *WebClient.DownloadStringAsync*. For other content, you can use *WebClient.OpenReadAsync*. Figure 11-1 shows a phone application (the *SimpleWebClient* solution in the sample code) that uses the *WebClient* to retrieve arbitrary web pages and render the resulting HTML text in a *TextBlock*.

webclient

url

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" >
<head> <title>App Hub - windows phone and xbox live indie games development</title> <meta http-equiv="X-UA-Compatible" content="IE=8" /> <meta name="keywords" value="XNA, XNA Game Studio, XNA Game Studio Express, XNA Framework, XNA partners, XACT, PIX, DirectX, Direct3D, XNA games, XNA gaming, XNA GSE, game studio, XNA articles, XNA tutorials, game tutorials, starter kit, game engine, game physics, XNA Creators Club Online, App Hub, creators club C# express, C#, game creator, game development, game developer, game programming, game programmer, videogame, video game, ai, artificial
```

FIGURE 11-1 Use the *WebClient* class to download or upload web content.

```
private WebClient webClient;

public MainPage()
{
    InitializeComponent();
    webClient = new WebClient();
    webClient.DownloadStringCompleted +=
        webClient_DownloadStringCompleted;
}

private void getPageButton_Click(object sender, RoutedEventArgs e)
{
    webClient.DownloadStringAsync(new Uri(urlText.Text));
}

private void webClient_DownloadStringCompleted(
    object sender, DownloadStringCompletedEventArgs e)
{
    if (e.Error == null)
    {
        pageText.Text = e.Result;
    }
    else
    {
        pageText.Text = e.Error.ToString();
    }
}
```


Although *WebClient* operations (such as *DownloadStringAsync*) run on a background thread, the events generated (such as *DownloadStringCompleted*) are raised on the UI thread. An advantage of this is that there's no need to dispatch any work back to the UI thread in this scenario. A disadvantage is that you're inevitably doing work on the UI thread, and you might prefer to avoid or minimize this for performance/responsiveness reasons.

WebClient: The *OpenReadAsync* Method

If you're retrieving data that's not all strings, you can use *OpenReadAsync* instead. Here's an example (the *AvatarWebClient* solution in the sample code) that fetches user-specified avatar images from Microsoft Xbox Live, as shown in Figure 11-2. Each time you fetch an avatar, you add it to the *ListBox*.



FIGURE 11-2 Use *OpenReadAsync* for retrieving non-string data such as these avatars.

In the XAML, apart from the *TextBlock*, *TextBox*, and *Button*, there is a *ListBox* whose items are data-bound *Image* controls:

```
<StackPanel x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <StackPanel Orientation="Horizontal">
        <TextBlock Text="gamertag: " Margin="12,20,0,12"
            FontSize="{StaticResource PhoneFontSizeMedium}" />
        <TextBox Name="gamerTagText" Text="something" Margin="0"
            Width="260" Height="72" VerticalAlignment="Top" />
        <Button Content="get" Height="72" Margin="-10,0,0,0"
            Name="getAvatarButton" VerticalAlignment="Top"
            Width="95" Click="getAvatarButton_Click" />
    </StackPanel>
```

```

<ListBox Name="avatarList" Height="530" HorizontalAlignment="Center">
    <ListBox.ItemTemplate>
        <DataTemplate>
            <Image Source="{Binding}" Height="250"/>
        </DataTemplate>
    </ListBox.ItemTemplate>
</ListBox>
</StackPanel>

```

In the code-behind, there is an *ObservableCollection* of *ImageSource* items. When the user clicks the *Button*, you call *OpenReadAsync* on the *WebClient*, using the supplied gamertag to build a *Uri* to the corresponding Xbox Live avatar image. On *OpenReadCompleted*, fetch the resulting image and add it to the collection. The magic of data binding takes care of the rest.

```

public partial class MainPage : PhoneApplicationPage
{
    private WebClient webClient;
    private ObservableCollection<ImageSource> avatarImages;

    public MainPage()
    {
        InitializeComponent();
        avatarImages = new ObservableCollection<ImageSource>();
        avatarList.ItemsSource = avatarImages;
        webClient = new WebClient();
        webClient.OpenReadCompleted +=
            new OpenReadCompletedEventHandler(webClient_OpenReadCompleted);
    }

    private void getAvatarButton_Click(object sender, RoutedEventArgs e)
    {
        try
        {
            webClient.OpenReadAsync(new Uri(String.Format(
                "http://avatar.xboxlive.com/avatar/{0}/avatar-body.png",
                gamerTagText.Text)));
        }
        catch (Exception ex)
        {
            MessageBox.Show(ex.ToString());
        }
    }

    private void webClient_OpenReadCompleted(object sender, OpenReadCompletedEventArgs e)
    {
        if (!e.Cancelled && e.Error == null)
        {
            BitmapImage bmp = new BitmapImage();
            bmp.SetSource(e.Result);
            avatarImages.Add(bmp);
        }
    }
}

```

The *HttpWebRequest* Class

The code that follows (the *SimpleHttpReq* solution in the sample code) presents the *HttpWebRequest* (also part of the standard .NET BCL) equivalent of the previous simple *WebClient* example (you fetch text from the web and display it in a *TextBlock*). You first need to create an *HttpWebRequest* for the selected URL, and then invoke the asynchronous *BeginGetResponse* method. When you call *BeginGetResponse*, you need to pass in the *HttpWebRequest* object so that it is accessible in the asynchronous callback. When the callback is invoked, extract the *WebResponse* from the result, and then fetch the embedded data stream to populate the *TextBlock*. Note that unlike *WebClient*, the response callbacks for *HttpWebRequest* are invoked on a background thread; hence, the use of the page's *Dispatcher* to update the UI.

```
private void getPageButton_Click(object sender, RoutedEventArgs e)
{
    HttpWebRequest webRequest =
        (HttpWebRequest)HttpWebRequest.Create(new Uri(urlText.Text));
    webRequest.BeginGetResponse(httpRequestCallback, webRequest);
}

private void httpRequestCallback(IAsyncResult result)
{
    HttpWebRequest request = (HttpWebRequest)result.AsyncState;

    try
    {
        WebResponse response = request.EndGetResponse(result);
        Stream responseStream = response.GetResponseStream();
        StreamReader reader = new StreamReader(responseStream);

        Dispatcher.BeginInvoke(() =>
            { this.pageText.Text = reader.ReadToEnd(); });
    }
    catch (WebException ex)
    {
        Dispatcher.BeginInvoke(() => { pageText.Text = ex.Message; });
    }
}
```

WebBrowser Control

The *WebBrowser* control available for Windows Phone is based on the desktop Silverlight version. This control is not a full web browser; rather, it provides the core functionality to render HTML content, but it has no UI controls or chrome of its own. This is useful if you simply want to render local HTML (either static or dynamically-generated). It is also useful if you want to navigate to remote web pages,

and to provide your own custom chrome. The phone version of the *WebBrowser* control is slightly different from the desktop version in the following ways:

- It does not allow the use of ActiveX controls on web pages.
- It can invoke scripts loaded from any site. Unlike desktop Silverlight, it is not restricted to the same site as the XAP package.
- It can access isolated storage for the hosting application.
- It has the same cross-site restrictions for HTML loaded from the web, but not for HTML loaded from static or dynamic local content.
- It can be treated as a normal UI control that can have transforms and projections performed on it, and it participates in the Z-order of UI elements.

You can use the *WebBrowser* by adding the following line to your XAML:

```
<phone:WebBrowser Source="http://create.msdn.com" />
```

The result of this is shown in Figure 11-3, which is a screenshot from the *TestWbc* solution in the sample code. Zoom (pinch-and-stretch) and scroll manipulations work, as do any links on the page. Embedded ActiveX controls, including Silverlight, will not work. By default, scripts will not work either, although this can be changed.



FIGURE 11-3 Use the *WebBrowser* control to render HTML content.

If you don't specify a name for the *WebBrowser* control in your XAML, and then run the Capabilities Detection tool, this will not report any required capabilities. If you then go ahead and remove the boilerplate capabilities from your manifest, the application will fail at runtime. In fact, you need both the *ID_CAP_WEBBROWSERCOMPONENT* and *ID_CAP_NETWORKING* capabilities. To be sure, simply specify a name for the control in XAML, as follows:

```
<phone:WebBrowser x:Name="wbc" Source="http://create.msdn.com" />
```

Without chrome, the previous example offers no way for the user to navigate forward and back. You could add suitable UI for this, if you need it (App Bar buttons would be most appropriate here). You would also have to add your own implementation of the navigation history, because the *Web Browser* control does not provide any back/forward navigation functionality. In addition to the history list itself, you would also need to keep track of where the current page is in the history list and whether the current navigation is to a page in the history list or not. However, note that the *Web Browser* control is intended to be used for simple rendering of HTML content. Although you *could* do a lot of work to replicate the behavior of a full browser, this is not encouraged. Instead, if you need browser functionality, you can use the *WebBrowserTask*, as is discussed in Chapter 9, "Phone Services."

Silverlight and Javascript

It is possible to interoperate between JavaScript on an HTML page and your application code, bi-directionally. There are three pieces to this:

- If you want any script on the page to run, you need to set the *WebBrowser.IsScriptEnabled* property to *true*.
- To receive data from scripts on the HTML page, you need to hook up the *WebBrowser.Script Notify* event, and retrieve the data in the *NotifyEventArgs* parameter.
- To invoke script on the HTML page, you can invoke the *WebBrowser.InvokeScript* method, specifying the name of the function to invoke, and passing any input data.

You can work with script on any suitable HTML page, local or remote, although you would almost certainly restrict your application to working with known pages and scripts—specifically, those that you control. For this reason, the pages are likely to be either on a server you control or local to the application. Figure 11-4 shows an application (*WbcScript* in the sample code) that loads a local HTML page into a *WebBrowser* control.

CONTOSO

wbc script

input

output

hello world

script to app

app to script

FIGURE 11-4 You can interoperate between a phone application and web page script.

To make it more obvious which parts of this are in the web page and which parts are in the client application, there is a border around the *WebBrowser* control. The HTML source for the page is listed in the code sample that follows. There are two JavaScript functions:

- **OutputFromApp** This is invoked within the application. The function takes in a text parameter and sets it into the *textFromApp* DIV on the HTML page.
- **InputToApp** This is invoked when the user taps the Send button on the HTML page. The function takes the text from the *textToApp* input textbox on the page, and then sends it to anything listening to the external *Notify* event (an event that you hook up in the client phone application).

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head >
  <title>SimplePage</title>
  <meta name="mobileoptimized" content="480" />
  <script type="text/javascript">
    function OutputFromApp(text) {
      textFromApp.innerHTML = text;
    }

    function InputToApp() {
      window.external.Notify(textToApp.value);
    }
  </script>
</head>
<body>
```

```

<div style="font-family: 'Segoe WP Semibold'; font-size: x-large">
    input
    <input id="textToApp" type="text"
        style="font-family: 'Segoe WP Semibold'; font-size: large; width: 292px;" />
    <input id="sendButton" type="button" value="send" onclick="InputToApp()"
        style="font-family: 'Segoe WP Semibold'; font-size: large; width: 90px;" />
    <br />
    <br />
    output
    <div id="textFromApp"
        style="font-family: 'Segoe WP Semibold'; font-size: x-large; color: #008000;" />
</div>

</body>
</html>

```

Note the *mobileoptimized* tag: you can use this to specify an integer value that corresponds to the intended display width of the screen. In the case of Windows Phone, this should always be 480. The browser will honor this value, and force the page into a single-column layout at the specified width. The tag is also used by search engines to determine whether the page is mobile-optimized.

In the client phone application, enable scripting and hook up the *ScriptNotify* event. Then, to load the local HTML file, you fetch it from the XAP (using *GetResourceStream*, although it is built as *Content*) and read it into a string. You can then use the *WebBrowser.NavigateToString* method to render the page. When the user taps Send on the web page, the page JavaScript invokes *Notify*, which results in a callback to the *ScriptNotify* event handler. In this handler, you fetch the incoming data from the JavaScript function and set it into a *TextBox*. Finally, when the user taps the Invoke button in the client phone application, you call *InvokeScript* to invoke one of the JavaScript functions on the page and pass in some text data.

```

private void MainPage_Loaded(object sender, RoutedEventArgs e)
{
    wbc.IsScriptEnabled = true;
    wbc.ScriptNotify += new EventHandler<NotifyEventArgs>(wbc_ScriptNotify);

    StreamResourceInfo sri =
        App.GetResourceStream(new Uri("SimplePage.htm", UriKind.Relative));
    using (StreamReader reader = new StreamReader(sri.Stream))
    {
        String html = reader.ReadToEnd();
        wbc.NavigateToString(html);
    }
}

private void wbc_ScriptNotify(object sender, NotifyEventArgs e)
{
    scriptToApp.Text = e.Value;
}

private void InvokeScript_Click(object sender, RoutedEventArgs e)
{
    wbc.InvokeScript("OutputFromApp", appToScript.Text);
}

```

Web Services

Windows Phone can consume web services, including Microsoft ASP.NET and WCF. The following example uses a simple WCF service that provides facts about the Magical Manatee, a creature in Brazilian folklore with mystical powers. The *ServiceContract* is listed in the following code. It defines just one method, *GetFact*, which returns a string.

```
[ServiceContract]
public interface IMagicalManateeFacts
{
    [OperationContract]
    string GetFact();
}
```

The code that follows (the *WCF Simple\MagicalManateeService* solution in the sample code) presents the server-side implementation. The service contains a static array of strings, and for each *GetFact* call, it simply returns a (pseudo-)random string from the array.

```
public class MagicalManateeFacts : IMagicalManateeFacts
{
    public string GetFact()
    {
        int len = FactsStrings.Length;
        Random rand = new Random();
        int num = rand.Next(len - 1);
        return FactsStrings[num];
    }

    private static string[] FactsStrings =
    {
        @"The Magical Manatee uses Tabasco Sauce for eye drops.",
        @"The Magical Manatee can get Blackjack with just one card.",
        ... etc
    };
}
```

The service must be hosted in some executable process. In this example, the host is a console application (see the sample code that accompanies this book for details). When the application starts, it creates a *ServiceHost* for the *MagicalManatee* service component, specifying the URL endpoint where clients can reach it. You then open the *ServiceHost* to start listening for client calls; you continue to listen until the user presses a key on the console to close the host and terminate the application.

The phone client has a single *ListBox* and an App Bar button, as shown in Figure 11-5. This is the *WCF Simple\MagicalManateeClient* solution in the sample code.

In creating the client application, you can generate a client-side proxy to the WCF service in the usual way: ensure that the service is running, select Add Service Reference, and then point to the service URL, as shown in Figure 11-6. This example uses localhost with port 8001, but for a production system, you would obviously use a real address.

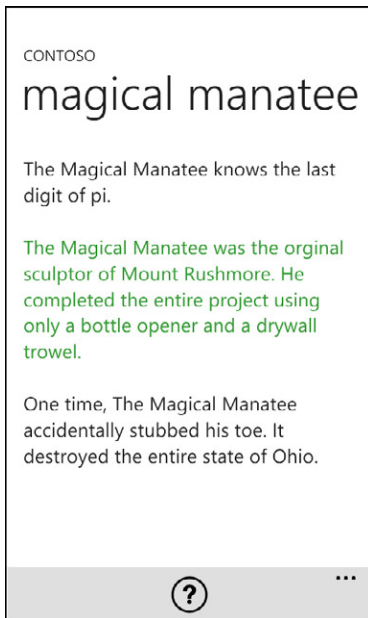


FIGURE 11-5 It's common for a phone client application to access web services.

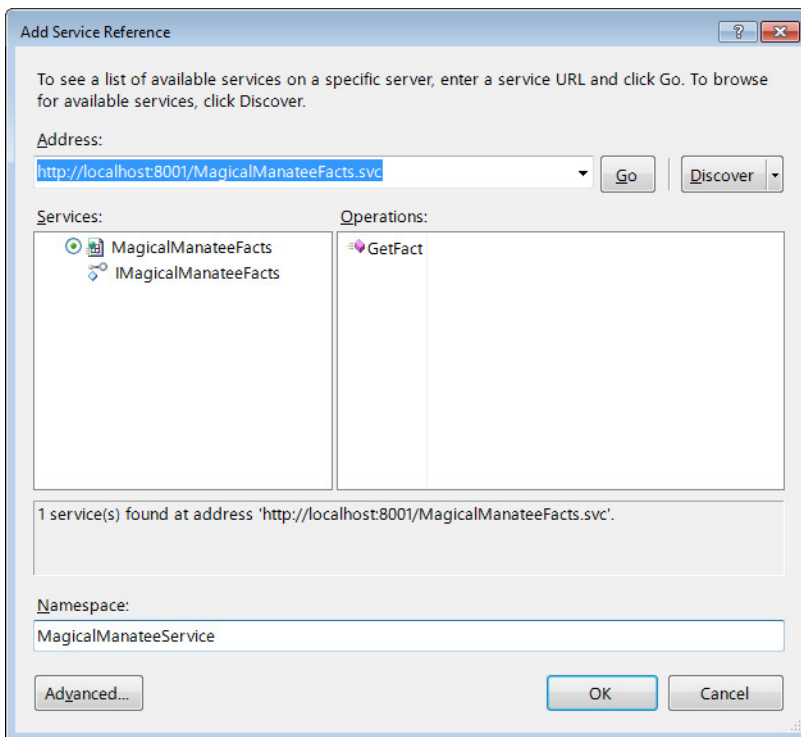


FIGURE 11-6 Use the Add Service Reference Wizard to generate proxy code for a web service.

This will generate a proxy as well as a `ServiceReferences.ClientConfig` file. Note that the binding is set to *basicHttpBinding*. You cannot change this for a phone application; this is a limitation of the .NET Compact Framework.

```
<configuration>
  <system.serviceModel>
    <bindings>
      <basicHttpBinding>
        <binding name="BasicHttpBinding_IMagicalManateeFacts"
          maxBufferSize="2147483647"
          maxReceivedMessageSize="2147483647">
          <security mode="None" />
        </binding>
      </basicHttpBinding>
    </bindings>
    <client>
      <endpoint address="http://localhost:8001/MagicalManateeFacts.svc"
        binding="basicHttpBinding"
        bindingConfiguration="BasicHttpBinding_IMagicalManateeFacts"
        contract="ChuckClient.MagicalManateeService.IMagicalManateeFacts"
        name="BasicHttpBinding_IMagicalManateeFacts" />
    </client>
  </system.serviceModel>
</configuration>
```

Now you can implement the App Bar button *Click* handler to invoke the service.

```
private MagicalManateeFactsClient client;
private static bool isAccentBrush;

private void appBarGetFact_Click(object sender, EventArgs e)
{
    if (client == null)
    {
        client = new MagicalManateeFactsClient();
        client.GetFactCompleted += chuckClient_GetFactCompleted;
    }
    client.GetFactAsync();
}
```

All phone application calls to web services are asynchronous—only asynchronous client proxies are generated, and you need to wire up the *GetFactCompleted* event and call *GetFactAsync*. In this example, you implement the *GetFactCompleted* handler to add a new *TextBlock* for each Magical Manatee fact received, and then add it to the *ListBox* (via the page *Dispatcher*, to ensure that it's executed on the UI thread). This example alternates the brush used for the *TextBlock Foreground* between the standard foreground brush and a brush based on the current accent color.

```
private void client_GetFactCompleted(object sender, GetFactCompletedEventArgs e)
{
    String result = String.Empty;
    if (e.Error == null)
    {
        result += e.Result + Environment.NewLine;
    }
}
```

```

else
{
    result = e.Error.ToString() + Environment.NewLine;
}

Dispatcher.BeginInvoke(delegate
{
    TextBlock tb = new TextBlock();
    tb.FontSize =
        (double)App.Current.Resources["PhoneFontSizeMediumLarge"];
    tb.TextWrapping = TextWrapping.Wrap;

    if (isAccentBrush)
    {
        tb.Foreground =
            (Brush)App.Current.Resources["PhoneAccentBrush"];
    }
    else
    {
        tb.Foreground =
            (Brush)App.Current.Resources["PhoneForegroundBrush"];
    }
    isAccentBrush = !isAccentBrush;

    tb.Text = result;
    factsList.Items.Add(tb);
});
}

```

WCF Data Services

A data service is an HTTP-based web service that exposes server-side data by using the Open Data (OData) web protocol. You can use WCF Data Services to publish data from a server application by using a REST-based interface, in either XML or JavaScript Object Notation (JSON) format. This is particularly interesting for mobile clients, because OData formats involve significantly less overhead than traditional SOAP formats.

The OData Client and XML Data

Figure 11-7 illustrates the high-level architecture of a generic WCF data service client/server solution. In the server application, you can define an Entity Data Model (EDM). This is a set of classes that represent some backing data, typically tables and rows in a database. Then, you would define a WCF data service to consume the EDM and expose your selected data to the web. On the client, you would typically generate a service proxy and data-bind the proxied entities to your UI.

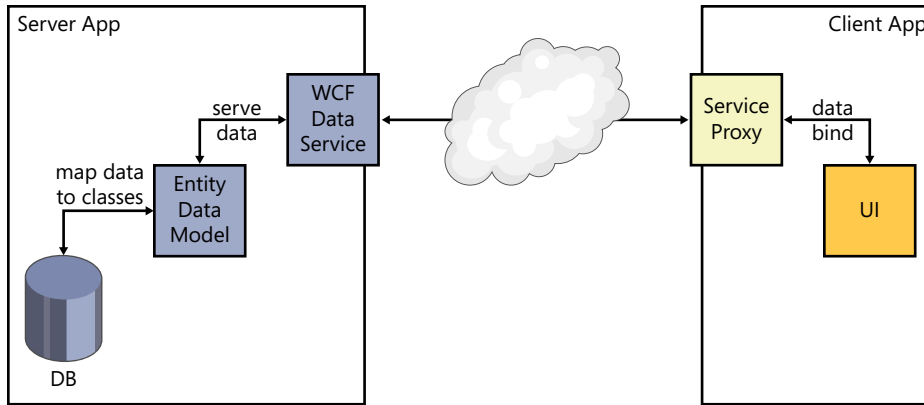


FIGURE 11-7 At a high-level, there are five major components in a WCF data service client/server solution.

Here are the basic steps for building an OData solution with a Windows Phone client, which are discussed in detail, later on:

Server

1. Create an ASP.NET application (by using either the regular ASP.NET Visual Studio project template or the empty ASP.NET template).
2. Using Microsoft Visual Studio's Add Item project wizard, add an ADO.NET Entity Data Model item, mapped to the database tables/views that you want to expose.
3. Add a WCF Data Service item, and then connect it to your EDM.

Client

1. Run the server, and then run the DataSvcUtil tool to generate the client-side proxy.
2. Add the generated proxy classes to your Windows Phone client project.
3. In the client XAML, data-bind UI elements to the proxied data entities.
4. In the code-behind, (asynchronously) execute the proxy service queries to return the data collection.

The following example uses the *Customer* table in the *AdventureWorks* database, which is one of the Microsoft SQL Server sample databases.



More Info You can get the Microsoft SQL Server sample databases as free downloads from codeplex at <http://msftdbprodsamples.codeplex.com/>.

On the server side, you have an ASP.NET Web Application with an ADO.NET Entity Data Model mapped to the *Customer* table, plus a WCF Data Service. This is the *WCF Data Services\Customer WebApp* solution in the sample code.

The Entity Data Model Wizard will generate *Entities* (*ObjectContext*) and *Customer* model (*Entity*) classes. The *Entities* class roughly corresponds to your database (collection of tables and views that you select in the wizard), whereas the *Entity* classes correspond to the individual tables and views, as shown in the following example:

```
public partial class AdventureWorksLT2008R2Entities : ObjectContext
{
    public ObjectSet<Customer> Customers
    {
        get
        {
            if ((_Customers == null))
            {
                _Customers = base.CreateObjectSet<Customer>("Customers");
            }
            return _Customers;
        }
    }
    private ObjectSet<Customer> _Customers;
    ... etc
}

[EdmEntityTypeAttribute(
    NamespaceName="AdventureWorksLT2008R2Model", Name="Customer")]
[Serializable()]
[DataContractAttribute(IsReference=true)]
public partial class Customer : EntityObject
{
    [EdmScalarPropertyAttribute(EntityKeyProperty=true, IsNullable=false)]
    [DataMemberAttribute()]
    public global::System.Int32 CustomerID
    {
        get
        {
            return _CustomerID;
        }
        set
        {
            if (_CustomerID != value)
            {
                OnCustomerIDChanging(value);
                ReportPropertyChanging("CustomerID");
                _CustomerID = StructuralObject.SetValidValue(value);
                ReportPropertyChanged("CustomerID");
                OnCustomerIDChanged();
            }
        }
    }
}
```

```

private global::System.Int32 _CustomerID;
partial void OnCustomerIDChanging(global::System.Int32 value);
partial void OnCustomerIDChanged();

[EdmScalarPropertyAttribute(EntityKeyProperty=false, IsNullable=true)]
[DataMemberAttribute()]
public global::System.String CompanyName
{
    ... etc
}
... etc
}

```

The WCF Data Service Wizard will generate a customer data service, as follows:

```

public class CustomerDataService : DataService< /* put your data source class name here */ >
{
    public static void InitializeService(DataServiceConfiguration config)
    {
        config.DataServiceBehavior.MaxProtocolVersion =
            DataServiceProtocolVersion.V2;
    }
}

```

You must update the data service class to associate it with the EDM and to specify what access to allow (read, write, append, delete, and so on) to which entities. The following example specifies all access to all entities:

```

public class CustomerDataService : DataService<AdventureWorksLT2008R2Entities>
{
    public static void InitializeService(DataServiceConfiguration config)
    {
        config.SetEntitySetAccessRule("*", EntitySetRights.All);
        config.DataServiceBehavior.MaxProtocolVersion =
            DataServiceProtocolVersion.V2;
    }
}

```

Build and test the server. Most browsers will decide to render the results as an RSS feed. To see the underlying data, you can select View Source. Alternatively (in Internet Explorer), click to Tools | Internet Options, click the Content tab, and then in the Feeds And Web Slices section, click Settings. Clear the Turn On Feed Reading View check box, as shown in Figure 11-8.

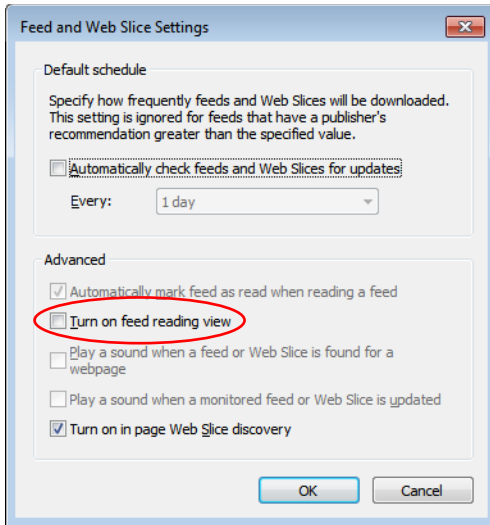


FIGURE 11-8 To see the data service data in XML, turn off feed reading view in Internet Explorer.

With this change, the browser will render the returned data as XML. The listing below shows just the first *Customer* in the list (Orlando Gee), returned by using the URL of our example service, that is, <http://localhost:8001/CustomerDataService.svc/Customers>.

```
<feed xml:base="http://localhost:8001/CustomerDataService.svc/" xmlns:d="http://schemas.
microsoft.com/ado/2007/08/dataservices" xmlns:m="http://schemas.microsoft.com/ado/2007/08/
dataservices/metadata" xmlns="http://www.w3.org/2005/Atom">
  <title type="text">Customers</title>
  <id>http://localhost:8001/CustomerDataService.svc/Customers</id>
  <link rel="self" title="Customers" href="Customers" />
  <entry>
    <id>http://localhost:8001/CustomerDataService.svc/Customers(1)</id>
    <link rel="edit" title="Customer" href="Customers(1)" />
    <category term="AdventureWorksLT2008R2Model.Customer"
      scheme="http://schemas.microsoft.com/ado/2007/08/dataservices/scheme" />
    <content type="application/xml">
      <m:properties>
        <d:CustomerID m:type="Edm.Int32">1</d:CustomerID>
        <d:NameStyle m:type="Edm.Boolean">>false</d:NameStyle>
        <d:FirstName>Orlando</d:FirstName>
        <d:LastName>Gee</d:LastName>
        <d:CompanyName>A Bike Store</d:CompanyName>
        <d:EmailAddress>orlando0@adventure-works.com</d:EmailAddress>
        <d:Phone>245-555-0173</d:Phone>
      </m:properties>
    </content>
  </entry>
  <entry>
    <id>http://localhost:8001/CustomerDataService.svc/Customers(2)</id>
    <!--etc -->
  </entry>
  <!--etc -->
</feed>
```

To consume this service in a Windows Phone app, you can't use the Visual Studio Add Service Reference approach because WCF Data Services are not yet supported in the Visual Studio Windows Phone application templates. Instead, you need to run the server, and then run the DataSvcUtil tool to generate the client-side proxy. Here's a suitable command line:

```
"%windir%\Microsoft.NET\Framework\v4.0.30319\DataSvcUtil.exe" /version:2.0 /  
dataservicecollection /language:CSharp /out:CustomerData.cs /uri:http://localhost:8001/  
CustomerDataService.svc
```

The `/language`, `/out`, and `/uri` switches are self-explanatory. You need the `/version` and `/dataservicecollection` switches to generate classes suitable for data binding.

Add the generated `CustomerData.cs` to the phone application project. You can see this at work in the *WCF Data Services\DataServiceClient(Simple)* solution in the sample code. Also, add a reference to the OData Client Library (`System.Services.Data.Client.dll`). Then, you can consume the data service proxy in your client code.

The current example defines a `ListBox` with an `ItemTemplate` which contains `TextBlock` controls that are data-bound to the service data, as shown in Figure 11-9.



FIGURE 11-9 All the hard work is in talking to the data service. After that, it's just a question of doing something useful with the data that's returned.

The following is the definition of the `ItemTemplate` and the `ListBox` in the XAML. Each `ListBox` item is made up of a two-column `Grid` with two `TextBlock` controls, one bound to the `CompanyName`, and the other bound to the `Phone` property.


```

<phone:PhoneApplicationPage.Resources>
    <DataTemplate x:Key="CustomerItemTemplate">
        <Grid>
            <Grid.ColumnDefinitions>
                <ColumnDefinition Width="270"/>
                <ColumnDefinition Width="*"/>
            </Grid.ColumnDefinitions>
            <TextBlock Grid.Column="0" Text="{Binding CompanyName}"
                Style="{StaticResource PhoneTextAccentStyle}"/>
            <TextBlock Grid.Column="1" Text="{Binding Phone}"/>
        </Grid>
    </DataTemplate>
</phone:PhoneApplicationPage.Resources>

<ListBox Name="customerList"
    ItemTemplate="{StaticResource CustomerItemTemplate}"/>

```

The code that follows is in the *MainPage* code-behind. You override *OnNavigatedTo* to call a custom method *ExecuteQuery*, which sets up the query (in this example, you query for all *Customers*) and makes an asynchronous call to execute the query. The callback retrieves the data results and dispatches them on the UI thread to the *ListBox*. Note that you're instantiating the entities object as part of class construction, but this might take a long time (that is, more than a few hundred milliseconds), in which case, you should defer construction to the *OnNavigatedTo* override to ensure that the application is as responsive as possible on startup. Also, in a production-strength application, you would typically check to see if invoking the query is actually necessary. Specifically, you could test to see if this is a fresh start or if the application is being resurrected after tombstoning. For scenarios in which you have already done this (and persisted some/all of the data to isolated storage), you can avoid the expensive server call.

```

public partial class MainPage : PhoneApplicationPage
{
    private AdventureWorksLT2008R2Entities entities =
        new AdventureWorksLT2008R2Entities(
            new Uri("http://localhost:8001/CustomerDataService.svc"));

    protected override void OnNavigatedTo(NavigationEventArgs e)
    {
        ExecuteQuery();
    }

    private void ExecuteQuery()
    {
        DataServiceQuery<Customer> query = entities.Customers;
        query.BeginExecute(Customers_Callback, query);
    }
}

```

```

private void Customers_Callback(IAsyncResult result)
{
    DataServiceQuery<Customer> query =
        (DataServiceQuery<Customer> )result.AsyncState;
    var customers = query.EndExecute(result);
    this.Dispatcher.BeginInvoke(
        () => { customerList.ItemsSource = customers; });
}
}

```

In the enhanced version shown in Figure 11-10 (the *WCF Data Services\DataServiceClient(Filterable)* solution in the sample code), a filter capability is provided. There's an additional *ListBox* that is data-bound to an array of strings to represent the alphabet. The user can scroll the list to find a specific letter; you then filter the query on the data service to return only those customers for which the *CompanyName* starts with that letter.



FIGURE 11-10 It's common to offer client-side filtering features for remote data services.

Here are the interesting changes. In the page, there is an array of strings, and a field to record the current filter string. You set the *ItemsSource* of the new *ListBox*, named *alphabetList*, to this array.

```

private String[] alphabet =
{
    "*", "A", "B", "C", "D", "E", "F", "G", "H",
    "I", "J", "K", "L", "M", "N", "O", "P", "Q",
    "R", "S", "T", "U", "V", "W", "X", "Y", "Z" };
public String filter = "";
private AdventureWorksLT2008R2Entities entities =
    new AdventureWorksLT2008R2Entities(
        new Uri("http://localhost:8001/CustomerDataService.svc"));

```

```

public MainPage()
{
    InitializeComponent();
    this.alphabetList.ItemsSource = alphabet;
}

```

You invoke the *ExecuteQuery* not only in *OnNavigatedTo* but also in the *SelectionChanged* handler for the alphabet *ListBox*. The *ExecuteQuery* is slightly more complex: the inner LINQ query has a where clause (which returns an *IQueryable<T>*, which you then have to cast to the *DataServiceQuery<T>* that you need to execute against the data service).

```

private void alphabet_SelectionChanged(object sender, SelectionChangedEventArgs e)
{
    filter = (String)this.alphabetList.SelectedItem;
    ExecuteQuery();
}

private void ExecuteQuery()
{
    DataServiceQuery<Customer> query = null;
    if (filter == "")
    {
        query = entities.Customers;
    }
    else
    {
        var innerQuery =
            from c in entities.Customers
            where c.CompanyName.StartsWith(filter)
            select c;
        query = (DataServiceQuery<Customer>)innerQuery;
    }
    query.BeginExecute(Customers_Callback, query);
}

```

JSON-Formatted Data

Data formatted by using JSON is significantly smaller than XML-formatted data for the same web service call. There's no proxy-generation tools support for JSON, but the code you need to write is simple enough. Instead of generating a proxy with the *DataSvcUtil* tool, you need to write a class manually that corresponds to the data received. You can see this at work in the *WCF Data Services\ DataServiceClient(JSON)* solution in the sample code.

Here's a class that maps to the *Customer* data. All mappable fields are set up as public properties (they need to be serializable). You only need to define the fields that you care about (in this case, *CompanyName* and *Phone*). Note that the data in a JSON-formatted payload starts with "d".

```

public class Customers
{
    public Customer[] d { get; set; }
}

```

```
public class Customer
{
    public String CompanyName { get; set; }
    public String Phone { get; set; }
}
```

The revised version of *ExecuteQuery* now has to create an *HttpRequest* manually and specify in the HTTP Accept header that you want to receive JSON-formatted data.

```
private void ExecuteQuery()
{
    HttpRequest request = (HttpRequest)HttpRequest.Create(
        "http://localhost:8001/CustomerDataService.svc/Customers");
    request.Accept = "application/json";
    request.BeginGetResponse(Customers_Callback, request);
}
```

The callback extracts the data stream from the response and uses the *DataContractJsonSerializer* (defined in *System.ServiceModel.Web.dll*) to deserialize it into a *Customers* object.

```
private void Customers_Callback(IAsyncResult result)
{
    HttpRequest request =
        (HttpRequest)result.AsyncState;
    HttpResponse response =
        (HttpResponse)request.EndGetResponse(result);
    DataContractJsonSerializer deserializer =
        new DataContractJsonSerializer(typeof(Customers));
    Stream responseStream = response.GetResponseStream();
    Customers customers =
        (Customers)deserializer.ReadObject(responseStream);

    this.Dispatcher.BeginInvoke(
        () => { customerList.ItemsSource = customers.d; });
}
```

Table 11-1 provides a detailed comparison of the data in XML format and JSON format for *Customers(1)*.

To achieve the same kind of filtering capability (wherein the app provides a filtered request to the server, which then returns only the filtered data) with a JSON-based client, you can use a filter as part of the request URI string. The code that follows is an enhanced version; most of the changes are the same as for the *DataServiceQuery* version. This is the *WCF Data Services\DataServiceClient(JSON-filterable)* solution in the sample code. The interesting differences are listed here—first, for convenience you declare a string that represents the unchanging part of the request URI.

```
private String requestBase =
    "http://localhost:8001/CustomerDataService.svc/Customers";
```

Then, in the *ExecuteQuery* method, you can append a suitable filter query to the end of this request URI.

```

private void ExecuteQuery()
{
    HttpWebRequest request = null;
    if (filter == "")
    {
        request = (HttpWebRequest)HttpWebRequest.Create(requestBase);
    }
    else
    {
        string requestString = String.Format(
            "{0}$filter=startswith(CompanyName,'{1}')" , requestBase, filter);
        request = (HttpWebRequest)HttpWebRequest.Create(requestString);
    }
    request.Accept = "application/json";
    request.BeginGetResponse(Customers_Callback, request);
}

```

TABLE 11-1 Comparision Between XML and JSON Format

XML	JSON
<pre> <?xml version="1.0" encoding="utf-8" standalone="yes" ?> - <entry xml:base="http://localhost:8001/ CustomerDataService.svc/" xmlns:d="http:// schemas.microsoft.com/ado/2007/08/dataservices" xmlns:m="http://schemas.microsoft.com/ado/2007/08/ dataservices/metadata" xmlns="http://www.w3.org/2005/ Atom"> <id>http://localhost:8001/CustomerDataService.svc/ Customers(1)</id> <title type="text" /> <updated>2011-03-14T02:14:00Z</updated> - <author> <name /> </author> <link rel="edit" title="Customer" href="Customers(1)" /> <category term="AdventureWorksLT2008R2Model. Customer" scheme="http://schemas.microsoft.com/ ado/2007/08/dataservices/scheme" /> - <content type="application/xml"> - <m:properties> <d:CustomerID m:type="Edm.Int32">1</d:CustomerID> <d:NameStyle m:type="Edm.Boolean">false</ d:NameStyle> <d:Title>Mr.</d:Title> <d:FirstName>Orlando</d:FirstName> <d:MiddleName>N.</d:MiddleName> <d:LastName>Gee</d:LastName> <d:Suffix m:null="true" /> <d:CompanyName>A Bike Store</d:CompanyName> <d:SalesPerson>adventure-works\pamela0 </d:SalesPerson> <d:EmailAddress>orlando0@adventure-works.com </d:EmailAddress> <d:Phone>245-555-0173</d:Phone> <d>PasswordHash>L/R1wxzp4w7RwmEgXX+/ A7cXaePEPcp+KwQh12fJL7w=</d>PasswordHash> <d>PasswordSalt>1KjXYs4=</d>PasswordSalt> <d:rowguid m:type="Edm.Guid">3f5ae95e-b87d-4aed- 95b4-c3797afcb74f</d:rowguid> <d:ModifiedDate m:type="Edm.DateTime">2005-08- 01T00:00:00</d:ModifiedDate> </m:properties> </content> </entry> </pre>	<pre> { "d" : [{ "__metadata": { "uri": "http://localhost:8001/ CustomerDataService.svc/Customers(1)", "type": "AdventureWorksLT2008R2Model.Customer"}, "CustomerID": 1, "NameStyle": false, "Title": "Mr.", "FirstName": "Orlando", "MiddleName": "N.", "LastName": "Gee", "Suffix": null, "CompanyName": "A Bike Store", "SalesPerson": "adventure-works\\ pamela0", "EmailAddress": "orlando0@adventure-works. com", "Phone": "245-555-0173", "PasswordHash": "L/R1wxzp4w7RwmEgXX+/A7cXaePEPcp+KwQh12fJL7w=", "PasswordSalt": "1KjXYs4=", "rowguid": "3f5ae95e- b87d-4aed-95b4-c3797afcb74f", "ModifiedDate": "\/ Date(1122854400000)\/" }] } </pre>

Bing Maps and Geolocation

The Bing Maps service exposes a range of APIs for use in a wide variety of application types. For Windows Phone, you would typically use the *Map* control and the Bing Maps web services, both of which are described in the following sections.

Using the Map Control

To use Bing Maps and Bing services, you need a Windows Live ID and a Bing Maps developer account, both of which are free and easy to obtain. Go to <http://www.bingmapsportal.com>, associate an account with your Live ID, and then select the option to create keys, as shown in Figure 11-11. You can create up to five keys, and for each one, you supply an arbitrary application name and URL (these don't have to bear any relation to your real application name/URL).

★ Create Key - Bing Maps Account Center

bing
Maps Account Center

Map APIs

- Update or view account details
- Create or view keys
- View my Bing Maps API usage

Map Apps

- Submit a map app
- View my map apps

Resources

- Bing Maps Platform
- Bing Maps SDKs
- Bing Maps Forums
- Bing Maps Blog
- Bing Map App SDK beta

Contacts

Get your answers at:

[Bing Maps Account Center Help](#)

For information or inquiries about pricing, licensing or volume licensing, contact:

maplic@microsoft.com

My keys

You can create up to five Bing Maps keys. You need a key to authenticate your Bing Maps application. If you need more than 5 keys, please contact mpnet@microsoft.com.

Create key

* Application name
Test

Application URL
<http://localhost>

* Application Type
Developer What's this?

Submit

Click [here](#) to view/download complete list of keys.

FIGURE 11-11 Use the Bing maps portal to create/view Bing account keys.

This will generate a key, which you should copy and save somewhere locally. For simplicity, you can paste it into your application code, typically as a field in the *App* class. You can see an example of this in the *SimpleBingMaps* solution in the sample code. Note, however, that this is not a secure approach to use for a production application. Chapter 13 discusses alternative, more secure approaches than simply hard-coding such “secrets” into your application.

```
internal const string BingMapsAccountId = "<< BING APP KEY >>";
```

In your main page, set up an *ApplicationIdCredentialsProvider*, using this key:

```
private readonly CredentialsProvider _credentialsProvider =
    new ApplicationIdCredentialsProvider(App.BingMapsAccountId);
public CredentialsProvider CredentialsProvider
{
    get { return _credentialsProvider; }
}
```

Note that *ApplicationIdCredentialsProvider* implements *INotifyPropertyChanged*, so you can use it for data binding. Drag a *Map* control from the toolbox onto your page. In the XAML, data-bind it to this *CredentialsProvider*.

```
<my:Map Name="map1" CredentialsProvider="{Binding CredentialsProvider}" />
```

These few simple steps will give your map functionality in your application, as shown in Figure 11-12.



FIGURE 11-12 Building a basic Bing maps application is simple.



Note Pinch-and-stretch will work out of the box, as will a double-tap to zoom in.

Geolocation

Basic use of the *GeoCoordinateWatcher* was covered in Chapter 9. In this example, you combine the *GeoCoordinateWatcher* with Bing Maps. This is the *TestBingMaps* solution in the sample code.

To use the phone's geolocation capabilities, add a reference to *System.Device* so that you can declare a *GeoCoordinateWatcher* and *GeoCoordinate* fields. In *OnNavigatedTo*, you instantiate and start the watcher, sinking the *PositionChanged* events. You stop the watcher in *OnNavigatedFrom*.

```
private GeoCoordinateWatcher coordWatcher;
private GeoCoordinate position;

protected override void OnNavigatedTo(NavigationEventArgs e)
{
    if (coordWatcher == null)
    {
        coordWatcher = new GeoCoordinateWatcher();
        coordWatcher.PositionChanged += gcw_PositionChanged;
    }

    coordWatcher.Start();
}

protected override void OnNavigatedFrom(NavigationEventArgs e)
{
    if (coordWatcher != null)
    {
        coordWatcher.Stop();
    }
}
```

In the *PositionChanged* event handler, update the value of the *GeoCoordinate*. You'll use this to set the map position.

```
private void gcw_PositionChanged(object sender, GeoPositionChangedEventArgs<GeoCoordinate> e)
{
    position = e.Position.Location;
}
```

Finally, in a suitable App Bar button handler, you'll set the map position. At the same time, set the *ZoomLevel* to an arbitrary value of 16, just to make it more obvious that the position locator is working as expected.


```
private void appBarGeoLocate_Click(object sender, EventArgs e)
{
    if (position != null)
    {
        map1.Center = position;
        map1.ZoomLevel = 16;
    }
}
```

The results are shown in Figure 11-13.



FIGURE 11-13 An application is more useful if it combines Bing maps and geolocation.

Bing Maps Web Services

The Bing Maps Simple Object Access Protocol (SOAP) services are listed in Table 11-2.

TABLE 11-2 Bing Maps SOAP Services

Namespace	URL
GeocodeService	http://dev.virtualearth.net/webservices/v1/geocodeservice/geocodeservice.svc?wsdl
SearchService	http://dev.virtualearth.net/webservices/v1/searchservice/searchservice.svc?wsdl
ImageryService	http://dev.virtualearth.net/webservices/v1/imageryservice/imageryservice.svc?wsdl
RouteService	http://dev.virtualearth.net/webservices/v1/routeservice/routeservice.svc?wsdl

The following example (the *TestGeocodeService* application in the sample code), which is shown in Figure 11-14, uses the *GeocodeService* to find the geocode (latitude,longitude) for a given street address.



The screenshot shows a web application interface with the following elements:

- Header: "CONTOSO" and "geocode svc"
- Form field: "street address" with the value "1 Microsoft Way, 98052"
- Button: "get geocode"
- Form field: "geocode" with the value "47.64,-122.13"

FIGURE 11-14 The *GeocodeService* is the most commonly used Bing Maps web service.

First, we add a service reference to the *GeocodeService*, as shown in Figure 11-15. Click the Advanced button, set the Collection Type to *System.Array*, and then clear the Reuse Types In Referenced Assemblies check box; otherwise, the wizard generates a blank service reference. You don't want the types from the assemblies referenced by the service because they'll be full .NET CLR and not compatible with the phone. The same applies to the collection classes.

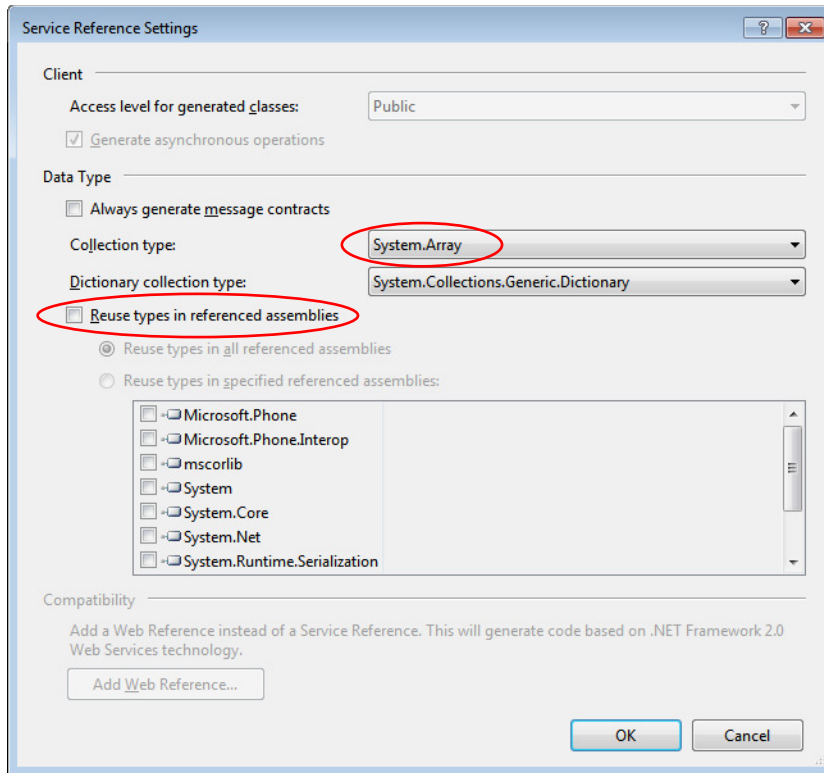


FIGURE 11-15 Avoid incompatibilities in the generated service reference.

In addition to the client proxy code, this will also generate a client-side service config file, typically named `ServiceReferences.ClientConfig`. This will include declarations like the basic HTTP binding, which you will need to reference later in your code.

```
<basicHttpBinding>
  <binding name="BasicHttpBinding_IGeocodeService" maxBufferSize="2147483647"
    maxReceivedMessageSize="2147483647">
    <security mode="None" />
  </binding>
</basicHttpBinding>
```

In the main page, you implement the *Button Click* handler by creating a *GeocodeRequest*, handling its *GeocodeCompleted* event and invoking the asynchronous web service call *GeocodeAsync*.

```
internal const string BingMapsAccountId = "<< BING APP KEY >>";

private void getGeocode_Click(object sender, RoutedEventArgs e)
{
    if (!String.IsNullOrEmpty(streetText.Text))
    {

```

```

        GeocodeRequest request = new GeocodeRequest();
        request.Credentials = new Credentials();
        request.Credentials.ApplicationId = BingMapsAccountId;
        request.Query = streetText.Text;
        GeocodeServiceClient geocodeService =
            new GeocodeServiceClient("BasicHttpBinding_IGeocodeService");
        geocodeService.GeocodeCompleted +=
            new EventHandler<GeocodeCompletedEventArgs>
                (geocodeService_GeocodeCompleted);
        geocodeService.GeocodeAsync(request);
    }
}

```

When you receive the *GeocodeCompleted* event, extract the *Latitude* and *Longitude* values from the results, and then set them into the last *TextBox*.

```

private void geocodeService_GeocodeCompleted(object sender, GeocodeCompletedEventArgs e)
{
    GeocodeResponse response = e.Result;
    if (response.Results.Length > 0)
    {
        geocodeText.Text = String.Format("{0},{1}",
            response.Results[0].Locations[0].Latitude,
            response.Results[0].Locations[0].Longitude);
    }
    else
    {
        geocodeText.Text = "not found";
    }
}

```

Deep Zoom (*MultiScaleImage*)

Deep Zoom is a WPF/Silverlight technology with which you can build applications that use extremely high-resolution images that can be incrementally downloaded to a desktop or mobile device. This provides obvious bandwidth-consumption benefits and a responsive UI, even in the face of very large image files. The way it works is that images are comprised of image tiles, at different resolutions. As the user zooms in and out of the image, only those tiles required for the requested zoom operation are downloaded. Although it is possible to use local image tiles with desktop Silverlight, this does not work with Windows Phone—all Deep Zoom images must be on a remote server, not packaged with the application itself. Figure 11-16 shows a simple application (the *TestDZ* solution in the sample code) that uses Deep Zoom. The phone application uses a Deep Zoom project on the server side.

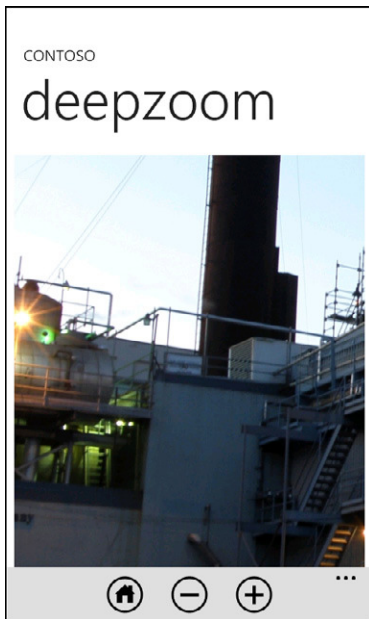


FIGURE 11-16 You can easily adapt a desktop Deep Zoom project for a Windows Phone application.

There are two stages to creating a Deep Zoom application: creating the image tiles that reside on a server; and creating the client application on the phone that downloads the server-side images. To create the image tiles, you need to use the Deep Zoom Composer tool, which is a free download from <http://www.microsoft.com/download/en/details.aspx?displaylang=en&id=24819>. Full documentation is included in the download, but the basic steps are as follows:

1. Create a new Deep Zoom project.
2. Import one or more source images. You can drag image files onto the Deep Zoom Composer window.
3. Compose the images, including optionally layering different images at different zoom levels into one composite experience.
4. Export the project. At this point, you can choose from several options for the export. The most useful for the purposes of Windows Phone development is to use the Deep Zoom Classic + Source template type. This generates a desktop Silverlight solution as well as generating all the image tiles. You can see an example of this in Figure 11-17.

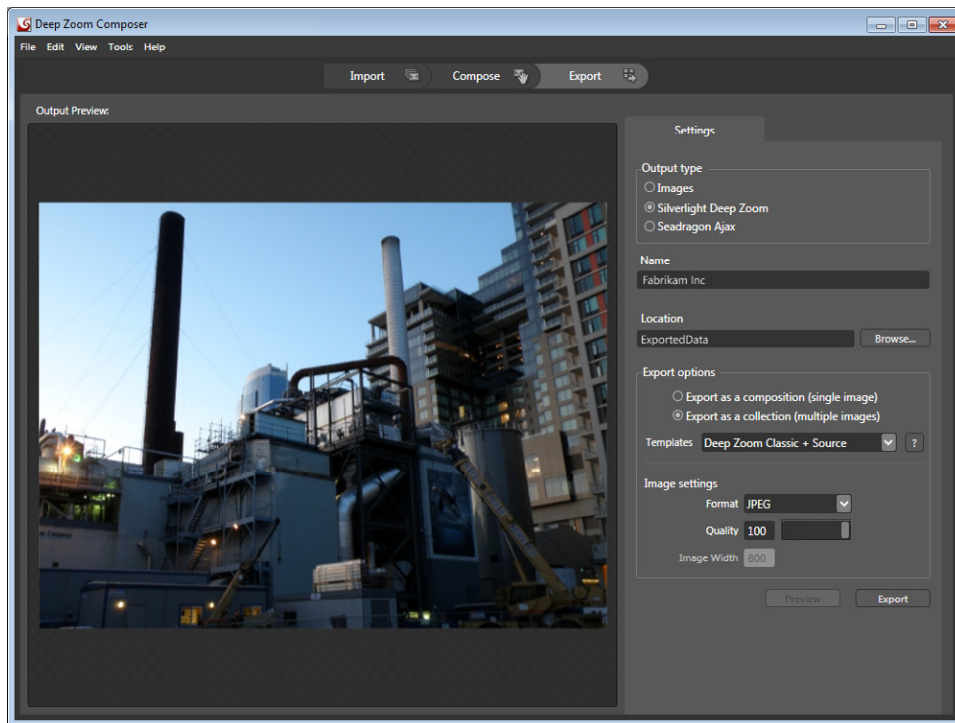


FIGURE 11-17 Create a Deep Zoom project by using the Deep Zoom Composer tool.

The generated solution includes an XAML file for the main application page, which includes the declaration for a *MultiScaleImage* control. The *MultiScaleImage* control is a standard control in System.Windows.dll. The *Source* property is set to the default *dzc_output.xml*, which is the top-level manifest that points to the collection of image tiles. The image tiles reside in a server-side web project, also generated as output by the Deep Zoom Composer. At this point, you could deploy the web project to a server. The desktop Silverlight project would also typically be deployed from a server-hosted HTML page. This piece is not required for a Windows Phone solution, but to keep things simple, you can just build and execute the entire solution as is. This will deploy to the local computer and run the browser to connect to the HTML page that hosts the Silverlight solution. You can then copy the browser URL and use it as the base path for the *Source* of your own *MultiScaleImage* control. Create a phone solution, and then add the following line to your page XAML, making sure that the URL corresponds to the path of the generated *dzc_output.xml* in the web project:

```
<MultiScaleImage
    x:Name="msi" Source="http://localhost:61306/DeepZoomProjectSite/ClientBin/GeneratedImages/
    dzc_output.xml"/>
```

You can also cannibalize parts of the desktop Silverlight project and use them in a standard Windows Phone solution. Specifically, almost all of the code related to mouse manipulation can be used directly—just copy it into your phone project. This provides all the scroll/drag behavior, and supports zooming by tapping the image. Most of this code is in three anonymous delegates, which you can copy from the *Page.xaml.cs* in the desktop Silverlight project to the *MainPage.xaml.cs* in your

phone project. These delegates handle *MouseLeftButtonDown* to track the starting point of a subsequent *MouseMove*; *MouseLeftButtonUp* to zoom in or out by a factor of two (depending on whether the Shift key is down—this is not applicable to non-keyboard/SIP input in a phone application); and *MouseMove* to move the *ViewportOrigin* of the *MultiScaleImage* control (thereby dragging it around the screen).

```
public MainPage()
{
    InitializeComponent();

    this.MouseLeftButtonDown += delegate(object sender, MouseButtonEventArgs e)
    {
        lastMouseDownPos = e.GetPosition(msi);
        lastMouseViewPort = msi.ViewportOrigin;
        mouseDown = true;
        msi.CaptureMouse();
    };

    this.MouseLeftButtonUp += delegate(object sender, MouseButtonEventArgs e)
    {
        if (!duringDrag)
        {
            newzoom *= 2;
            Zoom(newzoom, msi.ElementToLogicalPoint(this.lastMousePos));
        }
        duringDrag = false;
        mouseDown = false;
        msi.ReleaseMouseCapture();
    };

    this.MouseMove += delegate(object sender, MouseEventArgs e)
    {
        lastMousePos = e.GetPosition(msi);
        if (mouseDown && !duringDrag)
        {
            duringDrag = true;
            double w = msi.ViewportWidth;
            Point o = new Point(msi.ViewportOrigin.X, msi.ViewportOrigin.Y);
            msi.UseSprings = false;
            msi.ViewportOrigin = new Point(o.X, o.Y);
            msi.ViewportWidth = w;
            zoom = 1 / w;
            msi.UseSprings = true;
        }

        if (duringDrag)
        {
            Point newPoint = lastMouseViewPort;
            newPoint.X += (lastMouseDownPos.X - lastMousePos.X) /
                msi.ActualWidth * msi.ViewportWidth;
            newPoint.Y += (lastMouseDownPos.Y - lastMousePos.Y) /
                msi.ActualWidth * msi.ViewportWidth;
            msi.ViewportOrigin = newPoint;
        }
    };
}
```

As an alternative to just copying the code from the desktop project, you could consider migrating the mouse event code to use *ManipulationXXX* event handlers or *OnManipulationXXX* overrides, instead. This would keep your code more in line with the touch-oriented phone model, as opposed to the mouse-oriented desktop model. With the SDK v7.1, you could also use the simpler *Gesture* event support.

The desktop project also includes four buttons and their *Click* handlers (zooming in, zooming out, resetting to the original zoom level, and zooming to fullscreen). The fullscreen behavior is not relevant to Windows Phone, but all the code for the other three buttons can be copied and pasted. Revisit Figure 11-16 and note that the on-page buttons have been replaced with App Bar buttons, but the code-behind is the same.

```
private void Zoom(double newzoom, Point p)
{
    if (newzoom < 0.5)
    {
        newzoom = 0.5;
    }
    msi.ZoomAboutLogicalPoint(newzoom / zoom, p.X, p.Y);
    zoom = newzoom;
}

private void GoHome_Click(object sender, EventArgs e)
{
    this.msi.ViewportWidth = 1;
    this.msi.ViewportOrigin = new Point(0, 0);
    ZoomFactor = 1;
}

private void ZoomOut_Click(object sender, EventArgs e)
{
    Zoom(zoom / 2, msi.ElementToLogicalPoint(
        new Point(.5 * msi.ActualWidth, .5 * msi.ActualHeight)));
}

private void ZoomIn_Click(object sender, EventArgs e)
{
    Zoom(zoom * 2, msi.ElementToLogicalPoint(
        new Point(.5 * msi.ActualWidth, .5 * msi.ActualHeight)));
}
```

Note that the code generated by the Deep Zoom Composer tool sets the zoom factor to 1.3 in the *ZoomIn* and *ZoomOut* handlers. To be consistent with touch/mouse manipulation, the code above sets the factor to 2 in both cases. Also note that if you want to experiment with a phone solution without going to the trouble of building and deploying server-side image tiles, you can point your *MultiScaleImage* to the Deep Zoom team's hosted tiles at <http://static.seadragon.com>, as shown here:

```
<MultiScaleImage x:Name="msi" Source="http://static.seadragon.com/content/misc/
contoso-fixster.dzi"/>
```




Note If you want to see these tiles in a regular browser window, go to <http://zoom.it>.

Windows Azure

Windows Azure is a cloud services platform hosted through Microsoft data centers. Customers who have accounts on Azure Windows Azure accounts can build and deploy applications to the cloud. The applications are hosted in virtual computers running on geographically distributed data centers. Customer data can be hosted in SQL Azure databases and/or in Windows Azure tables and *blob containers*. You would then typically deploy headless components—called “worker roles”—and internet-facing web services and web applications—called “Web Roles.” Finally, you can also build client applications (desktop, web or mobile) to connect to the server-side components. Figure 11-18 summarizes the high-level architecture of Windows Azure applications. In this diagram, the round-cornered blocks are custom-defined, whereas the rectangular blocks are part of Windows Azure.

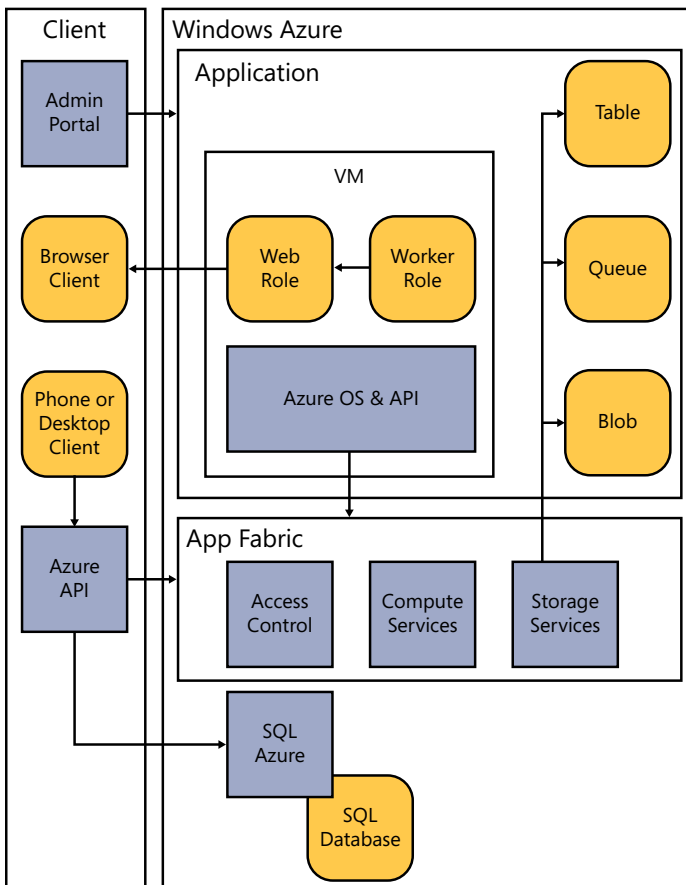


FIGURE 11-18 The Windows Azure application architecture offers a comprehensive set of cloud features.

When you build a Windows Azure application, you can build one or more Web Roles and/or one or more worker roles. Each web or worker role runs in a virtual computer on top of the Windows Azure operating system, which itself sits on top of Windows Internet Information Services (IIS). Microsoft provides an admin portal with which you can manage your server-side application, including starting and stopping, specifying the number of instances to run, and so on. The portal is a web application accessible from any browser. Your Web Role can consist of headless web services or it can include web UI. If your Web Role includes web UI, then the user can connect to it via a browser. You can also create Windows desktop and/or Windows Phone client applications that connect to Windows Azure. These will typically use the Windows Azure API, most of which resides in the Microsoft.WindowsAzure.StorageClient.dll (part of the Windows Azure SDK).

The Windows Azure AppFabric SDK includes access control, compute services, and storage services. Together, these provide secured access to your data in the cloud. There are two main types of cloud data:

- **Windows Azure Storage** This is intended for relatively simple, flat data. You can use tables (which behave like a very simple form of database table, supporting simple queries), queues (which behave exactly like queues, for reliable, persistent messaging between services), and blobs (for any unstructured data such as images).
- **SQL Azure** This is a full-blown SQL Server implementation hosted within Windows Azure. It has equivalent capabilities to regular SQL Server, including automatic geographically distributed server replication, provided by default.

Just as you can choose an appropriate permutation of web and/or worker roles, you can also choose which, if any, of the storage options your application needs. You can access the storage services either directly via a REST API over the Internet from any application that can use HTTP/HTTPS, or from a server-side component running within Windows Azure by using the StorageClient.dll.

The Windows Azure SDK is available as a free download from <http://www.microsoft.com/windowsazure/sdk/>, and includes:

- Windows Azure Tools for Microsoft Visual Studio, including project templates.
- Windows Azure SDK, including the client API in the StorageClient.dll.
- Visual Web Developer 2010, if you do not have Visual Studio 2010.
- ASP.Net MVC3.
- Windows Azure AppFabric SDK.
- Required IIS feature settings and related hotfixes.
- The storage emulator, which provides desktop emulation of Windows Azure storage services for local blob, queue, and table storage. This also includes the DSInit command-line tool for configuring the storage emulator for your specific desktop and local SQL environment.

- The compute emulator, which provides desktop emulation of the Windows Azure compute services, including a UI application for managing service deployments and role instances, and the CSPack and CSRun command-line tools for packaging applications for deployment.

Windows Azure Web Services

From the perspective of the Windows Phone app, consuming a web service exposed by Windows Azure is no different from consuming a web service exposed in any other way. Setting up the Windows Azure web service, however, does require specific steps, including an Azure account registration.

This example (including all the applications in the *CloudManatee* solution in the sample code) is a Windows Azure equivalent of the MagicalManatee WCF web service example that you created earlier. After you have downloaded and installed the Windows Azure SDK, create a Windows Azure project, and then add a WCF Service Web Role, as shown in Figure 11-19.

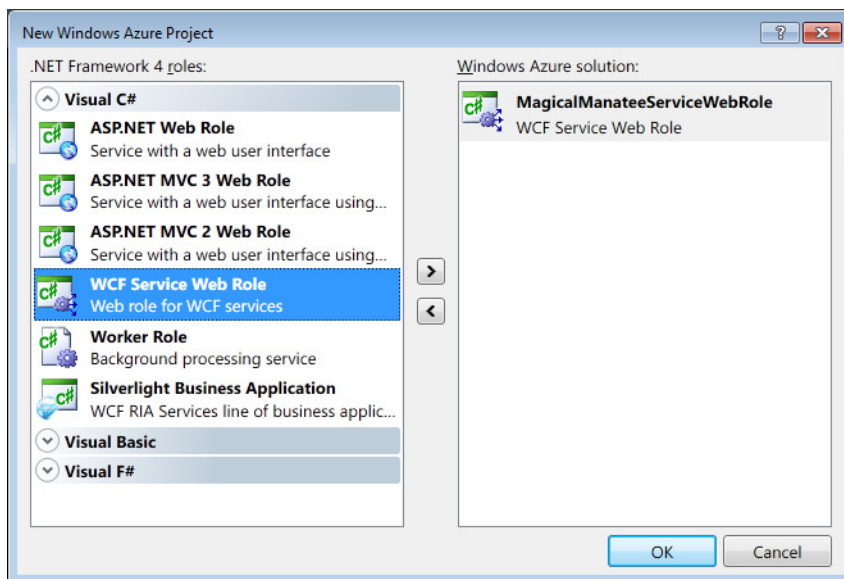


FIGURE 11-19 Add a WCF Service Web Role to a Windows Azure solution.

- **Create the service.** Rename and edit the *IService1.cs*, *Service1.svc*, and *Service1.svc.cs* to define your service. All this is the same as for regular WCF web services. Press F5 to build and run it. This will run the service and open a client browser window to the svc URL.
- **Create the client.** With the service still running, create a Windows Phone app, and then add a service reference to the service. Write code to exercise the service methods. All this is the same as for regular WCF web services client. Build and test the client.
- **Provision the service in Azure.** Log on to the Windows Azure Management Portal (<http://windows.azure.com>), and then create the storage account for your application, as shown in Figure 11-20.



Note Even if you don't have explicit data to store, you need a storage account for diagnostics logs. Select your subscription from the list, and then enter the name for your storage account. This must be a unique name. Azure uses this value to generate the endpoint URLs for the storage account services.

FIGURE 11-20 For most Windows Azure solutions, you must create a new storage account.

Create a new affinity group from the drop-down list. You use this to deploy both the hosted service and storage account to the same location, thus ensuring high bandwidth and low latency between the application and the data on which it depends. When you confirm this dialog, the provisioning process starts—this usually takes a few minutes. Once provisioning is complete, you need to copy the generated access keys for use in your code. You can get these keys from the View Access Keys button in the management portal toolbar.

In Visual Studio, open the `ServiceConfiguration.cscfg` file located in your service. Replace the developer diagnostics connection string value with your account information. That is, replace this:

```
<Setting name="Microsoft.WindowsAzure.Plugins.Diagnostics.ConnectionString"
value="UseDevelopmentStorage=true" />
```

with this (using your own account name and key, as copied to the clipboard in an earlier step):

```
<Setting name="Microsoft.WindowsAzure.Plugins.Diagnostics.ConnectionString"
value="DefaultEndpointsProtocol=https;AccountName=magicalmanateeservice;AccountKey=XXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX" />
```

Go back to the Management Portal and create a new hosted service, as shown in Figure 11-21. Enter an arbitrary (and unique) service name and a name for the base part of the URL. You should normally use the same name for both the storage account and hosted service.

Create a New Hosted Service

Choose a subscription
 My Subscription

Enter a name for your service
 MagicalManateeService

Enter a URL prefix for your service
 magicalmanateeservice .cloudapp.net

Choose a region or affinity group
☐ Choose a Region ☒ magicalmanateeservice - Anywhere US

Deployment options
☐ Deploy to stage environment
☐ Deploy to production environment
☒ Do not deploy
☒ Start after successful deployment

Deployment name

Package location
 Browse Locally... Browse Storage...

Configuration file
 Browse Locally... Browse Storage...

Add Certificate

OK Cancel

FIGURE 11-21 Create a new Hosted Service for your Windows Azure web services.

Use the same affinity group as for your storage account. Select the option labeled Do Not Deploy. While you *can* create and deploy your service to Azure in a single operation by completing the Deployment Options section, it is normal to use the “deferred deployment” path because the same steps are used when updating your deployment.

There are several alternatives for deploying applications to Azure. Using the Azure Tools for Visual Studio, you can both create and deploy the service package to the Azure environment directly from Visual Studio. Another deployment option is the Azure Service Management PowerShell Cmdlets (available at <http://archive.msdn.microsoft.com/azurecmdlets>) with which you can script deployment of your application. With the Management Portal, you can deploy and manage your service by using only your browser.

Back in Visual Studio, right-click the cloud project (not the Web Role), and then select Package, as shown in Figure 11-22.

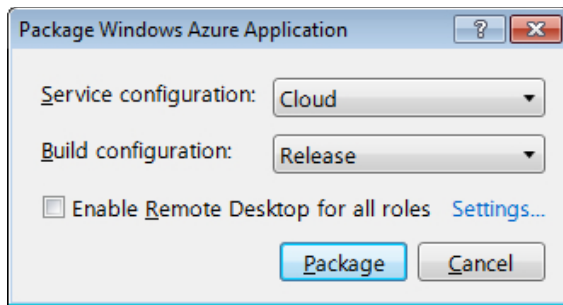


FIGURE 11-22 It is generally more useful to deploy the Windows Azure project in two phases.

This creates the packaged solution and its configuration file. By default, these are located in the app.publish subfolder of your solution target folder. Note that in order to package and publish from Visual Studio, you will be prompted to download and import a credentials file from the Azure Management Portal.

Back in the Management Portal, select your hosted service, and then create a New Staging Deployment. A hosted service has two separate deployment slots: staging and production. With the staging deployment slot, you can test your service in the Azure environment before you deploy it to production. Specify an arbitrary name for this deployment (for example, v1.0), and then browse to the location for the package and configuration files, as shown in Figure 11-23.

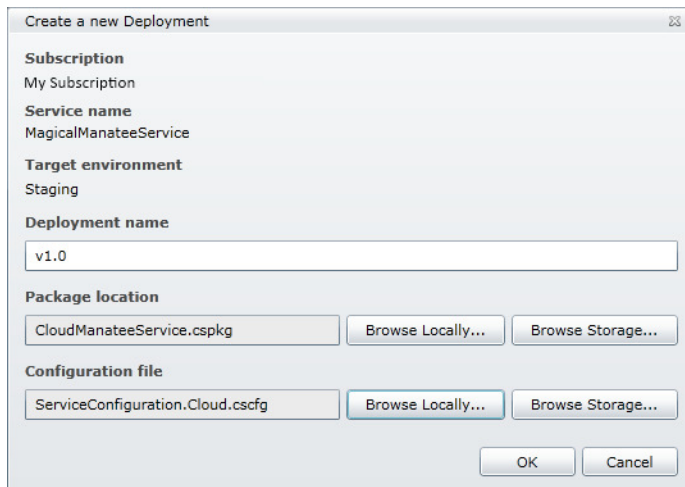


FIGURE 11-23 When you create a new deployment, you're typically creating a new version of your solution.

When you confirm this, the system analyzes your service configuration. If it finds anything questionable, it will issue a warning. For example, if any of your roles are set to have only one instance deployed (which is the default code generated in Visual Studio), the system will warn that this is not best practice from an availability perspective. You can override this warning, and you can always go back and edit the number of instances in the portal later. After confirming the deployment, the service is provisioned, and you can then test it. In the Management Portal, select your deployment and copy (don't click) the link in the text box labeled DNS Name, as shown in Figure 11-24.

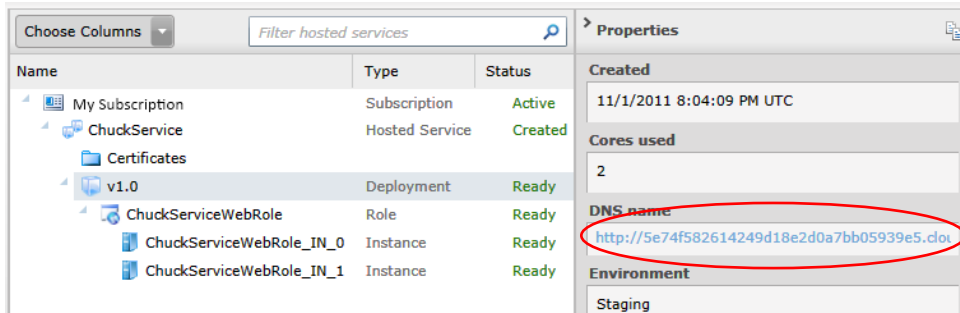


FIGURE 11-24 When you deploy your solution, you get a unique URL for the service name.

Paste this into a browser and append the service .svc extension; for example:

<http://a18daf2763ff41eb88f753fb6c6f8b72.cloudapp.net/MagicalManateeFacts.svc>

When you have verified that the service is working correctly in the staging environment, you can promote it to final production. When you deploy the application to production, Azure reconfigures its load balancers so that the application is available at its production URL—in this example, that will be: *<http://magicalmanateeservice.cloudapp.net/MagicalManateeFacts.svc>*.

To switch between staging and production, select your service, and then on the ribbon, click Swap VIP. In the Swap VIP dialog box, click OK to swap the deployments between staging and production.

Wait for the promotion process to complete, and then test the client phone app again. The final result on the client is exactly the same as the version illustrated back in Figure 11-5. The only difference between this version of the client and the earlier version is that you now target an Azure-hosted web service instead of a web service hosted on a non-Azure server. This difference is almost completely transparent to the client. To redirect the client to point to the Azure web service, all you need to do is to change the endpoint address in the ServiceReferences.ClientConfig file, as follows:

```
<endpoint
  address="http://magicalmanateeservice.cloudapp.net/MagicalManateeFacts.svc"
  binding="basicHttpBinding"
  bindingConfiguration="BasicHttpBinding_IMagicalManateeFacts"
  contract="MagicalManateeService.IMagicalManateeFacts"
  name="BasicHttpBinding_IMagicalManateeFacts" />
```

Windows Azure Toolkit for Windows Phone

The Developer Platform Evangelism team at Microsoft publishes and maintains the Windows Azure Toolkit for Windows Phone. This is available as a free download from <http://watwp.codeplex.com/>.

The toolkit includes Visual Studio project templates for building Windows Azure solutions that incorporate Windows Phone client applications, a set of class libraries optimized for use on the phone, sample applications, and documentation. One of the most useful pieces in the toolkit is its support for authentication. To access data stored in Windows Azure, a client must have the name and key information for the storage account. You would not want to expose this information to real client applications across the Internet, and you would not want these details stored on the phone. To solve this issue, the toolkit provides a set of proxies and services with which you can access Windows Azure Storage in a secure fashion. This way, the storage account information remains safe in the Web Role that hosts these services:

- The Azure Tables and Queues proxies forward requests to the real Windows Azure Storage Services. These proxies support several authentication mechanisms such as Membership and Access Control Service (ACS).
- The Shared Access Signature service is a WCF REST Service that delivers Shared Access Signatures (SAS) for containers and blobs. An SAS is a set of URL query parameters that includes an expiry time, the permission set to be granted, the blob/container resources to be made available, and the signature that the Blob service should use to authenticate the request. Once the phone client receives the SAS, it can use it to perform requests to the Blob Service REST API.

To get a feel for the toolkit, the proxies, and the supporting class libraries, you can create a new application by using the toolkit-supplied Windows Phone 7 Cloud Application solution template in Visual Studio. This will generate a solution with three projects: the Cloud project (which is primarily a deployment packaging project); a Cloud Services project (which contains the Web Roles, web services, and web application); and a Phone client application. Altogether, the template generates about 190 files. However, the solution is not as complicated as it seems: it is heavily layered, with strict separation of concerns, using a Model View ViewModel (MVVM) design—and most of the files are quite small.

Starting the Cloud project starts both the web app and the web services. The web app, shown in Figure 11-25, provides a tabbed-UI for managing users and permissions.

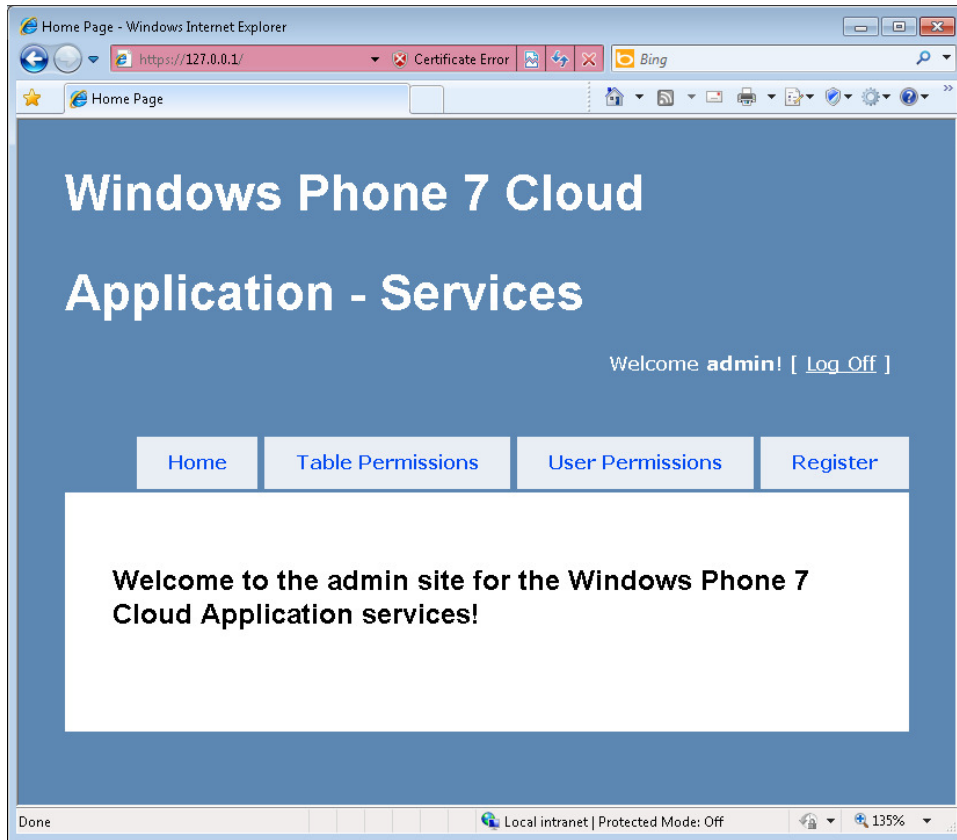


FIGURE 11-25 A Windows Phone 7 cloud application has a web app management portal.

The web services include both authentication services to verify that the client has the required permissions to access the cloud resources as well as the data services for transferring data to and from Azure storage. The phone application starts with a *LoginPage*, which upon successful login by the user, transitions to the *MainPage*. The *MainPage* has a *PivotControl* with pivot items for a list of tables, list of table data, and list of blob data. The phone UI is shown in Figure 11-26.

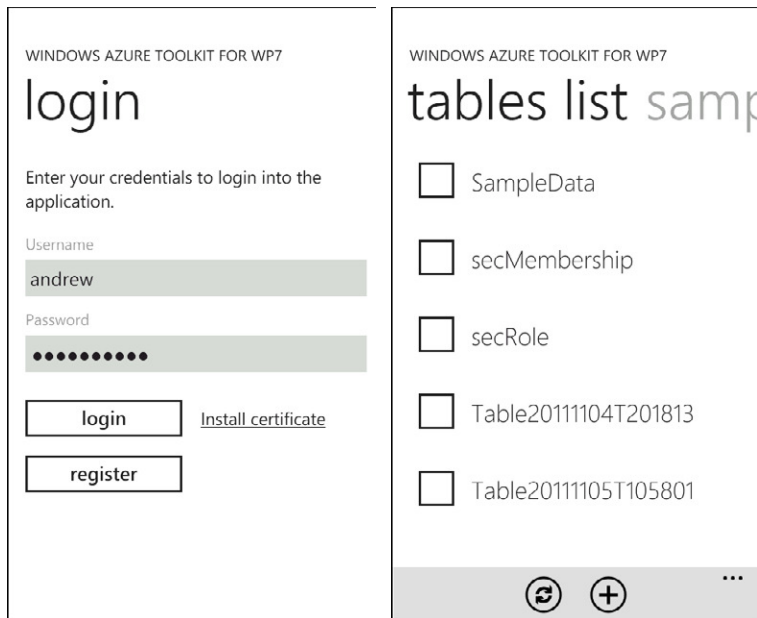


FIGURE 11-26 The Windows Phone client application login page and pivot page, connecting to Windows Azure.

Figures 11-27, 11-28, and 11-29 summarize the flows between the client and server applications, and the various toolkit-provided services and proxies that are used. First, login: the *LoginPage* is backed by a corresponding viewmodel, which uses the *StorageClient.dll* to connect to the *AuthenticationService*.

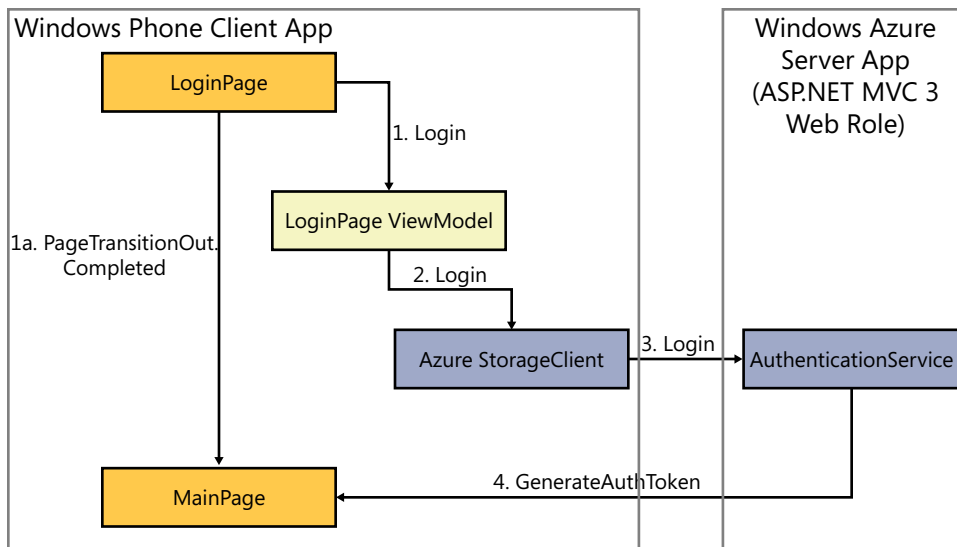


FIGURE 11-27 A phone client application logs in via the *AuthenticationService*.

After logging in, the user can select any of the pivot items on the *MainPage*, but each type of data represented is also secured on the server. For example, to access a list of tables and the table data within them, the user must not only be registered with the application but must also be authorized to access the table data. The web application provides UI for the administrator to manage storage and user permissions. The pivot items for table listings and table data are both backed by corresponding viewmodels, which use the *StorageClient.dll* to talk to the *AzureTablesHandler* proxy. In turn, this proxy submits an *HttpRequest* to fetch table data from cloud storage.

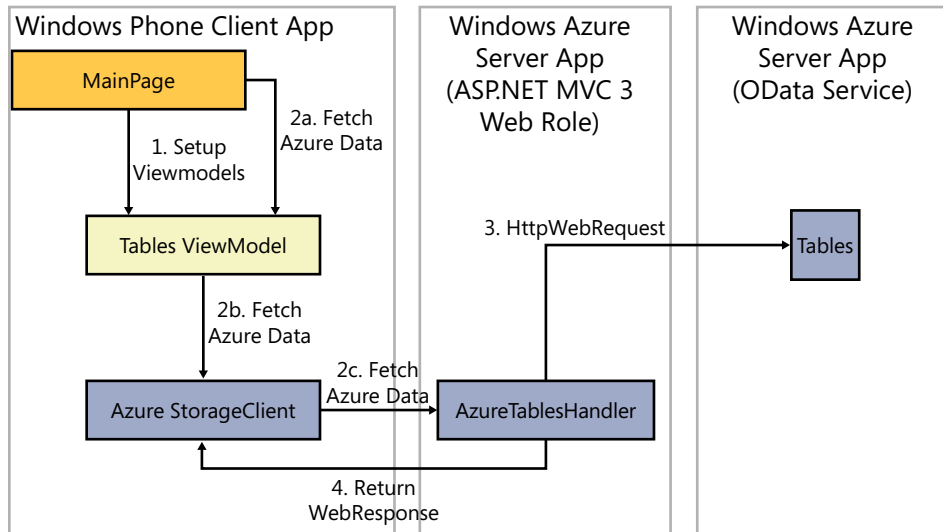


FIGURE 11-28 The phone client application accesses table data via the *AzureTablesHandler* proxy.

Access to blob containers and blob data is via the *SharedAccessSignature* service. Internally, this uses the *StorageClient.dll* to connect to the blobs in Azure via a REST API.

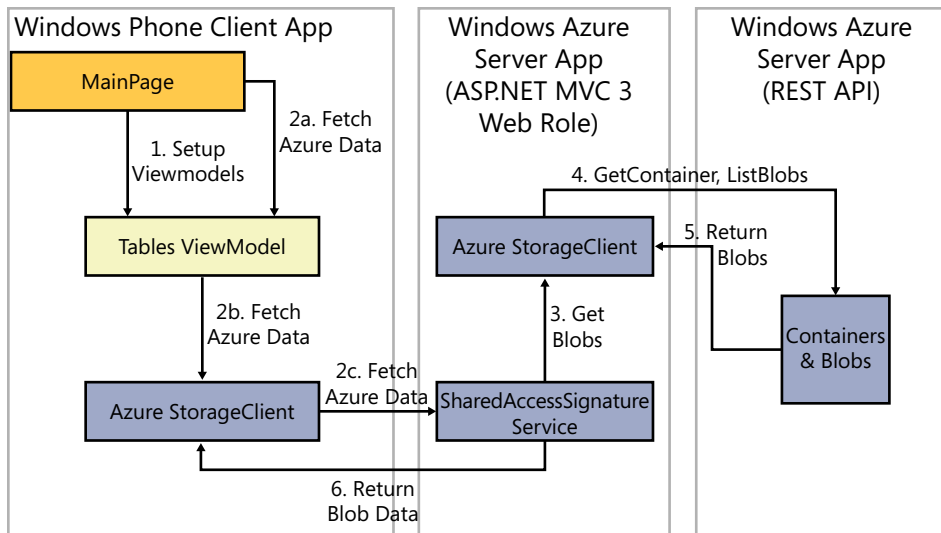


FIGURE 11-29 The phone client application accesses blob data via the *SharedAccessSignature* Service.

This application exercises many of the features of the toolkit and acts as guidance for building your own phone applications for connecting to Windows Azure Storage. The toolkit also includes sample applications for accessing SQL Azure databases by using bitly and by using Project Hawaii. bitly is discussed in the upcoming section. Project Hawaii is a joint initiative between Microsoft Research and universities around the world. It enables students to develop inventive cloud-enhanced mobile applications. Students at participating universities can use Windows Phone for accessing Windows Azure for computation and data storage.

bitly

bitly is a tool with which users can create a very short URL that maps to any arbitrarily long URL. The resulting short URL is unique and publicly accessible. For example, you can turn the link [http://msdn.microsoft.com/en-us/library/ff637516\(VS.92\).aspx](http://msdn.microsoft.com/en-us/library/ff637516(VS.92).aspx) into <http://bit.ly/c2RmNr>. bitly exposes an API that you can use in your Windows Phone applications. To use the bitly API, you must first create a free account at <http://bitly.com/>. The bitly API is documented at <http://code.google.com/p/bitly-api/wiki/ApiDocumentation>. Figure 11-30 shows a Windows Phone application (*TestBitly* in the sample code) that uses the bitly API to generate a shortened URL from a long URL.

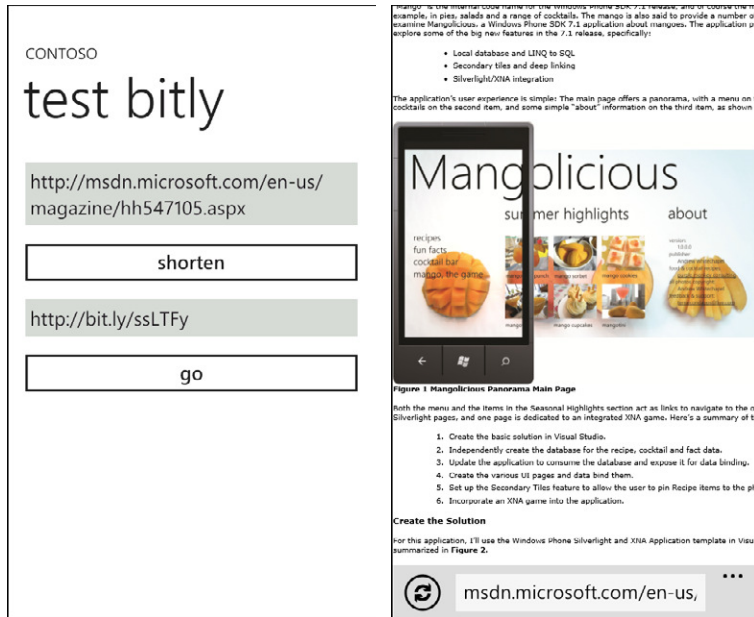


FIGURE 11-30 You can use the bitly API to generate a short URL, and then navigate to that URL.

The user can enter a long URL in the first *TextBox*. When he taps the Shorten button, you compose a URI string that incorporates the base bitly API website along with your API username and account key, and the long URL as query string parameters. As noted with Bing services, you should not hard-code your username and account key in a production application. See Chapter 13 for ways to secure these credentials. You then make an asynchronous *WebClient* request, and retrieve the XML returned as an *XDocument*. This will contain the short URL as well as a status code. In the event of an error, it will also contain an error string.

```
private void Shorten_Click(object sender, RoutedEventArgs e)
{
    String login = "<< YOUR BITLY USERNAME >>";
    String apiKey = "<< YOUR BITLY API ACCOUNT KEY >>";
    String longUrl = LongUrl.Text;
    String uriString = String.Format(
        "http://api.bitly.com/v3/shorten?login={0}&apiKey={1}&format=xml&longUrl={2}",
        login, apiKey, longUrl);
    Uri bitlyUrl = new Uri(uriString);

    WebClient client = new WebClient();
    client.DownloadStringCompleted += (s, ev) =>
    {
```

```

        if (ev.Error == null)
        {
            XDocument doc = XDocument.Parse(ev.Result);
            XElement errorCodeResponse =
                doc.Descendants("status_code").FirstOrDefault();
            String shortenedUrl = longUrl;
            if (errorCodeResponse != null)
            {
                int statusCode = Convert.ToInt32(errorCodeResponse.Value);
                if ((StatusCode)statusCode == StatusCode.OK)
                {
                    XElement node = doc.Descendants("data").FirstOrDefault();
                    shortenedUrl = node.Descendants("url").SingleOrDefault().Value;
                }
            }
            Dispatcher.BeginInvoke(() => { ShortUrl.Text = shortenedUrl; } );
        }
    };

    client.DownloadStringAsync(bitlyUrl);
}

```

When you get the short URL back, set it into the second *TextBox*. After that, the user can tap the second button to navigate to the short URL, using a *WebBrowserTask*. The result is shown in Figure 11-30.

```

private void Go_Click(object sender, RoutedEventArgs e)
{
    WebBrowserTask browserTask = new WebBrowserTask();
    browserTask.URL = ShortUrl.Text;
    browserTask.Show();
}

```

Note that the preceding listings use a custom *StatusCode* enum, whose values are derived from the bitly API documentation.

```

public enum StatusCode
{
    OK = 200,
    RateLimitExceeded = 403,
    Invalid = 500,
    UnknownError = 503,
    Unspecified = 1,
}

```

Facebook

The Facebook C# SDK is available as a free download at <http://facebooksdk.codeplex.com>. The SDK supports web, desktop Silverlight, and Windows Phone applications that are intended to integrate with Facebook. To create a Phone application that integrates with Facebook, you must provision the application on Facebook. To do this, go to the Facebook developer page (<http://developers.facebook.com/>),

log in with a valid Facebook account, and then create an application. The tool will present you with a prompt you for an application name as well as a captcha challenge. It will then allocate you an App ID and App Secret. As noted with the Bing and bitly examples, you should not hard-code these credentials in a production application. See Chapter 13 for security options.

Back in Visual Studio, create a regular phone application, and put a *WebBrowser* control on your page. The example shown in Figure 11-31 offers a *Button* to start connecting to Facebook, a *Web Browser* control (in this screenshot, it shows the result of the user's first tap on the Go button), and a *TextBox* at the bottom in which you place the results when you fetch Facebook data. This is the *Test Facebook* solution in the sample code.

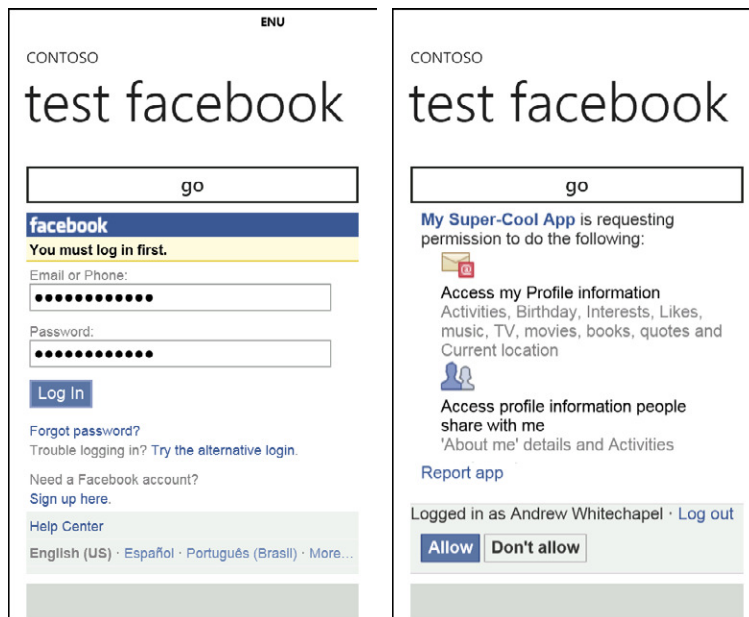


FIGURE 11-31 When you log into Facebook, you're presented with a permissions prompt.

In the XAML, note that it is important to set the *WebBrowser* control's *IsScriptEnabled* property to *true*; otherwise, the connection will fail, because the Facebook login mechanism includes Javascript for redirection.

```
<StackPanel x:Name="ContentPanel">
    <Button x:Name="Go" Content="go" Click="Go_Click"/>
    <phone:WebBrowser
        x:Name="wbc" Height="430" Margin="{StaticResource PhoneHorizontalMargin}"
        IsScriptEnabled="True"/>
    <TextBox x:Name="FbData" TextWrapping="Wrap" Height="110" />
</StackPanel>
```

In the code-behind, add a reference to the Facebook.dll, deployed as part of the SDK. Then, declare a string for the App ID and an array of strings for the permissions that your application will

request from the user when she logs in. As noted previously, you should not hard-code the App ID in a production application. See Chapter 13 for security options.

The code that follows lists all of the possible permissions. Normally, you would want to request only those permissions that you actually need. The more permissions you ask for, the higher the chance the user will reject the request to connect.

```
private const string appID = "<< APP ID >>";

private readonly string[] extendedPermissions = new[]
{
    "user_about_me", "user_activities", "user_location", "user_likes",
    "user_interests", "friends_activities", "friends_about_me", "read_stream",
    "read_friendlists", "email", "user_birthday", "publish_stream", "offline_access"
};
```

Handle the *Click* event on the button by navigating the *WebBrowser* control to the Facebook login page. This is not a fixed URL; rather, it is composed of the specific user information, permissions, and authentication type that this application uses. As of the time of this writing, contrary to the documentation, you need to specify “wap” for the *display* property, not “touch” for a phone application. You build a string of all the properties you need, and pass this to the *FacebookOAuthClient* *GetLoginUrl* method, which will return a valid URL that you can then pass on to the *WebBrowser* control for navigation.

```
private void Go_Click(object sender, RoutedEventArgs e)
{
    FacebookOAuthClient oauth = new FacebookOAuthClient { AppId = appID };
    Dictionary<String, object> parameters = new Dictionary<String, object>
    {
        { "response_type", "token" },
        { "display", "wap" }
    };

    if (extendedPermissions != null && extendedPermissions.Length > 0)
    {
        parameters["scope"] = String.Join(" ", extendedPermissions);
    }
    Uri loginUrl = oauth.GetLoginUrl(parameters);

    wbc.Navigated += new EventHandler<NavigationEventArgs>(wbc_Navigated);
    wbc.Navigate(loginUrl);
}
```


In this example, the URL returned by *GetLoginUrl* looks something like the following listing (with the dummy "<< APP ID >>" replaced, of course), which can be useful to know in case you ever need to either compose or decompose the URL manually:

```
http://www.facebook.com/dialog/oauth/?response_type=token&display=wap&scope=user_about_me,user_
activities,user_location,user_likes,user_interests,friends_activities,friends_about_me,read_
stream,read_friendlists,email,user_birthday,publish_stream,offline_access&client_id=<< APP ID
>>&redirect_uri=http://www.facebook.com/connect/login_success.html
```

After the user logs in, she will be redirected to the Facebook permissions prompt page, as previously shown in Figure 11-31.

If the user chooses to allow your application to have the permissions requested, the browser will be redirected again, and you will be able to parse the returned URI into a *FacebookOAuthResult* object. If this is successful, you can then start requesting Facebook data. In this example, you request the basic user profile data by passing */me* to the *FacebookClient GetAsync* method. When this returns, simply set the profile data into the *TextBox*, as shown in Figure 11-32.

```
private void wbc_Navigated(object sender, NavigationEventArgs e)
{
    FacebookOAuthResult result;
    if (FacebookOAuthResult.TryParse(e.Uri, out result))
    {
        if (result.IsSuccess)
        {
            var fb = new FacebookClient(result.AccessToken);
            fb.GetCompleted += new EventHandler<FacebookApiEventArgs>(fb_GetCompleted);
            fb.GetAsync("/me");
        }
        else
        {
            MessageBox.Show(result.ErrorDescription);
        }
    }
}

private void fb_GetCompleted(object sender, FacebookApiEventArgs e)
{
    var name = e.GetResultData();
    Dispatcher.BeginInvoke(() => { FbData.Text = name.ToString(); });
}
```



FIGURE 11-32 You can use the Facebook API to retrieve profile data.

Windows Live

At the time of this writing, Microsoft has released an early version of the Live SDK, with which you can connect to Windows Live, including SkyDrive, Hotmail, and Messenger. Figure 11-33 shows a very simple phone application that connects to Windows Live and retrieves some basic profile data for the logged-in user (the *TestLive* solution in the sample code). Note that the caption in the *SignInButton* changes dynamically from “Sign in” to “Sign out” according to the current sign-in state of the application.



FIGURE 11-33 A phone application that connects to Windows Live provides a sign-in button.

Before you can successfully connect a phone application to Windows Live, you must provision the application on Live itself. To do this, you log into Live in the normal way, using your Windows Live ID. Then go to the application management site (<http://manage.dev.live.com>) and click the Create Application link. You can give your application any arbitrary name, as shown in Figure 11-34, where the name is “My First Live App.” Live will generate a Client ID and Client secret. You don’t need the secret for this application. Make sure that you select the Mobile client app option.

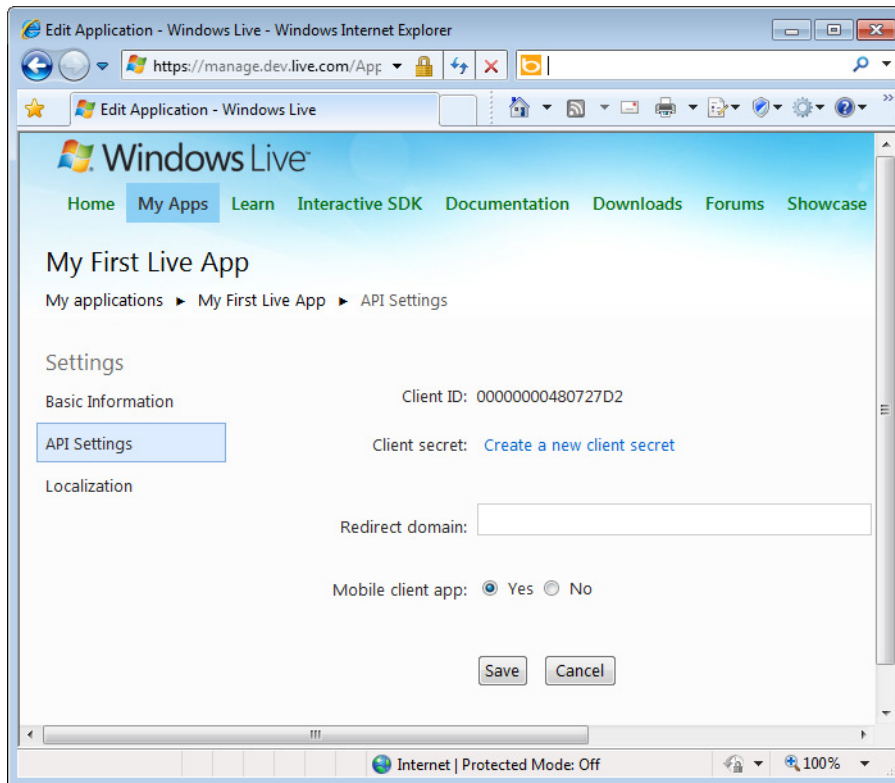


FIGURE 11-34 You must provision an application in Windows Live before deploying to the phone.

To build the phone application, you need to download the Live SDK from Microsoft download. Version 5.0 is available at <http://www.microsoft.com/download/en/details.aspx?displaylang=en&id=28195>. This includes a couple of critical DLLs that provide access to Live. Next, create a regular Windows Phone application. Add references to the `Microsoft.Live.dll` and `Microsoft.Live.Controls.dll` assemblies (these were installed with the Live SDK). You can then add a `SignInButton` control to your XAML. You can type this in manually, and you also need to add the corresponding namespace declaration. Alternatively, you can add it to the Toolbox in Visual Studio, and then drag it from the Toolbox to the design surface. Using this approach, the namespace is added for you. Here's the XAML declarations for the sample application:

```
xmlns:my="clr-namespace:Microsoft.Live.Controls;assembly=Microsoft.Live.Controls"
...
<my:SignInButton
    x:Name="liveSignIn" SessionChanged="liveSignIn_SessionChanged" />
```

In the code-behind, initialize the *Scopes* and *ClientId* properties of the *SignInButton* control. *Scopes* are the permissions that your application needs when connecting to Live. These are similar in concept to the capabilities that you specify for a phone application, but they are specific to Live. When your application runs, the user will be shown the list of scopes that your application is requesting so that he can make an informed decision as to whether to allow your application to connect to Live on his behalf, as shown in Figure 11-35.

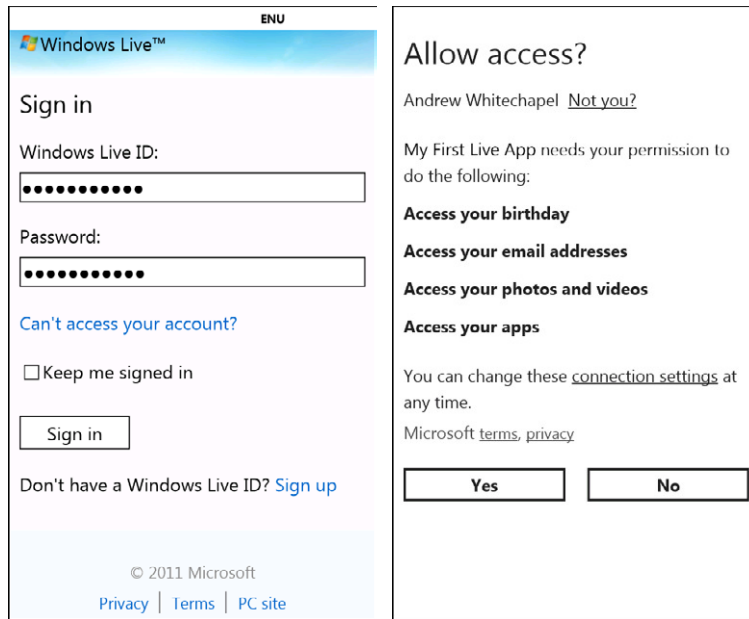


FIGURE 11-35 When you log into Windows Live, you're presented with a scopes access prompt.

The three basic scopes are *wl.signin*, *wl.basic*, and *wl.offline_access*, all of which are required for a phone application. There are extended scopes for accessing birthday information, photos, emails, contacts, and so on, but these are not used in this application.

```
private LiveConnectClient liveClient;

public MainPage()
{
    InitializeComponent();

    liveSignIn.Scopes = "wl.signin wl.basic wl.offline_access";
    liveSignIn.ClientId = "<< APP CLIENT ID >>";
}
```

In the XAML, the *SessionChanged* event is hooked up on the *SignInButton*. The implementation for this handler constructs a *LiveConnectClient* object from the event arguments. This event handler will be invoked first when the *SignInButton* is loaded, at which point there will be no session. After that, it will be invoked after the user has clicked the *SignInButton* to log in to Live and he has accepted the proposed scopes. At this point, you hook up the *GetCompleted* event and invoke the asynchronous *GetAsync* method. This calls the Windows Live REST API, with a specific entity request. In this case, you're retrieving the user data, as specified by the "me" identifier.

```
private void LiveSignIn_SessionChanged(
    object sender, LiveConnectSessionChangedEventArgs e)
{
    if (e.Session != null)
    {
        liveClient = new LiveConnectClient(e.Session);
        liveClient.GetCompleted += OnGetLiveData;
        liveClient.GetAsync("me", null);
    }
    else
    {
        liveClient = null;
        liveResult.Text = e.Error != null ? e.Error.ToString() : string.Empty;
    }
}
```

When the *GetAsync* call returns, the custom *OnGetLiveData* event handler is invoked. Assuming this was successful, you'll now have a set of user data that includes name, id, profile page link, birthday, employer, gender, emails, and so on. For your purposes, you're only interested in the name property, which you retrieve from the result set and set into the UI of the phone application.

```
private void OnGetLiveData(object sender, LiveOperationCompletedEventArgs e)
{
    if (e.Error == null)
    {
        object name;
        if (e.Result.TryGetValue("name", out name))
        {
            liveResult.Text = name.ToString();
        }
        else
        {
            liveResult.Text = "name not found";
        }
    }
    else
    {
        liveResult.Text = e.Error.ToString();
    }
}
```

SkyDrive

Figure 11-36 shows an enhanced version of this application (the *TestLive_Photos* solution in the sample code), which fetches photos from a SkyDrive album for the logged-in user.

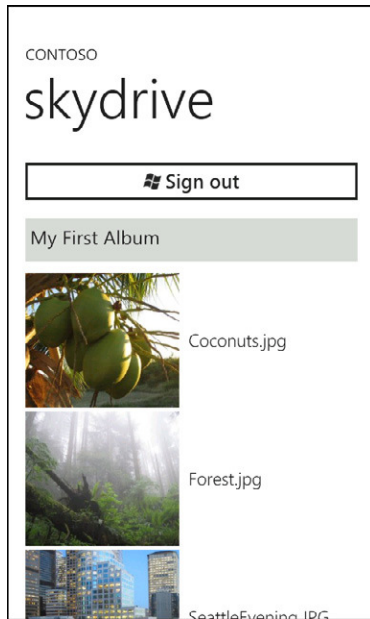


FIGURE 11-36 You can use the Windows Live API to download photos from SkyDrive.

The additional features are as follows. First, you define a *SkydrivePhoto* class to represent the key data that you want to fetch for each photo from SkyDrive. Note that the Live/SkyDrive REST API returns more properties (such as *Description*, *ID*, and so on), but you're only interested in the *Title* and *Url*. For other data, you might want to make this implement *INotifyPropertyChanged*, but that's not required for this application.

```
public class SkydrivePhoto
{
    public string Title { get; set; }
    public string Url { get; set; }
}
```

In the XAML, you add a *ListBox* with an *ItemTemplate* that includes an *Image* for the photo itself, plus a *TextBlock* for the *Title*. These two controls are data-bound to the SkyDrive properties.

```
<StackPanel x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <my:SignInButton
        x:Name="liveSignIn" SessionChanged="liveSignIn_SessionChanged" />
    <TextBox x:Name="albumName" Height="80" />
    <ListBox
        x:Name="PhotoList" Height="460" >
        <ListBox.ItemTemplate>
```

```

        <DataTemplate>
            <Grid>
                <Grid.RowDefinitions>
                    <RowDefinition Height="180"/>
                </Grid.RowDefinitions>
                <Grid.ColumnDefinitions>
                    <ColumnDefinition Width="200"/>
                    <ColumnDefinition Width="*/>
                </Grid.ColumnDefinitions>
                <Image
                    Grid.Row="0" Grid.Column="0" Width="200" Height="175"
                    Source="{Binding Url}" />
                <TextBlock
                    Grid.Row="0" Grid.Column="1" Text="{Binding Title}"
                    TextWrapping="Wrap" VerticalAlignment="Center"
                    Style="{StaticResource PhoneTextTitle3Style}" />
            </Grid>
        </DataTemplate>
    </ListBox.ItemTemplate>
</ListBox>
</StackPanel>

```

In the page code-behind, declare a *LiveConnectSession* field so that you can cache this between method calls. As before, you'll get the session value in the *SessionChanged* event handler. Previously, you needed to use this only in the one handler, but now you'll need it in a second handler. This is because you need to make two calls to retrieve SkyDrive data: one for the collection of albums, and a second call for the collection of photos within a given album. You also declare an *Observable Collection* of *SkydrivePhoto* objects. This will represent all the photos you download for the photo album that you target.

```

private LiveConnectClient liveClient;
private LiveConnectSession liveSession;
private ObservableCollection<SkydrivePhoto> photos =
    new ObservableCollection<SkydrivePhoto>();
public ObservableCollection<SkydrivePhoto> Photos
{
    get { return photos; }
}

```

In the *SessionChanged* event handler, modify the *GetAsync* call to fetch the collection of albums for this user.

```
liveClient.GetAsync("me/albums", null);
```

The implementation of your *OnGetLiveData* callback is rather different from the previous version. Instead of retrieving a simple string value, you're now fetching a collection of albums, indicated by the "data" prefix in the result set. In this example, you're interested only in the first album in the collection. Extract the name so that you can display it in the UI (in place of the user name that was displayed before). You also extract the ID because you need to pass this to the second *GetAsync* call, appending */files*, to fetch the collection of photos within this album.


```

private void OnGetLiveData(object sender, LiveOperationCompletedEventArgs e)
{
    if (e.Error == null)
    {
        if (e.Result.ContainsKey("data"))
        {
            List<object> data = (List<object>)e.Result["data"];
            IDictionary<string, object> album = (IDictionary<string, object>)data[0];
            albumName.Text = (string)album["name"];
            String albumId = (string)album["id"];
            LiveConnectClient albumClient = new LiveConnectClient(liveSession);
            albumClient.GetCompleted +=
                new EventHandler<LiveOperationCompletedEventArgs>(
                    albumClient_GetCompleted);
            albumClient.GetAsync(albumId + "/files", null);
        }
    }
}

```

Finally, in the second *GetCompleted* handler, extract the collection of photos, again identified by the “data” prefix. For each photo, you extract the name and source properties. The source property is the URL of the photo on SkyDrive.

```

void albumClient_GetCompleted(object sender, LiveOperationCompletedEventArgs e)
{
    if (e.Error == null)
    {
        List<object> data = (List<object>)e.Result["data"];
        foreach (IDictionary<string, object> item in data)
        {
            SkydrivePhoto photo = new SkydrivePhoto();
            photo.Title = (string)item["name"];
            photo.Url = (string)item["source"];
            photos.Add(photo);
        }
    }
}

```

Summary

This chapter examined the basic support that the application platform provides for connecting to the web. This included specifically the *WebClient* and *HttpWebRequest* classes, and the *WebBrowser* control. Note that version 7.1 introduces support for Sockets, which is discussed in Chapter 17, “Enhanced Connectivity Features.” You also looked at the many different ways that you can connect a phone application to web services, including WCF data services and Windows Azure hosted services. Finally, you looked at the additional support that is available for connecting to Bing Maps, bitly, Facebook, and Windows Live (including SkyDrive). As noted previously, it should be emphasized that this chapter should be read in conjunction with Chapter 13, which discusses security, including the security aspects of web connectivity.

Push Notifications

An important tenet of the user-focused application model in Windows Phone is that information on the phone can be continually refreshed so that it is always current. The traditional model for updating data on a mobile device is for software on the device to poll a remote server periodically to fetch, or *pull*, new data. The problem with any pull model is that the hit-rate is variable. That is, if the polling interval is small, some number of the polling operations will result in no new data; this consumes battery power for no benefit. Conversely, if the polling interval is large, there is a risk of losing some updates and the user viewing too much stale data. A *push* model, wherein the remote server pushes data to the phone only when it is updated, significantly improves the experience.

In this chapter, you will examine the Windows Phone 7 push notification model, which includes a server-side piece and a client-side piece that you need to build, plus a cloud service (built by Microsoft) for channeling notifications.

Architecture

For a push implementation to work, the server must know where to send the new data. It requires some way to identify each mobile device as an endpoint for updates. The model also more or less mandates some retry or queuing logic to allow for the circumstance in which the target device is offline or out of network. On top of that, there must be a way for a user to opt his phone in or out of receiving updates, potentially on a per-application basis.

There's a lot of infrastructure inherent in this model, which would be common across all services that want to push data to the phone. Fortunately, Microsoft provides the Microsoft Push Notification Service (MPNS), which handles many of the common server-side features of this model. Your server sends new data as simple HTTP messages to the MPNS, specifying which phones should receive the messages. The MPNS then takes care of propagating the messages and pushing them to all the target devices. Each client application running on the device that wants to receive notifications talks to the MPNS to establish an identity for the device. The overall behavior is illustrated in Figure 12-1.

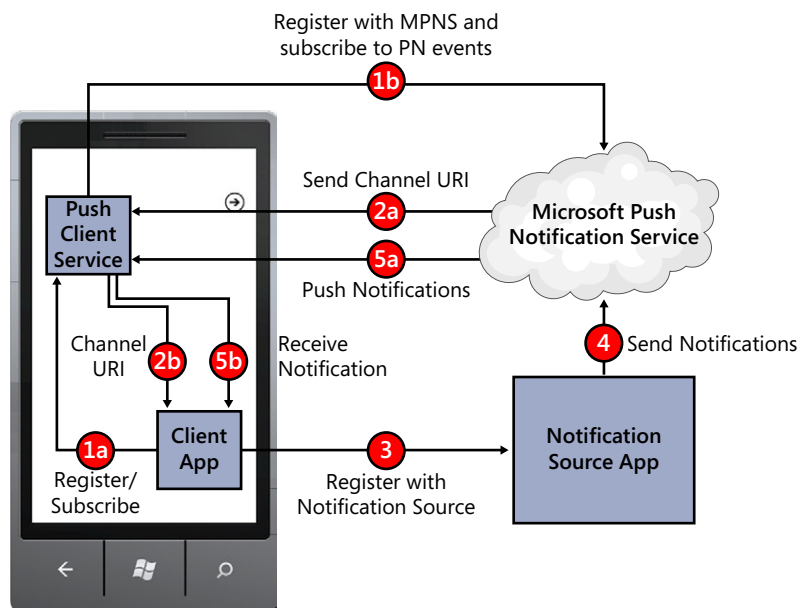


FIGURE 12-1 An overview of the Push Notification architecture.

The following list presents a bit more detail about the process:

1. Your phone application initiates the communication by registering with the MPNS. This is a simple matter of using *HttpNotificationChannel.Open*. Behind the scenes, this uses the Push Client service running on the device to communicate with the MPNS.
2. The MPNS returns a channel URI back to the phone. This URI serves as a unique identifier for the phone; it will be used by the server application as the target URL for a web request. Again, the MPNS actually sends this to the Push Client service on the device, which forwards it to your application.
3. To register for this server's notifications, the phone must send this device identifier (channel URI) to the server application (the application that will send the notifications). The server application can be anything that can make and receive web requests (web app, desktop app, and so on). For this to happen, the server application typically exposes a web service that the phone application can call to perform registration.
4. When it's ready to send a notification to a particular device, the server application makes an *HttpWebRequest* for the given channel URI (typically, it does this for multiple registered devices). This web request goes to the MPNS.
5. The MPNS then pushes the corresponding notification to the specified device (again, transparently through the Push Client service).

Whenever the server application sends a notification to the MPNS, it will receive a response that provides some information about the result, including the connection status of the target device and whether the notification was actively received or suppressed.

There are three types of push notification, which are described in Table 12-1. The payload for all types of notification must be no more than 1,024 bytes, and additional constraints apply to each type. There's a limit of one push notification channel per application, and this channel will be used for all types of notification. There's also a limit of 15 active push notification channels per device in version 7 (30 in version 7.1). The MPNS has a daily limit of 500 pushes per channel—this is per application/device combination, not 500 in total. Note that this limit also doesn't apply if you create an authenticated push channel, which you will read about in Chapter 13, "Security."

TABLE 12-1 Push Notification Types

Type	Description	Constraints	Typical Scenario
Tile	Handled by the phone OS and rendered on the start page when your application is pinned to the start page. The display includes three items: <i>Count</i> , <i>Title</i> , and <i>Background</i> image, all of which are specified in the Tile notification received on the phone. Each of the three items can update independently. Tile notifications will update the pinned tile, regardless of whether the application is currently running. The images must be either local to the phone application itself or specify a reachable HTTP URL.	Title can be any length, but only the first ~15 characters of the <i>Title</i> will be displayed. Images will be scaled to 173x173 pixels. They can be either JPG or PNG format. The <i>Count</i> is capped at 99; that is, you can send any number you like, but if you send a number higher than 99, the <i>Count</i> will be set to 99. Any remote image must be ≤80 KB, and must download in ≤30 seconds.	Status updates; for example, count of unread emails for an email client, current temperature for a weather application.
Toast	Include a <i>Title</i> and a message body (<i>Content</i>). If your application is not running or is obscured, the phone OS will display a popup toast at the top of the screen for 10 seconds, including both the title and the message. The user can tap the toast to launch the application. If your application is running and not obscured, then there is no default display and it's up to your application to handle the message, as appropriate.	Maximum ~40 characters <i>Title</i> , or ~47 characters <i>Content</i> , or ~41 characters <i>Title</i> + <i>Content</i> .	Breaking news, alerts.
Raw	No default visual display. With a raw push notification, you can send any arbitrary data (text or binary) to your application on the phone.	Can be received only when the application is running.	Arbitrary data for use in your application.

Figure 12-2 illustrates the UI elements of Toast and Tile notifications.

To build a solution that uses push notifications, you build two main pieces: the server-side application that generates and sends the notifications, and the client-side application that runs on the phone to receive and process incoming notifications. In fact, the client-side application is an optional piece, because you might send only tile notifications that do not require client-side code to process them. These are explored in the following sections. Note that the general security considerations for web services also apply to push notifications. These are not discussed in this chapter, but your push notification solution is not complete unless you factor in security. Chapter 13 discusses security across the board and includes specific guidelines for push notification security.

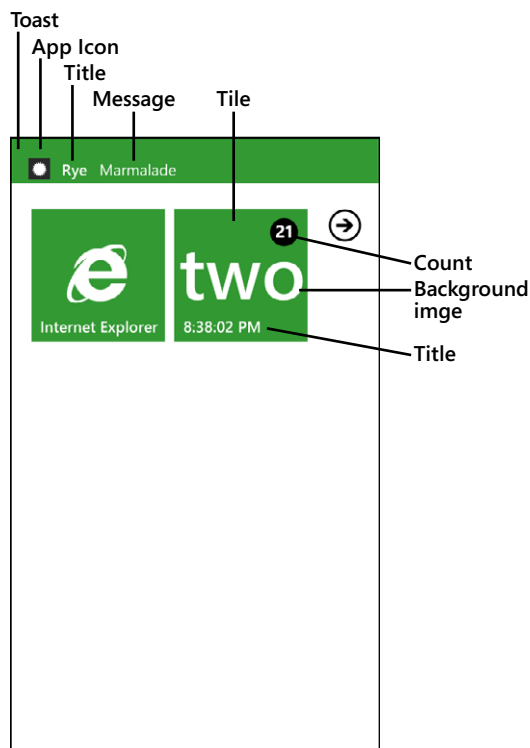


FIGURE 12-2 The Toast and Tile elements.

Push Notification Server

This example sends all three types of notification; it is a simple Windows Presentation Foundation (WPF) application that offers a user interface (UI) with which the user can enter suitable values for the various parts of the three notification types. This is the *Push_Simple\PnServer* solution in the sample code. The application provides a *Response* list with which the server reports on status and responses from notifications that have been sent. The values in the response report are the *WebResponse Status Code* and the values from the *X-DeviceConnectionStatus*, *X-SubscriptionStatus*, and *X-Notification Status* headers for the *WebResponse*, as shown in Figure 12-3.

Note that for all the message elements but one, the data is constructed entirely on the server. The exception is the "two.png" value shown in the screenshot. This is the path for an image file that will be used as the background image for a Tile notification. In this example, this is a relative path to a file that deploys as part of the phone application (although the path can also identify a remote URL for the image file).

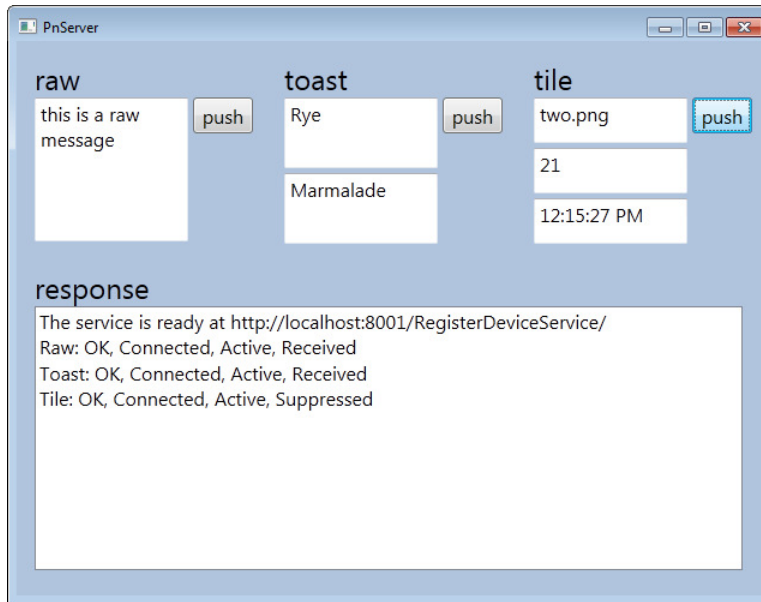


FIGURE 12-3 A Simple WPF Push Notification server.

There are two broad sets of functionality that you need to expose from your server: the code that generates and sends the notifications, and the code that allows client applications to register to receive notifications. For the latter, the standard approach is to expose a web service with *Register* (and *Unregister*) web methods. In your application, you define a simple *ServiceContract* named *IRegisterDeviceService* and implement it in a custom *RegisterDeviceService* class. In this class, you maintain a static collection of device URIs; this is the list of phones that register to receive notifications. You make the collection static so that it is accessible both to the server host application and the service object itself. The *Register* method simply adds the incoming *URI* to the collection, and the *Unregister* method removes it.

```
[ServiceContract]
public interface IRegisterDeviceService
{
    [OperationContract]
    void Register(Uri deviceUri);

    [OperationContract]
    void Unregister(Uri deviceUri);
}

public class RegisterDeviceService : IRegisterDeviceService
{
    private static Collection<Uri> devices = new Collection<Uri>();
    public static Collection<Uri> Devices
    {
        get { return devices; }
        set { devices = value; }
    }
}
```

```

public void Register(Uri deviceUri)
{
    if (deviceUri != null && !Devices.Contains(deviceUri))
    {
        Devices.Add(deviceUri);
    }
}

public void Unregister(Uri deviceUri)
{
    Devices.Remove(deviceUri);
}
}

```

Here's the app.config, which is entirely taken up with service configuration:

```

<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.serviceModel>
    <behaviors>
      <serviceBehaviors>
        <behavior name="">
          <serviceMetadata httpGetEnabled="true" />
          <serviceDebug includeExceptionDetailInFaults="true" />
        </behavior>
      </serviceBehaviors>
    </behaviors>
    <services>
      <service name="PnServer.RegisterDeviceService">
        <endpoint
          address=""
          binding="basicHttpBinding"
          contract="PnServer.IRegisterDeviceService">
          <identity>
            <dns value="localhost" />
          </identity>
        </endpoint>
        <endpoint
          address="mex"
          binding="mexHttpBinding"
          contract="IMetadataExchange" />
      </service>
    </services>
  </system.serviceModel>
</configuration>

```


In the service host application, the *MainWindow* has a field for the WCF *ServiceHost* object, and string templates for the toast and tile messages. Each type of notification is formatted as XML, with different elements and attributes for the different types of notification.

```
private ServiceHost serviceHost;

const String toastMessageFormat =
    "<?xml version=\"1.0\" encoding=\"utf-8\"?>" +
    "<wp:Notification xmlns:wp=\"WPNotification\">" +
    "    <wp:Toast>" +
    "        <wp:Text1>{0}</wp:Text1>" +
    "        <wp:Text2>{1}</wp:Text2>" +
    "    </wp:Toast>" +
    "</wp:Notification>";

const String tileMessageFormat =
    "<?xml version=\"1.0\" encoding=\"utf-8\"?>" +
    "<wp:Notification xmlns:wp=\"WPNotification\">" +
    "    <wp:Tile>" +
    "        <wp:BackgroundImage>{0}</wp:BackgroundImage>" +
    "        <wp:Count>{1}</wp:Count>" +
    "        <wp:Title>{2}</wp:Title>" +
    "    </wp:Tile>" +
    "</wp:Notification>";
```

In the *MainWindow* constructor, you instantiate the *ServiceHost* and start the service.

```
public MainWindow()
{
    InitializeComponent();

    serviceHost = new ServiceHost(typeof(RegisterDeviceService));
    try
    {
        serviceHost.Open();
        ShowStatus(
            "The service is ready at " + serviceHost.BaseAddresses[0]);
    }
    catch (Exception ex)
    {
        ShowStatus(ex.ToString());
        serviceHost = null;
    }
}
```

There are button *Click* handlers to trigger sending each of the three notification types. Each of these handlers invokes a custom *SendNotification* method, which has all the common code for sending any notification. The parameters that you pass to this method distinguish the different types and therefore govern how the XML payload is ultimately composed. Note that raw messages are type 3, toasts are type 2, and tiles are type 1. It is common to define an enum for these values, but the raw numeric values are retained here to make it clear that this is what the system uses under the covers.

```

private void sendRaw_Click(object sender, RoutedEventArgs e)
{
    ShowStatus(SendNotification(rawMessage.Text, 3));
}

private void sendToast_Click(object sender, RoutedEventArgs e)
{
    String message = String.Format(
        toastMessageFormat, toastTitle.Text, toastMessage.Text);
    ShowStatus(SendNotification(message, 2));
}

private void sendTile_Click(object sender, RoutedEventArgs e)
{
    tileTitle.Text = DateTime.Now.ToLongTimeString();
    String message = String.Format(
        tileMessageFormat, tileBackground.Text,
        tileCount.Text, tileTitle.Text);
    ShowStatus(SendNotification(message, 1));
}

```

All the heavy lifting of sending notification messages is done in a centralized method. This iterates through the collection of device URIs and opens an *HttpRequest* for each one, with the payload formatted as necessary for each message type. For toast and tile notifications, you need to add the *X-WindowsPhone-Target* header; this is not used for raw notifications. The toast target specifier is "toast", whereas the tile target specifier is "token". If you're wondering why there's a naming discrepancy, it's because prior to the first release of Windows Phone, tiles used to be called tokens, internally. Once you've composed the appropriate notification payload, you send the message and fetch the *HttpResponse* that results. You return payload and header information from the response as a collection of strings.

```

private List<String> SendNotification(String message, short notificationClass)
{
    List<String> responses = new List<String>();
    byte[] payload = Encoding.UTF8.GetBytes(message);
    if (payload.Length > maxPayload)
    {
        responses.Add(String.Format(
            "The message must be <= {0} bytes: {1}", maxPayload, message));
    }
    else
    {
        if (RegisterDeviceService.Devices.Count == 0)
        {
            responses.Add("No devices");
        }
        else
        {

```

```

foreach (Uri uri in RegisterDeviceService.Devices)
{
    // Create an HTTP web request for each device Uri.
    HttpWebRequest request = (HttpWebRequest)WebRequest.Create(uri);
    request.Method = WebRequestMethods.Http.Post;
    request.ContentLength = payload.Length;
    request.ContentType = "text/xml";

    // Add message headers.
    request.Headers.Add("X-MessageID", Guid.NewGuid().ToString());
    request.Headers.Add("X-NotificationClass", notificationClass.ToString());
    if (notificationClass == 1)
    {
        request.Headers.Add("X-WindowsPhone-Target", "token");
    }
    else if (notificationClass == 2)
    {
        request.Headers.Add("X-WindowsPhone-Target", "toast");
    }

    // Send the message.
    using (Stream requestStream = request.GetRequestStream())
    {
        requestStream.Write(payload, 0, payload.Length);
    }

    // Fetch the response.
    HttpWebResponse webResponse;
    try
    {
        webResponse = (HttpWebResponse)request.GetResponse();
        responses.Add(String.Format("{0}: {1}, {2}, {3}, {4}",
            notificationClass == 1 ? "Tile" :
            notificationClass == 2 ? "Toast" : "Raw",
            webResponse.StatusCode,
            webResponse.Headers["X-DeviceConnectionStatus"],
            webResponse.Headers["X-SubscriptionStatus"],
            webResponse.Headers["X-NotificationStatus"]));
    }
    catch (WebException ex)
    {
        webResponse = (HttpWebResponse)ex.Response;
        responses.Add(ex.Message);
    }
}
}
return responses;
}

```

To report these response strings (or other status), simply add the strings to the *ListBox*.

```
private void ShowStatus(List<String> responses)
{
    foreach (String response in responses)
    {
        ShowStatus(response);
    }
}

private void ShowStatus(String response)
{
    responseList.Items.Add(response);
}
```

Push Notification Client

In the client (the *Push_Simple\PnClient* solution in the sample code), raw and toast notifications are handled by the application, if the application is running and not obscured, as shown in Figure 12-4.

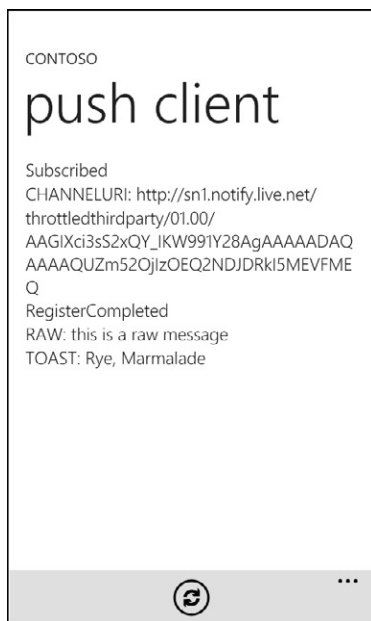


FIGURE 12-4 The Push Notification client.

If the application is not running, toast notifications are rendered as popups, as shown in Figure 12-5. The OS handles tile notifications.

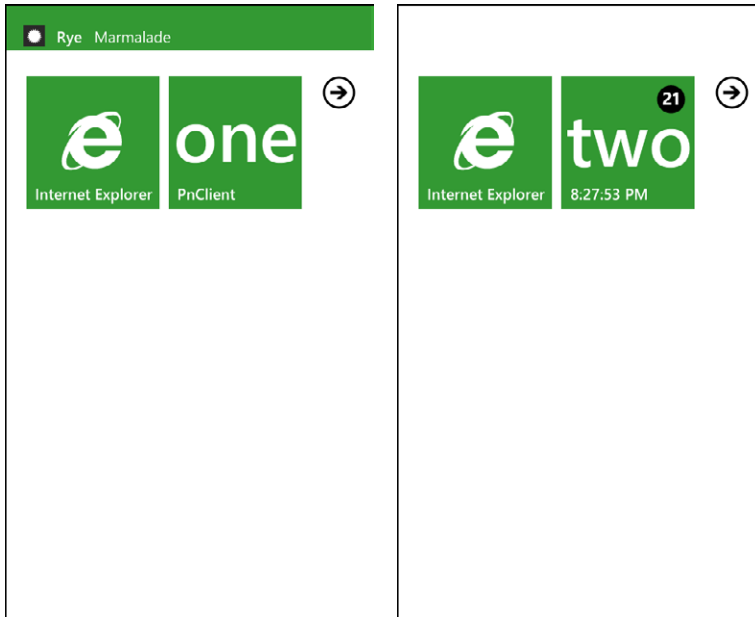


FIGURE 12-5 Your application can receive both toast and tile push notifications.

In the client's `mainpage.xaml.cs`, declare fields for the channel name (an arbitrary string), the `HttpNotificationChannel` for working with the MPNS, and a client-side proxy to the server application's web service. You also have an `ObservableCollection<T>` to hold all the message strings. This is data-bound to the `ListBox` in the UI.

```
private String channelName = "Contoso Notification Channel";
private HttpNotificationChannel channel;
private RegisterDeviceServiceClient serviceClient;
private Uri channelUri;
private bool isRegistered;
private ObservableCollection<String> notifications;
public ObservableCollection<String> Notifications
{
    get { return notifications; }
    private set { }
}
}
```

You now need to override `OnNavigatedTo` to perform initialization. You must instantiate the WCF service proxy and handle the event that's raised when your async call to the `Register` method returns. Later in the code, you will provide a mechanism for the user to unregister, and you need to hook up the callback for the asynchronous unregister event. The `OnNavigatedTo` is a suitable place to do this. This will ensure that you don't hook up the event multiple times. You must also subscribe to the push client.

```
protected override void OnNavigatedTo(NavigationEventArgs e)
{
    if (serviceClient == null)
    {
        serviceClient = new RegisterDeviceServiceClient();
        serviceClient.RegisterCompleted += serviceClient_RegisterCompleted;
        serviceClient.UnregisterCompleted += serviceClient_UnregisterCompleted;
        SubscribeToNotifications();
    }
}
```

To subscribe to notifications, find the named *HttpNotificationChannel*—or create it if it doesn't exist—and open it. You must handle the *ChannelUriUpdated* event so that you can get the Channel URI from MPNS. You also handle the other interesting channel events, *ShellToastNotification* and *HttpNotification*, for toast and raw notifications, respectively. If the channel hasn't already been bound to tile and toast notifications, go ahead and bind it now by using the *BindToShellTile* and *BindToShellToast* APIs.

```
private Uri SubscribeToNotifications()
{
    if (channel == null)
    {
        channel = HttpNotificationChannel.Find(channelName);

        if (channel == null)
        {
            channel = new HttpNotificationChannel(channelName);
            channel.ChannelUriUpdated += channel_ChannelUriUpdated;
            channel.Open();
        }
        else
        {
            if (!isRegistered)
            {
                serviceClient.RegisterAsync(channelUri);
            }
        }

        channel.ShellToastNotificationReceived += channel_ShellToastNotificationReceived;
        channel.HttpNotificationReceived += channel_HttpNotificationReceived;
        Notifications.Add("Subscribed");

        if (!channel.IsShellTileBound)
        {
            channel.BindToShellTile();
        }
        if (!channel.IsShellToastBound)
        {
            channel.BindToShellToast();
        }
    }
    return channel.ChannelUri;
}
```

When you receive a *ChannelUriUpdate* event, you need to call the server application's web service in order to register this client, passing it the Channel URI (device identifier) you've been given by the MPNS.

```
private void channel_ChannelUriUpdated(
    object sender, NotificationChannelUriEventArgs e)
{
    channelUri = e.ChannelUri;
    String message = String.Format("CHANNELURI: {0}", channelUri);
    Dispatcher.BeginInvoke(() => Notifications.Add(message));
    serviceClient.RegisterAsync(channelUri);
}
```

Note that you should allow for the possibility of a server restart while the client has a channel open. Of course, your client application won't know when this has happened, but you can mitigate it somewhat by always registering the channel during *OnNavigatedTo* (as in the preceding code), by periodically refreshing the channel, or by providing the user with a UI-driven mechanism for refreshing. This last approach is the least useful because most users won't know why they need to do this; however, it is a useful feature during testing. To this end, you also provide an App Bar button to toggle registration. Regardless of whether the registration is triggered by the UI or programmatically in your handler for the *ChannelUriUpdate* event, you simply cache the registered state and report status in the UI.

```
private void register_Click(object sender, EventArgs e)
{
    if (channelUri != null)
    {
        if (!isRegistered)
        {
            serviceClient.RegisterAsync(channelUri);
        }
        else
        {
            serviceClient.UnregisterAsync(channelUri);
        }
    }
}

private void serviceClient_RegisterCompleted(object sender, AsyncCompletedEventArgs e)
{
    if (e.Error == null)
    {
        isRegistered = true;
        Dispatcher.BeginInvoke(() => Notifications.Add("RegisterCompleted"));
    }
    else
    {
        Dispatcher.BeginInvoke(() => Notifications.Add(e.Error.Message));
    }
}
```

```

private void serviceClient_UnregisterCompleted(object sender, AsyncCompletedEventArgs e)
{
    if (e.Error == null)
    {
        isRegistered = false;
        Dispatcher.BeginInvoke(() => Notifications.Add("UnregisterCompleted"));
    }
    else
    {
        Dispatcher.BeginInvoke(() => Notifications.Add(e.Error.Message));
    }
}

```

When you receive a push notification (raw or toast) event, you also display it in the UI, appropriately dispatched to the UI thread. To retrieve the data, you dig into the *Collection* property on the *NotificationEventArgs* parameter that is passed into the event handler. Notification data is sent in this collection, which is a dictionary, so the elements are key-value pairs. In this application, there will be only one pair of data, although you will allow for it being a null element (in which case, you use an empty string).

```

private void channel_HttpNotificationReceived(
    object sender, HttpNotificationEventArgs e)
{
    byte[] bytes;
    using (Stream stream = e.Notification.Body)
    {
        bytes = new byte[stream.Length];
        stream.Read(bytes, 0, (int)stream.Length);
    }
    String rawMessage = Encoding.UTF8.GetString(bytes, 0, bytes.Length);
    String message = String.Format("RAW: {0}", rawMessage);
    Dispatcher.BeginInvoke(() => Notifications.Add(message));
}

private void channel_ShellToastNotificationReceived(
    object sender, NotificationEventArgs e)
{
    String title = e.Collection.Values.First();
    String rawMessage =
        e.Collection.Values.Skip(1).FirstOrDefault() ?? String.Empty;
    String message = String.Format("TOAST: {0}, {1}", title, rawMessage);
    Dispatcher.BeginInvoke(() => Notifications.Add(message));
}

```

The preceding code illustrates all the core requirements for the client and server applications. However, there are a few additional features that you could incorporate, to improve robustness and usability; these are described in the following sections.

Additional Server Features

On the server side, you can use the following enhancements:

- Batching intervals, which you can use to group notifications in batches.
- The *XmlWriter* or *XDocument* classes, for building the notification payload.
- Tracking, to monitor additional notification response information.

These features are described in the following sections.

Batching Intervals

The server currently uses strings, 1, 2, 3 to identify the *X-NotificationClass*, and these must be added to the request headers. However, the values 1, 2, 3 really correspond to batching indicators, and there are nine possible values. There are three categories, as shown in Table 12-2.

TABLE 12-2 Notification Batching Intervals

Value	Notification Type	Meaning
1	Tile	Send immediately
2	Toast	Send immediately
3	Raw	Send immediately
11	Tile	Batch and send within 450 seconds (7.5 minutes)
12	Toast	Batch and send within 450 seconds (7.5 minutes)
13	Raw	Batch and send within 450 seconds (7.5 minutes)
21	Tile	Batch and send within 900 seconds (15 minutes)
22	Toast	Batch and send within 900 seconds (15 minutes)
23	Raw	Batch and send within 900 seconds (15 minutes)

This allows the MPNS to batch notifications together, including from multiple applications. The primary purpose of this is to maintain an optimal balance of user experience (UX) on the phone; sending notifications in batches will improve battery performance because it makes maximum use of the network while it is up, rather than bringing it up for every single notification. In the following enhancement (the *Push_Additional\PnServer* solution in the sample code), there is an additional *HeaderedContentControl* that includes a *ComboBox* with which the user can select one of the three batching intervals. This will be applied to all notifications until it is changed.

```
<HeaderedContentControl
  Grid.Row="1" Grid.ColumnSpan="3" Header="batching interval">
  <Grid>
    <ComboBox Name="batchList" Width="100" Height="30">
      <ComboBoxItem Content="immedate" Tag="0"/>
      <ComboBoxItem Content="450 sec" Tag="10"/>
      <ComboBoxItem Content="900 sec" Tag="20"/>
    </ComboBox>
  </Grid>
</HeaderedContentControl>
```

The *SendNotification* method is updated to allow for the batching interval. Specifically, change this line:

```
request.Headers.Add("X-NotificationClass", notificationClass.ToString());
```

to this:

```
int batch = Int16.Parse(
    ((ComboBoxItem)batchList.SelectedItem).Tag.ToString())
    + notificationClass;
request.Headers.Add("X-NotificationClass", batch.ToString());
```

Be aware that the MPNS does not provide a true end-to-end confirmation that the notification was delivered. In particular, if you batch up your notifications, you will still get an immediate response notification based on the last known state of the target device and application, even though the notification might not be sent until up to 15 minutes after the fact.

XML Payload

Building the XML payload from string templates is useful from a developer's perspective; it helps to make it obvious what the XML schema is and what a typical payload for each notification type looks like. However, it is a somewhat error-prone approach. For example, it is very fragile in the face of replacement values that contain reserved characters such as "<", and so on. A more robust approach is to construct the XML in code by using *XmlWriter* methods *WriteStartElement*, *WriteEndElement*, and so on. Alternatively (and preferably), you can use *XDocument*.

Using this approach, you can eliminate the string templates altogether. Then, to send a toast notification, you change this method:

```
private void sendToast_Click(object sender, RoutedEventArgs e)
{
    String message = String.Format(
        toastMessageFormat, toastTitle.Text, toastMessage.Text);
    ShowStatus(SendNotification(message, 2));
}
```

to the implementation shown in the following, using *XmlWriter*:

```
private void sendToast_Click(object sender, RoutedEventArgs e)
{
    MemoryStream stream = new MemoryStream();
    XmlWriterSettings settings = new XmlWriterSettings();
    settings.Encoding = new UTF8Encoding(false);
    XmlWriter writer = XmlWriter.Create(stream, settings);

    writer.WriteStartDocument();
    writer.WriteStartElement("wp", "Notification", "WPNotification");
    writer.WriteStartElement("wp", "Toast", "WPNotification");

    writer.WriteStartElement("wp", "Text1", "WPNotification");
    writer.WriteValue(toastTitle.Text);
    writer.WriteEndElement();
```

```

        writer.WriteStartElement("wp", "Text2", "WPNotification");
        writer.WriteValue(toastMessage.Text);
        writer.WriteEndElement();

        writer.WriteEndElement();
        writer.WriteEndDocument();
        writer.Flush();

        ShowStatus(SendNotification(stream.ToArray(), 2));
    }

```



Note It's important to construct your own *UTF8Encoding* object so that you can specify that the byte order mark (BOM) should be eliminated.

To send a tile notification, you can make similar changes to the *sendTile Click* handler. The earlier version of our custom *SendNotification* method took in a string as the message parameter and converted it to a byte array. Using *XmlWriter* (or *XDocument*) makes it easy to provide a byte array in the first place, so you can simplify the *SendNotification* method to take this as a parameter instead of the original message string.

```

private List<String> SendNotification(byte[] payload, int notificationClass)
{
    List<String> responses = new List<string>();
    //byte[] payload = Encoding.UTF8.GetBytes(message);

    ...unchanged code omitted for brevity.
}

```

You can take this one step further by using *XDocument* instead of the more traditional *XmlWriter* approach. Doing so does mean pulling in the LINQ assemblies, but you're likely to be doing that anyway for other reasons. Using *XDocument*, you can rewrite the cumbersome *XmlWriter* code for toasts and tiles as shown in the following example. You can see this at work in the *Push_Better\PnServer* solution in the sample code.

```

private static readonly XNamespace WpNs = "WPNotification";

private void sendToast_Click(object sender, RoutedEventArgs e)
{
    XDocument doc = new XDocument();
    doc.Add(
        new XElement(WpNs + "Notification",
            new XAttribute(XNamespace.Xmlns + "wp", WpNs.NamespaceName),
            new XElement(WpNs + "Toast",
                new XElement(WpNs + "Text1", toastTitle.Text),
                new XElement(WpNs + "Text2", toastMessage.Text))));

    UTF8Encoding encoding = new UTF8Encoding(false);
    byte[] payload = encoding.GetBytes(doc.ToString());
    ShowStatus(SendNotification(payload, 2));
}

```

```

private void sendTile_Click(object sender, RoutedEventArgs e)
{
    XDocument doc = new XDocument();
    doc.Add(
        new XElement(WpNs + "Notification",
            new XAttribute(XNamespace.Xmlns + "wp", WpNs.NamespaceName),
            new XElement(WpNs + "Tile",
                new XElement(WpNs + "BackgroundImage", tileBackground.Text),
                new XElement(WpNs + "Count", tileCount.Text),
                new XElement(WpNs + "Title", tileTitle.Text))));

    UTF8Encoding encoding = new UTF8Encoding(false);
    byte[] payload = encoding.GetBytes(doc.ToString());
    ShowStatus(SendNotification(payload, 1));
}

```

Response Information

The application has logic to test for connected subscribers, so for each notification sent, the results are most likely to be *status code = OK*, *connection status = Connected*, *subscription status = Active*. The notification status will be either *Received* or *Suppressed*. Raw notifications are received if the application is running; otherwise, it is suppressed. Tile notifications are received if the application is pinned to the start menu and is not running; otherwise, it is suppressed. Toast notifications are received whether the application is running or not, and the platform will show toast UI if the application is not in the foreground.

So far, you have been reporting the most useful notification response information in the server application's UI. In a more sophisticated application, you might well want to track other information such as the message ID and timestamp. A full list of the headers, with sample values, is shown in the example code that follows. These are documented at [http://msdn.microsoft.com/en-us/library/ff941100\(VS.92\).aspx](http://msdn.microsoft.com/en-us/library/ff941100(VS.92).aspx). Your server application will realistically not be UI-driven; although it might have some management UI, and you would most likely track the notifications sent and responses received in a database (or perhaps in Windows Azure table storage). Most of the potentially useful information is in the *m_HttpResponseHeaders* member of the *HttpWebResponse* object that you get back in the server after sending a notification (as listed in the following example). Be aware that some of these are not specific to push notifications.

```

X-DeviceConnectionStatus: Connected
X-NotificationStatus: Received
X-SubscriptionStatus: Active
X-MessageID: 82ce700e-b409-4aeb-bb6c-235edfd01495
ActivityId: 7ebf51ac-3fb6-4f55-b8c8-4dd5504408d0
X-Server: SN1MPNSM020
Content-Length: 0
Cache-Control: private
Date: Tue, 22 Nov 2011 19:25:54 GMT
Server: Microsoft-IIS/7.5
X-AspNet-Version: 4.0.30319
X-Powered-By: ASP.NET

```

Additional Client Features

On the client side, there are several enhancements that you should consider layering on top of the basic push features, including the following:

- Persistent client settings, with which the application can keep track of its push status in the face of user navigation.
- Handling the special *ErrorOccurred* push notification event.
- Providing a mechanism for the user to opt in or out of push notifications for your application.
- Implementing a custom viewmodel for push settings.

Persistent Client Settings

Recall that toast and tile push notifications are sent even when the application is not running, and both can be used to launch the application. This means that the application needs to keep track of its MPNS subscription/registration status if it wants to retain the option to unsubscribe or unregister. One way to deal with this is to persist the channel URI and registration state in *IsolatedStorageSettings*, writing them out in the *OnNavigatedFrom* override. Then, in the *OnNavigated* override, if you're navigating back to the application after you've already registered the device with MPNS, you can retrieve the Channel URI from the application settings. You can see this at work in the *Push_Additional\PnClient* solution in the sample code. However, keep in mind that this is only marginally useful, and you should generally consider it best practice to simply *Find* or *Open* the channel on each launch, so that you don't need to persist this information.

```
protected override void OnNavigatedTo(NavigationEventArgs e)
{
    Uri uri;
    if (IsolatedStorageSettings.ApplicationSettings.TryGetValue<Uri>
        ("ChannelUri", out uri))
    {
        channelUri = uri;
    }
    bool reg;
    if (IsolatedStorageSettings.ApplicationSettings.TryGetValue<bool>
        ("IsRegistered", out reg))
    {
        isRegistered = reg;
    }

    if (serviceClient == null)
    {
        serviceClient = new RegisterDeviceServiceClient();
        serviceClient.RegisterCompleted +=
            new EventHandler<AsyncCompletedEventArgs>(serviceClient_RegisterCompleted);
        SubscribeToNotifications();
    }
}
```

In *OnNavigatedFrom*, you perform the corresponding “save” operation and persist the Channel URI and registration status settings to isolated storage.

```
protected override void OnNavigatedFrom(NavigationEventArgs e)
{
    IsolatedStorageSettings.ApplicationSettings["ChannelUri"] = channelUri;
    IsolatedStorageSettings.ApplicationSettings["IsRegistered"] = isRegistered;
    IsolatedStorageSettings.ApplicationSettings.Save();
}
```

The *ErrorOccurred* Event

So far, your code has used a reasonable level of try/catch exception handling, including for standard HTTP web error codes. However, the MPNS will also report specific errors that you can consume in your application. To do this, in the *SubscribeToNotifications* method, add an event sink for the *ErrorOccurred* event, as shown in the example that follows. You can see this at work in the *Push_Better\PnClient* solution in the sample code.

```
if (channel == null)
{
    channel = new HttpNotificationChannel(channelName);
    channel.ChannelUriUpdated += channel_ChannelUriUpdated;
    channel.ErrorOccurred += channel_ErrorOccurred;
    channel.Open();
}
```

For testing purposes and for a simple implementation, the event handler could report the error—or rather, a suitably user-friendly version of the error message—to the screen. In a more sophisticated application, you would want to look at the *ErrorType* property (and possibly the *ErrorCode* property) and take the appropriate corrective action. For example, if you get a *ChannelOpenFailed* or *PayloadFormatError*, the channel is now useless, so you should clean it up and optionally recreate it. On the other hand, if you get bad data from the server or too many notifications in a short span of time, there’s really not much you can do on the client, beyond reporting. If you get a *PowerLevelChanged* event, this is an informative warning that the server will stop sending tile and toast notifications. If phone power drops to critical level, then MPNS will stop sending even raw notifications to the device. The obvious purpose of this event is to reduce power consumption on the phone when the battery is low.

```
private void channel_ErrorOccurred(object sender, NotificationChannelErrorEventArgs e)
{
    String description = String.Empty;
    switch (e.ErrorType)
    {
        case ChannelErrorType.ChannelOpenFailed:
        case ChannelErrorType.PayloadFormatError:
            channel.Close();
            channel.Dispose();
            channel = null;
            SubscribeToNotifications();
            description = "Channel closed and re-initialized.";
    }
}
```

```

        break;
    case ChannelErrorType.MessageBadContent:
        description = "Bad data received from server.";
        break;
    case ChannelErrorType.NotificationRateTooHigh:
        description = "Too many notifications received.";
        break;
    case ChannelErrorType.PowerLevelChanged:
        if (e.ErrorAdditionalData == (int)ChannelPowerLevel.LowPowerLevel)
        {
            description =
                "No more toast or tile notifications will be "
                + "received until power levels are restored.";
        }
        else if
            (e.ErrorAdditionalData ==
             (int)ChannelPowerLevel.CriticalLowPowerLevel)
        {
            description =
                "No notifications of any kind will be received"
                + "until power levels are restored.";
        }
        break;
    }
    Dispatcher.BeginInvoke(() => Notifications.Add(
        String.Format("ERROR: {0} - {1}", e.Message, description)));
}

```

User Opt-In/Out

The marketplace certification requirements include two items that are specific to push notifications. The first time your application uses the *BindToShellToast* method, you must ask the user for explicit permission to receive toast notifications. You must also provide a UI mechanism with which the user can turn off toast notifications at any time subsequently. A simple example of the first requirement is shown in Figure 12-6. This feature is also implemented in the *Push_Better\PnClient* solution in the sample code.

This is because toast notifications use the same alert mechanism as other system notifications, such as incoming phone calls and incoming SMS messages. These alerts are executed at idle-level priority; that is, they will be displayed immediately, regardless of anything else the user is doing on the phone at the time, so long as the CPU is not at maximum utilization. The user must be given the choice as to whether she considers your application's notifications to warrant this high priority. In addition, notifications consume battery power, and even though the MPNS itself will throttle the rate at which notifications are sent, any potentially excessive use of battery power should also be under the user's control.

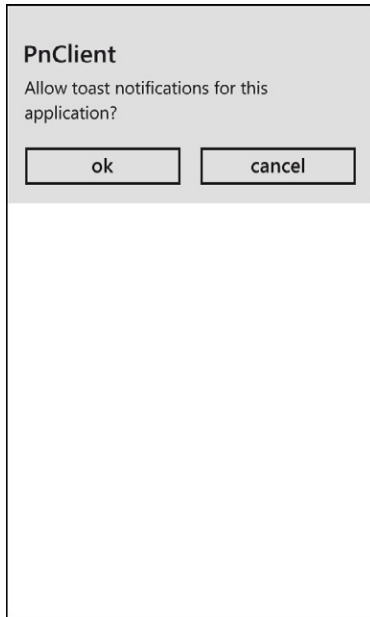


FIGURE 12-6 The prompt to accept or decline toast notifications.

To accommodate this requirement, you need to prompt the user, typically with a *MessageBox*, and then persist the user's choice. You can enhance the client application with a couple of additional *bool* fields to record whether the user wants to allow toasts and whether you've already asked her once. These need to be included in the persistent application settings.

```
protected override void OnNavigatedTo(NavigationEventArgs e)
{
    bool push;
    if (IsolatedStorageSettings.ApplicationSettings.TryGetValue<bool>
        ("IsToastOk", out push))
    {
        isToastOk = push;
    }
    bool prompted;
    if (IsolatedStorageSettings.ApplicationSettings.TryGetValue<bool>
        ("ToastPrompted", out prompted))
    {
        toastPrompted = prompted;
    }
    ...unchanged code omitted for brevity.
}

protected override void OnNavigatedFrom(NavigationEventArgs e)
{
    IsolatedStorageSettings.ApplicationSettings["IsToastOk"] = isToastOk;
    IsolatedStorageSettings.ApplicationSettings["ToastPrompted"] = toastPrompted;
    IsolatedStorageSettings.ApplicationSettings.Save();
}
```


Some applications use only toast notifications. Also, some applications require both toast notifications and either raw or tile notifications. This means that the user's choice about allowing toasts might determine whether your application uses notifications at all. However, it is more common to keep the different types of notification separate. In the following example, you go ahead with raw and tile notifications, regardless, but only use toasts if the user has explicitly agreed to them.

```
private Uri SubscribeToNotifications()
{
    ...unchanged code omitted for brevity.
    if (!channel.IsShellToastBound)
    {
        if (!isToastOk && !toastPrompted)
        {
            MessageBoxResult pushPrompt =
                MessageBox.Show(
                    "Allow toast notifications for this application?",
                    "PnClient",
                    MessageBoxButton.OKCancel);
            toastPrompted = true;
            if (pushPrompt == MessageBoxResult.OK)
            {
                isToastOk = true;
                channel.BindToShellToast();
            }
        }
    }
    return channel.ChannelUri;
}
```

Implementing a Push ViewModel

You are only required to ask the user about toast notifications once; in fact, you probably don't want to ask him more than once. On the other hand, you are also required to offer a mechanism by which the user can change his decision at any later stage. This more or less mandates a settings page of some kind. As soon as you implement a settings page, the limitations of the simple client implementation you've been working with so far become more obvious, specifically because you now need to access connection information across at least two pages. The classic design solution here is to encapsulate all the connection information into a viewmodel class and declare a public property of that type in the *App* class, where it will be accessible to all pages in the application.

Here's an example (this is the *Push_ViewModel\PnClient* solution in the sample code) wherein the application class declares a static *PushViewModel* property, and invokes the *LoadContext* and *SaveContext* methods in the appropriate lifecycle events.

```

public partial class App : Application
{
    private static PushViewModel push = new PushViewModel();
    public static PushViewModel Push
    {
        get { return push; }
    }

    private void Application_Launching(object sender, LaunchingEventArgs e)
    {
        push.LoadContext();
    }

    private void Application_Activated(object sender, ActivatedEventArgs e)
    {
        push.LoadContext();
    }

    private void Application_Deactivated(object sender, DeactivatedEventArgs e)
    {
        push.SaveContext();
    }

    private void Application_Closing(object sender, ClosingEventArgs e)
    {
        push.SaveContext();
    }
}

```

All of the connection-related fields and properties are moved from the *MainPage* class to the *PushViewModel* class, along with all of the server registration and notification subscription methods. Most of these methods are taken wholesale from the original implementation in *MainPage*. There are a couple of interesting modifications: notice that the *LoadContext* includes both loading of settings from isolated storage, and also the registration and subscription calls. These are moved from *MainPage.OnNavigatedTo*. The *SaveContext* code is moved from *MainPage.OnNavigatedFrom*. The registration code is moved from the App Bar *Click* handler to a new *Register* method in the *PushViewModel* class.

```

public class PushViewModel : INotifyPropertyChanged
{
    private String channelName = "Contoso Notification Channel";
    private HttpNotificationChannel channel;
    private RegisterDeviceServiceClient serviceClient;
    private Uri channelUri;
    private bool isRegistered;
    private bool toastPrompted;

    private ObservableCollection<String> notifications = new ObservableCollection<String>();
    public ObservableCollection<String> Notifications
    {
        get { return notifications; }
    }
}

```

```

public void LoadContext()
{
    bool prompted;
    if (IsolatedStorageSettings.ApplicationSettings.TryGetValue<bool>
        ("ToastPrompted", out prompted))
    {
        toastPrompted = prompted;
    }
    bool push;
    if (IsolatedStorageSettings.ApplicationSettings.TryGetValue<bool>
        ("IsToastOk", out push))
    {
        isToastOk = push;
    }

    if (serviceClient == null)
    {
        serviceClient = new RegisterDeviceServiceClient();
        serviceClient.RegisterCompleted += serviceClient_RegisterCompleted;
        serviceClient.UnregisterCompleted += serviceClient_UnregisterCompleted;
        SubscribeToNotifications();
    }
}

public void SaveContext()
{
    IsolatedStorageSettings.ApplicationSettings["ToastPrompted"] = toastPrompted;
    IsolatedStorageSettings.ApplicationSettings["IsToastOk"] = isToastOk;
    IsolatedStorageSettings.ApplicationSettings.Save();
}

private Uri SubscribeToNotifications()
{
    {
        ...original code moved unchanged from MainPage class.
    }

    private void channel_ChannelUriUpdated(
        object sender, NotificationChannelUriEventArgs e)
    {
        {
            ...original code moved unchanged from MainPage class.
        }

        private void channel_HttpNotificationReceived(
            object sender, HttpNotificationEventArgs e)
        {
            {
                ...original code moved unchanged from MainPage class.
            }

            private void channel_ShellToastNotificationReceived(
                object sender, NotificationEventArgs e)
            {
                {
                    ...original code moved unchanged from MainPage class.
                }

                public void Register()
                {
                    if (channelUri != null)

```

```

        {
            if (!isRegistered)
            {
                serviceClient.RegisterAsync(channelUri);
            }
            else
            {
                serviceClient.UnregisterAsync(channelUri);
            }
        }
    }

    private void serviceClient_RegisterCompleted(object sender, AsyncCompletedEventArgs e)
    {
        ...original code moved unchanged from MainPage class.
    }

    private void serviceClient_UnregisterCompleted(object sender, AsyncCompletedEventArgs e)
    {
        ...original code moved unchanged from MainPage class.
    }

    private void channel_ErrorOccurred(object sender, NotificationChannelErrorEventArgs e)
    {
        ...original code moved unchanged from MainPage class.
    }
}

```

You also need a *Dispatcher* in the viewmodel. This is because the notifications will come in on a non-UI thread, but you're data-binding the list of notifications to the UI. To avoid cross-thread exceptions, you need to marshal any updates to the notifications collection to the UI thread. The standard way to achieve this is to use the *Dispatcher* class. Every UI element—in fact, every type derived from *DependencyObject*—has a *Dispatcher* field of type *Dispatcher*. Your viewmodel is not a UI element, but you can use the *Deployment.Current.Dispatcher*, which always exists in a phone application and is globally available. If there is no UI active, you simply add the message to the collection, but if there is UI active, you use the global *Dispatcher* to ensure that you update the collection on the UI thread.

```

private void AddMessage(String message)
{
    if (Deployment.CurrentDispatcher != null)
    {
        Deployment.Current.Dispatcher.BeginInvoke(() => Notifications.Add(message));
    }
    else
    {
        Notifications.Add(message);
    }
}

```

The other side of this design is that in the *MainPage* class, the *Click* event for the Register App Bar button is now routed through to the *Register* method in the *PushViewModel*. You also implement a second App Bar button and wire its *Click* event to navigate to the new *SettingsPage*.

```
public partial class MainPage : PhoneApplicationPage
{
    public MainPage()
    {
        InitializeComponent();
        messageList.ItemsSource = App.Push.Notifications;
    }

    private void register_Click(object sender, EventArgs e)
    {
        App.Push.Register();
    }

    private void settings_Click(object sender, EventArgs e)
    {
        NavigationService.Navigate(new Uri("/SettingsPage.xaml", UriKind.Relative));
    }
}
```

The *SettingsPage* itself is trivial; it uses the *ToggleSwitch* from the Microsoft Silverlight Toolkit to give the user the option to turn toast notifications on or off, as shown in Figure 12-7.

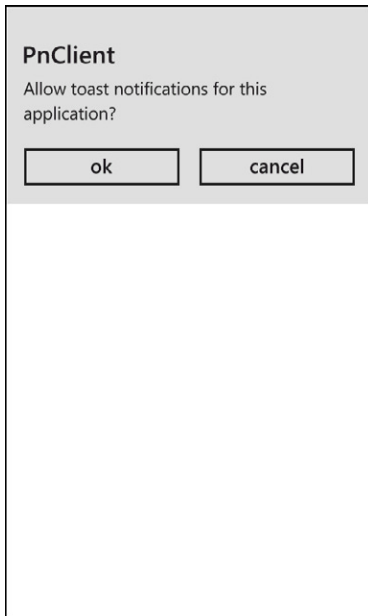


FIGURE 12-7 The toast notifications settings page.

In the *SettingsPage* XAML, data-bind the *ToggleSwitch* to the *IsToastOk* property on the *PushViewModel*.

```
<StackPanel x:Name="LayoutRoot" Background="Transparent" Margin="{StaticResource
PhoneHorizontalMargin}">
    <toolkit:ToggleSwitch
        Header="Allow toast notifications" IsChecked="{Binding IsToastOk, Mode=TwoWay}"
        FontSize="{StaticResource PhoneFontSizeLarge}"/>
    <TextBlock
        Style="{StaticResource PhoneTextTitle3Style}" TextWrapping="Wrap"
        Text="When a toast notification is received, it brings up the radio stack which can
        reduce battery life." />
</StackPanel>
```

Back in the *PushViewModel*, expose the *IsToastOk* property and implement *INotifyPropertyChanged* so that it can take part in data binding. The collection of notifications is already covered because you're using an *ObservableCollection<T>*, which itself implements *INotifyCollectionChanged*. For the *IsToastOk* property, when it is changed (which in your application is done only via the *SettingsPage ToggleSwitch*), you conditionally either bind or unbind to toast notifications.

```
public class PushViewModel : INotifyPropertyChanged
{
    private bool isToastOk;
    public bool IsToastOk
    {
        get { return isToastOk; }
        set
        {
            if (value != isToastOk)
            {
                isToastOk = value;
                PropertyChangedEventHandler handler = PropertyChanged;
                if (null != handler)
                {
                    handler(this, new PropertyChangedEventArgs("IsToastOk"));
                }

                if (isToastOk)
                {
                    channel.BindToShellToast();
                }
                else
                {
                    channel.UnbindToShellToast();
                }
            }
        }
    }

    public event PropertyChangedEventHandler PropertyChanged;
}
```



Note For an even richer UX, you could provide two settings: one for toggling toasts on/off, and another for toggling all notifications on/off. This won't be useful in all applications, but if your application uses toasts plus tiles and/or raw notifications, and if it can function correctly with only some or none of the notification features, then a finer-grained settings option might make sense.

The Push Notification Server-Side Helper Library

The Push Notification Server-Side Helper Library is a set of classes that simplify the creation of server applications that need to send push notifications. The library is available as a free download from <http://create.msdn.com/en-us/education/catalog/article/pnhelp-wp7>. The download includes source code for the library as well as a tutorial and sample code showing how to use it. The library provides the following additional features:

- A set of classes that encapsulate the server-side behavior of sending each of the different notification types, including a base *PushNotificationMessage* class, and derived *RawPushNotificationMessage*, *TilePushNotificationMessage*, and *ToastPushNotificationMessage* classes.
- A set of support classes such as the *HttpWebResponseExtensions* class that provides extension methods for *HttpWebResponse*; the *MessageSendPriority* class that provides a simple encapsulation of the core batching intervals; the *MessageSendResult* class that encapsulates the web response from a sent notification; and simple enumerations for subscription, notification, and connection status.
- A client implementation that illustrates a traditional Model-View ViewModel (MVVM) approach to data-binding the notifications and connection context, including the user's settings preferences for accepting each type of notification.
- Demonstration code to illustrate some of the more common server-side notification patterns.

The key feature of the library is the hierarchy of notification message classes, which is illustrated in Figure 12-8.

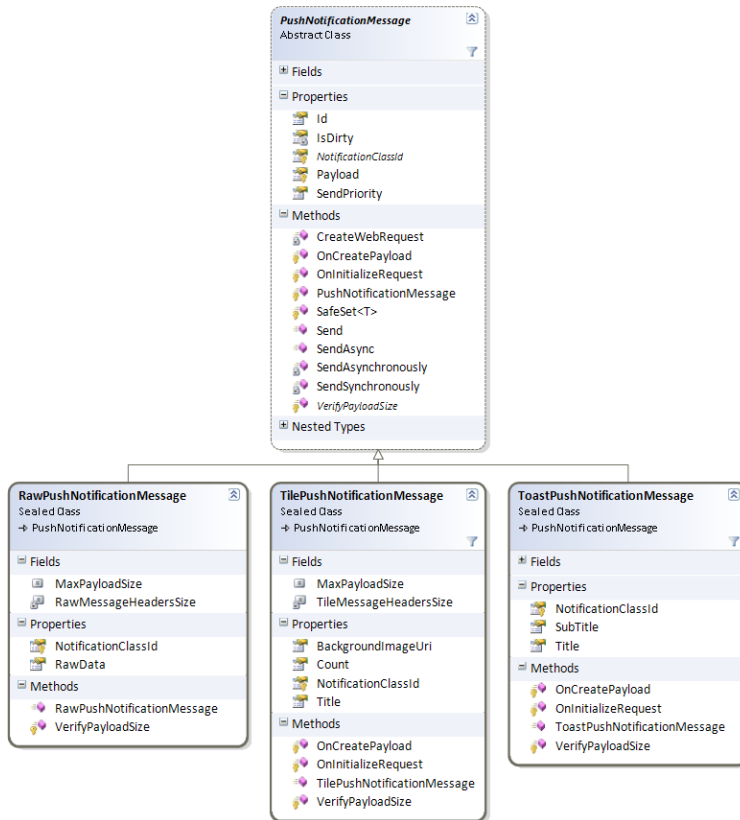


FIGURE 12-8 The Push Notification Server-Side Helper Library classes.

Using these classes simplifies the server-side code for sending notifications. For example, the code that follows sends a raw notification; you instantiate the specific notification type class, and then invoke either the synchronous (blocking) *Send* method or the asynchronous *SendAsync* method. For the asynchronous method, rather than having to wire up an event handler for the result independently, you pass in a delegate as a parameter to the *SendAsync* method.

```

var rawMsg = new RawPushNotificationMessage(MessageSendPriority.High)
{
    RawData = Encoding.ASCII.GetBytes(RawMessage)
};

foreach (var subscriber in PushService.Subscribers)
{
    rawMsg.SendAsync(
        subscriber.ChannelUri,
        result =>
        {
            Log(result);
            OnRawSent(subscriber.UserName, result);
        },
        Log);
}
  
```


In addition to the helper classes, the library also demonstrates how you can implement some of the common patterns for sending notifications, specifically:

- **One-time push** This is the simplest pattern and the basis of all notifications. This is what you've been using to explore push notifications throughout this chapter.
- **Push counter resets upon logon** The classic example of this is email, wherein the email client tile shows a counter for the number of unread emails since the last time the user opened the email client application. The same technique is used for unread SMS messages, missed phone calls, and so on. The trigger for resetting the count can be opening the application (which will generally log the user on to the server implicitly) or some other domain-specific action.
- **Ask to pin application tile** If you send a tile notification but the tile is not pinned to the start screen, the notification will not be delivered to the phone. In this case, the server will be notified that the application tile is not pinned. It is part of the usability model in Windows Phone that there is no programmatic way to pin a tile to the start screen. This is kept strictly under user control. When the server application detects that the tile is not pinned, it can work with this constraint and simply send a follow-up raw notification that can be used by the client application to inform the user that she can receive tile updates if she performs the pin operation.
- **Create custom server-side image** The location of the image for the tile background must be either local to the client application or at a remote URL that can be reached by the client. If the image is on the phone, it must be static. However, if it is at a remote URL, it can be either static or dynamic. This pattern shows how to generate static images on the server for use as tile backgrounds.
- **Scheduled tile updates** The normal pattern is that tile updates are sent by the server application at times dictated by logic on the server. An alternative pattern is where the client application can use the *ShellTileSchedule* class to register for periodic tile updates from the server.

Common Push Notification Service

So far in this chapter, you've considered the Microsoft Push Notification Service. In fact, there are two other commonly used push notification systems: the Apple Push Notification Service (APNS) for iOS devices, and the Cloud 2 Device Messaging Framework (C2DM) for Android devices. It is possible that an application developer would want to target not just the MPNS, but one or both of the other systems, as well.

Although the three systems have similar goals and architectures, they're obviously implemented differently. There are differences in how your application would need to talk to each service, so building an application to target multiple systems involves additional work. This work would need to be repeated for each application. This is where the Common Push Notification Service (CPNS) comes in. The CPNS provides a common framework, and a set of common classes that simplify connecting

to any or all of the three systems. Furthermore, it's sufficiently open-ended that it could easily be extended to cover other systems that might become available in the future. The CPNS is released as open source by the Microsoft Interoperability Strategy Group; it is not a formal service like the MPNS, rather it is a set of class libraries that you can deploy to your servers (or to Windows Azure) to mediate between your application server and the MPNS/APNS/C2DM services. The basic architecture of an application that uses the CPNS is shown in Figure 12-9.

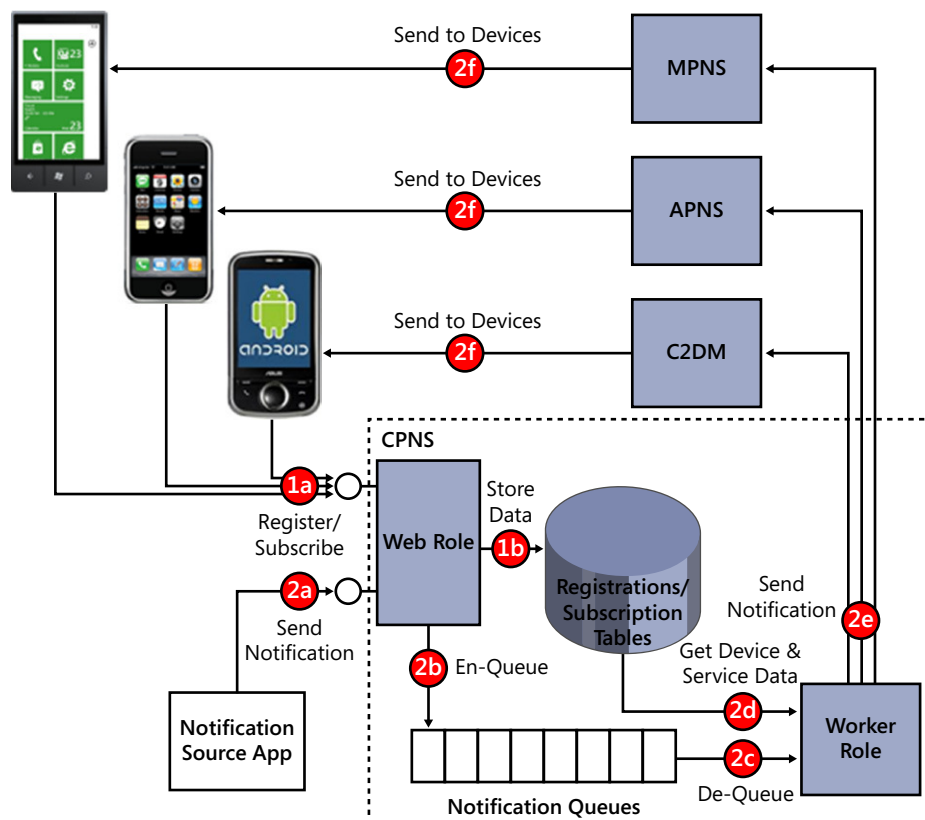


FIGURE 12-9 An overview of the Common Push Notification Service (CPNS) architecture.

The CPNS defines three primary entities:

- **Devices** This is the set of user phones that are registered for a given system.
- **Notifications** The three systems support different types of notification, but all of them support notifications of some kind.
- **Subscriptions** Notifications can be grouped into logical subscriptions. For example, a user might choose to subscribe to all notifications for a particular sports team, or he might choose to subscribe to a subset of news topics (perhaps, technology, business, and sports) from a news feed.

The CPNS maintains internal data stores that correspond to these three entity types. As it happens, the CPNS is implemented as an Azure service. It consists of a web role that exposes REST endpoints for device, subscription, and notification management and a worker role that sends push notifications. It uses Azure storage for information about each device, notification, and subscription, as well as the different push notification systems. Most of the data is held in Azure table storage, whereas the ongoing notifications are held in an Azure queue.

Figure 12-10 is a screenshot of the sample server application UI. From this, you can see that you can send either a system-specific message (for Windows Phone 7, iPhone, or Android), or a non-system-specific message. In the latter case, the message will be sent to all devices that are subscribed to receive notifications about coffee, for all three of the push notification systems.

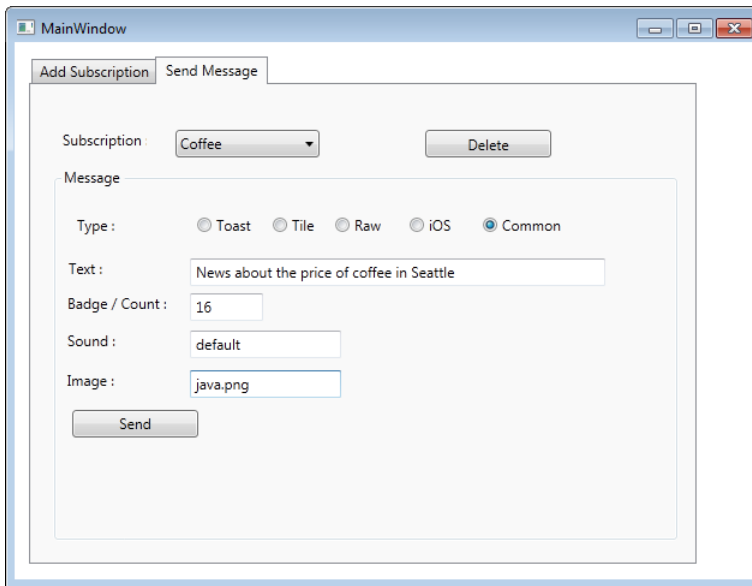


FIGURE 12-10 Sending a notification to multiple push systems.

Of course, the same message is implemented in a different way for each system:

- When sending to a Windows Phone 7 device, the message is sent as a tile notification.
- For an iPhone, it's sent as an iPhone push notification.
- For Android, notifications are similar to Windows Phone 7 raw notifications; the payload is arbitrary and each client application can choose how to handle it.

To send push messages to iPhone, the sender service must be registered with the Apple Developer portal and use certificate authentication. Similarly, for sending notifications to Android devices, you need to sign up to use C2DM (<http://code.google.com/android/c2dm/signup.html>).

The CPNS exposes service REST endpoints, backed by Azure web and worker roles. Your client application would register with the CPNS, and the web role adds the registration and subscription data to table storage. Your server application also uses the CPNS to send notifications, and the web role puts each notification into queue storage. Then, the worker role consumes the queue, matching each notification to its target device(s), using the specified target notification system(s).

Here's how you would use the CPNS. Keep in mind that the following code listings are a simplified abstraction of the core functionality. In your server application (the custom data provider that initiates the push notifications), you would send each notification to the CPNS instead of directly to MPNS/APNS/C2DM. You send the message payload without any of the push notification XML formatting or headers. For example, the code that follows is all you need to send a toast notification. The key point to note is that your server code is extremely simple—you don't need to worry about the specifics of the target system.

```
string toast =  
    "http://127.0.0.1/push.svc/message/toast/CoffeeSubscription?mesg=Hello World";  
HttpRequest request = (HttpRequest)WebRequest.Create(toast);
```

The CPNS side of this request constructs a *ToastMessage* class object from the incoming *HttpRequest* payload, and then adds this to an Azure queue. The *ToastMessage* class is one of the derivatives of a base *PushMessage* class defined in the CPNS library.

```
ToastMessage toastMsg = new ToastMessage(subscriptionId, message);  
this.msgQueue.Enqueue(toastMsg);
```

Then, there is an independent Azure worker role which processes the main queue of incoming notification requests. For each notification, it determines which target system(s) are requested, and adds the message to one or more of the queues specific to each system.

```
while (true)  
{  
    PushMessage message;  
    if ((message = this.msgQueue.Dequeue()) != null)  
    {  
        if ((message.MessageType & MessageType.Apns) == MessageType.Apns)  
        {  
            apnsConnection.Enqueue(message);  
        }  
        if ((message.MessageType & MessageType.Mpns) == MessageType.Mpns)  
        {  
            mpnsConnection.Enqueue(message);  
        }  
        if ((message.MessageType & MessageType.C2dm) == MessageType.C2dm)  
        {  
            c2dmConnection.Enqueue(message);  
        }  
    }  
}
```

Meanwhile, another operation in the worker role is processing each system-specific queue, adding the required payload formatting and custom headers for that system. It then sends a *WebRequest* to the corresponding target system (MPNS, APNS, or C2DM).

```
var request = (HttpRequest)WebRequest.Create(uri);
request.Method = WebRequestMethods.Http.Post;
request.ContentType = "text/xml";
request.Headers.Add("X-MessageID", Guid.NewGuid().ToString());
request.Headers["X-WindowsPhone-Target"] = "toast";
request.Headers.Add("X-NotificationClass", ((int)this.ToastInterval).ToString());

while (!sent && tries < retries)
{
    using (var requestStream = request.GetRequestStream())
    {
        requestStream.Write(messageBytes, 0, messageBytes.Length);
    }
}
```

You can use the CPNS as it is, modifying it in minor ways to meet your specific requirements, or you can use it as an informational example of just one way to support push notifications that target multiple notification systems. For example, the CPNS as released uses Windows Azure internally, and you might choose to implement your own backing storage and notification queuing mechanisms instead. Further details are available at <http://windowsphone.interoperabilitybridges.com/articles/common-push-notification-service-sample>.

Summary

In this chapter, you have seen how the Windows Phone 7 push notification model provides a way for you to build applications that can easily keep information on the phone up to date. This involves a server-side application for generating notifications and a client-side application for receiving the incoming notifications and rendering the data appropriately on the phone. The three different notification types (raw, tile, and toast) each provide different data and behavior, appropriate for different use cases. Raw notifications are entirely under your control; tile notifications integrate seamlessly with the standard Metro UX. Notifications also integrate with system-level alerts; therefore, they require user approval. Finally, you examined two open-source helper libraries, which can help simplify your applications, and help you target non-Windows Phone 7 devices. Security issues related to push notifications are addressed in Chapter 13.

Security

The Windows Phone 7 OS and application platform are very secure, but there's no such thing as 100 percent secure, and developers have a responsibility to follow secure development guidelines and do their part toward protecting the user from security attacks. Mobile devices present particular security challenges, arising mainly from the fact that they're mobile: it is easier for an attacker to steal your phone than your PC. As a developer, this means that you should carefully consider the impact of storing potentially valuable data on the phone, especially if the phone is not protected by a PIN. This includes personal information such as credit card or other account details, all usernames and passwords, and unique identifiers used to access public web services or websites—none of this data should be kept on the phone. In this chapter, you'll examine the security features built into the platform, some development lifecycle techniques that you can adopt to improve the security of your applications, and some specific authentication and authorization technologies.

Device Security

Windows Phone 7 does have some level of built-in security support. For example, the user can set a PIN on the device which is then required to unlock it for use. Also, removable SD cards are not supported, which significantly reduces the security vulnerability of the phone. Inserting an SD card represents a potential attack vector, because you might have no control over what is put on the card before it gets to your phone. However, for cases in which a Windows Phone uses an SD card, that card is associated with the device by a 128-bit key, so even if an attacker manages to get hold of your SD card, it would be difficult for him to read it on another device. Version 7 also does not support PC tethering, although this feature was introduced in version 7.1.

To install applications on a Windows Phone, the user must have a Windows Live ID. This confers a number of additional benefits, including security features. If the user loses his phone, he can go to the Windows Phone Live portal (<http://windowsphone.live.com>) to manage the lost phone remotely. Features available on the portal include locating the phone, locking it, ringing it, and erasing it. To use any of these features, you navigate to the portal in any browser, and then sign in with your Windows Live ID. If you have multiple phones, they will all be listed under your profile and available to manipulate. Figure 13-1 illustrates locating a phone and ringing it.

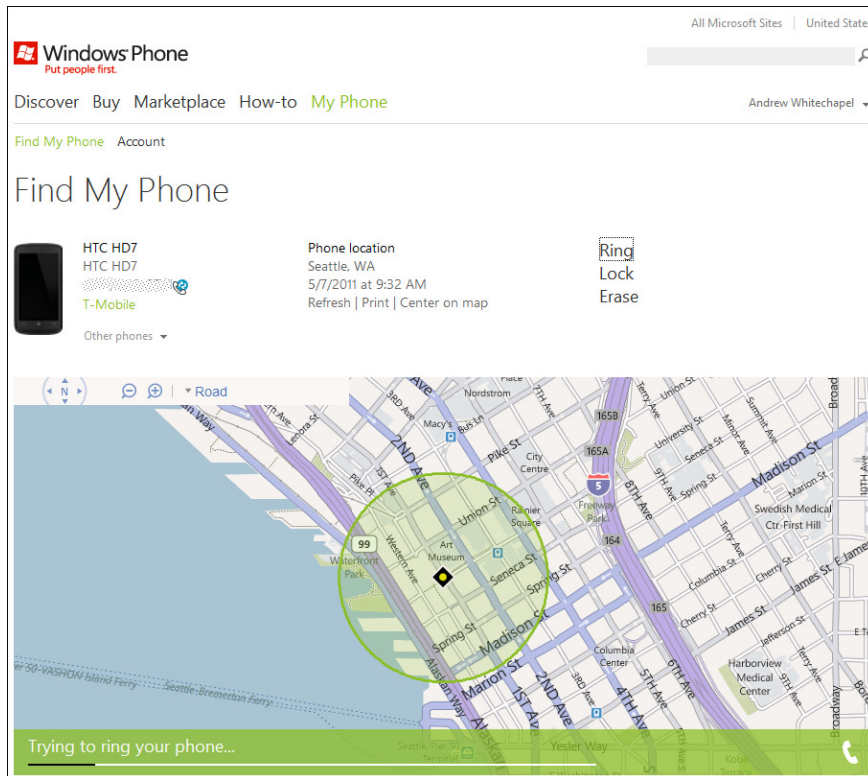


FIGURE 13-1 Locating and ringing my phone via the Windows Phone Live portal.

Application Safeguards

On top of the user-centric device security features, the Windows Phone application platform has also been built to use a range of techniques that help to protect users and their data by imposing constraints on applications. These include checks on how applications are installed on the phone and further restrictions within the deployment channel itself; that is, within the marketplace ingestion process. Even after an application is installed on the phone, there are further launch-time and runtime safeguards that protect the overall user experience (UX), the security and privacy of user data, as well as the ongoing cost of using the phone.

Application Deployment

Apart from developer-unlocked phones, applications can be deployed onto a retail phone only through the Windows Phone marketplace portal. The structured marketplace submission and provisioning process includes a suite of certification tests to identify certain behaviors that might be associated with problems. If detected, this process prevents applications that exhibit those behaviors from becoming available in the marketplace. The marketplace is the only legitimate source of application acquisition, mandatory code signing, and application licenses. This approach helps to maintain a consistent set of standards for Windows Phone applications.

All marketplace applications have to go through a formal certification process before they are made available to users. Code signing occurs automatically once the application has successfully passed the certification testing within the marketplace ingestion workflow. The application and repackaged XAP files are signed by Microsoft with the Microsoft-issued Authenticode certificate that is assigned to the developer when it originally registered on the marketplace. Any pre-existing signatures in a submitted application or XAP files are replaced/overwritten during this process. Only those applications that are signed with a Microsoft-issued Authenticode certificate can be installed and run on a retail Windows Phone.

Also, of course, for a developer to submit an application to marketplace, that developer must be registered with the marketplace. This requires the developer to provide information about herself—including credit card information—and this is subject to verification.

Marketplace ingestion covers other aspects of publication apart from security (these are examined in detail in Chapter 14, “Go to Market”). Ingestion includes the following security-specific constraints:

- **Application code validation** An application must not invoke native code via P/Invoke or COM interoperability; if it does, it will fail certification.
- **Malicious software screening** The application must be free of viruses and any malicious software.
- **Microsoft Intermediate Language (MSIL) type safety verification** The phone OS implements multiple sandboxes to help protect the integrity of the device and the applications running on it. An application must contain only type-safe MSIL code to pass certification; any use of “unsafe” language constructs such as pointers and pointer arithmetic, type casts, *fixed* buffers, and *stackalloc* will cause the application to fail type-safety verification, and consequently fail certification.

Figure 13-2 summarizes the marketplace developer registration and application publication processes.

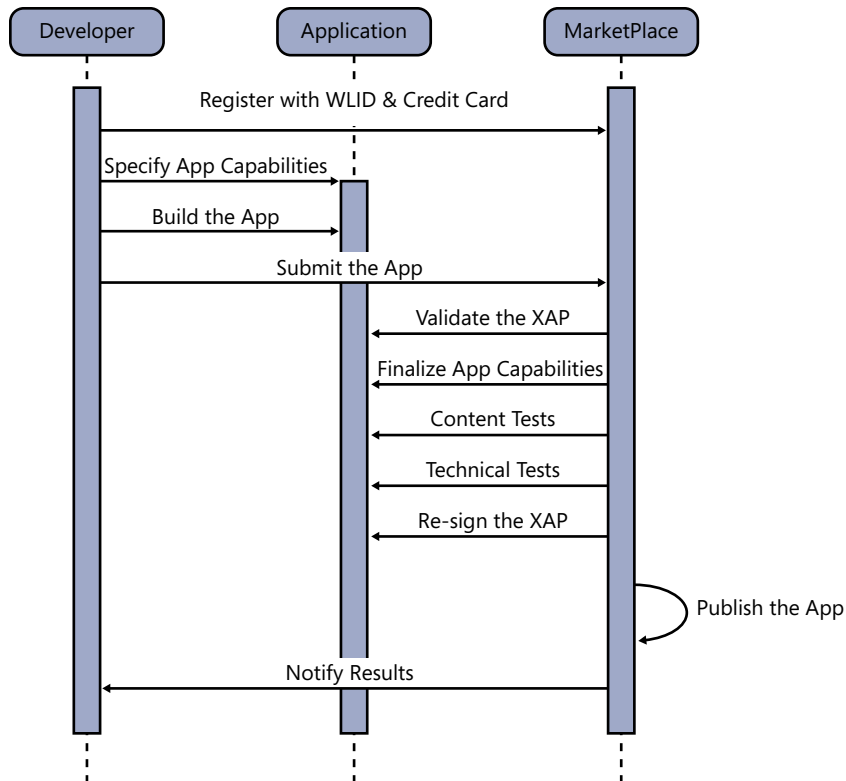


FIGURE 13-2 The marketplace ingestion and publication process.

There are currently no supported mechanisms for loading applications onto the Windows Phone 7 platform outside of the marketplace. Such *side-loading* mechanisms, including downloading from a PC or from a storage card or from another device over Bluetooth are not enabled. Even if you could side-load an application on to the phone, there would still be no way to launch that application. The only way to launch applications is via the application list or the Start menu (or Launchers and Choosers, in the case of built-in applications). For an application to be available in the user interface (UI), it must be correctly registered in the internal application database, and side-loading cannot achieve this. Even upgrades, maintenance releases, and patches for applications must be routed through the public marketplace and run through the marketplace security constraints. Figure 13-3 summarizes the application installation process. Note that installation includes a step to configure the security sandbox that will be used for the application at runtime.

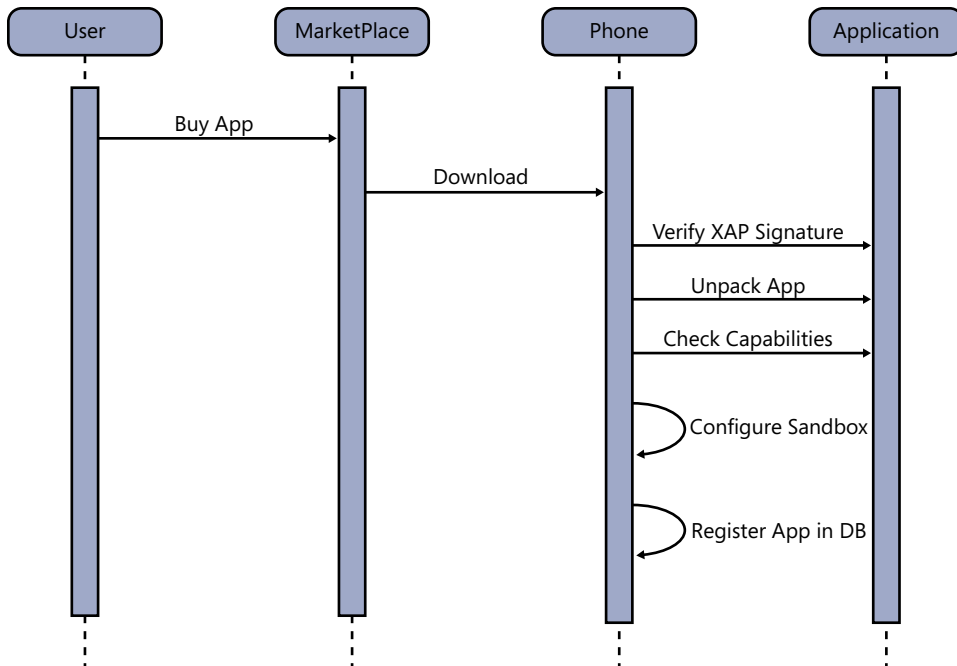


FIGURE 13-3 Installing an application from the marketplace.

Managed Code Constraints

All marketplace applications can be developed only in managed code. Not only does this help to improve developer productivity, but it also improves the robustness of the applications. The strong typing, bounds checking, and memory management features of managed code minimize or eliminate many of the common programming errors that can lead to exploitation of the application by hackers as well as excessive and unintended resource consumption. In general, managed applications are far less vulnerable to issues such as buffer overflows, format string errors, memory management errors, and so on, which are common in unmanaged code.

The Windows Phone 7 application platform also strictly controls access to Windows APIs. Many areas of functionality are effectively blocked, including registry access, filesystem access, many network features, process APIs, all inter-process communication APIs, and so on. All of these restrictions are designed to improve the overall robustness of the platform by disabling potentially damaging APIs and increasing isolation between applications as well as between applications and the system itself.



Note One inter-process communication mechanism that is supported is the use of named mutexes. This was introduced in version 7.1 specifically to support communication between a background agent and its associated foreground application. It is strictly scoped to the application's security chamber.

Windows Phone 7 applications run inside a managed sandbox which implements the Microsoft Silverlight security model. Silverlight is supported by a stripped-down version of the Common Language Runtime (CLR), and on the phone this uses the Microsoft .NET Compact Framework. Silverlight needs only one sandbox that is equivalent to the browser sandbox for running scripts. Managed code in Silverlight follows the new CLR Security model, based on the notion of “transparency.” This divides code into three layers, based on whether the assembly is trusted, and on three custom attribute annotations: *Transparent*, *SafeCritical*, and *Critical*, which are described in the following:

- **Transparent code** This is the lowest trust level for code. *Transparent* code can run only with the same permission as the caller. All third-party application code and significant portions of framework libraries code is *Transparent* code. *Transparent* code has the following constraints:
 - It cannot contain unverifiable code. This means that all of the code must be verifiably type-safe.
 - It cannot call native code via a P/Invoke or COM interop.
 - It cannot access *Critical* code or data.
- **SafeCritical code** The bridge between *Transparent* code and *Critical* code. *SafeCritical* code performs security boundary checks such as parameter validation, and ensures that *Transparent* code is clean to perform the critical operations. Basically, *SafeCritical* code is expected to perform the necessary due diligence on the caller before, in its turn, calling *Critical* code. All *SafeCritical* code must be part of the platform itself; it cannot be part of a marketplace application.
- **Critical code** This has the highest privileges and is restricted only by the permissions of the application. This includes the ability to interact with the system through P/Invokes or even to contain unverifiable code. All *Critical* code must be part of the platform.

Table 13-1 summarizes all three layers.

TABLE 13-1 Code Transparency Layers

Managed Layer	Code Annotated with	Role	Accessibility
SecurityCritical	<i>System.Security.SecurityCritical</i>	Fully trusted code. Can perform pointer arithmetic and P/Invoke.	Can be accessed only by <i>SafeCritical</i> layer. Must be loaded by the OS as a trusted assembly.
SafeCritical	<i>System.Security.SecuritySafeCritical</i>	Acts as a bridge between <i>Transparent</i> and <i>Critical</i> code.	Can be accessed by all layers. Must be loaded by the OS as a trusted assembly.
Transparent	<i>System.Security.SecurityTransparent</i> or Unannotated	Can call into <i>SafeCritical</i> code. All user application code is <i>Transparent</i> ; any annotation on user code is ignored by the runtime.	Can be accessed by all layers.

Note that only Microsoft assemblies (that is, those that ship in the device as part of the managed platform) are allowed to have these annotations; hence, only they can directly access any native code. All marketplace application code and libraries are themselves treated as *Transparent* code and can only access *Transparent* and *SafeCritical* methods in the managed platform APIs.



Note Windows Phone 7 does also support a hybrid application model, in which the application can include both native and managed code, but this feature is not exposed to marketplace application developers, and is used only by the phone hardware manufacturers and mobile operators. A hybrid application can in fact use COM interop to call into its own COM libraries, which in turn can call native functionality.

Chambers and Capabilities

On top of the basic Silverlight transparency layers, the Windows Phone application platform introduces the following additional security/robustness techniques:

- **Chambers** A chamber is a new process isolation technology. There are four chamber types. The first three have fixed permission sets, whereas the last one (Least-Privileged Chamber) has permissions driven by capabilities. All marketplace applications use the Least-Privileged Chamber only. The following is a description of each chamber:
 - **Trusted Computing Base (TCB)** This chamber is assigned the greatest privileges, with unrestricted access to all device resources, including the ability to modify security policy. The kernel and kernel-mode drivers run in the TCB.
 - **Elevated-Rights Chamber (ERC)** This chamber can access all resources except security policy. The ERC is for services and user-mode drivers intended for use by other phone applications.
 - **Standard-Rights Chamber (SRC)** This is the default chamber for pre-installed applications.
 - **Least-Privileged Chamber (LPC)** This is the default chamber for all third-party applications; that is, all non-Microsoft applications that can be downloaded from the Windows Phone marketplace. Windows Phone applications run in a sandboxed process. This means that they are isolated from each other and interact with phone features in a strictly structured way.
- **Capabilities** A capability (or privilege) grants access to one or more resources on the phone that have security, privacy, or cost implications for the user. The platform provides a capabilities-driven security model, in which an application is executed within a security sandbox whose limits are determined by the capabilities required by the application, as described in the following:
 - The sandboxed process within which a particular application runs has a customized set of security privileges. The platform is designed to minimize the attack surface area of each application by only granting it the privileges that it needs in order to run. For example, if an application does not require the use of the location services, the application platform will seek to execute it in a sandboxed process which does not have access to location services.

- Certain privileges that an application might need have a direct impact on information access or cost. In such cases, the Windows Phone marketplace will disclose this information to the end user before the application is purchased. Some examples include using network-based services, whereby a user could incur additional roaming costs if the use of the services were not disclosed by the application. Pre-installed applications are also required to disclose this information to the end-user upon first use of the application.
 - The capabilities model helps to decrease the attack surface: capabilities are used to create a security chamber in which the application will execute. This chamber is created once during installation, stored in the application database on the phone, and then always used whenever the application is launched, subsequently. A chamber is a security and isolation boundary for a process; it places limits on what the hosted process can do, based on policies, as configured for the instance of the chamber.
 - The capabilities model also helps to ensure proper disclosure. Users must be notified if an application's functionality has implications on their privacy, security, or costs. The user is told what capabilities an application uses before she installs it; if she chooses to proceed with the installation, this is her acceptance of these capabilities. As it pertains to location services, before the functionality is activated, the user must explicitly opt in, and she must also be given a mechanism to opt out again at any later stage. For other capabilities, the user's only opt-out path is to uninstall the application. For example if an application uses a microphone and a user is not aware of it, then it might be possible for the application to record the user's conversation and send it to the attacker, thus posing a security and privacy risk. The capability model ensures that a user is able to review what capabilities an application supports before she uses it.
- **Execution Model** Before an application can run on the phone, it must be installed via the official marketplace. Marketplace ingestion imposes security and other constraints, and when the marketplace triggers the installation of an application on the phone, this includes registering it in the platform's application database. This registration includes extracting metadata from the manifest about the application's required capabilities. These, in turn, will be used to configure the sandbox in which the application runs.
- All applications on the phone are launched via an Execution Manager, which is a core part of the application platform. The Execution Manager will only launch applications that are correctly registered in the database, and it will ensure that the runtime sandbox is configured to precisely match the application's registered capabilities.
 - The Execution Manager will only load applications that have been signed by the Microsoft certificate used in marketplace ingestion. So, even if you could side-load an invalidly signed XAP onto the device, it would not get loaded.
 - When you build an application, the generated package includes metadata that declaratively specifies the capabilities required by the application. When the user installs an application, the Package Manager validates the capabilities in the XAP and maps them to corresponding security groups maintained in the application database, which are then

associated with the Security ID (SID) and LPC created for the application. When the user launches the application, the Execution Manager associates the required capabilities with the SID and LPC under which the application's host process executes.

- Launching an application is not a “fire-and-forget” operation. Rather, the Execution Manager continues to monitor the running application to ensure that it behaves appropriately. The application must not allow unhandled exceptions to propagate beyond itself; it must respond within a reasonable time to events such as the user navigating away from (or back to) the application; it must not make excessive use of CPU or memory resources; and so on. An application that misbehaves at runtime will be terminated in order to protect the overall UX on the phone.

Figure 13-4 shows a summary of the load/run sequence.

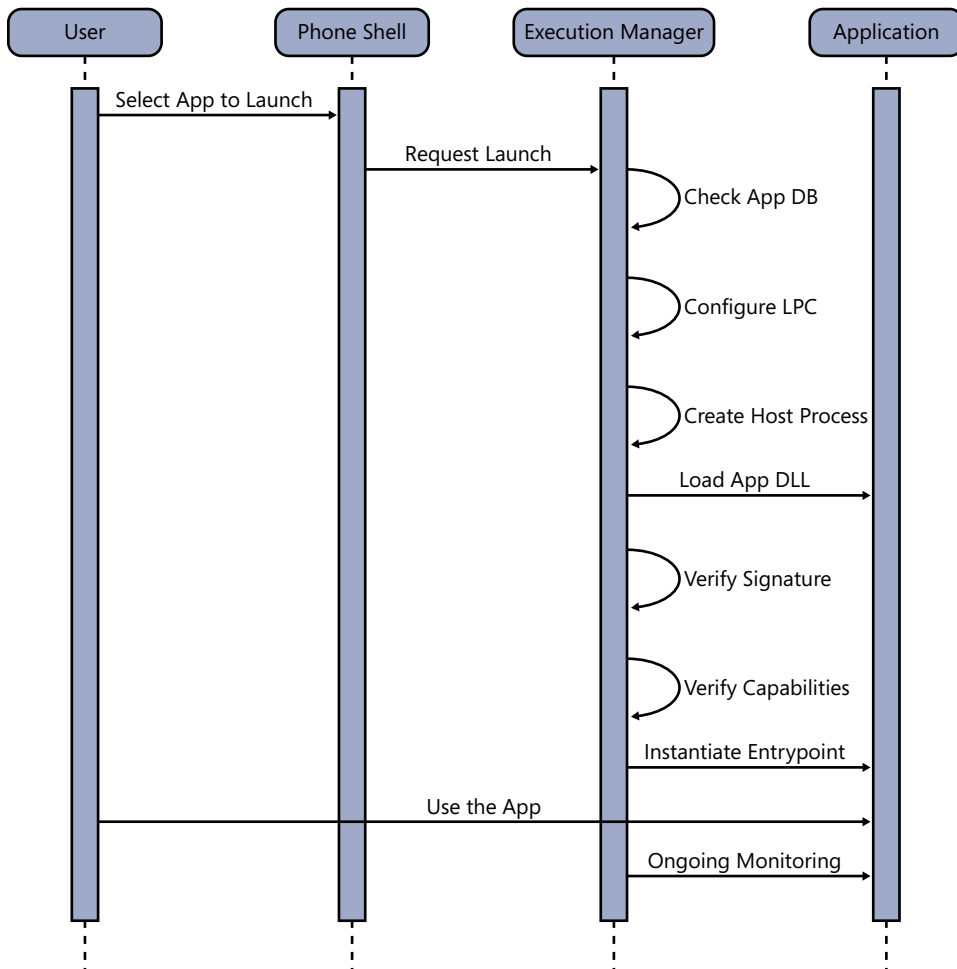


FIGURE 13-4 The application launch and execution sequence.

Missing Security Features

The Windows Phone 7 platform is targeted primarily as a consumer device rather than a business device. Although the phone does include many enterprise features, the priorities for this release place a greater emphasis on UX, battery life, and overall performance. Table 13-2 lists some prominent security features which would be required for enterprise line of business (LOB) applications but are not currently supported (or exposed to marketplace applications) in the Windows Phone 7 platform.

TABLE 13-2 Missing Security Features for Enterprise/LOB Applications

Feature	Description
Private application deployments	Windows Phone 7 applications can only be deployed through the Marketplace. Side-loading of applications is not supported. This makes the phone more secure, but makes it difficult to deploy a "private" enterprise application.
Credential Manager (CredMan)	On earlier versions of Windows Mobile, CredMan manages username and password credentials needed to authenticate clients who access remote resources. Without a way to store credentials <i>securely</i> , applications must always prompt the user to provide his username/password dynamically; you should avoid storing these on the phone wherever possible. One reasonable approach is to prompt the user before storing such data, and then do so only if he also has a PIN to protect the phone itself.
Client certificates	No managed API to access client certificates. In addition, there is no UI to manage certificates on the device.
Device encryption	Although a phone can be locked by using a PIN, and individual applications can encrypt their data, there is no way to encrypt the device as a whole (that is, there is no equivalent to BitLocker).
Local Authentication Subsystem (LASS)	No managed access to LASS. Marketplace applications cannot use Windows integrated authentication (NTLM, Kerberos).
Windows Identify Foundation (WIF)	No direct support for WIF federated authentication. An application can perform federated authentication, but this requires the application itself to implement WIF protocols.
Internet Protocol Security (IPSec)	IPSec provides a number of network access protections, including application-transparent encryption services for IP network traffic. None of this is available on Windows Phone.
Information Rights Management (IRM)	There is no support for viewing or editing IRM-protected documents in version 7; however, this support was added in version 7.1.
Data Protection API (DPAPI)	A comprehensive cryptography API for securing passwords and encryption keys. This is not available on version 7, although the phone does include a minimal set of cryptographic APIs, and DPAPI support was introduced in version 7.1.
Windows Live ID (WLID)	Although a WLID is required to register a phone, client-side programming for Windows Live is not directly supported in the platform. However, there is a Live SDK, available for download.

The absence of some of these features makes it challenging to develop and deploy applications for enterprise/LOB scenarios. Note that support for some of these features is gradually being introduced. For example, the Live SDK is available as a separate download; you can use this to connect to Windows Live, including SkyDrive, Hotmail, and Messenger. Windows Phone 7.1 also introduces support for Sockets (see Chapter 17, "Enhanced Connectivity Features") and DPAPI (see Chapter 18, "Data Support").

Also note that Windows Phone 7 does include comprehensive support for Exchange ActiveSync (EAS), version 14.0, and this is very important for enterprise users. System administrators can use EAS to configure security policy, which is then enforced on all phones connected to the organization's network. Administrators can require users to set up a PIN that must be used before the device can sync Exchange data (including email, contacts, and calendar). PINs can also have an expiry policy,

which requires users to enter a new PIN periodically. A security policy can prevent users from recycling previously used PINs, and can enforce rules about what constitutes a valid PIN in terms of pattern (for example, 1234 is typically not valid) and length. EAS also gives the organization the ability to control the idle timeout period (upon which the phone is locked). It can even enable the ability to erase a device remotely and reset it to factory defaults. This last feature can be combined with a policy that specifies the maximum allowed number of failed unlock attempts.

Data Encryption

Any data stored on the phone by an application is stored in isolated storage. This is an isolated virtual filesystem that is sequestered from other applications. By default, it is not encrypted. If an attacker gains physical access to a phone, it would be possible for him to then mount the phone's filesystem as a navigable filesystem on a PC, and thereby gain access to all the data stored on the device. To mitigate this threat, an application can choose to encrypt its persisted data. To this end, a core set of cryptographic functionality is supported on Windows Phone 7, as summarized in Table 13-3.

TABLE 13-3 Cryptographic Functionality Supported in Windows Phone 7

Algorithm	Classes	Description
Advanced Encryption Standard (AES)	<i>AesManaged</i>	This is a symmetric-key algorithm, meaning that the same key is used for both encrypting and decrypting the data. This is a standard algorithm used by the United States government and others.
SHA-1, SHA-256	<i>SHA1Managed</i> , <i>SHA256Managed</i>	A hash is used as a unique value of fixed size that represents a large amount of data. Hashes of two sets of data should match if and only if the corresponding data also matches. Small changes to the data result in large unpredictable changes in the hash.
HMACSHA-1, HMACSHA-256	<i>HMACSHA1</i> , <i>HMACSHA256</i>	<p>A type of keyed-hash algorithm that is constructed from the SHA1 or SHA256 hash function and used as a Hash-based Message Authentication Code (HMAC). The HMAC process mixes a secret key with the message data, hashes the result with the hash function, mixes that hash value with the secret key again, and then applies the hash function a second time.</p> <p>Used to determine whether a message has been tampered with, assuming that the sender and receiver share a secret key. The sender computes the hash value for the original data and sends both the original data and hash value as a single message. The receiver recalculates the hash value on the received message and checks that the computed HMAC matches the transmitted HMAC. Be aware that this does not provide confidentiality.</p>
RFC-2898	<i>Rfc2898DeriveBytes</i>	A password-based cryptography specification. The <i>Rfc2898DeriveBytes</i> class implements PBKDF2 (a password-based key derivation function) and includes methods for creating a key and initialization vector (IV) from a password and a salt.

As noted earlier, it is important to remember that the version 7 application platform does not support secure storage of passwords or encryption keys, nor does it include any kind of built-in key management facilities. It also does not contain managed APIs for asymmetric cryptography; for example, certificate-based scenarios such as digital signature verification.

The application (the *SimpleEncryption* solution in the sample code) shown in Figure 13-5 offers two pages. On one page (shown on the left), the user can enter some text plus an arbitrary password, and then click the Encrypt button to encrypt the text and save it to a file in isolated storage. On the second page, the user is given a list of all files in the application's isolated storage; she can select any of these, provide the corresponding password, and then click the Decrypt button to decrypt the file contents to a *TextBox*.

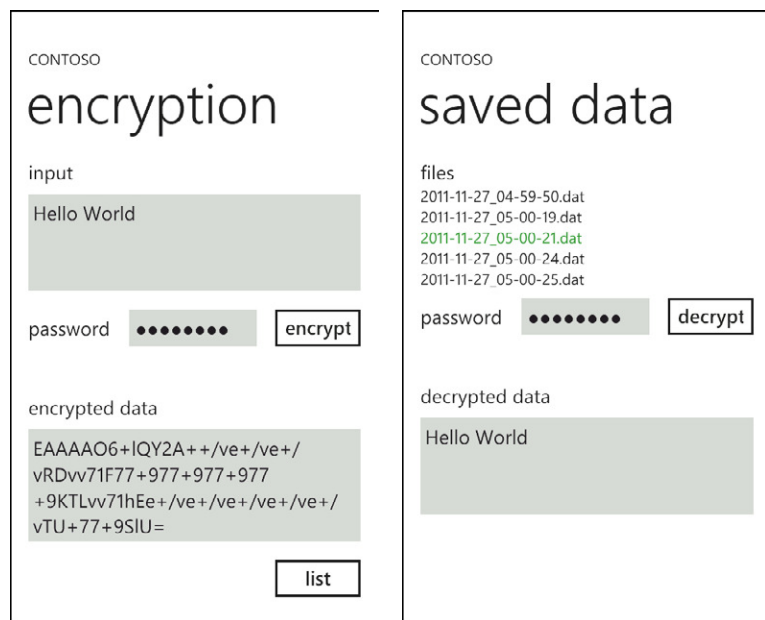


FIGURE 13-5 Encrypting data (on the left) and decrypting it (on the right).

There's just one piece of data that's common to both pages: the password salt. You store this in the *App* class (this is an arbitrary string). Note that hard-coding this is not very secure; you'll fix this later, but for now it serves to illustrate the main flow.

```
public const String PasswordSalt = "PasswordSalt";
```

To encrypt the data, take the password salt and the password itself that the user enters, and use these with *Rfc2898DeriveBytes* to generate a key. You then set this key into an *AesManaged* object and create a symmetric encryptor based on this key and the default IV. Next, perform the encryption by writing to a *CryptoStream* object, using the encryptor.

```
using (IsolatedStorageFile store = IsolatedStorageFile.GetUserStoreForApplication())
{
    String filePath = String.Format("{0:yyyy-MM-dd_hh-mm-ss}.dat", DateTime.Now);
    using (IsolatedStorageFileStream fileStream =
        store.OpenFile(filePath, FileMode.Create))
    {
        using (Aes aes = new AesManaged())
        {
            Rfc2898DeriveBytes key = new Rfc2898DeriveBytes(
```

```

        password.Password, Encoding.UTF8.GetBytes(App.PasswordSalt));
int maxKeySize = 256;
int keyByteCount = maxKeySize / 8;
aes.Key = key.GetBytes(keyByteCount);
fileStream.Write(BitConverter.GetBytes(aes.IV.Length), 0, sizeof(int));
fileStream.Write(aes.IV, 0, aes.IV.Length);
ICryptoTransform transform = aes.CreateEncryptor();

using (CryptoStream cryptoStream = new CryptoStream(
    fileStream, transform, CryptoStreamMode.Write))
{
    byte[] plainText = Encoding.UTF8.GetBytes(input.Text);
    cryptoStream.Write(plainText, 0, plainText.Length);
    cryptoStream.FlushFinalBlock();
}
}
}

```

To display the encrypted data on screen, you first convert it to a Base64 string so that it is more readable. This is only for testing/illustration purposes; you would not normally have a reason to do this in a published application.

```

using (IsolatedStorageFileStream fileStream =
    store.OpenFile(filePath, FileMode.Open))
{
    using (StreamReader reader = new StreamReader(fileStream))
    {
        byte[] encryptedBytes = Encoding.UTF8.GetBytes(reader.ReadToEnd());
        encrypted.Text = Convert.ToBase64String(encryptedBytes);
    }
}
}

```

Decryption follows a similar pattern. Again, you derive the same key from the password salt and the user-supplied password. Then, read the initialization vector into an *AesManaged* object so that you can then create a decryptor object. Next, read the *CryptoStream*.

```

using (IsolatedStorageFile store = IsolatedStorageFile.GetUserStoreForApplication())
using (IsolatedStorageFileStream fileStream = store.OpenFile(filePath, FileMode.Open))
{
    using (Aes aes = new AesManaged())
    {
        Rfc2898DeriveBytes deriveBytes = new Rfc2898DeriveBytes(
            password.Password, Encoding.UTF8.GetBytes(App.PasswordSalt));
        int maxKeySize = 256;
        int keyByteCount = maxKeySize / 8;
        aes.Key = deriveBytes.GetBytes(keyByteCount);

        byte[] dataSize = new byte[sizeof(int)];
        if (fileStream.Read(dataSize, 0, dataSize.Length) == dataSize.Length)
        {
            byte[] buffer = new byte[BitConverter.ToInt32(dataSize, 0)];
            if (fileStream.Read(buffer, 0, buffer.Length) == buffer.Length)
            {
                aes.IV = buffer;
            }
        }
    }
}

```

```

        ICryptoTransform transform = aes.CreateDecryptor();

        try
        {
            using (CryptoStream cryptoStream = new CryptoStream(
                fileStream, transform, CryptoStreamMode.Read))
            {
                using (StreamReader reader = new StreamReader(
                    cryptoStream, Encoding.UTF8))
                {
                    decrypted.Text = reader.ReadToEnd();
                }
            }
        }
        catch (Exception ex)
        {
            decrypted.Text = ex.ToString();
        }
    }
    else
    {
        decrypted.Text = "Failed to read byte array from stream.";
    }
}
else
{
    decrypted.Text = "Invalid byte array in stream";
}
}
}

```

Notice the call to *Rfc2898DeriveBytes.GetBytes*. For maximum strength, this code specifies a 256-bit key; the key will be an array of 32 bytes. AES supports encryption using 128, 192, and 256-bit keys. The longer the key is, the stronger is the encryption—and the slower the encryption process. In many cases, a 128-bit key is sufficient, but unless you're encrypting large volumes of data, the inherent performance degradation associated with a 256-bit key might not be significant.

Note that it is important not to store the encryption key on the device itself. You must also, of course, not store the password on the device (unless the user directs you to do so). Instead, you must derive the key dynamically, using *Rfc2898DeriveBytes*, each time it is required, and from a user-supplied password. If you want to be super-secure, you could also avoid storing the password salt on the device, and have the user enter this at runtime, too. Finally, if there is ever a need to generate random numbers in a security context, you should always use the *RNGCryptoServiceProvider* class, and not the standard *Random* class. In the following enhancement (the *StrongerEncryption* solution in the sample code), you increase the strength of the security by eliminating the hard-coded password salt. Instead, you generate it dynamically by using *RNGCryptoServiceProvider* to produce a random array of bytes. In fact, you can do this for both the password salt and for the IV used in the *AesManaged* object.

```

//public const String PasswordSalt = "PasswordSalt";
private static byte[] passwordSalt;
public static byte[] PasswordSalt
{
    get
    {
        if (passwordSalt == null)
        {
            int maxKeySize = 256;
            int keySize = maxKeySize / 8;
            passwordSalt = new byte[keySize];
            RNGCryptoServiceProvider rng = new RNGCryptoServiceProvider();
            rng.GetBytes(passwordSalt);
        }
        return passwordSalt;
    }
    private set { }
}

private static byte[] iV;
public static byte[] IV
{
    get
    {
        if (iV == null)
        {
            int maxBlockSize = 128;
            int blockSize = maxBlockSize / 8;
            iV = new byte[blockSize];
            RNGCryptoServiceProvider rng = new RNGCryptoServiceProvider();
            rng.GetBytes(iV);
        }
        return iV;
    }
    private set { }
}

```

Then, on the main page, in the encryption method, create the encryption key by using the randomly generated salt and the randomly generated IV. Note that you also specify the number of iterations for the *Rfc2898DeriveBytes* to use (in this case, 1000, which is a good minimum number to use in this scenario).

```

using (Aes aes = new AesManaged())
{
    Rfc2898DeriveBytes key = new Rfc2898DeriveBytes(
        password.Password,
        //Encoding.UTF8.GetBytes(App.PasswordSalt));
        App.PasswordSalt, 1000);

    int maxKeySize = 256;
    int keyByteCount = maxKeySize / 8;
    aes.Key = key.GetBytes( keyByteCount);

    // Use the randomly-generated IV.
    aes.IV = App.IV;
}

```

```

fileStream.Write(BitConverter.GetBytes(aes.IV.Length), 0, sizeof(int));
fileStream.Write(aes.IV, 0, aes.IV.Length);
ICryptoTransform transform = aes.CreateEncryptor();

using (CryptoStream cryptoStream = new CryptoStream(
    fileStream, transform, CryptoStreamMode.Write))
{
    byte[] plainText = Encoding.UTF8.GetBytes(input.Text);
    cryptoStream.Write(plainText, 0, plainText.Length);
    cryptoStream.FlushFinalBlock();
}
}

```

The same randomly generated password salt must be used for successful decryption, as well. This does mean that the application still stores this information with the file, so this approach adds only minimal additional security. For more robust security, you could also compute a hash, based on the encrypted data, the IV, and the password by using *HMACSHA256*. You could store this hash so that when it comes time to decrypt the data, you could recompute the hash and cross-check it with the previous hash. This mitigates the possibility that the encrypted data has been tampered with.

SDL Tools

The internal product teams at Microsoft use a wide range of tools for building secure software, as part of the Security Development Lifecycle (SDL). Several of these tools are relevant to Windows Phone developers, and they have been made available for free download, as described in Table 13-4.

TABLE 13-4 SDL Tools

Tool	Description
SDL Guidelines	A document that describes the SDL concepts as well as guidelines on how to adopt an SDL process in your own development. As of this writing, many aspects of the Microsoft SDL are being adopted by the United States military. You can download the guidelines from http://www.microsoft.com/download/en/details.aspx?displaylang=en&id=12379 .
Threat Modeling Tool	Used to create and analyze threat models. This is a way of describing the solution from a security perspective, listing all the possible threats, vulnerabilities, and attack vectors, and then devising suitable mitigations for each. Using the tool, you can map the security constraints of your application. It can also help you to identify problem areas. You can download the tool from http://www.microsoft.com/security/sdl/adopt/threatmodeling.aspx .
FxCop	Used for static analysis of managed code assemblies. The tool includes hundreds of common rules for maintainability, reliability, security, and so on. The tool is extensible, and you can add your own custom rules. The concept being that you run FxCop as part of your build process to flag areas in your code that don't meet the required standards. Note that Microsoft Visual Studio 2010 Premium includes an enhanced version of FxCop built in to the IDE. For other versions of Visual Studio, or to use FxCop stand-alone, an installer for FxCop is included in the Windows SDK. On a 32-bit computer, this is typically located at %ProgramFiles%\Microsoft SDKs\Windows\<version>\Bin\FxCop; on a 64-bit computer, you can find it at %ProgramFiles(x86)%\Microsoft SDKs\Windows\<version>\Bin\FxCop. Be aware that a default install of the SDK doesn't necessarily install the FxCop installer, so you might need to go back to the SDK install and select Add Features. You can download it from http://www.microsoft.com/download/en/details.aspx?displaylang=en&id=6544 .

Microsoft offers a range of SDL tools, but not all of them are relevant for Windows Phone development. In particular, MiniFuzz (file fuzzing tool), BinScope (binary file verification), and Banned.h (banned native APIs) are not useful.

Threat Modeling

Threat modeling is a structured approach to mapping the security context of your solution. It provides a process framework for describing your architecture in security terms, identifying vulnerabilities and threats to your system, and ensuring that you implement mitigations to all identified threats.

This is how you use the SDL Threat Modeling tool. First, you draw a diagram—this uses an embedded Microsoft Visio design surface within the tool. The diagram includes the major logical processes, data stores, and external interactors. It also includes the significant logical data flows between these various entities, as shown in Figure 13-6.

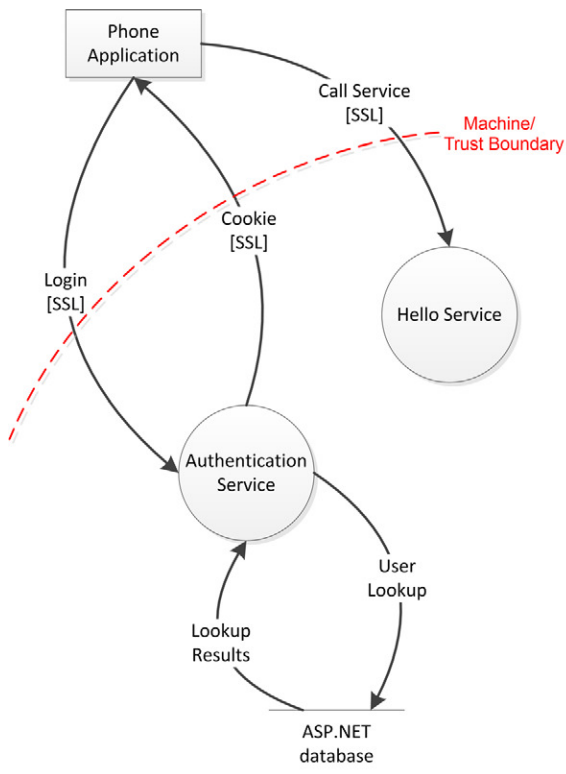


FIGURE 13-6 A simple threat model context diagram.

The tool analyzes the solution based on your diagram and generates a list of potential threats. You then go through all the threats and provide mitigations, as shown in Figure 13-7. You would also typically open bugs in your bug database for each one until you have implemented the mitigation.

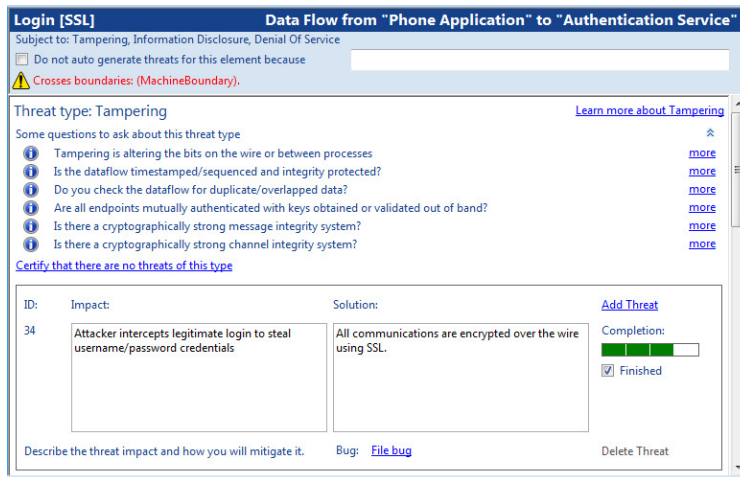


FIGURE 13-7 Analyzing the threats and determining mitigations.

In a rigorous SDL process, you would submit your final threat model for review with designated security experts within your organization. By using this tool, you can identify threats in a structured way, track them, and ensure that you have addressed all of them with acceptable mitigations.

Static Code Analysis/FxCop

FxCop ships with a command-line launcher, FxCopCmd.exe. For Visual Studio Premium and above, there's a slightly enhanced version built into the IDE itself. For any project, you can go to the Analyze menu, and then in the Project settings, select the Code Analysis tab. From there, you can configure code analysis; that is, choose the rule sets that you want to apply, and then run the tool, as shown in Figure 13-8. Results will be displayed in the standard Output window.

When you specify a rule set to use, this identifies an XML file such as the SecurityRules.ruleset. This is found typically in a location such as %ProgramFiles%\Microsoft Visual Studio 10.0\Team Tools\Static Analysis Tools\Rule Sets\SecurityRules.ruleset. The XML file, in turn, points to one or more managed assemblies that actually contain the rules; in this example, it would be a file such as %ProgramFiles%\Microsoft Visual Studio 10.0\Team Tools\Static Analysis Tools\FxCop\Rules\SecurityRules.dll.

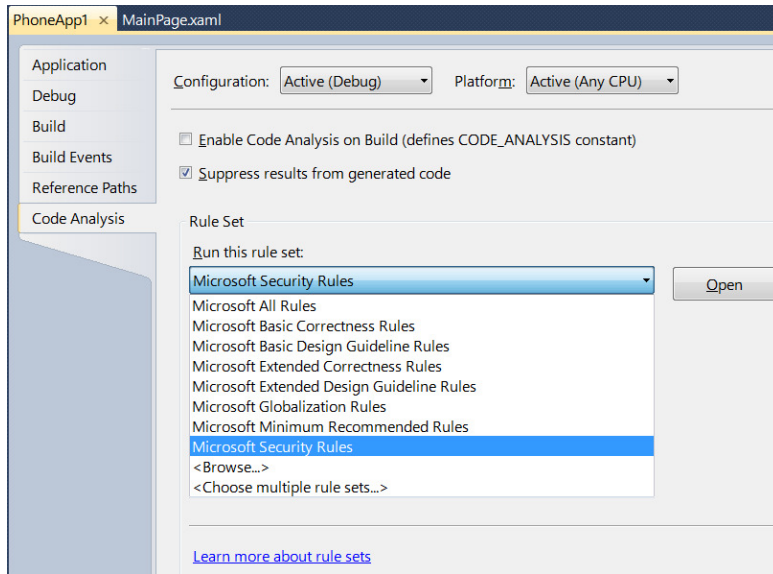


FIGURE 13-8 Configuring static code analysis in Visual Studio.

If you like, you can create a custom rule set by selecting your own combination of existing rules. To do this, you can open one or more standard rule sets in the rule set editor in Visual Studio (via the Open button on the Code Analysis tab). You can then add or remove specific rules in the set, and you can change the action that occurs when code analysis determines that the rule has been violated (None, Warning, Error).

You can also create a custom rule assembly, and then use this rule in a custom rule set. You can see this at work in the *MyCodeAnalysisRule* solution in the sample code. To start, create a new regular Windows class library project (not a Windows Phone project), and then add references to these two FxCop assemblies (assuming that you installed FxCop to the default location):

```
%ProgramFiles%\Microsoft Fxcop 10.0\FxCopSdk.dll
%ProgramFiles%\Microsoft Fxcop 10.0\Microsoft.Cci.dll
```

Because these two assemblies are not in the Global Assembly Cache (GAC), adding them to a project like this means that they will be added by using relative paths. These paths will be relative to the current project (the custom rules project) on the current computer. When the consuming developer wants to use this custom rule in his project on his computer, the paths will almost certainly be invalid. To fix this, you need to change the relative paths in the project file to use the *\$(CodeAnalysisPath)* *MSBuild* property, instead. To do this, in Solution Explorer, right-click the project, and then select Unload Project. When the project has been unloaded, right-click it again, and then select Edit xxx.csproj. You can then edit the project XML. Specifically, replace:

```
<Reference Include="FxCopSdk">
  <HintPath>..\..\..\..\..\Program Files\Microsoft Fxcop 10.0\FxCopSdk.dll</HintPath>
</Reference>
<Reference Include="Microsoft.Cci">
```

```
<HintPath>..\..\..\..\Program Files\Microsoft Fxcop
10.0\Microsoft.Cci.dll</HintPath>
</Reference>
```

with this:

```
<Reference Include="FxCopSdk">
  <HintPath>$(CodeAnalysisPath)\FxCopSdk.dll</HintPath>
  <Private>False</Private>
</Reference>
<Reference Include="Microsoft.Cci">
  <HintPath>$(CodeAnalysisPath)\Microsoft.Cci.dll</HintPath>
  <Private>False</Private>
</Reference>
```

Next, you must add a rule metadata file to the project. This is an XML file that describes your rule. The following rule metadata can be defined in the RuleMetadata file:

- The name of the rule. The *FriendlyName*, *Category*, and *CheckId* will be surfaced to the developer in Visual Studio when she composes a rule set that includes this rule.
- Rule description.
- One or more rule resolutions; that is, instructions to the developer on how to resolve the rule violation.
- The *MessageLevel* (severity) of the rule. This can be set to one of the following:
 - *CriticalError*
 - *Error*
 - *CriticalWarning*
 - *Warning*
 - *Information*
- An attribute of the *MessageLevel* element is the *Certainty* of the violation. This represents the accuracy percentage of the rule. In other words, this field describes the rule author's confidence in how accurate she estimates the rule to be.
- The *FixCategory* of this rule. This specifies whether fixing this rule would require a breaking change; that is, a change that could break other assemblies that reference the one being analyzed.
- The help *Url* for this rule.
- The support *Email* to contact about this rule.
- The name of the *Owner* of this rule.

For example:

```
<?xml version="1.0" encoding="utf-8" ?>
<Rules FriendlyName="My Code Analysis Rule">
  <Rule TypeName="MustNotUsePInvokes" Category="CustomRules.Interop" CheckId="CR0001">
    <Name>Phone apps must not use P/Invokes</Name>
    <Description>A Windows Phone 7 application must not use P/Invokes, or it will fail
      marketplace certification.</Description>
    <Resolution>The use of {0} is not permitted - you must remove this.</Resolution>
    <MessageLevel Certainty="100">CriticalError</MessageLevel>
    <FixCategories>NonBreaking</FixCategories>
    <Url>http://www.contoso.com</Url>
    <Email />
    <Owner />
  </Rule>
</Rules>
```

You must set the build action property of the rule metadata file to *EmbeddedResource* so that it is included in the compiled assembly.

In the code, you need to define at least two classes: one is an abstract class that derives from *BaseIntrospectionRule*; the second is a concrete class that derives from your abstract class.

In the abstract class, you must define a constructor that takes three parameters: the first parameter is the type name of the rule; the second is the resource name (that is, a string composed of the default namespace plus whatever you named the rule metadata XML resource); and the last is your rule assembly. All of the individual rules in your custom rule assembly will typically derive from the same abstract base class. This allows you to specify the XML resource only once.

```
public abstract class BaseRule : BaseIntrospectionRule
{
    protected BaseRule(string ruleName)
        : base(ruleName, "MyCodeAnalysisRule.MyCodeAnalysisRule", typeof(BaseRule).Assembly)
    { }
}
```

For your one and only concrete rule class, define a class that encapsulates the rule that you must not use p/invoke in a Windows Phone application. The use of p/invoke would be flagged during marketplace ingestion as well as by the Capabilities Detection tool. In fact, it would fail to run on the emulator or a developer-unlocked phone, but it is useful to pre-empt these by using a code analysis rule as part of your regular build cycle.

Apart from deriving from your abstract base class, the only other requirement is that you must override the *Check* method. The *BaseIntrospectionRule* base class has six overloads of this method, taking in parameters of different types that correspond to the type of code entity that is to be checked (type names, class members, parameters, and so on). In this example, you're interested in class members, which is where p/invoke (*DllImports*) would be declared. For simplicity, focus on the p/invoke methods (as opposed to other imported types). For each method declared in the project, if the *PInvokeFlags* are not *None*, this must be a p/invoke declaration, in which case you add this to the

ProblemCollection. The *GetResolution* method formats an array of parameters into the resolution text defined in the XML file (using string substitution); in this example, you feed the method signature into the resolution string that ends up being displayed in the errors list in the output window in Visual Studio when code analysis is run.

```
public class MustNotUsePInvokes : BaseRule
{
    public MustNotUsePInvokes() : base("MustNotUsePInvokes")
    {
    }

    public override ProblemCollection Check(Member member)
    {
        Method method = member as Method;
        if ((method == null) || (method.NodeType != NodeType.Method))
        {
            return null;
        }
        if (method.PInvokeFlags == PInvokeFlags.None)
        {
            return null;
        }
        Problem item = new Problem(base.GetResolution(new object[] { method }));
        base.Problems.Add(item);
        return base.Problems;
    }
}
```

When you’ve built your rule assembly, you must copy it to the FxCop Rules folder, typically this resides at %ProgramFiles%\Microsoft Visual Studio 10.0\Team Tools\Static Analysis Tools\FxCop\Rules. You’ll have to shut down Visual Studio and restart before this will be picked up.

To use the custom rule, you must first create an editable custom rule set. You can either open one of the standard rule sets and save it with a new name or create one from scratch. To start from an existing rule set, go to the Analyze menu, and then select Configure Code Analysis for this project. From the drop-down list of rule sets, select the one that you want to start from, and then click Open. In the rule set editor, click Show Rules That Are Not Enabled to see all of the custom rule assemblies in the FxCop Rules folder. Select your custom rule assembly, and then select one or more of your custom rules (in this example, there’s only one), as shown in Figure 13-9.

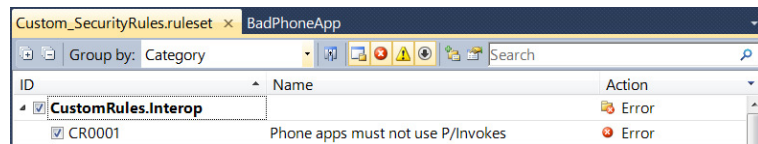


FIGURE 13-9 Selecting a custom rule set in the rule set editor.

When you save your rule set (with a .ruleset extension), the rule set file will contain a reference to each custom rule that you added. If you want to create a rule set file from scratch, you can use the listing that follows as a starting point. Observe that the only custom entries are one or more *Rule Id* elements, each of which corresponds to a rule that you want to include in the rule set.

```
<?xml version="1.0" encoding="utf-8"?>
<RuleSet Name="Custom Security Rules" Description="This rule set contains custom security rules
for Contoso." ToolsVersion="10.0">
  <Rules AnalyzerId="Microsoft.Analyzers.ManagedCodeAnalysis" RuleNamespace="Microsoft.Rules.
Managed">
    <Rule Id="CR0001" Action="Error" />
  </Rules>
</RuleSet>
```

The following phone application includes a *DllImport* declaration, which, in fact, will violate your custom rule.

```
public partial class MainPage : PhoneApplicationPage
{
    [DllImport("user32.dll")]
    static extern bool ExitWindowsEx(uint uFlags, uint dwReason);

    public MainPage()
    {
        InitializeComponent();
    }
}
```

When code analysis for this project is configured to include your custom rule set, and code analysis is run, it will produce a build error, as shown in Figure 13-10.

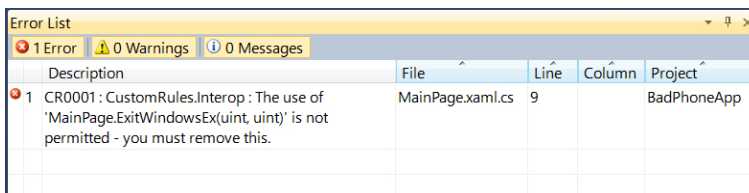


FIGURE 13-10 A build error caused by a custom rule in the custom rule set.

Web Service Security

It is a common pattern for a Windows Phone client application to communicate with a remote web service. Web service–specific security comes into play, regardless of whether you are building a closed system for which you own both the client application and the web service (and the hosting server), or a system wherein your client application is communicating with a public web service that you do not own. There are two levels of security to consider: authentication and authorization. Authentication is about confirming identity. For example, your client application might need to provide some credentials such as a username and password to prove that it is known to the web service. Authorization is

about controlling what access an authenticated application has to secured resources. For example, a web service might constrain the type of data it returns according to the role to which your application belongs. The server must first authenticate the user/client application, and can then make authorization decisions for that authenticated user/application.

Authentication

There are three types of web service authentication that are relevant to Windows Phone applications:

- **Basic authentication** This is a simple protocol, supported by virtually all browsers. With Basic authentication, when the browser makes a request to a web service (or web application), the server returns an HTTP 401 status code. This code informs the user agent that authentication is needed. This prompts the browser to present a logon dialog box to the user. The user then enters his username and password, and these are returned to the server. The credentials are base-64 encoded, which simply transforms the text to ensure that only HTTP-supported characters are used. Herein is the main flaw in Basic authentication: credentials are effectively passed “on the wire” in clear text. For this reason, Basic authentication should be used only in combination with Secure Sockets Layer (SSL), which encrypts the entire communication. Basic authentication plus SSL is a perfectly valid, secure, and very commonly used authentication scheme.
- **Forms authentication** With this approach, requests to the server from unauthenticated users are redirected to a logon page defined by the server application. The UX is therefore very similar to using basic authentication; the difference being that a custom server-side logon page is used instead of a standard client browser dialog. Again, credentials are passed from client to server in clear text, so forms authentication should also only be used with SSL.
- **Windows authentication** This also employs username and password credentials, but the difference is that these are taken implicitly from the context of the currently logged-on user rather than explicitly via a dialog box or logon page. For these credentials to be useful, the current user must have a valid account on the server or in the server’s trusted domain. This is only really useful in intranet scenarios. Clearly, it is not useful for public-facing web services, nor is it useful for Windows Phone client applications.

Forms Authentication

To explore Forms authentication, you can build a web application that combines UI elements with web services. This is the *FormsAuthClient* and *FormsAuthServer* applications in the sample code. In this solution, you want your client phone application to connect to a web service, and you’ll use the web application to provide the server-side logon functionality. Figure 13-11 illustrates the end result from the client perspective.



Note If you want to open the sample code version of the *FormsAuthServer* solution in Visual Studio, you must first set up an application directory for the solution in the Internet Information Services (IIS) Manager. This is part of what Visual Studio does for you when you create a website solution from scratch. To do this, from the Start menu, run the IIS Manager, and then in the Connections tree list, expand out the Default Web Site node. Right-click this node, and then select the Add Application option. In the Add Application dialog, specify *FormsAuthServer* as the alias, and then point the Physical Path to the folder for the solution. The correct folder is the one that contains the web.config file.

The figure displays two side-by-side screenshots of a web application titled "forms auth" under the "CONTOSO" header. Each screenshot shows a login form with "username" and "password" fields, a "go" button, and a "result" field.

Left Screenshot (Successful):

- username: Andrew
- password: [masked]
- go: [button]
- result: Hello (2:13:20 PM)

Right Screenshot (Unsuccessful):

- username: Monkey
- password: [masked]
- go: [button]
- result: Invalid credentials

FIGURE 13-11 Successful (on the left) and unsuccessful (right) attempts to connect to a web service with forms authentication.

Forms Authentication: Server Side

First, you'll create the server-side pieces. In Visual Studio, create a new website. Rather than creating a project, go to the File menu, select New, and then click Web Site. Specify a regular ASP.NET website, select HTTP for the web location, and then provide a suitable URL. For example, enter *http://localhost/FormsAuthServer*. This will generate the project files in the local IIS filesystem. This will create a virtual directory in IIS under the Default website for your computer. This folder will contain all the project code and other files. Note, however, that the solution files (.sln and .suo) will be in the default project location for Visual Studio on your computer, typically in a path such as C:\Users\<username>\Documents\Visual Studio 2010\Projects\.

Now, create a new folder in the solution, named *Hello*. Later on, you will restrict access to your custom web service by virtue of restricting access to the folder where you put the web service implementation. Add a Windows Communications Foundation (WCF) Service to the project. By default, the generated contract and implementation files will be in the special *App_Code* folder, and the *HelloService.svc* file will be in the root folder of the application. If you want to restrict access to the *HelloService*, you must move the *HelloService.svc* file into the *Hello* folder. Do not, however, move the generated *.cs* files; leave them in the *App_Code* folder. You might wonder why it's necessary to go to the bother of creating a *Hello* folder, particularly when you already have an *App_Code* folder; why not simply put the *HelloService.svc* in the *App_Code* folder, and protect that instead? The answer is that the *App_Code* folder has special meaning to IIS. If you attempt to expose a web service from there, you'll get an "invalid segment" error.

Next, although it's not technically necessary, you would generally want to rename the default *IService* contract and *Service* implementation class to something slightly more distinct. For this case, use *IHelloService* and *HelloService*, respectively. Also, change the contract to return a string that you can display in the phone application UI. Here's the contract:

```
[ServiceContract]
public interface IHelloService
{
    [OperationContract]
    String GetData();
}
```

The implementation of this contract returns a dynamically generated string that will be unique for each call, which will help to simplify testing. Note that you must set the service to use Microsoft ASP.NET compatibility mode; this is required if you want to apply forms authentication to a WCF service.

```
[AspNetCompatibilityRequirements(RequirementsMode = AspNetCompatibilityRequirementsMode.
Allowed)]
public class HelloService : IHelloService
{
    public String GetData()
    {
        return String.Format("Hello ({0})", DateTime.Now.ToLongTimeString());
    }
}
```

The behavior with a forms authentication web application is that when the user navigates to the site in the browser, if she's not already authenticated, she will be redirected to a logon page. You don't actually want to force the user to switch from your phone application to a browser session to enter her username and password credentials in the server-side logon page. Instead, you'll set up the phone application UI to allow her to enter her credentials in the application itself, and you'll then forward these to the authentication service, programmatically.

This means that you must expose the authentication service to the application. You're not going to create a custom authentication service from scratch. Instead, you can simply expose the default authentication service that ships with ASP.NET. So, you don't need any code-behind, and you can simply add a text file to expose the service. To do this, add a text file to the root of the web application folder, named `AuthenticationService.svc`.

```
<%@ ServiceHost
    Language="C#"
    Service="System.Web.ApplicationServices.AuthenticationService"
    Factory="System.Web.ApplicationServices.ApplicationServicesHostFactory" %>
```

Note that the standard ASP.NET Visual Studio project template enables forms authentication by default (although only, of course, for the generated web application, not for the additional web services that you're adding). Visual Studio generates a default logon page as part of the application and includes an authentication mode element in the primary `web.config`. You won't make use of the logon page in this solution (because your client application is not browser-based), but you do need the authentication mode set to *Forms*. This should be set by default in the primary `web.config` at the root of the project.

```
<authentication mode="Forms">
    <forms loginUrl="~/Account/Login.aspx" timeout="2880" />
</authentication>
```

The first change you need to make in the `web.config` is to the service hosting environment. You need to make this use ASP.NET compatibility mode so that you can use Forms authentication with your WCF service. The *serviceHostingEnvironment* element is a child of the *system.ServiceModel* element.

```
<serviceHostingEnvironment aspNetCompatibilityEnabled="true"/>
```

Next, enable the WCF *AuthenticationService* endpoint in your web application, by adding a service node in the `web.config`. At the same time, add a second service node for the *HelloService*. Note that you specify a binding configuration that allows cookies. The reason for this is that you need to persist security information across calls to two different services. The client application will first call the *AuthenticationService* to supply logon credentials. When these are accepted by the server, the client will then call the *HelloService* and will need to present proof that the credentials have in fact been accepted. This proof is supplied in the form of a cookie, so you enable both services to use cookies.

```
<services>
    <service
        name="System.Web.ApplicationServices.AuthenticationService"
        behaviorConfiguration="StandardBehavior">
        <endpoint
            binding="basicHttpBinding" bindingConfiguration="CookiesBinding"
            bindingNamespace="http://asp.net/ApplicationServices/v200"
            contract="System.Web.ApplicationServices.AuthenticationService"/>
        </service>
    <service
        name="HelloService"
        behaviorConfiguration="StandardBehavior">
```

```

    <endpoint
      binding="basicHttpBinding" bindingConfiguration="CookiesBinding"
      contract="IHelloService" />
    <endpoint address="mex" binding="mexHttpBinding" contract="IMetadataExchange" />
  </service>
</services>

```



Note For simplicity, in this example, the cookie-enabled binding configuration is unsecured. For an actual production service, you would use SSL (Transport security) with Forms authentication.

```

<bindings>
  <basicHttpBinding>
    <binding name="CookiesBinding" allowCookies="true">
      <security mode="None"/>
    </binding>
  </basicHttpBinding>
</bindings>

<behaviors>
  <serviceBehaviors>
    <behavior name="StandardBehavior">
      <serviceMetadata httpGetEnabled="true" />
      <serviceDebug includeExceptionDetailInFaults="true" />
    </behavior>
  </serviceBehaviors>
</behaviors>

```

Finally, enable the *AuthenticationService* by adding a *system.web.extensions* section.

```

<system.web.extensions>
  <scripting>
    <webServices>
      <authenticationService enabled="true" requireSSL="false"/>
    </webServices>
  </scripting>
</system.web.extensions>

```

So far, you've done all the core work for your web application to use Forms authentication, for your WCF web service to work with the ASP.NET web application, to hook up the standard ASP.NET authentication service, and to expose everything from the one website. Now, you need to determine the security constraints. In other words, now that you've set up Forms authentication, you need to set up some users, and then set up their authorizations.

Some of this work can be done by manually creating and editing text files, but you want to take advantage of the standard SQL membership provider that ASP.NET provides, so you need to use a tool to update the user/roles database. At the top of the Solution Explorer, right-click the ASP.NET Configuration tool button. This brings up the Web Site Administration tool. Navigate to the Security tab, and then create a new user. For this exercise, you can use an arbitrary username, password, and email address, as shown in Figure 13-12.

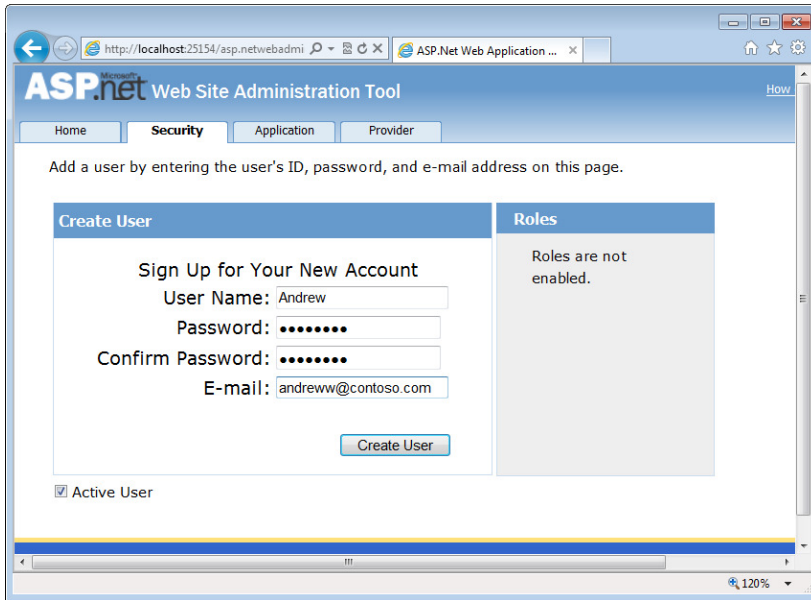


FIGURE 13-12 Creating a new user in the Web Site Administration tool.

Next, on the main Security tab, select Create Access Rules and constrain access to the Hello folder (where the HelloService.svc is found), such that anonymous users are denied and all authenticated users are allowed, as shown in Figure 13-13.

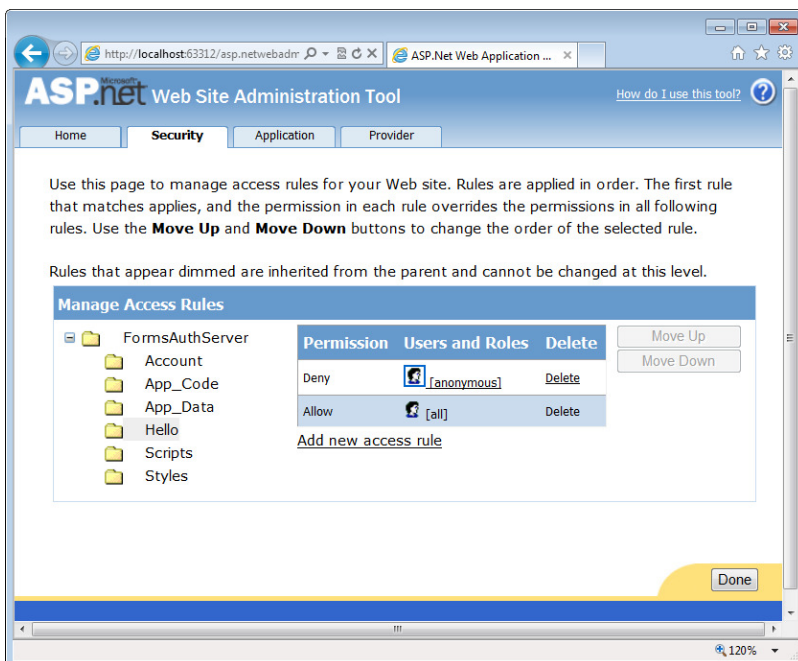


FIGURE 13-13 Denying access to anonymous users.

This will generate a new web.config in the specified folder with the specified authorization settings. Note that this security constraint covers only the *Hello* folder, and therefore, the *HelloService* web service. It does not cover the *AuthenticationService*, because that is exposed at the root folder of the web application, and you need that service to be accessible to anonymous users.

```
<configuration>
  <system.web>
    <authorization>
      <deny users="?" />
    </authorization>
  </system.web>
</configuration>
```

Be aware that this tool is part of Visual Studio, and is not likely to be available on a production server. For production servers (and, indeed, on your development computer), you can use the IIS Administration Tool to manage users and authorization. When Visual Studio generated the initial web application project, it also set up the website in IIS. So, when you open the IIS Administration tool, you'll see the website listed in the connections tree. You can expand this out, and select the *Hello* folder. Then, in the main panel, double-click the Authentication item. This should give you a list of possible authentication schemes, as shown in Figure 13-14. If Forms authentication is not listed, you need to go to Control Panel | Programs And Features | Windows Features, and then turn on IIS Forms Authentication. Also, ensure that this is enabled for the default website and for the FormsAuthServer website.

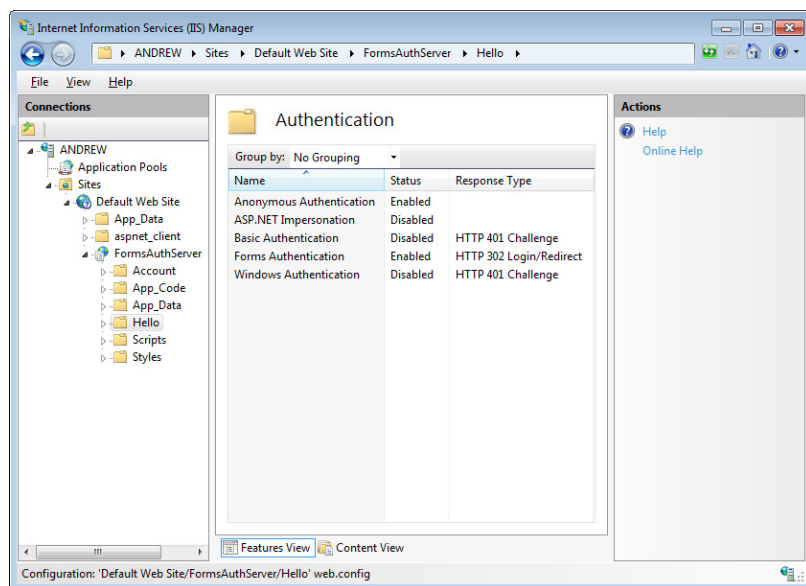


FIGURE 13-14 Authentication schemes in the IIS Administration tool.

When Forms authentication is enabled, select the *Hello* folder, and then in the main panel, double-click the .NET Authorization item to see the same authorization rules that you set in Visual Studio, as shown in Figure 13-15.

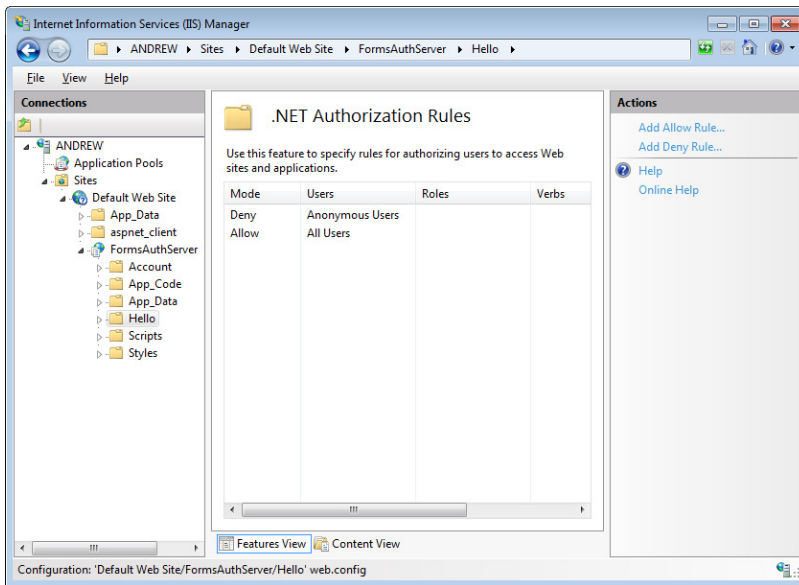


FIGURE 13-15 Authorization rules for the Hello service in the IIS Administration tool.

Forms Authentication: Client Side

Now, it's time to address the phone client application. First, set up the UI. This should include a few *TextBlock* and *TextBox* controls, a *PasswordBox* for the password field, and a *Button* to invoke the service. Next, add service references to both the *AuthenticationService* and the *HelloService*. Given the virtual directory structure used on the server side, these will likely be at URLs such as *http://localhost/FormsAuthServer/AuthenticationService.svc* and *http://localhost/FormsAuthServer/Hello/HelloService.svc*.

Adding service references will generate the client-side service proxy code, as normal. It will also generate a *ServiceReferences.ClientConfig* file, which specifies the service connections and configurations. The endpoints should match the referenced services.

```
<endpoint address="http://localhost/FormsAuthServer/Hello/HelloService.svc"
  binding="basicHttpBinding" bindingConfiguration="BasicHttpBinding_IHelloService"
  contract="HelloReference.IHelloService" name="BasicHttpBinding_IHelloService" />
<endpoint address="http://localhost/FormsAuthServer/AuthenticationService.svc"
  binding="basicHttpBinding" bindingConfiguration="BasicHttpBinding_AuthenticationService"
  contract="AuthenticationReference.AuthenticationService" name="BasicHttpBinding_
AuthenticationService" />
```

The only change you need to make to this is to ensure that cookie containers can be used across the two services. To do this, add the *enableHttpCookieContainer* attribute to both service bindings, and then set it to *true*.

```
<binding
  name="BasicHttpBinding_IHelloService"
  maxBufferSize="2147483647" maxReceivedMessageSize="2147483647"
  enableHttpCookieContainer="true">
  <security mode="None" />
</binding>
<binding
  name="BasicHttpBinding_AuthenticationService"
  maxBufferSize="2147483647" maxReceivedMessageSize="2147483647"
  enableHttpCookieContainer="true">
  <security mode="None" />
</binding>
```

The remaining work is relatively straightforward. When the user taps the Go button, you create a *CookieContainer* and add it to the *AuthenticationService* proxy object. Then, hook up the *Login Completed* event and invoke the *LoginAsync* method to pass the user-supplied username and password credentials to the server. You can use the third parameter to this method to pass additional user credentials, if the service requires it. When this asynchronous call returns, you can retrieve the returned cookie and add it to the proxy for the *HelloService*. Next, handle the *GetDataCompleted* event, and then invoke the *GetDataAsync* method. Finally, when that method returns, you can extract the result and display it in the UI—or an error message if the call failed. In this case, the most likely cause is that the user credentials were invalid, although that’s not the only possibility.

```
private void Go_Click(object sender, RoutedEventArgs e)
{
    AuthenticationServiceClient authClient = new AuthenticationServiceClient();
    authClient.CookieContainer = new CookieContainer();
    authClient.LoginCompleted += authClient_LoginCompleted;
    authClient.LoginAsync(this.Username.Text, this.Password.Password, "", true);
}

private void authClient_LoginCompleted(object sender, LoginCompletedEventArgs e)
{
    if (e.Error == null)
    {
        AuthenticationServiceClient client = (AuthenticationServiceClient)sender;
        HelloServiceClient helloClient = new HelloServiceClient();
        helloClient.CookieContainer = client.CookieContainer;
        helloClient.GetDataCompleted += helloClient_GetDataCompleted;
        helloClient.GetDataAsync();
    }
    else
    {
        this.Result.Text = "Login failed";
    }
}
```

```
private void helloClient_GetDataCompleted(object sender, GetDataCompletedEventArgs e)
{
    if (e.Error == null)
    {
        this.Resultt.Text = e.Result;
    }
    else
    {
        this.Resultt.Text = "Invalid credentials";
    }
}
```

If the user supplies credentials that match one of the users created on the server, all will be good. If not, then the server will attempt to redirect the user to a logon page. The phone application suppresses this (because you want the user to stay in your phone logon UI, not a server-side logon page) and instead displays an error message to the user.

So far, you've restricted access to your web service to only authenticated users. However, this is a fairly open security policy. It is more common to restrict different subsets of authenticated users to different sets of resources. One way to achieve this is to laboriously list all individual users who you either want to allow or deny access to some protected resource.

```
<authorization>
  <deny users="?" />
  <allow users="Andrew, Sally, Joe, Bill, Millie"/>
</authorization>
```

Clearly, this is not a scalable approach. The alternative is to make use of *roles*. That is, create logical groupings that map conveniently to the different sets of protected resources, and then assign individual users to one or more of those groups. This is another feature offered by the standard ASP.NET SQL membership provider and the standard Web Site Administration tool.

To set up roles in the tool, go to the Security tab, click the Roles panel, and then select Enable Roles. When the page refreshes, it brings up an additional link, Create Or Manage Roles. Click this link to add a new role; for example, the *Employees* role. Once you've created a role, you can then start adding users—either existing users, or new ones. For now, just add the one user to the role.

Next, create a new user, but do not add this user to the *Employees* role. You now have roles enabled, and you have at least one role set up with at least one user as a member and another user who is not a member of this role. Now you can set up authorization based on this role. First, add a new access rule in the tool: select the Hello folder, and add a rule to allow access to the *Employees* role. Then, add another access rule to deny access to all users. The order of these rules is important, which is why the tool provides Move Up and Move Down buttons, as shown in Figure 13-16. The ASP.NET rules will be evaluated in order, top to bottom. So, in this example, you first deny access to all anonymous users. Then, you allow access to any users in the *Employees* role. Finally, you deny access to all other users. The last rule, which allows access to all users, is the default rule which cannot be removed (because it is inherited from the parent website). This rule must be last in the list; this means that because the previous rule (deny all) already covers all other users, the default rule becomes a no-op.

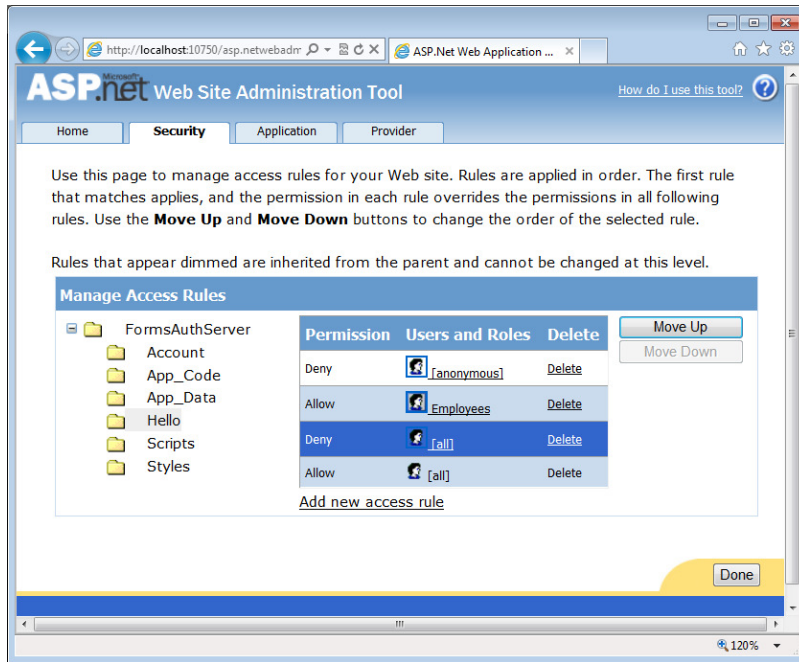


FIGURE 13-16 Configuring access for roles and users.

This configuration should update the web.config for the Hello folder, and the authorization section should list the explicit rules.

```
<authorization>
  <deny users="?" />
  <allow roles="Employees" />
  <deny users="*" />
</authorization>
```

Now, when the user enters her credentials on the phone, the user who is a member of the *Employees* role will successfully call the *Hello* service, but any users who are not a member of the *Employees* role will fail.

Note that the users and roles are stored in a SQL database for your website. By default, this will be in a file named ASPNETDB.MDF in the App_Data folder of your website. You can double-click this in Visual Studio to open it in the Server Explorer, and then you can drill down to examine the table structure and the table data, as shown in Figure 13-17. As you would expect, the passwords and other sensitive credentials in this database are encrypted (email addresses, however, are not).

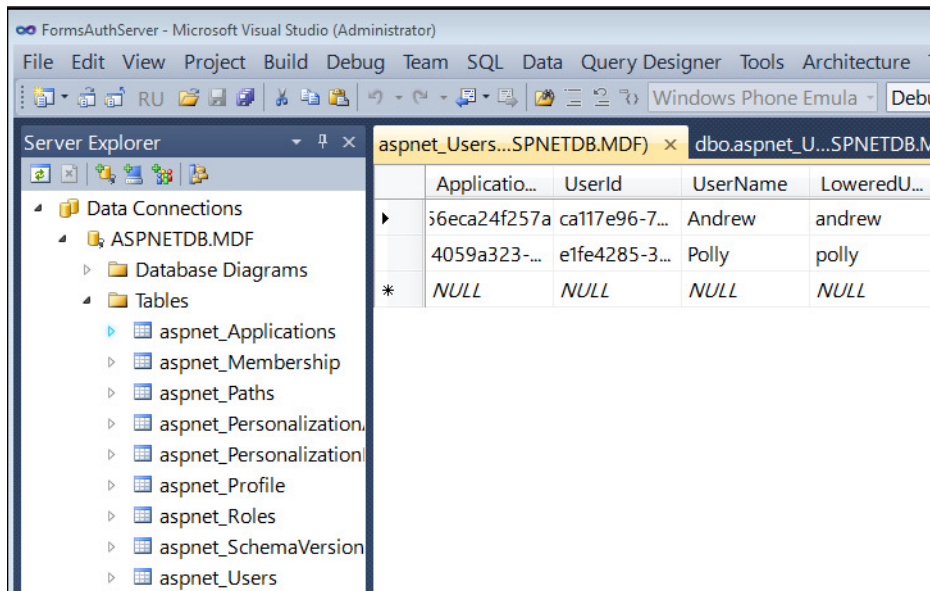


FIGURE 13-17 A standard ASP.NET SQL database.

It's worth repeating that so far, you have still been passing user credentials across the wire in clear text. Forms authentication should only be used in combination with SSL to encrypt the data on the wire. SSL is discussed later in this chapter.

Basic Authentication

In theory, Basic authentication is the simplest method to implement. Certainly, setting up Basic authentication for a web application or web service, and then connecting to these from a traditional browser client or Windows clients of various kinds, is simple and well understood. However, setting up a connection from a Windows Phone client is only minimally documented, and not at all obvious.

In this section, you'll re-implement the *MagicalManateeService* WCF web service from earlier chapters and configure it for Basic authentication. Before you start, ensure that IIS on your developer computer has Basic authentication installed. If not, go to Control Panel | Programs And Features, and then click Turn Windows Features On Or Off. From the list, check the Basic Authentication item. This is the *BasicAuth* client and server solutions in the sample code.

In Visual Studio, create a new WCF Service Application project and call it *MagicalManateeService*. Rename the default *IService* contract and Service implementation class to *IMagicalManateeFacts* and *MagicalManateeFacts*, respectively. Implement *MagicalManateeFacts* to return a string from a static collection of strings. For implementation specifics, see the version described in Chapter 11, "Web and Cloud."

In the web.config for the WCF service, define a binding configuration that specifies a security mode of *TransportCredentialOnly*, and then set the transport *clientCredentialType* attribute to *Basic*. Assign this binding configuration to the endpoint for the service.

```
<behaviors>
  <serviceBehaviors>
    <behavior name="basicAuthBehavior">
      <serviceMetadata httpGetEnabled="true"/>
      <serviceDebug includeExceptionDetailInFaults="true"/>
    </behavior>
  </serviceBehaviors>
</behaviors>

<bindings>
  <basicHttpBinding>
    <binding name="basicAuthBinding">
      <security mode="TransportCredentialOnly">
        <transport clientCredentialType="Basic"/>
      </security>
    </binding>
  </basicHttpBinding>
</bindings>

<services>
  <service
    name="MagicalManateeService.MagicalManateeFacts"
    behaviorConfiguration="basicAuthBehavior">
    <endpoint
      binding="basicHttpBinding"
      bindingConfiguration="basicAuthBinding"
      contract="IMagicalManateeFacts"/>
    </service>
</services>
```

Go to the Project settings, select the Web tab, and then click Use Local IIS Web Server. Enter a suitable URL or accept the default, which is something like `http://localhost/MagicalManateeService`. Next, click the Create Virtual Directory button. This will provision the application virtual directory in IIS. In the IIS Administration tool, refresh the connections list if necessary, then navigate to the new *MagicalManateeService* virtual directory. Double-click the Authentications item in the main panel, and then enable Basic Authentication, as shown in Figure 13-18. That's all the work you need to do for the server, but do take note of the warning in the tool: Basic authentication should be used only in combination with SSL. You'll see how to fix this later.

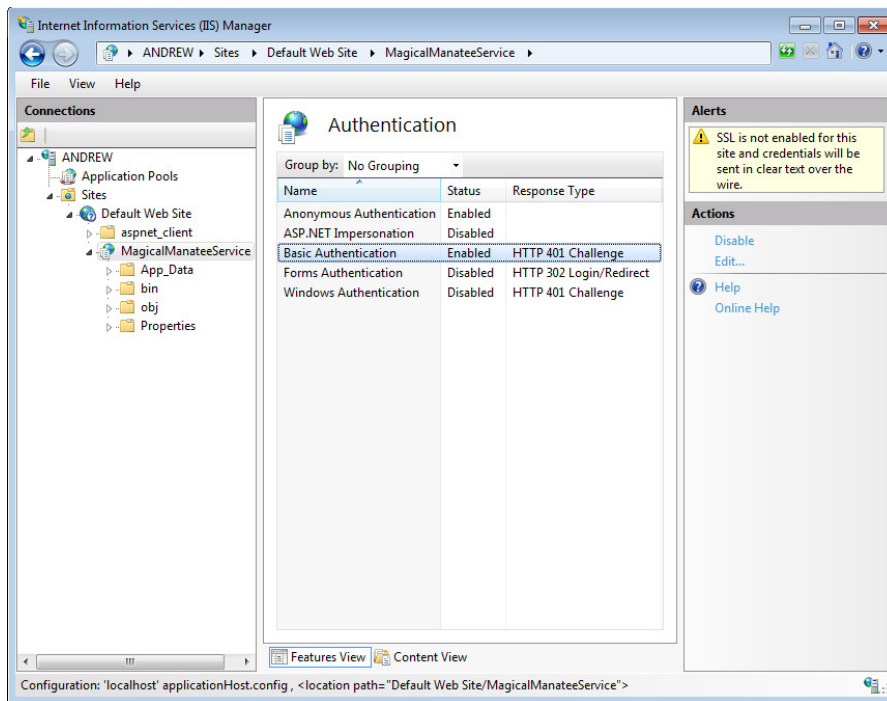


FIGURE 13-18 Enabling Basic Authentication for the MagicalManatee WCF Service.

The client phone application project provides a *TextBox* for the user to enter a username, and a *PasswordBox* to enter a password, as shown in Figure 13-19.

CONTOSO

basic auth

username

MYDOMAIN\Andrew

password

••••••••

go

results

The Magical Manatee grinds his coffee with his teeth and boils the water with his own rage.

The Magical Manatee brushes his teeth with a mixture of iron shavings, industrial paint remover, and wood-grain alcohol.

The Magical Manatee eats a bowl of diamonds every morning.

FIGURE 13-19 Basic authentication in a client application.

In the client project, add a service reference, specifying the URL to the running *MagicalManatee Service*; for example, `http://localhost/MagicalManateeService/MagicalManateeFacts.svc`. Because this is now set to use Basic authentication, the Add Service Reference Wizard will throw up a logon dialog when it tries to connect to the service. At this point, you should enter your domain credentials, and then click OK to continue. In addition to generating the service proxy class, this will create or update the client-side `ServiceReferences.ClientConfig`. The binding for the service in the client config will include the security mode element, set to *TransportCredentialOnly*. This maps to the equivalent setting in the server-side `web.config`.

```
<system.serviceModel>
  <bindings>
    <basicHttpBinding>
      <binding
        name="BasicHttpBinding_IMagicalManateeFacts"
        maxBufferSize="2147483647" maxReceivedMessageSize="2147483647">
        <security mode="TransportCredentialOnly" />
      </binding>
    </basicHttpBinding>
  </bindings>
  <client>
    <endpoint
      address="http://localhost/MagicalManateeService/MagicalManateeFacts.svc"
      binding="basicHttpBinding" bindingConfiguration="BasicHttpBinding_
IMagicalManateeFacts"
      contract="MagicalManateeService.IMagicalManateeFacts" name="BasicHttpBinding_
IMagicalManateeFacts" />
    </client>
  </system.serviceModel>
```

If you run the client now and try to connect to the service, you'll get a *NotFound WebException*. Obviously, you need to supply credentials on the call. The documentation on WCF client proxies states that you have the option to pass username and password details in the *ClientCredentials* property of the proxy object, as shown in the code example that follows. This code retrieves the username and password entered by the user in the *Username TextBox* and the *Password PasswordBox*.

```
if (service == null)
{
    service = new MagicalManateeFactsClient();
    service.GetFactCompleted += service_GetFactCompleted;
    service.ClientCredentials.UserName.UserName = Username.Text;
    service.ClientCredentials.UserName.Password = Password.Password;
}
service.GetFactAsync();
```

However, the documentation is for general-purpose client proxies only; the preceding code will work with Windows clients of all kinds, but not with Windows Phone clients. The reason is that a mobile device has additional security challenges, and the Windows Phone application platform is therefore more security constrained than a desktop. Specifically, the *ClientCredentials* values are ignored if the communication is using HTTP and not HTTPS.

Now, the correct thing to do here is to use HTTPS/SSL. This is the only way Basic authentication should be used in production. However, during development it can sometimes be useful to test your solution without enforcing SSL. In this context, you can force the use of Basic authentication over HTTP. The credentials would normally be added to the message headers as a base-64 encoded string. To achieve this by using the wizard-generated proxy classes, you need to work with the *InnerChannel* property of the service proxy object. Use this to initialize an *OperationContextScope* object. Then, construct a base-64 encoded string from your credentials, in the form "<domain>\<username>:password". Finally, add this string to the current *OperationContext*. Under the covers, this ultimately adds it to the message headers.



Note *NOT_SSL* is a custom conditional compilation symbol that is defined in this example for a corresponding custom solution configuration. The solution includes a custom configuration based on the standard Debug configuration as well as a custom configuration based on the standard Release configuration. The reason why you're doing this here instead of simply using the standard Debug configuration and the standard *DEBUG* compilation symbol is that it is common to test a Release build against internal servers as well as test the Debug build.

```
if (service == null)
{
    service = new MagicalManateeFactsClient();
    service.GetFactCompleted += service_GetFactCompleted;
#if !NOT_SSL
    service.ClientCredentials.UserName.UserName = Username.Text;
    service.ClientCredentials.UserName.Password = Password.Password;
#endif
}

#if NOT_SSL
using (OperationContextScope scope = new OperationContextScope(service.InnerChannel))
{
    HttpRequestMessageProperty message = new HttpRequestMessageProperty();
    message.Headers[System.Net.HttpRequestHeader.Authorization] =
        "Basic " + Convert.ToBase64String(Encoding.UTF8.GetBytes(
            Username.Text + ":" + Password.Password));
    OperationContext.Current.OutgoingMessageProperties.Add(
        HttpRequestMessageProperty.Name, message);
    service.GetFactAsync();
}
#else
    service.GetFactAsync();
#endif
```

Both Basic authentication and Forms authentication have the vulnerability that, if used with an unsecured channel (such as the default HTTP), the user credentials are passed across the wire in clear text. An attacker could intercept the communication and gain access to the user credentials. Both types of authentication should only be used in production systems in combination with SSL.

SSL

SSL is a standard for securing Internet connections. Its main purpose is to encrypt HTTP communications. The encryption keys are contained in SSL certificates, also known as Transport Layer Security (TLS) certificates—typically, X.509 documents—which are used by both the client and the server. SSL provides two critical assurances: first, that the server is authentic, and second, that an attacker cannot intercept and read or tamper with the data being exchanged between the client and the server (so long as you don't use a null cipher for the connection). It is an effective defense against server spoofing, channel tampering, and man-in-the-middle attacks.



Note A man-in-the-middle attack is a form of eavesdropping or message tampering, in which an attacker manages to intercept communications between two parties and fools them into thinking that they're talking directly to each other, when in fact the conversation is controlled by the attacker.

The server provides the SSL certificate for the session and sends the certificate to the client in the handshake phase of establishing the communication channel. The server's certificate must be valid and issued by a trusted authority. It must chain to one of the certificate authorities (CAs) in the phone's trusted authorities list. You can add a CA to the trusted authorities list. Note that SSL mutual authentication (whereby the client sends its certificate to the server) is not supported, because you cannot add a client certificate to the phone's trusted authorities list.

During development, you can create a self-signed root certificate and install this on the phone. Of course, you must not use a self-signed certificate in production, but this is a common approach during development. There is no certificate management UI on the phone, but installing a certificate on the phone can be done in two ways, as described in the following:

- Email the certificate to yourself as an attachment, and then open the email and the attachment on the phone. This approach works only on the phone, not the emulator.
- Make the certificate available on a website accessible to the phone. This approach works with both the emulator and a physical phone.

In both cases, you would tap the certificate to open it; the default behavior when opening a certificate file on the phone is to install it into the trusted authorities list.

In the following exercise, you'll create a web service that uses Basic authentication secured with SSL. You'll also set up a self-signed certificate and configure the HTTPS binding for the website to use this certificate. Then, you'll install the certificate on the phone so that the phone client application can connect to the secured service. You can see this at work in the SSL client and server solutions in the sample code. There's also a Windows Phone Certificate Installer helper library on codeplex that automates most of the following steps. This is described at the end of this section.

First, create a new WCF Service Application and rename the contract and implementation class to *ISafeService* and *SafeService*, respectively. Code a simple implementation of the service to return a unique string on each call.

```
[AspNetCompatibilityRequirements(
    RequirementsMode = AspNetCompatibilityRequirementsMode.Allowed)]
public class SafeService : ISafeService
{
    public String GetData()
    {
        return String.Format("Hello ({0})", DateTime.Now.ToLongTimeString());
    }
}
```

On the Properties page for the project, on the Web tab, under Servers, select Use Local IIS Web Server, and then modify the default URL to use HTTPS instead of HTTP (for example, <https://localhost/SafeService>). Then, click Create Virtual Directory. This sets up the website, the virtual directory in IIS, and configures it for HTTPS.

Now for the certificate. There are two ways to create this: if you're using Internet Information Services (IIS) 7.0, you can use the IIS Manager tool to create self-signed certificates, or alternatively, you can use the `makecert` command-line tool. Using the first approach, click the Start button on your computer, type **IIS**, and then select Internet Information Services (IIS) Manager. In the IIS Manager tool, in the Connections list, select your local computer. In the main panel, under IIS, double-click Server Certificates. From the panel on the right side, click the link to create a new self-signed certificate, and then specify a friendly name such as `MyCertificate`.

Apart from ease of use, using the IIS Manager tool has the added advantage that it installs the certificate in both your personal user certificate store and also the Trusted Root Certification Authorities (CA) store on the computer. The disadvantage is that you have no choice about the Common Name (CN) for your certificate. The CN is the internal name; this is the name the certificate is "issued to." This is not the friendly name. When using self-signed certificates in a phone application, the CN must be the same as the computer name you're testing against. For example, if your computer name is `andrew.contoso.com`, then your certificate CN must also be `andrew.contoso.com`. IIS Manager will set the CN to the local computer name. This will work, so long as you continue to develop/test on the same computer.

While still in the IIS Manager tool, select the website to which you deployed the `SafeService` (typically, the default website). In the Actions panel, click Bindings. In the Site Bindings dialog, add a new site binding, and then specify type HTTPS. Click the SSL certificate drop-down list, and then select `MyCertificate`, as shown in Figure 13-20. Confirm everything and close all dialogs.

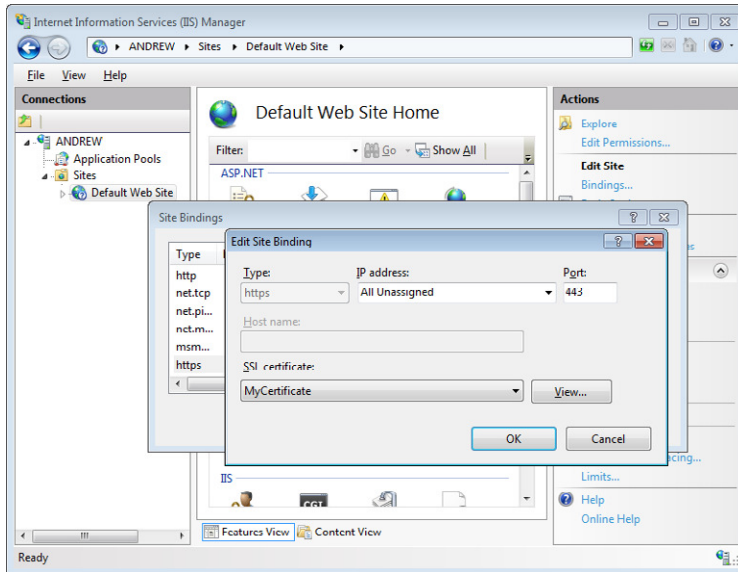


FIGURE 13-20 Configuring the website HTTPS bindings.

Next, select the SafeService application, double-click SSL settings, select the Require SSL check box, and then in the Actions panel, click Apply. This forces all communications with this website to go through HTTPS. Now, configure the web service for Basic authentication. For details on how to do this, review the previous section on Basic authentication.

Next, you need to make the certificate available to the phone. Recall that the IIS Manager tool generated the certificate and added it to the certificate store on the developer computer. So, you now need to export the certificate to a file. To do this, use the MMC Certificate Snap-in. From the Start button, type **MMC** to run the Microsoft Management Console. From the file menu, click Add/Remove Snap-In, and then select Certificates. Specify that this snap-in will always manage certificates for the Computer account (that is, the local developer computer). When the certificate snap-in is installed, expand out the Trusted Root Certification Authorities node, and then the Certificates node. From the list, select the MyCertificate certificate. Right-click, select All Tasks, and then click Export. The Certificate Export Wizard opens. In the wizard, export the certificate, including the private key (which will need password protection), as a Personal Information Exchange PKCS #12 (PFX) file, as shown in Figure 13-21. You can use any arbitrary name for this (for example, MyCertificate.pfx).

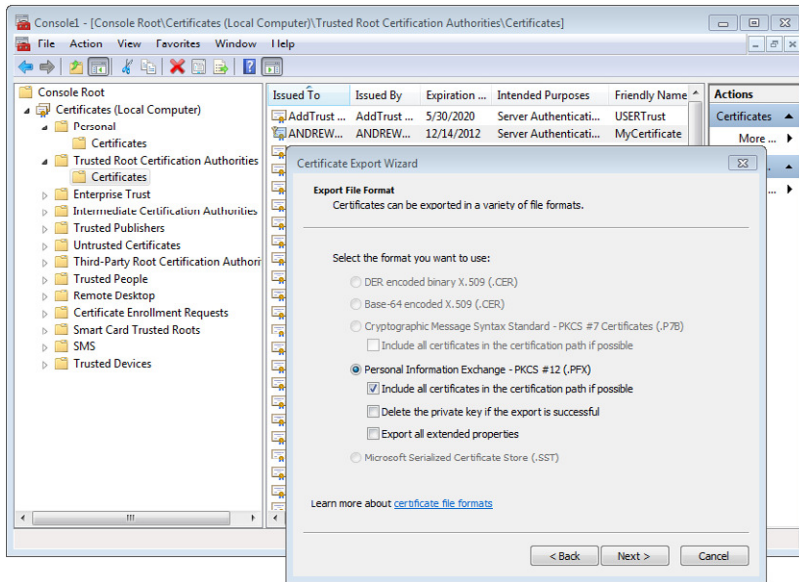


FIGURE 13-21 Exporting the self-signed certificate from the Trusted Root CA store.

If you're working directly with a device, you can now email the certificate to an email account that you have access to on the phone. If you're working with the emulator (where there are no email accounts), you need to make this PFX file available on a website. You can't use your existing website, because that is configured to *require* HTTPS, and you won't be able to access it until you've installed the certificate (catch-22). We could configure your website so that it doesn't require SSL; that wouldn't mean that you can't use SSL, just that it's not mandated. This would be one way to proceed.

Alternatively, you can create another website to host this certificate file. This is not an unreasonable approach (to house our certificate in one place, while you might have multiple other sites for your web services). Taking this approach, in Visual Studio, create a new WCF Service Application project called, for example, *CertificateHost*, and then delete the *IService* and *Service.svc* files (which also deletes the *Service.svc.cs* file). This leaves you with a blank website. On the Properties page, on the Web tab, change the settings to use the local IIS instead of the Visual Studio server, and then click the Create Virtual Directory button. In the IIS Administration tool, turn on directory browsing for this website. This ensures that you can browse to find files on the site. Add the exported *MyCertificate.pfx* file to the project, at the root folder.

Build the solution, and then run it (press F5, or right-click the *SafeService.svc* file in the Solution Explorer, and then select View In Browser). In Internet Explorer, you should see the "Problem With This Website's Security Certificate" anti-phishing warning page. Click the Continue To This Website (Not Recommended) option. You will be prompted to enter domain user credentials; enter these, and then confirm that you can get to the virtual directory and see the *MyCertificate.pfx* file listed there.



Note If you want a simpler alternative to creating a special virtual directory, you could just drop the MyCertificate.pfx file into the default website folder; that is, \inetpub\wwwroot. This might be appropriate if the certificate is used across multiple web applications and/or services for that website, although you should consider carefully before adding arbitrary files to your default website folder.

Now that everything is set up on the server side, you can create a client phone application. Figure 13-22 presents a simple example, wherein you expect the user to supply logon credentials and then tap the button to go to the web service.

CONTOSO

ssl client

username Andrew

password

call web service

results

clear

FIGURE 13-22 A client application connecting to an SSL-secured web service

Add a service reference, using the SafeService.svc URL. Enter your domain credentials to go ahead and generate the proxy code. At this point, build and run the client. The following error should appear:

"There was no endpoint listening at https://localhost/SafeService/SafeService.svc that could accept the message. This is often caused by an incorrect address or SOAP action. See InnerException, if present, for more details."

What's missing is that you have not yet installed the root CA certificate on the phone. To fix this, use Internet Explorer on the phone to navigate to the MyCertificate.pfx file on the CertificateHost website (using HTTP not HTTPS). The phone will recognize the .pfx file type and prompt you to install the certificate on the phone. Accept all the prompts to install, as shown in Figure 13-23 (using the same password with which you exported the certificate).

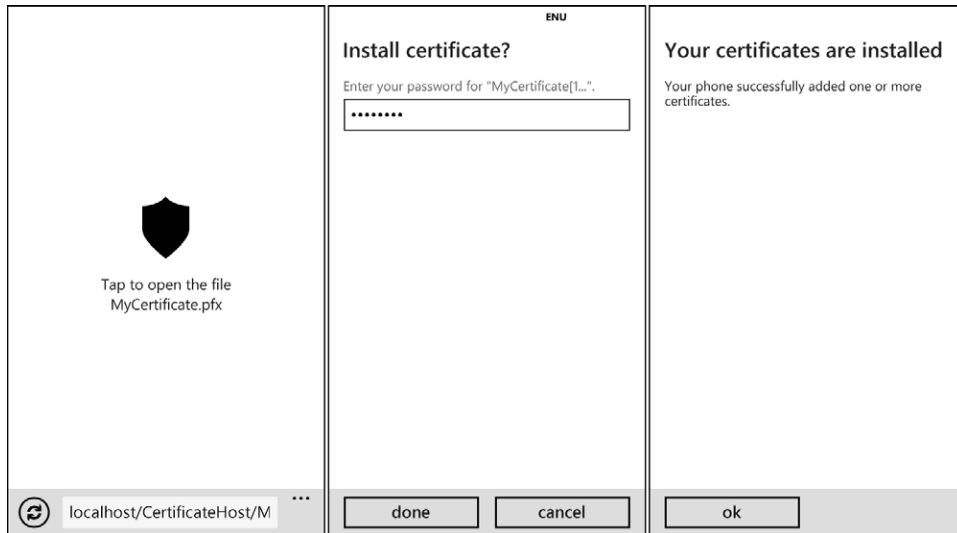


FIGURE 13-23 Installing the self-signed certificate on the phone.

Finally, go back (or forward) to the application on the phone and try connecting again. This time, it should work.

The preceding steps to set up a self-signed certificate, make it available on a website, and install it to the phone are complicated and error-prone. David Hardin of Microsoft has released a very useful helper library that dramatically streamlines this whole process. This is freely available for download in source-code format from <http://wp7certinstaller.codeplex.com/>. The WP7CertInstaller project includes the following pieces:

- A batch file that contains makecert commands to create a Root CA certificate and a second SSL certificate that chains to this Root CA.
- Source code for a website for hosting your certificates.
- Helper classes to automate the process of fetching certificates and installing them.
- A sample Windows Phone application that shows how to use the helpers.
- A Windows Azure cloud project, in case you want to host your certificates in Windows Azure.

The general flow for using WPCertInstaller is as follows:

- Modify the names in the makecert batch file to create the certificates you want, and then run it.
- Build the website or Windows Azure cloud project.
- Apply the SSL certificate to the HTTPS bindings for the website that you want to secure.

- At a suitable point in your phone application, before you want to connect to the secured website, you use the helper class to install the certificate. The sample project does this in the *Loaded* event handler for *MainPage*, which is a good early point to perform the certificate check.
- The helper class checks to see if the certificate is already installed; if it is not, the helper class launches a *WebBrowserTask*, and then navigates to the certificate URL.
- The user taps the certificate to install it, and then taps Back to return to your application. This time, the check for installed certificate succeeds, and the application can now connect to the secure website.

Push Notification Security

Chapter 12, “Push Notifications,” discusses push notifications and the Microsoft Push Notification Service (MPNS). Recall that there are three main pieces:

- A custom web service that generates the push notification messages, and then sends them to the MPNS.
- The MPNS, which distributes the messages to each subscribed phone.
- A client phone application that handles the notifications.

The communications between MPNS and the Push Client on the phone are secured via SSL. This is set up by Microsoft. However, communications between the phone and your push web service are secured only if you explicitly secure them. The general recommendation is the same as for any web service that can exchange sensitive data: you should protect it with SSL.

Setting up SSL authentication for your web service is no different for a push web service than for any other web service. An additional benefit of securing your service is that you are no longer constrained to a daily limit on the number of push notifications that you can send. Recall that, by default, unsecured web services are throttled at a rate of 500 notifications per day. To enable this feature, you must upload a TLS (SSL) certificate to the marketplace. The key-usage value of this certificate must be set to include client authentication, and the Root CA for the certificate must be one of the CAs that are trusted on the Windows Phone platform (these are listed in the documentation on the marketplace).

After you have submitted the certificate to marketplace, you can associate any subsequently submitted applications with this certificate. This option will be available during the application submission process. There is obviously a gap between setting up your secured web service and submitting your certificate on the one hand, and having a submitted application approved and published in the marketplace on the other hand. To bridge this gap, Microsoft will flag your web service as authenticated for a period of four months only; however, this time constraint is removed when your application is successfully published to marketplace.

To use the authenticated channel from your phone client application, when creating the *Http NotificationChannel*, set the service name to the CN in the certificate. Note that you cannot use a self-signed certificate for this purpose.

If you do authenticate your push web service in this way, you can also take advantage of a call-back registration feature. This feature enables the MPNS to callback on your registered URI when it determines that it cannot deliver a message to the device as a result of the device being in an inactive state.

It's also worth keeping in mind that the push service does not guarantee message delivery, so you should not use it in scenarios for which failure to deliver one or more messages has serious consequences.

OAuth 1.0

As of this writing, some public web services, such as Twitter, Flickr, Vimeo, Yahoo!, and SmugMug, use the OAuth 1.0 specification for security. OAuth is an Open Web specification that provides a method for users to grant third-party access to their resources without sharing their passwords. So, as a Twitter user, you can grant some arbitrary Windows Phone application permissions to access your Twitter information, without handing over your Twitter credentials to that application.

The sequence of operations is shown in Figure 13-24. Essentially, the phone client application asks for a Request token from the secured website. This token represents temporary credentials that are not user-specific. The token is only good for the next step of requesting user authorization, and does not provide any direct access to the protected user data. The user must directly logon to the secured website and grant the requested authorization. This cannot be done programmatically. Once authorization is granted, the secured website will return an Access token to the requesting client application. This access token can then be used to gain access to the user's protected data on his behalf.

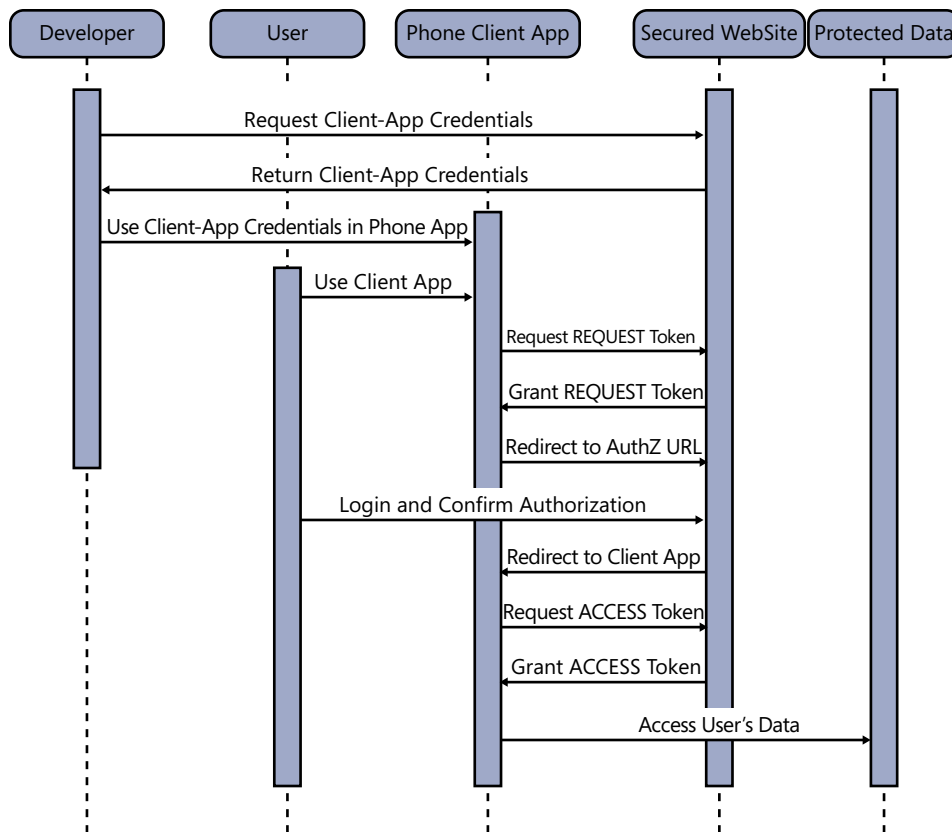


FIGURE 13-24 The OAuth 1.0 authorization sequence.

The key is that the username and password credentials of the user are utilized only by the user directly in the secured server's authorization page; they are never accessible to the client application at all. The authorization request specifies exactly what kind of access to which protected resources is being requested. The specifics of this will vary with each secured website. It might be that the developer needs to specify a list of specific data items to which it wants to be granted access, or a specific list of categories of data. Typically, the type of access is also required; that is, read-only, read-write, create, delete, and so on.



Note As stated here, the user enters his credentials in the secured server's authorization page, not in the client application. However, it is important to note that a malicious application could attempt to steal the user's credentials by injecting script into the web browser or by spoofing the entire authorization site. So, the most secure approach would be to gather the access token in a way that is completely outside the application. For example, you could instruct the user to press the Start button on the phone, navigate to Internet Explorer, log on to the real site, obtain the secure access token and copy it to the clipboard, and then return to the application and paste it in. Unfortunately, very few users would understand why you're asking them to do this, when they see other applications that host the browser inside the application, as just described.

OAuth 2.0

As of this writing, some public web services, such as Facebook, Google, and MySpace, use the OAuth 2.0 specification for security. The sequence of operations is shown in Figure 13-25. OAuth 2.0 is simpler to use than OAuth 1.0, but offers the same security model, and behind the scenes, it behaves in essentially the same manner.

The phone client application constructs an authorization URL string, which includes the developer's client application credentials and the target site to be accessed. Then, the client application navigates to the secured server's authorization site using this URL string. The server presents logon and authorization UI to the user. As with OAuth 1.0, the user must directly log on to the secured website and grant the requested authorization. This cannot be done programmatically. Once authorization is granted, the secured website will redirect to the originally requested target secured site. The phone client application handles the Navigated event on the *WebBrowser* control, and extracts the access token from the navigation event arguments. This access token can then be used to gain access to the user's protected data on his behalf.

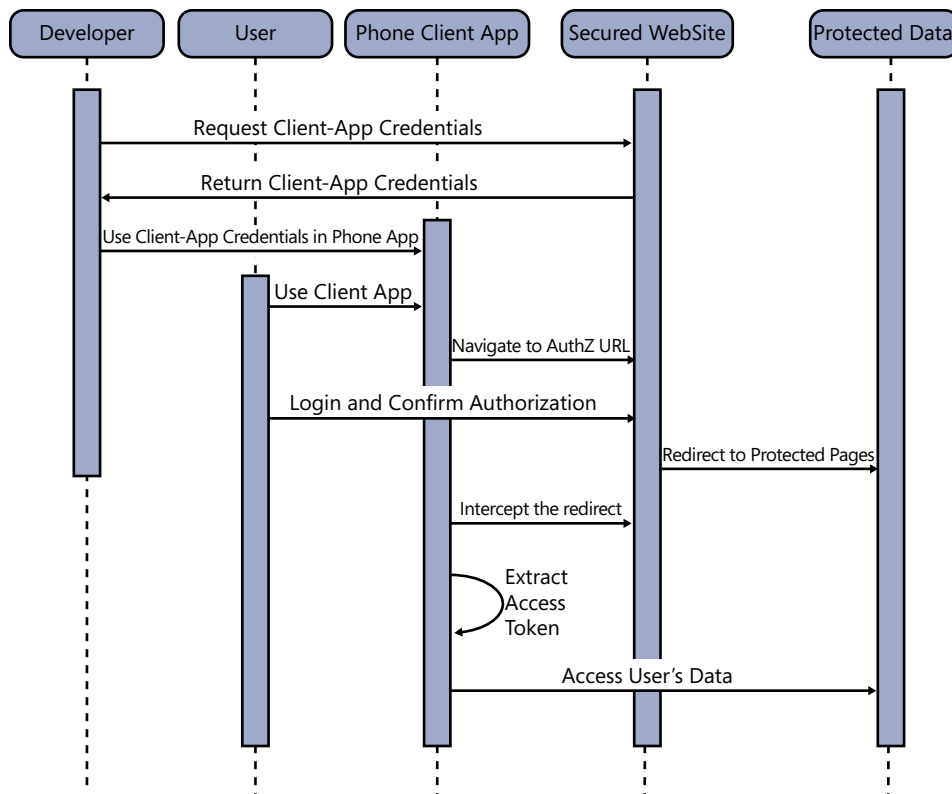


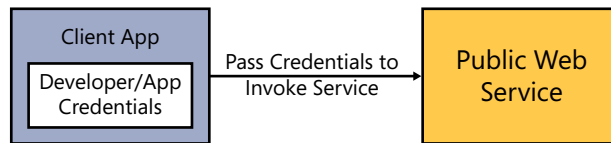
FIGURE 13-25 The OAuth 2.0 authorization sequence.

Securing Web Service IDs

Chapter 11 looked at a wide range of web connectivity scenarios. Many of these involved secured, third-party web services, for which your application is required to provide some credentials (an App ID or Client Secret, for example) in order to connect. In Chapter 11, you focused on setting up the connections by using the third-party SDK class libraries as well as the sequences of operations required to interact with each of the different web services. However, in all cases for which you were required to supply credentials, these were—simplistically—hard-coded into the client application. This is not secure.

As a matter of best practice, you should never hard-code security credentials in your application. A better approach is to eliminate the credentials from your client application, and instead, provide your own web service to hold them. Then, the client application can connect to this web service to fetch the credentials before invoking the secured public web service, as shown in Figure 13-26.

Less Secure



More Secure

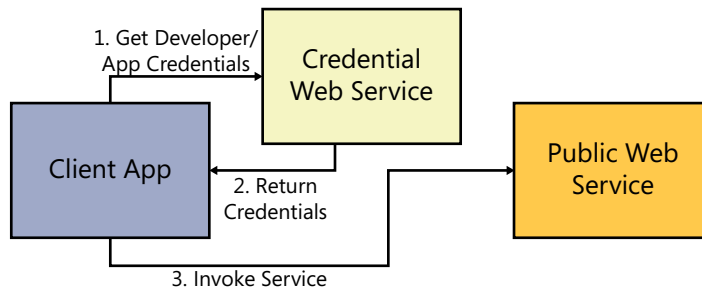


FIGURE 13-26 Eliminating connection IDs from the client application.

You could further secure this by using SSL for your credential web service, although the main concern here is to avoid having hard-coded credentials in the client application, which is achieved even without SSL. This approach can be applied to public web services such as Bing maps and geocode services, bitly, Twitter, Facebook, Windows Live, Windows Azure storage account keys, and so on. Most of these services use either OAuth 1.0 or OAuth 2.0, and therefore require the client application to provide developer/application credentials. It is these credentials that you want to protect.

This approach is slightly more secure than hard-coding credentials on the phone, but it is still not completely secure. With hard-coded credentials, an attacker could extract the credentials from your application's XAP with relative ease. With an indirect web service, the attacker would first have to reverse-engineer your code in order to discover your web service URL and interface, and then make calls to your web service to fetch the credentials. This is harder, but it is still not impossible. The bottom line is that while OAuth security works well for web applications, for which the credentials never leave the server, it works less well for mobile applications.

Implementing Security for the *WebBrowser* Control

Although the *WebBrowser* control can be used to render HTML content—including some scripting capability—it is important to note that the control is not the same as full-fledged Internet Explorer. Many features of the full browser, including security features, are not provided in the control. On top of that, there are some differences in behavior between the *WebBrowser* control in Desktop Silverlight and the version in Windows Phone, which are described here:

- Unlike Internet Explorer's padlock icon, the *WebBrowser* control does not offer any visual indication to inform the user whether a site is being accessed over SSL or not.
- Unlike Internet Explorer, the *WebBrowser* control does not display URLs for links anywhere, which can make it difficult for a user to be certain where the link will take them.
- Unlike Internet Explorer, users cannot navigate in the *WebBrowser* control from HTTPS pages to HTTP pages.
- Any cookies used by the sites to which the *WebBrowser* control has navigated are stored in an application-specific location. These are not accessible to other applications or via Internet Explorer, and vice versa. Even the hosting application cannot programmatically access these cookies.
- If validation of the SSL Server certificate for an SSL-configured website fails, then the *WebBrowser* control restricts access that site.
- If you have a *WebBrowser* control in your application, you must specify an *x:Name* for it, so that the marketplace ingestion process can properly detect and grant the correct capabilities for your application.
- Silverlight provides Cross-Site Scripting (XSS) protection when rendering HTML content directly from a URL. In this situation, scripting is disabled by default. However if the page is cached locally in isolated storage, then XSS protection is not enforced. Locally cached content is considered implicitly trusted, even though it might have originated from an untrusted site. Content loaded by using *NavigateToString* also has no XSS protection.
- The default setting for the *IsScriptEnabled* property is *false*; you should clearly change this only if you have an explicit need for scripting.
- When using the *InvokeScript* method with untrusted scripts, at a minimum, you should ensure that you do not provide any valuable or sensitive data. Better yet, simply do not use *InvokeScript* with any untrusted or unknown script.

Summary

The Windows Phone application platform is very security-conscious. It includes a range of security features, from the certification process of marketplace publication, through the install, load and run-time verification, and sandboxing features, the constraints on using less-secure coding practices, and support for a finite set of security APIs. Even with this support, there is still scope for a developer to build an application that is vulnerable to security attacks.

Applications should never store any valuable or personally identifiable data on the phone—especially if it is unencrypted—without explicit user consent (and use the new cryptography features in version 7.1). Even with the cryptographic support in the platform, it is still possible to use the APIs improperly and end up with an application that might be less secure than you think. When communicating with remote services (web services or web applications) you should never pass user credentials (or any other sensitive data) in the clear. For websites you own, you should set up standard authentication and authorization, and protect the channel with SSL. For public websites, you will almost certainly have to use SSL or OAuth.

Go to Market

This chapter focuses on the end-game of bringing your application to market. There are three aspects to this: fine-tuning the design and implementation to make it robust and perform well; the certification and publication process; and approaches to monetization of the application. In fact, many of the so-called “fine-tuning” techniques are really best practices that you would adopt earlier in the development cycle. They’re merely collected here as a kind of final checklist. It almost makes sense to read this chapter first so that you can see what you’re going to end up doing later on in order to get your application to market.

Threading

In traditional desktop applications, you have a main user interface (UI) thread, and you have the option to create background worker threads if you want. Windows Phone does provide an additional rendering thread, and version 7.1 also introduces a separate input thread. In general, however, you have the same threading opportunities in Windows Phone as you do for desktop Microsoft Silverlight. In addition, Silverlight for Windows Phone also forces all network calls to be non-blocking. You want the phone to be responsive at all times, so you should consider offloading work to a background thread wherever it makes sense. In particular, you should avoid blocking the UI whenever you need to perform work that takes a non-trivial amount of time. Blocking the UI is annoying to the user, and it might even lead him to believe that your application has hung.

In most scenarios, you can predict the cases for which you will need to use additional threads, but sometimes the need comes to light only after user-acceptance testing. Fortunately, the coding work to offload some operation to another thread is usually trivial, so this is something you can easily do late in the cycle. There is a small set of threading APIs that you can use, including *Thread*, *Background Worker*, and *ThreadPool*. Table 14-1 summarizes these APIs.

TABLE 14-1 Threading APIs

API	Description	Results Raised on the UI Thread?
<i>Thread</i>	A restricted version of the standard Microsoft .NET Thread type, wherein you have access to the properties of the <i>Thread</i> object you create.	No.
<i>BackgroundWorker</i>	A useful wrapper for doing work on a background thread while reporting progress and results on the UI thread. A <i>BackgroundWorker</i> can also be stopped before completion, if required. Especially useful if you only need a small number of background threads and if the results need to be displayed in the UI.	Progress and results are reported on the calling thread, which can be the UI thread.
<i>ThreadPool</i>	A fire-and-forget approach: you ask for your method to be queued up and executed by an arbitrary thread from the .NET threadpool. You have no direct access to the <i>Thread</i> object. Especially useful if you need to batch-process multiple operations that require multiple threads, and you don't need ongoing progress reports. In general, this is the preferred approach because it will yield the best overall performance in the long term, especially as the application becomes more complicated.	No.

Here's a simple application (the *TestThreading* solution in the sample code) that demonstrates the use of these APIs, as shown in Figure 14-1. These are no different from standard Silverlight behavior. In each case, the *Click* event handler for the button creates another thread to perform some operation while the UI remains unblocked and responsive. As each thread completes its work, you report the results in the corresponding *TextBox*. The *BackgroundWorker* version additionally reports ongoing progress.

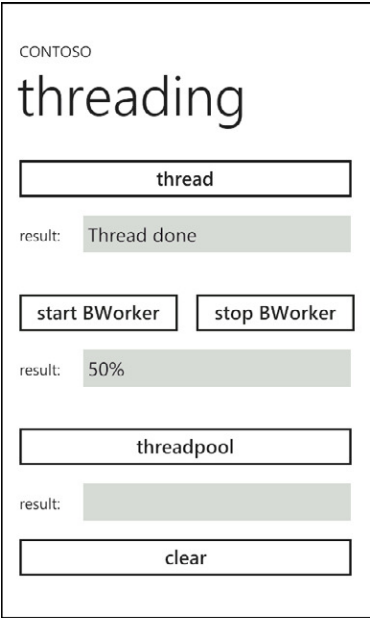


FIGURE 14-1 Threading APIs.

You *can* use the *Thread* class in the *System.Threading* namespace, but you probably don't need to because most scenarios are covered by either the *BackgroundWorker* class or the *ThreadPool*. One commonly applied rule of thumb is that if you are essentially running an infinite loop inside your background thread—such as a network socket listener, for example—then you should consider using a *Thread*. On the other hand, if it is a single operation, even if it is a long-running one, then use the *ThreadPool*.

Regardless, here's a simple example for illustration. You construct a *Thread* object, passing in a *ThreadStart* or *ParameterizedThreadStart* delegate object (or simply a method, which the compiler will resolve to a delegate). The only other property that you can set is the *Name*, which can be any arbitrary string. This can be useful for disambiguation in the debugger window, and it is generally recommended to set this. Then, call *Start*; this will run the delegate method on a new thread.

If you want to touch any of the UI elements from any thread apart from the main UI thread, you must be careful to marshal back to the UI thread. You do this with the *Dispatcher* property of a UI element (typically, of the current page). The reason for this is that the UI components are not completely thread-safe. Making components thread-safe is notoriously difficult and fragile. So, behind the scenes, each UI component records the identity of the thread that created it, checks to make sure that only that thread accesses it, and throws an exception if another thread attempts to use it.

```
private void threadButton_Click(object sender, RoutedEventArgs e)
{
    threadResult.Text = "";
    Thread t = new Thread(thread_DoWork);
    t.Name = "My Thread";
    t.Start();
}

public void thread_DoWork()
{
    Thread.Sleep(5000);
    Dispatcher.BeginInvoke(
        () => { threadResult.Text = "Thread done"; });
}
```

In each of the thread worker functions, you simulate a time-consuming operation with a *Thread.Sleep*, and then report results. Note that if you're in a situation for which you don't have a UI element at hand that you can use to access a UI *Dispatcher* (any UI element, including the page), you can use the *Deployment* object, instead.

```
Deployment.Current.Dispatcher.BeginInvoke(
    () => { threadResult.Text = "Thread done"; });
```

Rather than using the low-level *Thread* type, it is generally more useful to use the *BackgroundWorker* class. This class takes care of all the thread marshaling for you. It also provides a simple way to check ongoing progress, and it supports cancellation.

```

private BackgroundWorker bw = new BackgroundWorker();

private void startBwButton_Click(object sender, RoutedEventArgs e)
{
    if (!bw.IsBusy)
    {
        bw.ResultText = "";
        bw.WorkerReportsProgress = true;
        bw.WorkerSupportsCancellation = true;
        bw.DoWork += bw_DoWork;
        bw.ProgressChanged += bw_ProgressChanged;
        bw.RunWorkerCompleted += bw_RunWorkerCompleted;
        bw.RunWorkerAsync();
    }
}

private void stopBwButton_Click(object sender, RoutedEventArgs e)
{
    if (bw.WorkerSupportsCancellation)
    {
        bw.CancelAsync();
    }
}

```

You would typically provide three event handlers: one for the primary work itself; one for handling progress changes; and one for handling the “completed” event. If you want to support cancellation, your primary work method must check to see if a cancel request has been raised. If you want to support progress reports, your primary work method must invoke the *ReportProgress* method, passing a value that corresponds to the percentage of work completed. It’s up to you to decide to what this percentage relates.

```

private void bw_DoWork(object sender, DoWorkEventArgs e)
{
    BackgroundWorker worker = sender as BackgroundWorker;

    for (int i = 1; (i <= 10); i++)
    {
        if ((worker.CancellationPending == true))
        {
            e.Cancel = true;
            break;
        }
        else
        {
            Thread.Sleep(500);
            worker.ReportProgress((i * 10));
        }
    }
}

```

When you call *ReportProgress*, this internally raises the *ProgressChanged* event. In this example, you’re reporting the percentage progress in the UI. When the *BackgroundWorker* terminates, you should check the reason for termination. Was it cancelled, was there an error, or did it terminate because it finished its work?


```

private void bw_ProgressChanged(object sender, ProgressChangedEventArgs e)
{
    bwResult.Text = (e.ProgressPercentage.ToString() + "%");
}

private void bw_RunWorkerCompleted(object sender, RunWorkerCompletedEventArgs e)
{
    if ((e.Cancelled == true))
    {
        bwResult.Text = "BackgroundWorker cancelled";
    }

    else if (!(e.Error == null))
    {
        bwResult.Text = ("Error: " + e.Error.Message);
    }

    else
    {
        bwResult.Text = "BackgroundWorker done";
    }
}

```

If your requirement is to perform some number of operations, for each of which you need a background thread, and where UI reporting is a lesser requirement, then the simplest option is to queue work up and let the *ThreadPool* take care of it. A fresh (or recycled) thread will be used to service the operations in the queue. The *ThreadPool* internally will manage threads in the pool and will spin up new threads as appropriate, in a manner that balances CPU use on the device. This means that if you were to queue up, for example, 100 operations by using the *ThreadPool*, this would not result in 100 threads getting spun up all at once. Instead, the *ThreadPool* will run what it thinks is appropriate for the current hardware, and then queue the rest.

```

private void threadpoolButton_Click(object sender, RoutedEventArgs e)
{
    threadpoolResult.Text = "";
    ThreadPool.QueueUserWorkItem(threadpool_DoWork);
}

private void threadpool_DoWork(object stateInfo)
{
    Thread.Sleep(5000);
    Dispatcher.BeginInvoke(
        () => { threadpoolResult.Text = "ThreadPool thread done"; });
}

```

Note that you have a further choice: rather than using the *Dispatcher* to marshal calls to the UI thread, you can use *SynchronizationContext*, instead. The *Dispatcher* is specific to UI elements, whereas you can use *SynchronizationContext* for any thread. So, in this example, in the two places where you explicitly used *Dispatcher*, you could re-code to use *SynchronizationContext*. You can see this at work in the *TestThreading_SyncContext* solution in the sample code. The *SynchronizationContext* object must be initialized on the thread to which you're going to marshal back; in this case, that would be the UI thread. So, you can declare a *SynchronizationContext* object as a field in the page class, and then use this object in your event handlers that is invoked on background threads.

```
private SynchronizationContext context = SynchronizationContext.Current;

public void thread_DoWork()
{
    Thread.Sleep(5000);

    //Deployment.Current.Dispatcher.BeginInvoke(
    //    () => { threadResult.Text = "Thread done"; });
    context.Post(new SendOrPostCallback(
        arg => threadResult.Text = "Thread done"), null);
}
```



Note Under the covers, the *BackgroundWorker* class actually uses the *ThreadPool* to queue its work. It also uses an *AsyncOperation* object to take care of the thread marshaling, and *AsyncOperation* uses the *SynchronizationContext* internally.

Recall that all phone application network calls are asynchronous, consistent with desktop Silverlight. This includes web service calls, *WebClient*, and *HttpWebRequest* method calls. Consider the example that follows, using the MagicalManatee web service (for more details on this, go to Chapter 11, “Web and Cloud”). You must first instantiate the web service client proxy. Then, set up a handler for the method call-completed event (in this case, *GetFactCompleted*). Finally, invoke the web method asynchronously.

```
private void appBarGetFact_Click(object sender, EventArgs e)
{
    client = new MagicalManateeFactsClient();
    client.GetFactCompleted += client_GetFactCompleted;
    client.GetFactAsync();
}
```

The call then happens on a background thread, and the UI is not blocked. When the call returns, this invokes your event handler. If an exception occurred during the call, it won't be raised on the main thread; instead, it will be sent back with the *EventArgs* in the *xxxCompleted* event handler. So, you should always test to determine if there was an exception before processing the method call results:

```
private void client_GetFactCompleted(object sender, GetFactCompletedEventArgs e)
{
    String result = String.Empty;
    if (e.Error == null)
    {
        result = e.Result;
    }
    else
    {
        result = e.Error.ToString();
    }
    Debug.WriteLine(result);
}
```

By contrast, the *WebClient*'s events are always raised on the UI thread in Windows Phone 7, even if you create the *WebClient* on a background thread. Although this offers you the convenience that all your work is on the UI thread (no need for *Dispatcher* calls), the downside is that it risks blocking the UI. For this reason, the recommendation is to avoid it as much as you can. Instead, you should use *HttpWebRequest*. Be aware that this behavior changes in version 7.1, in which the *WebClient* events are raised on the thread where the *WebClient* was created (which is not necessarily the UI thread). When you use *HttpWebRequest*, the responses are raised on a background thread. So, you have the minor inconvenience of manually marshaling to the UI thread (if you need to) by using a *Dispatcher*. On the other hand, *HttpWebRequest* calls will not block the UI. The *HttpWebRequest* does some work internally on the thread on which it was created. For this reason, you might want to create it on a background thread to further minimize its impact on the UI thread. The *HttpWebRequest* still needs to post some work to the UI thread in order to start. The system will deadlock if you have your UI thread wait on the web request, even if it is initiated from a background thread.

Performance

Application performance is always important, and it is especially important for applications on mobile devices, which have relatively low processor and memory specifications, as compared to desktop computers. There are several performance issues to consider, including the following:

- Improving application responsiveness by offloading work from the UI thread to the Render thread.
- Designing efficient visual elements in your application UI.
- Miscellaneous performance tips related to specific controls or control usage.
- Performance issues related to non-UI application features, such as network calls, data formats, and code structure.

UI vs. Render Thread, and *BitmapCache* Mode

Desktop Silverlight has one UI thread, which does all the work. The Silverlight rendering engine used on Windows Phone is optimized for devices. It includes a second thread for rendering the UI, known as the Render or compositor thread. These two threads function as follows:

- **UI thread** The UI thread is still the primary thread in the application. It handles all input (including input events), parsing and creating objects from XAML, visual layout, property change notifications, data binding, and other managed code execution that has not explicitly been placed on a background thread. The UI thread also handles animations that are implemented through per-frame callbacks, for which the render thread needs the UI thread to perform work for each frame rendered.

- **Render thread** This is designed to be very lightweight, and it is mainly responsible for stitching together textures to hand off to the Graphics Processing Unit (GPU). The Render thread handles simple (double) animations, translate, scale, rotate and perspective transforms, opacity (but not opacity masks), and rectangular clipping—all of which can be hardware-accelerated via the GPU. Note that for scale transforms, whenever the scale crosses the 50 percent threshold of the original (or previously rasterized) texture size, the visual is re-rasterized on the UI thread.

Briefly, the initial template expansion and rasterization for an element occurs on the UI thread. These rendered elements are then cached in memory as bitmaps and handed off to the Render thread which works with the GPU to draw the frame and add it to the back buffer for display. From this point, if you don't make any changes to the cached element, it doesn't have to be redrawn. Any changes you might make to the cached element fall into two categories:

- A change that can be handled by the GPU. Some examples are rectangular clipping, *Translate Transform*, or *ScaleTransform*.
- A change that requires the element to be re-rendered. This can include a color change, non-rectangular clipping, opacity mask, padding or margin changes, an so on. In this case, the cached bitmap will be deleted, and the element will be re-rendered on the UI thread to generate a fresh bitmap for caching.

So, there are at least two threads in the system, plus one or more additional threads that the application itself can choose to create, either indirectly or directly, as summarized in Figure 14-2.



Note Version 7.1 also introduces a third system thread that is intended specifically to handle input.

In general, you want to take advantage of the Render thread as much as you can to offload work from the UI thread. To see where there might be opportunities for optimizing between the UI thread and the Render thread, you can turn on display of redraw regions and bitmap caching by using the *Application* settings, as shown in the following code snippet. Note that the standard Microsoft Visual Studio project templates generate code for this in the *App.xaml.cs*, which you can uncomment.

```
Application.Current.Host.Settings.EnableRedrawRegions = true;  
Application.Current.Host.Settings.EnableCacheVisualization = true;
```

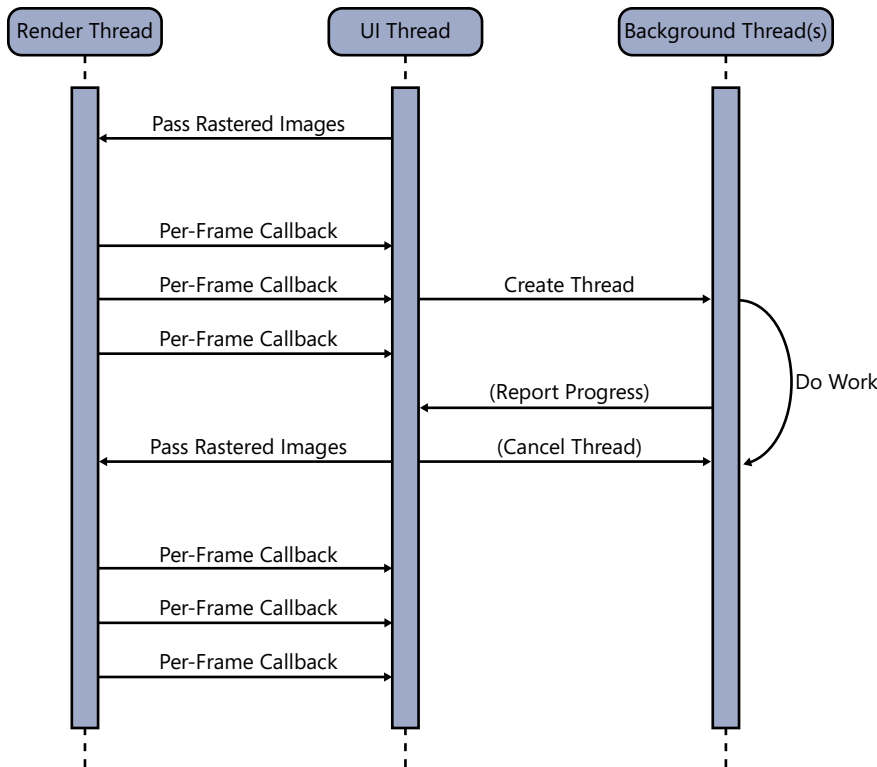


FIGURE 14-2 The Render thread, UI thread, and background threads.

Consider an application that moves a ball around the screen. Depending on how you write the code, this will result in different rendering behavior. The following sample application (the *Bouncing Ball* solution in the sample code) does not take advantage of the Render thread and GPU; instead, it implements timer-based callbacks on the UI thread. The application also responds to user touch, via a *GestureListener*, so even more work is being done on the UI thread. The application provides four AppBar buttons. With the first three, the user can toggle *EnableRedrawRegions*, *BitmapCache* mode, and *EnableCacheVisualization*, respectively. This allows you to see the effects of the design choices on the UI rendering behavior (you'll examine the purpose of the fourth button later).

```

private void appBarRedraw_Click(object sender, EventArgs e)
{
    Application.Current.Host.Settings.EnableRedrawRegions =
        !Application.Current.Host.Settings.EnableRedrawRegions;
}

private void appBarCache_Click(object sender, EventArgs e)
{
    if (ball.CacheMode == null)
    {
        ball.CacheMode = new BitmapCache();
    }
}

```

```

    else
    {
        ball.CacheMode = null;
    }
}

private void appBarCacheViz_Click(object sender, EventArgs e)
{
    Application.Current.Host.Settings.EnableCacheVisualization =
        !Application.Current.Host.Settings.EnableCacheVisualization;
}

```

Figure 14-3 illustrates the application with *EnableRedrawRegions* turned on, showing which elements are being redrawn with each frame (the colors are arbitrary; they cycle between purple, yellow, and magenta). As the ball bounces around the screen, the redraw region is a rectangle that expands or contracts to include all UI elements. Thus, the redraw rectangle varies in size, depending on where the ball is in relation to the other UI elements.

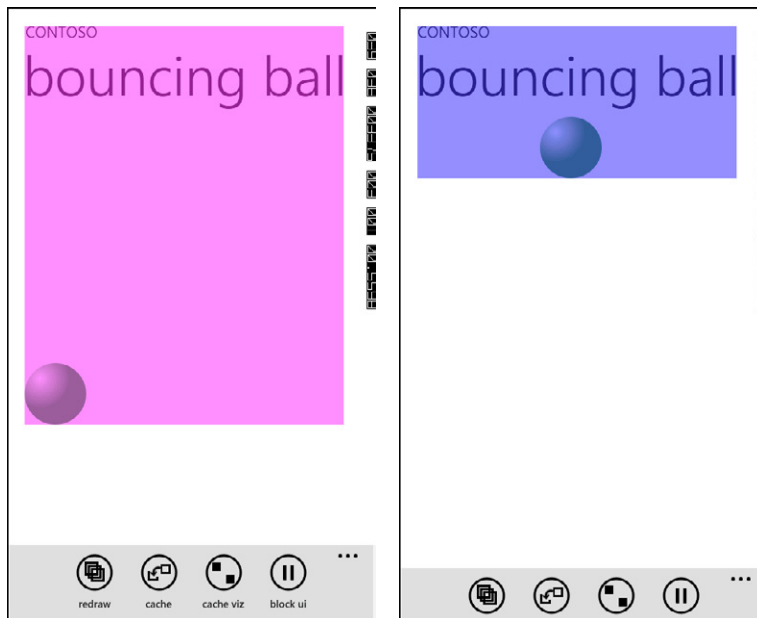


FIGURE 14-3 The regions of your application that need to be redrawn can vary over time.

The redrawing highlighted when the *EnableRedrawRegions* flag is turned on is essentially drawing in software (on the CPU) and does not offload any work to the Render thread and the GPU. In all cases, the first time a visual is drawn, it will be drawn in software, but the aim generally should be to draw a visual once on the UI thread, and then hand it off to the Render thread (which in turn, hands it off to the GPU) for all subsequent drawing. So, what you want to avoid is the situation in which you can see the same region being drawn repeatedly (as evidenced by the colors changing repeatedly). The bottom line is that if you see something in your application that's frequently changing color, it means it's being frequently redrawn. You should therefore examine your code to see if you can reduce this.

Figure 14-4 demonstrates the application again, this time with *EnableCacheVisualization* turned on, which shows the areas of the application that are cached. The un-cached surfaces are rendered in software and the cached surfaces are passed to the GPU and rendered in hardware. With this flag turned on, each element/texture that is handed off to the GPU is tinted blue and has a transparency applied. This way, you can see where textures are overlapping. The darkest shades indicate that multiple textures are lying atop one another.



Note Turning on *EnableCacheVisualization* degrades performance, so you should not attempt to measure frame rates while this is active. Note also that the behavior of this flag on Windows Phone is different from the behavior on desktop Silverlight: on desktop Silverlight, the tinted areas are areas that are *not* drawn by the GPU; on Silverlight for Windows Phone, the tinted areas are those that *are* drawn by the GPU.

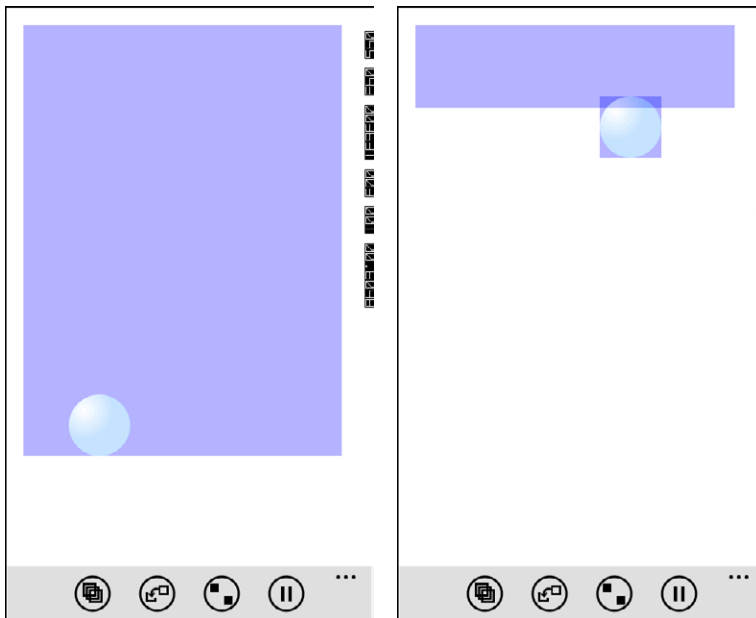


FIGURE 14-4 Cache visualization with *BitmapCache* mode turned off (on the left) and on (right).

You can specify that an element should have its rendered bitmap cached in XAML, as shown here:

```
<Ellipse x:Name="ball" Width="80" Height="80" CacheMode="BitmapCache" >
```

Or, you can specify that it should be cached in code:

```
ball.CacheMode = new BitmapCache();
```

The effect of setting *BitmapCache* mode is to skip the render phase for the element, which will have a significant effect on performance. The screenshot on the right in Figure 14-4 shows what happens if you set the *CacheMode* on the ball to *BitmapCache*. The UI thread works considerably less to display the ball, the frame rate on this application goes up, and the fill rate goes down. The frame rate, fill rate, and the other performance counters are discussed in more detail in Chapter 8, “Diagnostics and Debugging.”

When you use bitmap caching, you should group cached elements together, following non-cached elements in the visual tree. Do not interleave cached/non-cached elements. This way, non-cached elements can be included in a single intermediate background texture, which improves performance.



Note There's a downside to caching: it takes up additional memory, and the more you cache, the higher your fill rate will be. Therefore, you shouldn't simply cache everything. Instead, you should profile your application by using *EnableRedrawRegions* and *EnableCacheVisualization*, and look for opportunities to optimize caching. If you do this right, you should see the frame rate go up, and the *FillRate* count go down.

The fourth App Bar button is implemented to simulate an operation that needs to do work on the UI thread:

```
private void appBarBlockUI_Click(object sender, EventArgs e)
{
    Thread.Sleep(3000);
}
```

If you start the ball bouncing and then tap this button, the bouncing will stop for three seconds. This demonstrates the critical flaw in this application's design, because so much relies on work being done on the UI thread.

The previous example relied on user interaction, and a lot of work was done on the UI thread. Figure 14-5 shows an alternative example of a bouncing ball (the *BouncingStoryboard* solution in the sample code); this one involves only minimal user interaction (start and stop App Bar buttons) and does most of the work on the Render thread. In addition, the animation is fixed and declared in XAML via a storyboard. This contains two main animations: one that moves the ball from top to bottom (using the *Canvas.Top* attached property), and another that moves the ball from left to right (using *Canvas.Left*). The top-bottom animation itself contains a third animation, which uses one of the Silverlight *EasingFunctions* to provide a bouncing motion.

```
<Canvas.Resources>
    <Storyboard x:Name="bounceStory">
        <DoubleAnimation From="130" To="616"
            Duration="0:0:12" Storyboard.TargetName="ball"
            Storyboard.TargetProperty="(Canvas.Top)">
            <DoubleAnimation.EasingFunction>
                <BounceEase Bounces="12" Bounciness="1.2"
                    EasingMode="EaseInOut"></BounceEase>
            </DoubleAnimation.EasingFunction>
        </DoubleAnimation>
    </Storyboard>
</Canvas.Resources>
```



```

        </DoubleAnimation>
        <DoubleAnimation From="24" To="400"
            Duration="0:0:12" Storyboard.TargetName="ball"
            Storyboard.TargetProperty="(Canvas.Left)">
        </DoubleAnimation>
    </Storyboard>
</Canvas.Resources>

```

The big change here is to move the per-frame manual animation (which was all done on the UI thread) to a double animation implemented via a storyboard, which can all be done by the Render thread.

The App Bar buttons are there to start and stop the storyboard. There is also a button to toggle redraw regions, and one to block the UI thread. Because this ball is part of a simple animation and is automatically cached, there's no button to set *CacheMode*. Observe also that if you block the UI thread, in this application, the ball will continue to bounce because the animation is all being done on the Render thread.

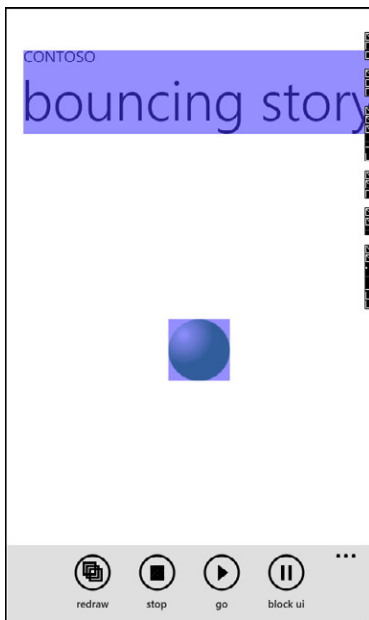


FIGURE 14-5 Using the Render thread for animations reduces the amount of redrawing.

The following objects will be automatically cached:

- The target of any storyboard-driven animation that uses the Render thread (as shown in the *BouncingStoryboard* solution).
- The target of any plane projection, either static or animated.
- All *MediaElement* objects.
- Child items in a *ScrollViewer* or *ListBox*.

As discussed in Chapter 8, the three most important performance counters to monitor are the UI Thread Frame Rate, the Render Thread Frame Rate, and the Fill Rate. If your UI thread is overloaded, you'll see the UI Thread Frame Rate drop, which is a sign that you need to offload work from the UI thread. From the other end, the Fill Rate corresponds to how hard the GPU is working, and as the Fill Rate exceeds 2, the Render Thread Frame Rate will drop. So, a high Fill Rate is a sign that you need to minimize your use of UI elements (by reducing the number and/or complexity of your elements, setting *BitmapCache* mode, avoiding interleaved cached/non-cached elements, and so on).

UI Layout and *ListBoxes*

The *ListBox* is an obvious element for which scale has significant performance implications. UI layout is the most expensive operation performed on the UI thread. There's nothing to stop you from creating complex layouts with nested *Grids*, *StackPanels*, nested *ListBoxes*, plus complex *ValueConverters*, custom controls, and so on. This is bad enough if you only have one of these on your page, but as soon as you use such a complex element as an item within a *ListBox*, the scale issues become more obvious. Plus, of course, there's nothing to stop you from putting many thousands of these items into your *ListBox*. On top of that, you could be sourcing your data from a remote service over the Internet, and the data might include large images or large volumes of redundant metadata. The potential permutations implied in this kind of model represent a potential recipe for bad performance and an unresponsive UI.

Here are some UI best practice guidelines for optimizing runtime performance and responsiveness when using *ListBoxes*:

- **Avoid using complex item data templates.** Most particularly, don't use nested *ListBoxes*, and don't use *UserControls* or custom controls. Also, ensure that you have the data template in a fixed-size container such as a *Grid* with an explicit *Height* set on it. As a performance optimization, the *ListBox* calculates the height of three screen's worth of items (the one currently visible, plus one above and below), and this doesn't work if your items vary in size.
- **Avoid using complex converters.** If possible, try to perform conversions in the data source request or as you pull the data into a local cache, before attempting to render it in the UI.
- **Offload work to background threads.** Typical candidates for this include the retrieval, processing, and caching of item data, leaving only the final data-binding on the UI thread. The trade-off here is that doing work on a background thread before dispatching to the UI thread might cause items to load more slowly. However, the advantage is that the UI will remain responsive. If you do it right, even the slower load might be apparent only on the initial batch of items, because you can continue working on the background while the user is exercising the UI. As you're doing work on the background thread, be sure to yield control frequently (perhaps by a simple *Thread.Sleep*) so that the OS can schedule the UI thread more frequently.
- **Virtualize your data if you can.** This is particularly relevant if you need to manipulate raw data and compute the final data for rendering in the UI. The *ListBox* virtualizes the UI (via the *VirtualizingStackPanel*, which is the default items host for the *ListBox*), such that if you have, for example, 1,000 items, only 3 screen's worth of UI elements are created, and then these

are recycled as the user scrolls other items into view. However, the data is not virtualized; the *ListBox* property you data-bind is *ItemsSource*, which is defined as an *IEnumerable*. This means that the only way the *ListBox* knows the size of the list is to enumerate all the items. This information is required so that the *ListBox* can configure itself correctly; for example, to calculate the scrollbar indicator. More significantly, it assumes that all of your data items are constructed completely before handing off to the UI. You can mitigate this by binding to collections that implement *IList*, which provides *Count*, *IndexOf*, and indexer (*this[]*) methods. In your implementation, you can be smart about constructing each of your computed items for the list. For example, you can defer the composition/computation processing for items until they're required for the UI.

- **Consider your caching strategy.** If you're virtualizing your data, you have the opportunity to do some intelligent caching. Depending on the context, you might also be able to segment your data into static and dynamic components. You can also consider the standard .NET technique of *WeakReferences*.
- **Minimize work while scrolling.** First, don't do anything on the UI thread while the user is scrolling, because that would make the UI less responsive. Second, don't do any work for items that are not visible to the user; this implies waiting for the user to stop scrolling before you can calculate which items are now visible. You could also segment your data templates according to whether the user is scrolling; that is, use a simple template if she's scrolling (perhaps just text, or a low-resolution image), and a more sophisticated template for items that are visible when she has stopped.

More UI Performance Tips

Apart from the aforementioned *ListBox*-specific issues, there are other UI-related performance optimizations that you can consider:

- **JPG versus PNG** It is faster to decode JPG images than PNG images, so you should prefer JPG over PNG. The only case for which you need to use PNG is if your images have transparency; otherwise, you should default to JPG in the absence of other constraints. Note that the difference is often very small, so it is not worth spending a lot of effort on this.
- **Static versus dynamic images** It is often faster to render static images than to create them dynamically at runtime. A complex image could be constructed in XAML, in code, or by loading an image file. Constructing it in XAML involves more steps, and is the slowest of the three approaches. Loading from a file is generally the fastest—decoding an image is relatively cheap, and the benefits mount up if you are reusing the visual multiple times. Of course, if the visual depends on something at runtime, then you might have no option but to create it in code.
- **Image scaling** It is common to use a fixed-size *Image* control and then pull in the image source from a file or resource, which might or might not be the same size as the control. There's an obvious performance penalty when scaling images. This is particularly so with respect to wasted memory for images that are larger than you need, so you should try to

get the source images at the right size in advance. This is another technique that becomes more critical if you're adding image items to a *ListBox*. Furthermore, Windows Phone has a maximum image size of 2048x2048 pixels. If you use larger images, they will automatically be sampled at a lower resolution. The algorithm to perform this sampling picks a simple ratio, so your image can end up with a resolution that is significantly less than 2048x2048. And of course, it will be slower to render.

- **Custom decoding** If you are scaling images manually, you should use the *PictureDecoder* API. You can use this to specify how to decode, and to what target size. Without this, the normal behavior would be to first decode at the image's native resolution, and then scale down. There's a slight performance gain and a significant memory gain in using *PictureDecoder*.
- **Visibility versus Opacity** If you need to hide/show UI elements, you have a choice between using the *Visibility* property or the *Opacity* property. An element with *Visibility=Collapsed* incurs no cost, the system will not walk the visual tree for the element, and events will not be propagated for the element. On the other hand, when the time comes to unhide it, setting *Visibility=Visible* will incur a heavy cost in creating the element's tree, so it might be slower to appear. Using *Opacity=0* means that the element's visual tree is in existence at all times, which will add to the overall cost of your UI. On the other hand, when the time comes to set *Opacity>0*, this will be extremely quick, provided the element is cached. If the element is not cached, you will pay a penalty. To recap, using *Opacity* to show/hide a visual that is not cached is the worst thing you can do, from a performance standpoint. Conversely, the optimal strategy is generally to use *Opacity* and to enable *BitmapCache* mode.
- **Resource versus Content** An image embedded as a resource becomes part of your assembly. One side-effect of this is that it is read twice at startup. The reason for this is that the Windows Phone application platform has to read your assemblies for security purposes to ensure that this is a valid, certified Windows Phone application. Then, the Common Language Runtime (CLR) also has to read it (for verification purposes, type resolution, JIT buffering, and so on). So, if you embed a large amount of images into your assembly, they'll all be read twice. If, instead, you set their build action to be *Content*, then they'll simply be added to your XAP, but not to your assembly. The trade-off is that while embedded resources slow down startup time, they are faster to load subsequently.
- **Desktop versus phone** Be very careful about reusing code or controls from desktop Silverlight applications. Even though they might work, they might work very slowly. The *Rating* control from the Silverlight toolkit is one example: this has a very large number of UI elements. Using even one instance of this should give you pause, but you should definitely avoid making this a part of an item control in a *ListBox*.
- **Inline XAML** It is very easy to declare complex UI elements in XAML, as opposed to defining the same elements in code directly. You can use the *XamlReader* class to dynamically load XAML at runtime. However, you will pay a performance penalty for this; parsing the XAML and executing the resulting code is always going to be slower than executing code that you've written to do the same work directly.

- **Panorama versus Pivot** Although the *Panorama* control is an *ItemsControl*, it is not virtualized: all of the content inside each of the *PanoramaItems* is rendered on initial load. If you think about it, this is what enables the *Panorama* to show part of the next (and sometimes, previous) items immediately upon load. You should stick to the Metro guidelines, which suggest that you should not be using a *Panorama* for heavy work anyway. The *Panorama* is intended to be a front-page “magazine” experience. It should be an attractive entry to your application that encourages the user to explore further. It should not be used for complex controls or complex lists. The standard themes used on *Panorama* further encourage this; for instance, the heading is huge and takes up a lot of space. That’s not a reason to fill up the remaining space, it’s an encouragement to be minimalist on the *Panorama*. By contrast, the *Pivot* does virtualize its content. Only the first *PivotItem* is populated on initial load, although the system does immediately trigger a load for the items to the right and left. So, effectively, three items are loaded at or shortly after startup. A three-item *Panorama* will have similar startup time to a three-item *Pivot*, but the more items you have, the more the load times diverge. The *Panorama* user experience (UX) encourages users to navigate back and forth between the items, so you can mostly assume that all items need to be available at all times. This is not true of the *Pivot*, for which some items might never be seen by the user in any given session. Thus, if you have, for example, complex lists or animations on an item, you should be proactive about creating/starting/stopping such elements.
- **Progress bars** The standard library provides a *ProgressBar*. The Toolkit provides a *PerformanceProgressBar*. For determinate scenarios (that is, where you can determine the percentage progress), use the standard *ProgressBar*. For indeterminate scenarios, use the *PerformanceProgressBar*. The critical difference is that the Toolkit *PerformanceProgressBar* does most of its work on the Render thread, whereas the standard *ProgressBar* uses the UI thread heavily, and also uses about three times more video memory. You can compare the difference by running the *TestProgressBars* solution in the sample code and turning on redraw regions. Applications built to target version 7.1 can take advantage of the built-in progress indicator in the *SystemTray*. Doing this affords performance benefits over using any XAML-based progress bars.

Non-UI Performance Tips

- **WebClient versus HttpWebRequest** In Windows Phone 7, when you make a web request by using *WebClient*, regardless of which thread you make it on, the result will always be returned on the UI thread. In fact, the *WebClient* class is a wrapper around *HttpWebRequest*, whose primary purpose is to simplify making web requests. This has performance implications; especially in data-binding scenarios in which your data-bound items are being sourced from the web. Conversely, requests made by using *HttpWebRequest* return on the background thread, as expected, so you should opt for *HttpWebRequest* in most cases. On top of that, even though *HttpWebRequest* responses are raised on a background thread, it still uses resources on the thread where it was created. So, the optimal approach for a network-heavy scenario is to use *HttpWebRequest*, and to create it on a background thread.

- **Network calls** Avoiding chatty network calls is a well understood performance optimization technique, but one worth repeating here. If your *ListBox* items are data-bound to a web service, you ideally want to pull in a batch of items in one network call and cache them locally in a list of some kind. You definitely don't want to be making an individual web method call for each item. You should also default to filtering on the server, as opposed to bringing down high volumes of redundant data which you then filter on the client.
- **Web service data formats** Traditional web services typically send data in Simple Object Access Protocol (SOAP) format. Newer web services, including Windows Communications Foundation (WCF) Data Services, expose data with the OData protocol, in either XML or JSON format. OData formats involve significantly less overhead than traditional SOAP formats. OData JSON, in turn, offers significantly less overhead than OData XML.
- **Static versus dynamic Bing maps** If all you need to do in your scenario is show a few static maps, you should use the Bing web services. Don't use the standard Bing dynamic maps unless you need their inherent richer interactive capabilities. If you're not using the Bing Maps control, you save on the control, and you also save on the assembly itself, which is not pulled into your XAP.
- **Pages in separate assemblies** If your application has pages that are rarely visited, consider factoring them out to a separate assembly. That way, you save on startup time and memory usage. It's important to minimize startup time; first, because slow startup is a bad UX, and second, because if your startup is too slow, you will fail marketplace ingestion. Factoring pages out to secondary assemblies means that the additional page(s), and their containing assembly, are only loaded if and when they're actually used at runtime. To do this, you would create a class library project and add the page(s) to that project. Then, add a reference to the class library from your main project. The syntax of the URI needed for a page in another assembly is "<OtherAssemblyName>;component/<PageName>.xml", which appears as follows in actual code:

```
NavigationService.Navigate(new Uri(
    "/MyPageLibrary;component/Page2.xml", UriKind.Relative));
```

- **Minimize constructors** Constructors for UI elements as well as handlers for the *Loaded* event are executed before the first frame of an application is presented. You can reduce startup time by reducing the work that you do on constructors and *Loaded* event handlers. Wherever it makes sense, you should move work from these methods to later methods. After the *Loaded* event, the next most commonly used methods are the *OnNavigatedTo* override and the *LayoutUpdated* event handler. It is generally better to do work in either of these methods than in the constructor or *Loaded* event handler. However, note that *OnNavigatedTo* is also

called during the main initialization flow for a page, so you should also minimize code here. Plus, if you implement *LayoutUpdated* for initialization (that is, one-time only) code, you must ensure that you immediately unhook the handler once it has been called the first time. This method will be called very frequently, so if you don't unhook the handler as soon as its work is done, the performance penalty will be extreme.

- **Isolated storage** Accessing isolated storage can be slow. First, you should be doing any such work on a background thread. Second, you should consider how you're structuring your isolated storage. If you have a large number of files, you'll experience poor performance if you need to search them or fetch a list or count of them. In this scenario, consider a hierarchical subfolder structure instead.

Silverlight Unit Testing Framework

Jeff Wilcox, a senior developer at Microsoft, built the Silverlight Unit Testing Framework (SLUTF), and has been maintaining it as a codeplex project for several years. The SLUTF is now part of the Silverlight Toolkit and is available as a free download from <http://silverlight.codeplex.com/>. The codeplex site includes downloads that target Silverlight 3, 4, and 5, and Windows Phone versions 7 and 7.1.

The idea of unit testing is to examine small units of code independent of the rest of the application, typically at the method level. It is not uncommon for an organization's development process to mandate method-level unit tests as a code check-in requirement. Unit testing is also a primary mechanism in test-driven development (TDD). Even if you don't adopt TDD, you're still encouraged to build unit tests for your application. The SLUTF can make this relatively painless. Here's how it works:

- Create your main application project.
- Create a test application (another Windows Phone application project).
- In your test application, add a reference to the SLUTF assemblies.
- In your test application, add a reference to your main application.
- In your test application, add one or more classes that contain methods to test your main application.

The SLUTF facilitates the last step by providing helper classes to manage and control your tests. Try it out. First, create a regular Windows Phone application, called *SimpleApp*, as shown in Figure 14-6.

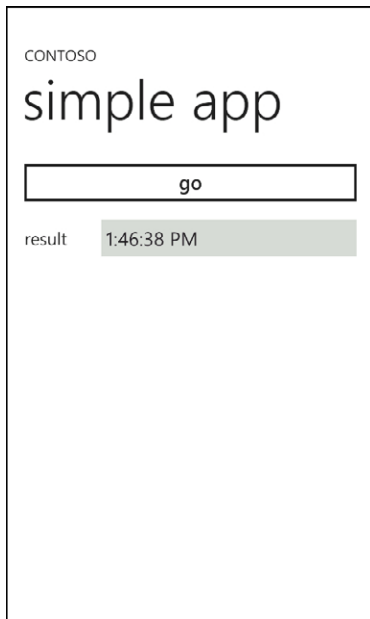


FIGURE 14-6 A simple application to exercise the Silverlight Unit Testing Framework.

The application is very simple: you offer the user a *Button*, a *TextBlock* label, and a *TextBox*. Note the names of these items, because you'll be using them later in your test application.

```
<Button x:Name="goButton" Content="go" Click="goButton_Click"/>
<TextBlock Text="result"/>
<TextBox x:Name="resultText"/>
```

When the user taps the *Button*, you fill in the *TextBox* with the current *DateTime*.

```
private void goButton_Click(object sender, RoutedEventArgs e)
{
    resultText.Text = DateTime.Now.ToLongTimeString();
}
```

Next, create a second Windows Phone application. This will be used for running tests on the phone against the main application. The test application will not use the standard XAML content for the *MainPage*, but you can't eliminate it altogether because it is used to generate the initialization code for the *MainPage* class. You can, however, reduce it to the bare minimum, as shown in the following:

```
<phone:PhoneApplicationPage
    x:Class="UnitTestApp.MainPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:phone="clr-namespace:Microsoft.Phone.Controls;assembly=Microsoft.Phone"/>
```


Add a reference to `Microsoft.Silverlight.Testing.dll` and `Microsoft.VisualStudio.QualityTools.UnitTesting.Silverlight.dll`, both of which are deployed with the Silverlight Toolkit. You'll get a warning about mismatched frameworks, but you can safely ignore this. Also, add a reference to your main application. Update the *MainPage* constructor to initialize the page content with the main test page from the SLUTF. Internally, this checks for the current platform, and in the case of Windows Phone, will return a *MobileTestPage*. Because the same testing framework is used for desktop Silverlight applications, too, it does not use conventional page navigation internally. Instead, it uses a "slide"-based model, wherein each slide is a UI element that takes up part or all of a phone page. For this reason, you need to hook up the *BackKeyPress* to perform the internal pseudo-navigation by invoking the *IMobileTestPage.NavigateBack* method. Without this, then any time the user presses the Back key, from any page, this would exit the test application.

```
public MainPage()
{
    InitializeComponent();

    Content = UnitTestSystem.CreateTestPage();
    IMobileTestPage testPage = Content as IMobileTestPage;
    if (testPage != null)
    {
        BackKeyPress += (s, e) => e.Cancel = testPage.NavigateBack();
    }
}
```

Now it's time to write some tests. If your application employs good Separation of Concerns (SoC), it should be easy to test the viewmodel, the view, and the model independently. You could instantiate your non-visual viewmodel and model types, and exercise their methods and properties. For this simple example application, which doesn't have a viewmodel, you can start by exercising the *MainPage* constructor. This should always succeed, so you add an assertion that should always be true. The class for your tests is attributed with the *TestClass* attribute, and each method has at least a *TestMethod* attribute. You can optionally add a *Description*, which will show up in the test UI at runtime.

```
[TestClass]
public class Tests : SilverlightTest
{
    [TestMethod]
    [Description("Construct the MainPage - always succeeds")]
    public void MainPageConstructor()
    {
        SimpleApp.MainPage mp = new SimpleApp.MainPage();
        Assert.IsNotNull(mp);
    }
}
```

When you run the test application, it finds any *TestClass* types in the assembly, and runs through all the *TestMethods*, as shown in Figure 14-7. Either tap the Run Everything button, or wait for the five-second timeout, when all tests will be executed.

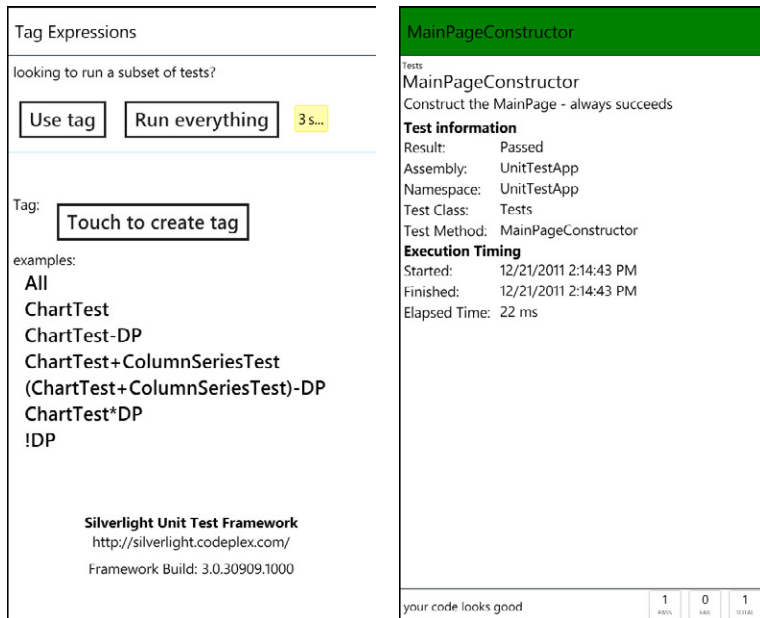


FIGURE 14-7 You can run the test application to exercise each of the test methods.

So far, you have only one *TestMethod*, which you expect to succeed. You'll notice the results include a count of passes and failures at the bottom. The green color is used throughout to indicate a successful test run (all tests passed).

In cases for which you expect certain operations to throw exceptions, you can exercise these code paths with a test that includes the *ExpectedException* attribute. In the listing that follows, you also include a method that performs the exact same test, but does not have the *ExpectedException* attribute. This test should fail; it is included here to illustrate what happens when you have an unexpected failure (as opposed to an expected failure).

```
[TestMethod]
[Description("Assign negative height - throws ArgumentException")]
[ExpectedException(typeof(ArgumentException))]
public void BadPageSizeThrows()
{
    SimpleApp.MainPage mp = new SimpleApp.MainPage();
    mp.Height = -1;
}

[TestMethod]
[Description("Fake failure - to illustrate failure UI")]
[Tag("Dummy")]
public void UnexpectedFailure()
{
    SimpleApp.MainPage mp = new SimpleApp.MainPage();
    mp.Height = -1;
}
```

Note the *Tag* attribute on the second method. This can be used in the test UI to filter out (or filter in) selected methods, based on their *Tag* values. Figure 14-8 (left) shows the result of running the first method, which includes the *ExpectedException* attribute; the exception is thrown, and the test method succeeds. Figure 14-8 (right) shows the result of running the second method, which does not include the *ExpectedException* attribute, thus, the same exception is thrown, and the test method fails. Any failure of any test method causes the entire test run to fail, and this is flagged as red in the report.

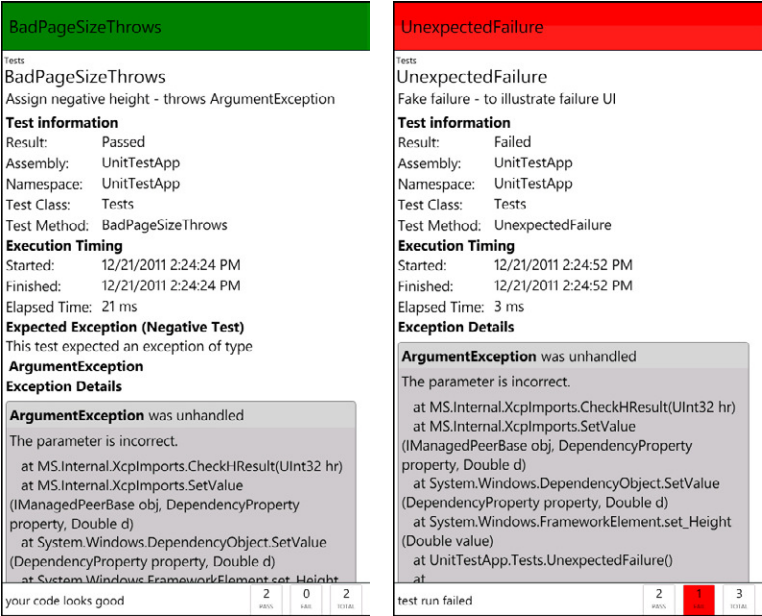


FIGURE 14-8 Test results for an expected exception (on the left) and an unexpected exception (right).

Figure 14-9 (left) shows how to use a *Tag*. In this case, any methods that have the *Tag* attribute “Dummy” are excluded. Figure 14-9 (right) is a screenshot of a later test run, which shows the history list for tags that you have used previously. The history list items are buttons, so you can select an item from the list to reapply it for the next run. Tag expressions use a language based on Extended Backus-Naur Form (EBNF) grammar, which is a remarkably rich feature that allows for inclusion, exclusion, union, intersection, complement, and difference. In other words, you can build complex expressions where you filter in or filter out tests, based on their tags.

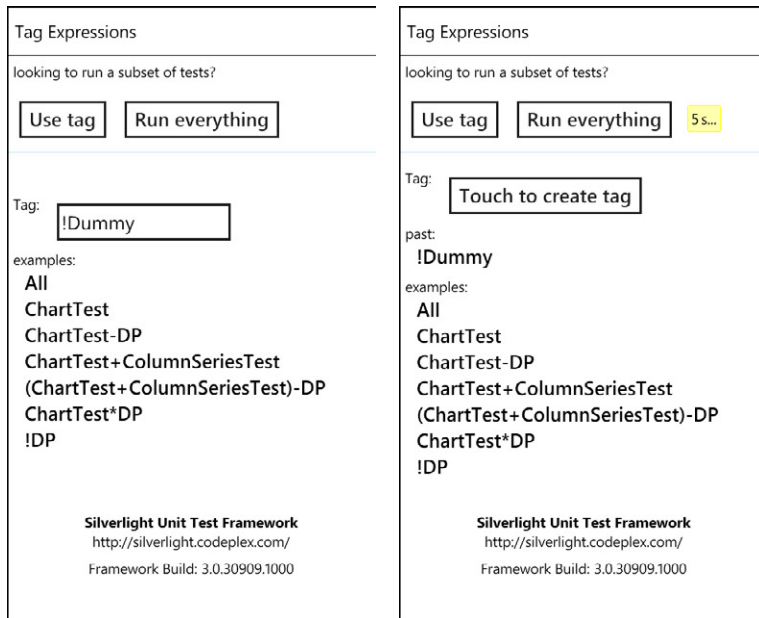


FIGURE 14-9 You can filter out methods by specifying an arbitrary tag.

In addition to using tags for filtering, you can explicitly exclude tests directly in your code by using the *Ignore* attribute. You can use the *TestInitialize* attribute for methods to be run before the other tests start, and *TestCleanup* for any post-run operations that you want to perform.

Exercising non-UI units of your solution is relatively straightforward. Exercising UI elements and simulating UI gestures, is a little more involved. The code that follows shows how you can achieve this. This test method instantiates a page, and then invokes the *FindName* method to find the "go" *Button* by name. Rather than assigning this to a *Button* variable, it is assigned to a *ButtonAutomationPeer* object. This type is defined in the standard *System.Windows.dll*; it includes an *Invoke* method so that you can call up the default behavior. In this case, this simulates the user tapping the button. As a result of this operation, you expect that the current *DateTime* value is inserted into the Result *TextBox* so that you can get hold of that directly and examine its *Text* property.

UI gestures will tend to be asynchronous. You can queue up any asynchronous method call via the *EnqueueCallback* method. All queued operations will be executed asynchronously, and the results will be deferred until you call the *EnqueueTestComplete* method.

```

[TestMethod]
[Asynchronous]
[Description("Invoke the goButton asynchronously")]
[Tag("Async")]
public void InvokeGoButton()
{
    SimpleApp.MainPage mp = new SimpleApp.MainPage();
    ButtonAutomationPeer goButtonPeer =
        new ButtonAutomationPeer(mp.FindName("goButton") as Button);
    IInvokeProvider goButtonInvoker = (IInvokeProvider)goButtonPeer;

    TextBox resultText = mp.FindName("resultText") as TextBox;

    EnqueueCallback(() => goButtonInvoker.Invoke());
    EnqueueCallback(() => Assert.IsTrue(!String.IsNullOrEmpty(resultText.Text)));
    EnqueueTestComplete();
}

```

The SLUTF is a great tool for building unit tests for Windows Phone applications, and you are strongly encouraged to make use of it in your own solutions.

Certification and Publication

The only supported way to install an application onto a retail phone (that is, a phone that has not been developer-unlocked) is via the Windows Phone Marketplace. The portal to the marketplace is called AppHub, and you can access it at <http://create.msdn.com>. This is the same place for submitting Xbox 360 XNA indie games.

When you're ready to submit your application to the marketplace, you should ensure that you have the following information ready:

- **Your AppHub registration** To submit an application, you need to log on to AppHub by using your Windows Live ID (WLID).
- **Your application's XAP** This must be the release build.
- **A title** This should match the title you used in your WMAManifest.xml
- **The category and sub-category under which you want your application to be listed** You can choose from the drop-down lists supplied on the marketplace at the time of submission.
- **Version number of your application** This is arbitrary, although it should obviously be meaningful to your own versioning scheme.

- **A full description** This will appear in the marketplace. This is not the same as the description in the WMAAppManifest.xml (which, anyway, is restricted to 255 characters). The marketplace description is limited to 2,000 characters.
- **(Optional) short description** This will be used if your application is used as the feature application in marketplace. The maximum is 25 characters.
- **One or more keywords** These are arbitrary, but you must supply at least one.
- **(Optional) support contact email address** This can be any contact address; it is not typically the same as your WLID.
- **Three icon images** These must all be 96-dpi PNGs and be very similar to the images used in your application's tile (except that they should not use transparency):
 - **173x173 pixels** This should be the same as the image that you use for when the user pins your application to the Start page, but without transparency.
 - **99x99 pixels** A slightly larger version of the standard icon used in the application list on the phone.
 - **200x200 pixels** This is displayed in the marketplace, in the Zune PC client.
- **(Optional) panorama background art** This must be a 1000x800-pixel, 96-dpi PNG. It will be used if your application is presented as the feature application in the marketplace and in Zune.
- **One or more screenshots** These must be 480x800 pixel, 96-dpi PNGs, all in portrait mode. You must supply at least one, but you can supply up to eight.
- **Pricing and primary offer currency** You can choose \$0.00 if your application is free.
- **(Optional) tester notes.** These are useful to provide special testing instructions for the testing and certification of your application, in case it is not clear from the UI. This is limited to 1,000 characters.



Note For version 7.1 projects, you can also use the Marketplace Test Kit to analyze your application prior to submitting it to the marketplace. You can read more on this in Chapter 20, "Tooling Enhancements."

The submission tool is intuitive, and there are only four pages in total. You can save at any point, so you can pause if you need to, and then go back to complete the submission later. Or, you can abandon the process at any point before final submission. The first page asks you to provide the application name and version, and to upload the XAP via the AppHub portal, as shown in Figure 14-10. The supported language(s) and required capabilities are detected automatically when your XAP is uploaded and scanned. Note the Requires Technical Exception field; this option is there in case your application requires an exception to the certification approval process for technical reasons. You would not normally select this option, because doing so will slow down the approval process, and approval could be withheld.

The screenshot shows a web browser window with the URL <https://windowsphone.create.msdn.com/AppSubr>. The page title is "App Submission". At the top, there is a navigation bar with five tabs: "upload", "describe", "price", "test", and "submit". The "upload" tab is currently selected. Below the navigation bar, the heading "submit an app!" is followed by a brief instruction: "Let's get started. Distribute your app by giving it a name and uploading the app package. You can also learn what to expect during this [submission and certification process](#)." Below this, there are several required fields marked with an asterisk:

- * App name for App Hub:** A text input field containing "My Super Cool App". Below it, a note says "App name only visible in App Hub".
- * Distribute to:** Two radio buttons are present. The first is "Public Marketplace" (selected). The second is "Private Beta Test. Learn more about [beta testing](#)."
- * Browse to upload a file:** A text input field containing "MySuperCoolApp.xap" and a "Browse" button. Below this, technical details are displayed: "Max size: 225 MB", "Expected format: *.xap", "File size: 13 KB", "OS: Windows Phone 7", "Detected languages: English", and "Detected capabilities: data services, Silverlight framework".
- * App version number:** Two dropdown menus showing "1" and "0".

At the bottom, there is a checkbox labeled "Requires technical exception?". To its right, a note states: "Submitting a technical exception will add several days to the certification approval process unless you have been previously approved. Additionally, exception request approval is guaranteed. [What is a technical exception and why do I need it?](#)". At the very bottom, there are two buttons: "Save and Quit" and "Next".

FIGURE 14-10 The first page of the submission tool, in which you enter your app's name and XAP.

Figure 14-11 depicts the second page, where you provide more metadata, including the descriptions to be used in the marketplace.

https://windowsphone.create.msdn.com/AppSubr Describe

tell us about your app

The information you provide here is displayed in the Marketplace catalog so users can learn about your app. Click on each of the languages listed below on the left to enter localized details for each language we detected in your app.
[Learn more about the fields on this page.](#)

Category

*** Required fields**

*** Category** lifestyle ▼

*** Subcategory** food + dining ▼

Details

English →

English app name: MySuperCoolApp

Short description: A super-cool app

*** Detailed description:** Sed ut perspiciatis unde omnis iste natus error sit voluptatem accusantium doloremque laudantium, totam rem aperiam, eaque ipsa quae ab illo inventore veritatis et quasi architecto beatae vitae dicta sunt explicabo. Nemo enim ipsam voluptatem quia voluptas sit aspernatur aut odit aut fugit, sed quia consequuntur magni dolores eos qui ratione voluptatem sequi nesciunt. Neque porro quisquam

*** Keywords:** super cool rocket monkey
[help choosing effective keywords](#)

Legal URL:

Email address: andrew@contoso.com
For app support

FIGURE 14-11 Certification, page 2: the app descriptions and images.

The bottom of the second page is where you upload your artwork, including screenshots, as shown in Figure 14-12. Be aware that although these must be in PNG format, they should not use transparency because they might be used in the Zune PC client, where transparency would be poorly rendered. So, for example, you should not use the same 173x173-pixel image here that you use within the application for the background, if that image uses transparency.

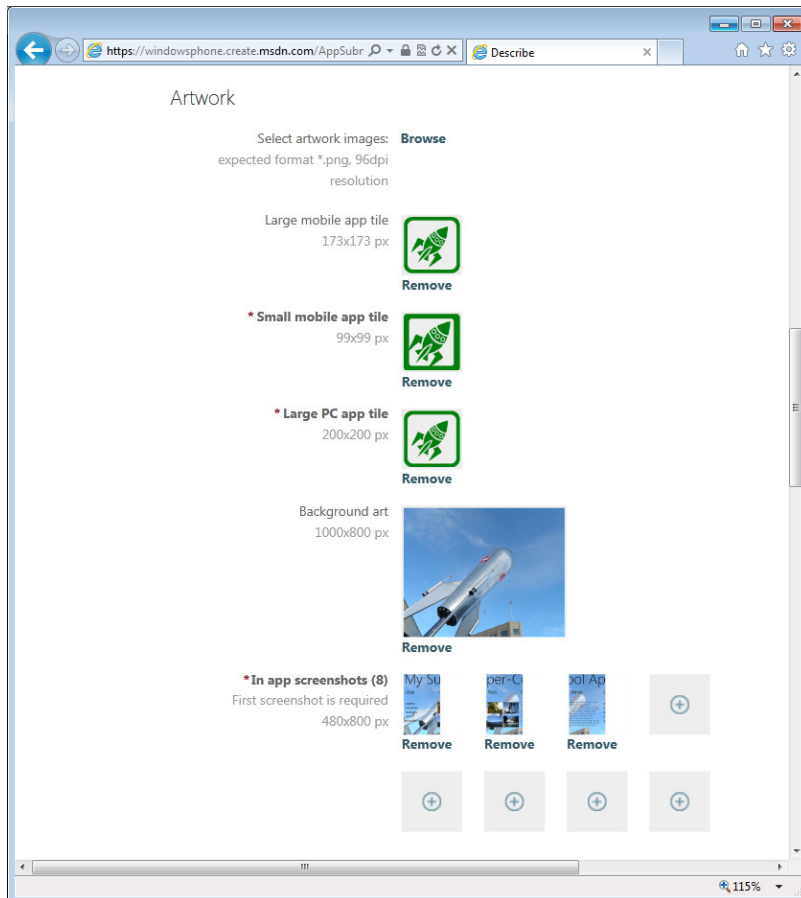


FIGURE 14-12 Certification, page 2: preparing to upload images.

On the third page (Figure 14-13), you specify the pricing that you want to apply for the app, and the regional distribution. If you want your application to be free, set the price to zero.

The screenshot shows a web browser window with the URL <https://windowsphone.create.msdn.com/AppSubr>. The page title is "set price and market availability". Below the title, there is a paragraph explaining that users can set the price tier and trial options, and select the markets for distribution. It also mentions that customers will see an approximate price equivalent in their currency and that sales in multiple countries may settle in different amounts due to currency fluctuations.

Below the paragraph, there is a link to "Learn more about taxes and other worldwide legal responsibilities." and another link to "Learn about game ratings on how to get certification."

The "Select Price tier:" section shows a dropdown menu set to "0.00" and "USD". Below this, there is a checkbox for "Enable trials to be downloaded:" which is currently unchecked.

The "Worldwide distribution" section is expanded, showing a list of regions. The "Africa, Asia and the Pacific" region is selected, and a list of countries is shown below it. Each country has a checkbox, a name, a price, and a currency code.

Country	Price	Currency
<input checked="" type="checkbox"/> Australia	0.00	AUD
<input checked="" type="checkbox"/> Hong Kong SAR	0.00	HKD
<input checked="" type="checkbox"/> India	0.00	INR
<input checked="" type="checkbox"/> Japan	0.00	JPY
<input checked="" type="checkbox"/> New Zealand	0.00	NZD
<input checked="" type="checkbox"/> Russia	0.00	RUB
<input checked="" type="checkbox"/> Singapore	0.00	SGD
<input checked="" type="checkbox"/> South Africa	0.00	ZAR
<input checked="" type="checkbox"/> South Korea	0.00	KRW

FIGURE 14-13 The third page of the submission tool is where you specify the price and market availability for your app.

On the fourth and final page, you can add any notes that you think would be useful to the testers. If your application behaves in an unexpected way in any scenario, you should document it here so that the testers don't fail it simply because they don't understand how it's supposed to work. On this page, you can also choose whether or not to publish automatically as soon as your application passes testing, as shown in Figure 14-14.

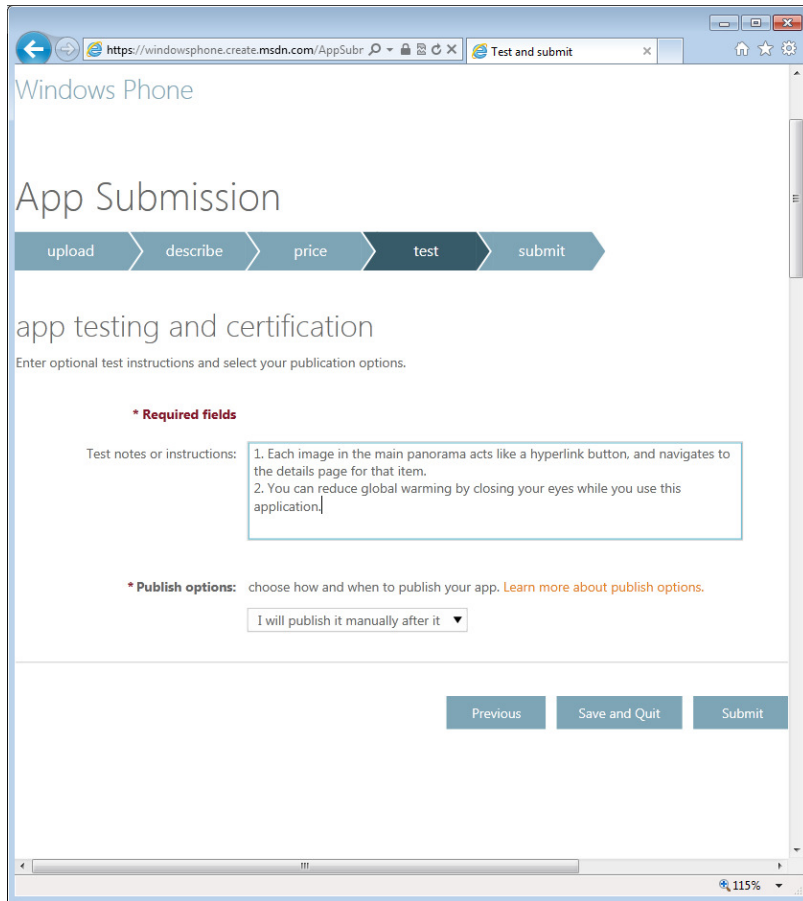


FIGURE 14-14 The final page of the submission tool.

When you're done filling in the submission form—assuming that you haven't missed any required field—you'll be returned to your dashboard, where your new application will be listed as pending certification.

When you submit your application, it goes through both static validation and automated testing to verify that it meets all the policies and requirements. If it passes certification, it is repackaged and signed before it is made available to the marketplace. The Windows Phone Application Certification Requirements are described at <http://go.microsoft.com/fwlink/?LinkID=183220>.

You can cancel a submission before or after it has been certified, but you cannot do anything to it while it is actually going through the certification process.



Note At the time of this writing, the marketplace does not encrypt XAPs. This means that if you're concerned about piracy, you might opt to obfuscate your application before you upload it to the marketplace.

Updates

You can update the marketplace catalog information for your application at any time. This includes the descriptions, artwork, keywords, pricing and regional availability, and so on. Although this goes through the same ingestion pipeline as a full submission, if you do not update the XAP, the time-consuming steps of validating and testing the XAP are skipped, so this kind of update is relatively quick. You can also upload a new version of your XAP at any time, and in all cases, you use the same submission form and upload mechanism. Just as with a full submission, you can check the progress of an update in marketplace through the portal.

When you perform an incremental build of your application in Visual Studio, and run it in the emulator or a device, the fresh XAP replaces the previous version. However, it does not replace any data stored in isolated storage. On the other hand, if you perform a clean and rebuild, this does wipe out isolated storage. This is because a clean/rebuild performs a complete uninstall/reinstall, as opposed to an update. In the marketplace, when you submit an update, this behaves the same way as an incremental build: it replaces the XAP, but does not touch isolated storage. The only way that isolated storage is automatically cleaned out is when the user uninstalls the application.

So, when you publish an update to your application, you might want to clean out isolated storage yourself, upon the first run of the updated application. Alternatively, if you want to keep the old data, you might need to make allowances for any changes in data schema that you have made. For example, you might want to convert all the old data to the new format upon first run. Be aware that this applies only to data stored in isolated storage files; it does not apply to *IsolatedStorageSettings*, application state, or page state, none of which are persisted across runs of the application.

A common requirement is for an application to check the marketplace periodically to see if there is an updated version and to prompt the user if it detects one. Although the platform includes *MarketplaceXXXTask* Launchers, which will bring up marketplace UI pages, there is no marketplace API with which you can fetch marketplace data programmatically, without showing UI. That said, it is fairly easy to construct web requests to the marketplace to fetch metadata for a given application, assuming that you know what you're looking for. That might not be as simple as it sounds, however, because some of the values in your *WMApManifest.xml* will be modified during marketplace ingestion. This includes the *Author*, *Publisher*, *Capabilities*, and—crucially—your application's *ProductID*.

The screenshot presented in Figure 14-15 shows a simple implementation of the key techniques in this approach (the *MarketplaceData* solution in the sample code). Using this application, you can check marketplace metadata for any application, and then display the data in the UI, but more realistically, you would use this approach to check for the current application's updates.

CONTOSO

marketplace

app name

version

released

rating

FIGURE 14-15 Fetching marketplace metadata.

The application would also realistically know its own application name, but in this example, you're asking the user to input the name. You then construct an *HttpRequest* to fetch the Zune marketplace atom feed for this product. The query template string specifies a *clientType* value of *WinMobile 7.1* (or rather, the URL-encoded "WinMobile%207.1"). This narrows down the query to Windows Phone products, eliminating PC, Zune, and Xbox products. By specifying "7.1," you will get back version information for both versions 7 and 7.1. If you want to narrow the search to version 7 only, you need to specify a *clientType* value of *WinMobile%207.0*.

```
private String queryTemplate =
    @"http://catalog.zune.net/v3.2/en-US/apps?q={0}&clientType=WinMobile%207.1&store=zest";
private XNamespace ns = "http://schemas.zune.net/catalog/apps/2008/02";

private void getData_Click(object sender, RoutedEventArgs e)
{
    String zuneQuery = String.Format(queryTemplate, appName.Text);
    HttpRequest webRequest =
        (HttpRequest)HttpRequest.Create(new Uri(zuneQuery, UriKind.Absolute));
    webRequest.BeginGetResponse(httpRequestCallback, webRequest);
}
```

Note also that you need to specify the default namespace. Most elements in the feed use either the default namespace or the atom namespace (aliased as "a"). If you want to extract any of these elements, for example, <a:content> or <a:author>, you'd need to specify the following namespace, also:

```
private XNamespace ns2 = "http://www.w3.org/2005/Atom";
```

When the web request returns, you load it into an *XDocument*, and then parse the document to find certain elements; specifically, the title, version number, release date, and average user ratings. You populate the UI with these element values. Note that the average user ratings are reported on a scale of 1 to 10, which maps 2:1 to the star rating you see in marketplace.

```
private void httpRequestCallback(IAsyncResult result)
{
    HttpWebRequest request = (HttpWebRequest)result.AsyncState;
    using (WebResponse response = request.EndGetResponse(result))
    {
        using (StreamReader reader = new StreamReader(response.GetResponseStream()))
        {
            XDocument doc = XDocument.Load(reader);
            XElement appVersion = doc.Descendants(ns + "version").First<XElement>();
            XElement releaseDate = doc.Descendants(ns + "releaseDate").First<XElement>();
            XElement userRating = doc.Descendants(
                ns + "averageUserRating").First<XElement>();

            Dispatcher.BeginInvoke(() =>
            {
                version.Text = appVersion.Value;
                release.Text = releaseDate.Value;
                rating.Text = userRating.Value;
            });
        }
    }
}
```

Using this approach, it would be simple to find the number of the latest version in the marketplace, compare it with the version number of the currently executing application, and then prompt the user if necessary to inform her that there's an updated version available. You could even take this a step further: if she indicates that she indeed wants to load the update, navigate to your application's page on the marketplace by using the *MarketplaceDetailTask* Launcher.

Note that the marketplace ingestion process changes over time, with new features added periodically. If you submit an update to an application that was originally published before some marketplace ingestion change, you might find that you need to do extra work to satisfy the new requirements. One classic example of this is the neutral resources language. At one point, the marketplace was changed to require that you add the *NeutralResourcesLanguage* attribute to your assembly. It will reject your uploaded XAP if it doesn't have this. You can do this by going to the project properties, assembly information dialog, and selecting the default language from the list. This adds the corresponding declaration to your assemblyinfo.cs. One twist to this is that if you select a language that is more specific than what you had previously (even implicitly), you will again be rejected. For example, if you select "en-US", this is more specific than an implicit default of "en". So, it is common to set the value to a less specific language; for example, add this line to your assemblyinfo.cs:

```
[assembly: NeutralResourcesLanguageAttribute("en")]
```

Marketplace Reports

The marketplace portal also provides reports on your application, including the number of downloads, payment history (for non-free applications), ratings and reviews, and crash dumps (if any). It is a certification requirement that your application should not allow any unhandled exceptions to propagate out of the application. However, the testing done during the ingestion process is not very comprehensive, so it is easy to miss some code path where an exception might escape. This means that your application might pass certification even if it does throw unhandled exceptions. To assist in mitigation efforts, whenever an application crashes, a tiny crash dump is collected and sent back to Microsoft. This eventually makes its way to the marketplace servers, where a summary is produced, and made available to the application publisher for retrieval.

To fetch marketplace crash dumps, log on to the AppHub, go to My Dashboard, select Windows Phone, and then click the Reports link. The reports page shows charts and tables of daily downloads and crash counts for all your applications, as shown in Figure 14-16.

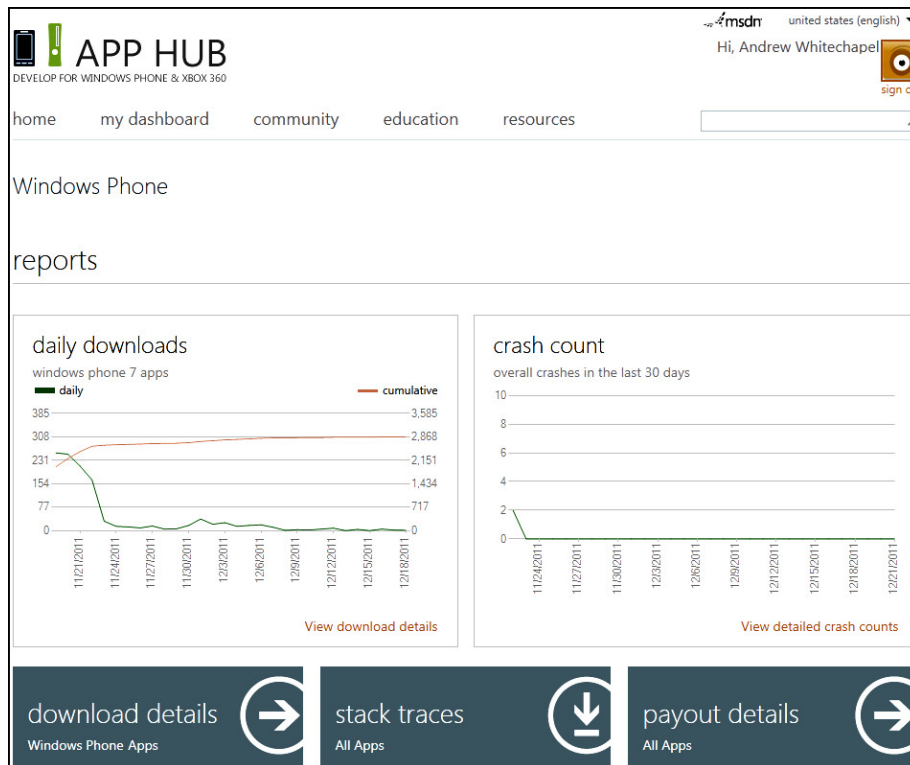


FIGURE 14-16 A marketplace download and crash-count report.

There are links at the bottom of the charts that you can click to drill down into the details for downloads and crashes. On the details pages, there are further links with which you can download the reports in Excel spreadsheet format. This is especially useful for crash dumps, because the spreadsheets will include the stack traces for each crash. These are obviously indispensable for diagnosing the cause of the crash.

Beta Testing

The marketplace supports a simple beta testing mode for applications. When you first submit your application, you have the option to make it available to only a selected set of users. You choose the users by specifying their WLIDs in the submission form. The system allows you to enter up to 100 users for beta testing. When you do this, AppHub will send you an email containing the download link for your application, which you can then send to your selected users. The marketplace will hide the application from general availability, and instead make it visible for download only to those users you selected. You need to handle test results from your beta users directly; there is no marketplace mechanism to support feedback in this scenario. There is a 90-day limit to the beta testing period, and this is enforced by installing a 90-day license for the application on the phone. After this time, your users will no longer be able to launch the application. You can add more test users at any time during this period. Note also that there is no option to terminate your test period before the 90 days is expired.

Submitting an application for beta distribution is free, and it does not count against your allowed number of application submissions. Anyone with a Windows phone (and a WLID) can be a beta tester, and he does not need to have a developer-unlocked device. The process is also fast because the application does not go through the full set of certification tests, and will generally become available to your beta testers within hours of submitting it to the program.

There is no special “beta-to-release” upgrade path, so when your beta test is complete, you will need to submit a fresh, full submission to publish your application in the normal way.

Versions

Apart from fixing bugs, improving the UI, providing fresher data, and adding features, the other main reason for updating is to take advantage of new features in the latest version of the platform and SDK. As of this writing, this means updating from version 7 to 7.1. However, some users will not have upgraded their devices to 7.1, so you would typically want to maintain both major versions of your application. This means forking your source code and applying bug fixes in two places, but it also means that users can continue to get the benefit of your updates, regardless of which platform version they use.

The marketplace allows you to maintain two versions, and to submit updates to each one independently. Note, however, that some of the marketplace metadata for your application is shared across both versions. Specifically, the catalog details (descriptions and screenshots), pricing and regional availability, ratings and reviews, and the hidden/live status are all shared by both versions. Apart from the XAP itself, the only marketplace data that is kept separate for each version is the version number and the published/unpublished status. This means the version number of your 7 version doesn't need to be the same as the version number for your 7.1 version. You might submit, for example, four version updates for your 7 version, and only two version updates for your 7.1 version, and so on.

There are a few other non-obvious issues:

- If you add new language support to your 7 version, and you also have a 7.1 version in the marketplace, then you must add the new languages to the 7.1 version first.
- If you currently have an update in progress going through marketplace ingestion for any version, you cannot start another update until the previous one has completed the process.
- The hidden/live status was introduced with version 7.1. Because of this, if you remove (unpublish) your 7.1 version, then you cannot change the value of the hidden/live status of your 7 version. The only way to change this is to republish a 7.1 version.
- You can submit an update to a version while it is in the hidden state. This is a useful step to take if your update includes significant bug fixes. Thus, while the fixed version is going through marketplace ingestion, you can keep the application hidden so that new users cannot download the old buggy version.
- When you publish a 7.1 version, users with 7 phones can only download the 7 version, and users with 7.1 phones can download only the 7.1 version. If you remove (unpublish) your 7.1 version, then users with 7.1 phones will still not be able to download the 7 version.
- You cannot update a 7.1 version with a 7 version; that is, platform downgrade is not possible.
- You can change your application name as part of an update, if you wish, although this would probably be confusing to users. Regardless, you may not change the *ProductID*. A change of the *ProductID* is considered as an entirely different application, not an update.
- If you have no previous version of your application, and your first published version is a 7.1 version, you will not be able to submit a 7 version, ever. For this reason, you should generally submit a 7 version first, even if you intend to submit a 7.1 version immediately thereafter.



Note An additional minor version, 7.1.1, was announced in February 2012. For details of this, including how it affects marketplace publication, see Chapter 16, “Enhanced Phone Services.”

Light-Up Features

One reason to publish an update for an old version 7 application is to enable a “light-up” scenario. That is, to take advantage of version 7.1 features if the application is running on Windows Phone 7.1. Be aware that this approach is not supported by Microsoft. To be clear, this strategy is not the same as publishing a 7.1 version of your application. Rather, it means publishing a 7 update to a 7 application, but including 7.1 features. How is this possible? The answer is runtime reflection. Consider the following line of code:

```
ApplicationBar.Mode = Microsoft.Phone.Shell.ApplicationBarMode.Minimized;
```

This code is version 7.1–specific: it uses the *Mode* property and the *AppBarMode* enumeration, both of which were introduced in version 7.1, and were not available in version 7. The purpose of these is to allow you to switch your App Bar between the default size and the new *Minimized* mode, which takes up less real estate on the screen, as shown in Figure 14-17. This is the *TestLightUp* solution in the sample code. The *Minimized* mode was designed to be used on Panorama controls (and also with *Opacity* set to something less than 1); it should generally not be used, otherwise.

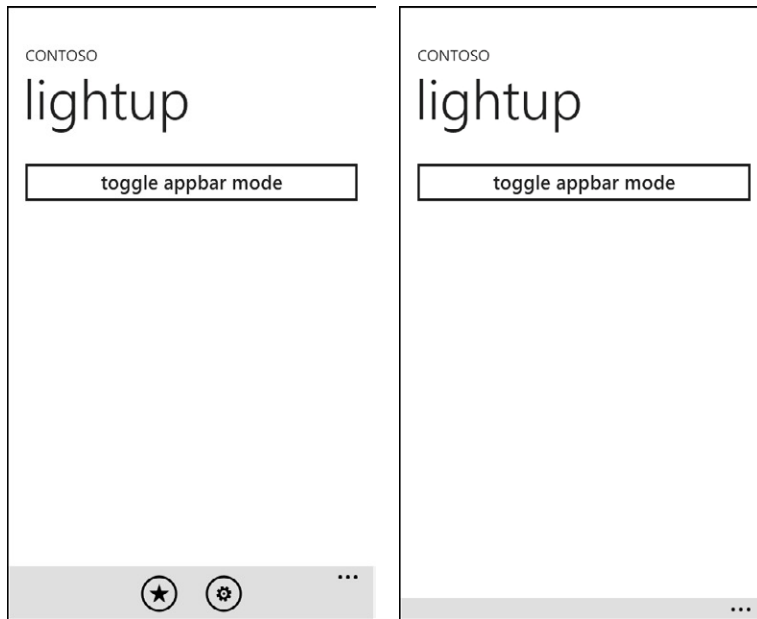


FIGURE 14-17 This application shows the *AppBar* in *Default* mode (on the left) and *Minimized* mode (right).

If you tried to include the preceding line of code in a version 7 application, it would fail at compile time. Instead, if the application finds itself running on OS 7.1, it can gain access to the *Mode* property of the *AppBar* object at runtime via the reflection APIs.

```
private bool isAppBarMinimized;
private const int Default = 0;
private const int Minimized = 1;

private void toggleAppBarMode_Click(object sender, RoutedEventArgs e)
{
    Type t = typeof(AppBar);
    PropertyInfo pi = t.GetProperty("Mode");
    if (pi != null)
    {
        pi.SetValue(this.AppBar, isAppBarMinimized ? Default : Minimized, null);
        isAppBarMinimized = !isAppBarMinimized;
    }
}
```

Using this approach, you can build in any number of light-up features in your application. When running on version 7, the test fails, and the code continues without the feature. On the other hand, when running on version 7.1, the test succeeds and the feature lights up. Note that reflection is a costly operation at runtime, in terms of performance. It also adds to your code size (albeit trivially); this is code that, by definition, is inoperative when running on 7 devices. Also, recall that if you have the version 7.1 SDK installed, then the emulator is always running 7.1, and although it emulates version 7 when you run a 7 solution, this is still the version 7.1 emulator running in compatibility mode. This means that even though it's easy to test the case where you're running on version 7.1, it is not possible to test the opposite, unless you have access to a version 7 device or another development PC that has the version 7.0 SDK installed.

Obfuscation

As of this writing, the marketplace does not encrypt XAPs. For this reason, if you want to increase the protection of your intellectual property, you might want to consider obfuscation. Visual Studio ships with a version of the PreEmptive Dotfuscator Community Edition, but this does not work very well with Silverlight XAPs. Instead, you can download the Dotfuscator Windows Phone Edition. This is free to use and has all of the features of the commercial version, but is specifically designed for Windows Phone applications. There are two main features in Dotfuscator: obfuscation and analytics instrumentation. You can download the tool from <http://www.preemptive.com/windowsphone7.html>.



More Info PreEmptive had a joint agreement with Microsoft to provide free runtime analytics reporting for Windows Phone, but this service was terminated at the end of 2011. Dotfuscator no longer collects or processes analytics data from instrumented phone applications, and the portal has been taken offline. On the other hand, the obfuscation features still work perfectly well.

This installation includes the Dotfuscator engine, a command-line version of the tool, and a GUI version, plus a couple of sample applications for demonstration purposes. Run the Dotfuscator GUI application, and then create a new project. Open your XAP or simply drag it on to the Input tab. By default, all obfuscation is disabled, and all analytics instrumentation is enabled. If you're interested only in obfuscation, you need to reverse this configuration. To set up a Dotfuscator project for obfuscation only, go to the Settings tab, and then disable instrumentation. At the same time, configure the Global options to enable Control Flow, Linking, PreMark, Removal, Renaming, and String Encryption, as shown in Figure 14-18. These settings are described in the following:

- **Control Flow** Dotfuscator transforms your control flow patterns to output code that is semantically equivalent to your original code, but contains no clues as to how the code was originally written.
- **Linking** This option links multiple assemblies into a smaller number of output assemblies, thereby reducing the size of your deployment.

- **PreMark** Also called *watermarking*, this is used to embed data such as copyright information or unique identifiers into your assembly without impacting its behavior. The purpose of this is to allow you to track unauthorized/pirated copies of your software back to the source.
- **Removal** Also called *pruning*, this feature removes unused types, methods, and fields.
- **Renaming** This feature renames your identifiers (classes, methods, fields, properties, and so on) to short (space-saving) names, which also has the benefit of making your code more difficult to understand when viewed in a decompiler such as Red Gate's Reflector or Telerik's JustDecompile.



More Info Reflector is a tool for browsing, analyzing, decompiling, and debugging .NET assemblies. This is a commercial tool that you can buy from <http://www.reflector.net/>. Telerik has also recently produced a tool that does the same job, called JustDecompile. This is available as a free download from <http://www.telerik.com/products/decompiler.aspx>.

- **String Encryption** Per the guidelines in Chapter 13, "Security," your code should not include any embedded security credentials or any other sensitive data. However, it might include non-sensitive or non-valuable strings, and this feature encrypts those strings to make the code even more difficult for an attacker to understand. Keep in mind that there is a runtime performance penalty: the encrypted strings need to be decrypted upon use.

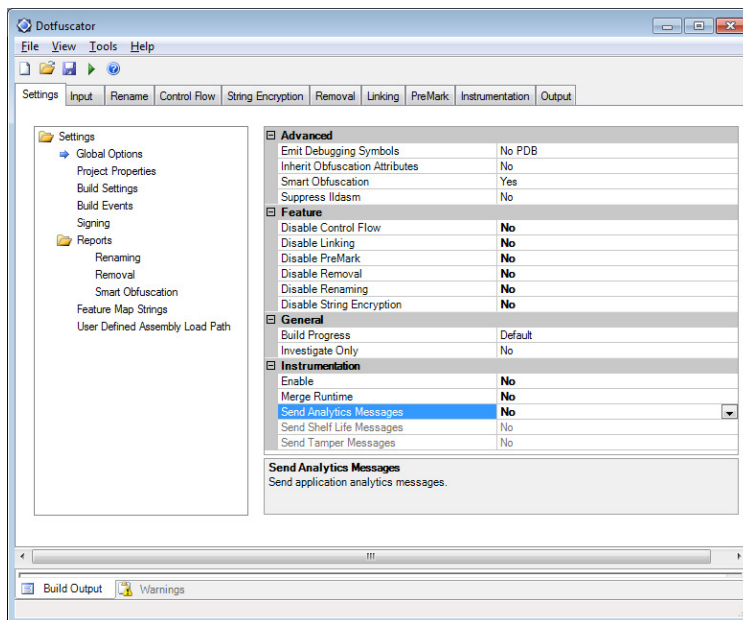


FIGURE 14-18 Dotfuscator project configuration for obfuscation only.

You should experiment with varying levels of obfuscation to find a compromise that meets your needs. This is particularly the case with Silverlight applications because of the extensive use of XAML,

which in turn makes extensive use of reflection. Reflection makes it difficult for the tool to statically analyze the assemblies to determine what level of obfuscation is safe, and will not result in runtime errors.

In addition to obfuscation, while Dotfuscator is transforming your code, it can also eliminate unused code and data, coalesce strings, merge assemblies, and so on, all of which have the added benefit of reducing the size of your assemblies. On the Input tab, select all options except Library. On the Rename tab, under Built-In Rules, clear the Fields Of Silverlight And WPF UserControls check box, because this would otherwise almost certainly introduce errors when the corresponding XAML is processed. On the Removal tab, under Options, select the check box to Remove Only Literals (Const Definitions). Removing unused metadata and code can end up removing code and metadata that in fact is not unused. Again, this is because of the way XAML is processed.

When you're happy with your configuration, save the project. This saves the configuration settings as an XML file. The location of this is arbitrary; however, you would typically put it in a subfolder of your solution. You can also specify the target folder for Dotfuscator output as part of your configuration, and by default this will be in a subfolder of the folder containing the XML configuration file.

When you build the project, this runs the analysis and obfuscation transforms. Build output is displayed in the list at the bottom, along with any build errors or warnings that might require you to change configuration settings. The resulting obfuscated code can be seen in the Output tab, as shown in Figure 14-19. Dotfuscator does not produce output source code; the Output tab is just for information. Instead, Dotfuscator will produce an output XAP, which you can then deploy to the emulator or device, or upload to the marketplace just like a normal XAP.

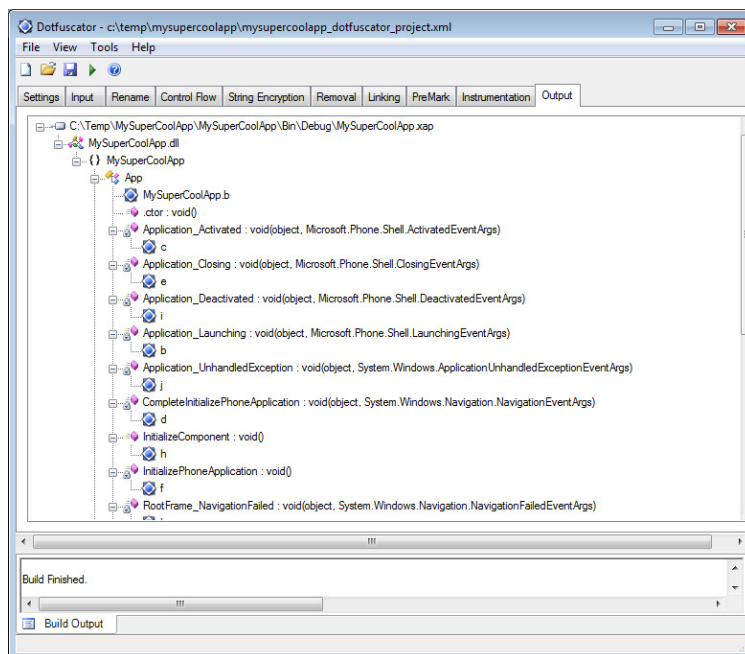


FIGURE 14-19 Obfuscated output code.

Note that some of the transforms that Dotfuscator performs can result in unexpected runtime behavior. Obfuscation with this kind of tool is an iterative process, and you might need to fine tune your configuration, especially in terms of what elements to exclude from obfuscation, until you achieve a build that is error-free at runtime.

Ads

One of the ways to monetize your application is to incorporate advertisements. There are several third-party options for this, although the standard Microsoft option is one of the best. To put advertisements in your application, you put one or more instances of the *AdControl* into your XAML or your code. By default, the *AdControl* includes a test advertisement banner. To use real advertisements, you also need to register your application with the Microsoft pubCenter, as shown in Figure 14-20.

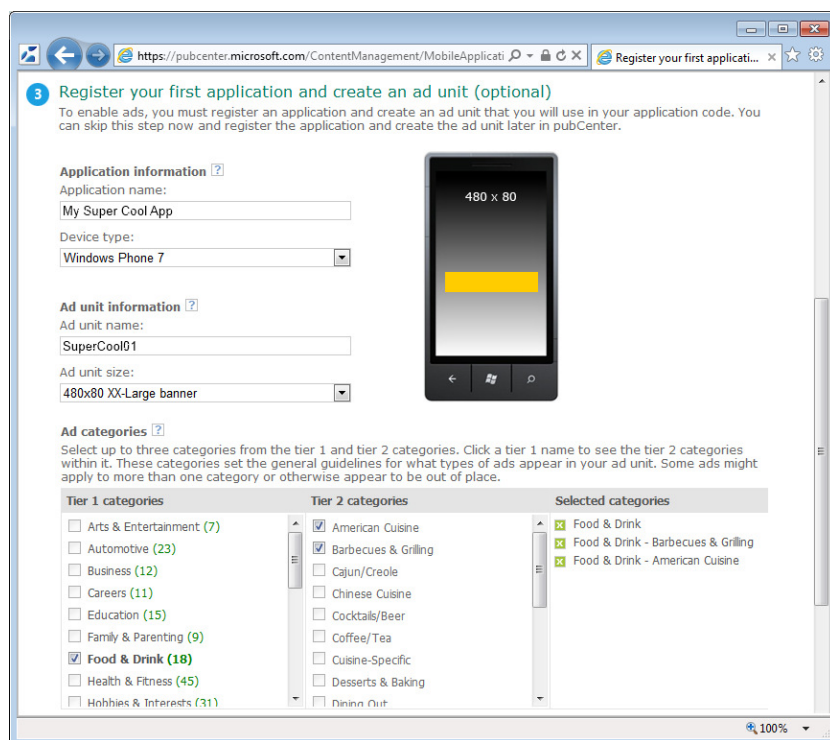


FIGURE 14-20 Registering an application and creating an ad unit.

The general steps are as follows:

- Sign up with pubCenter at <https://pubcenter.microsoft.com/Login>. You can use your WLID to sign in, and thereby link your pubCenter user name to your WLID. Apart from your name and email address—which will be automatically extracted from your WLID profile—you just need to specify the country/region where you live, and the currency in which you prefer to be paid.

- (Optionally) register your first application, and then create an ad unit. To register the application, you need to supply the application name. For an ad unit, you provide some arbitrary ad unit name, and then specify the ad unit size. An ad unit is the space in your application UI where ads appear. Ad units can contain one or more ads. You decide where you want the ad unit to appear on a page, and you can add different ad units to different pages in your application, or use a single ad unit for all pages. The standard size options are 350x50 pixels or 480x80 pixels. The 480x80 size tends to work better with the standard Windows Phone 7 screen size.
- You can also choose up to three ad categories (for example, Arts & Entertainment, Food & Drink, and so on). This is an important step, because the mechanism for retrieving ads for your application depends largely on the categories that you select. Ideally, you would pick categories that closely match the content and theme of your application, to increase the likelihood that the ads will be seen by the target audience selected by the ad network. The more people who see your ads, the more ad revenue you will receive.
- Finally, you have the option to exclude ads that link to specific URLs. You might want to do this to prevent ads from linking to your competitors' sites, for example.
- When you confirm all your entries, you'll be given an application ID (a GUID) and an ad unit ID (an integer value). To show ads in your application, you set these IDs into the corresponding properties on the instance(s) of the *AdControl* in your application, either in XAML or in code.
- After you publish your application, you will receive ads automatically.

The Microsoft Advertising SDK for Windows Phone is installed by default with the Windows Phone 7.1 SDK. It is also available as a separate, free download from <http://www.microsoft.com/download/en/details.aspx?id=8729>, which is useful if you're still using the version 7 SDK. Also, be aware that the Advertising SDK is updated independently of the Windows Phone SDK; therefore, you might want to use a later version of the Advertising SDK than the one that ships with the Windows Phone SDK. Here are some points to keep in mind about ads in your application:

- You cannot choose specific advertisements, although you can filter out ads that link to specific URLs.
- You will be paid 70 percent of the revenue received by Microsoft for your ads, less taxes. Payments are made monthly for any month where the pay-out is \$50.00 or more (including any unpaid amounts brought forward from previous months).
- As of this writing, the Microsoft Advertising SDK only functions in the United States, although this is set to expand worldwide. Consequently, the feature is available only to developers who have a company registered in the United States or a personal tax identifier issued by the United States government.
- Under the covers, the *AdControl* in your application requests an ad via a web service call to the Microsoft Ad Exchange. The Ad Exchange is a system wherein multiple ad networks can bid for the opportunity to display their ad in your application. The network with the highest bid wins the right to have their ad returned to the *AdControl* and displayed.

- Users will see a standard banner or text ad in your *AdControl* instance(s), and they can interact with the ad in standard ways. This means that they can initiate click-to-web, click-to-call, and click-to-marketplace actions. The terms “click-to-web,” and so on, are standard in the industry, even though a phone user will “tap” with a finger rather than “click” with a mouse. Click-to-web ads launch a browser task, and when the user taps the Back button, she returns to where she was in your application. Click-to-call ads launch the phone dialer. After the user completes her call to the advertiser, control is returned to your application.
- Using the *AdControl* increases the size of your application. Specifically, the Silverlight Ad Control assembly is approximately 98 KB.
- The *AdControl* makes use of the user’s data service, and in many cases this incurs a cost to the user. This is one reason why developers adopt the model of ads in the free version of their application, and no ads in the paid version.
- After publication, you can obtain reports on how your ads are performing by logging on to the pubCenter portal.

The simplest way to incorporate ads in your application is to drag the *AdControl* from the toolbox onto the page designer. This will add references to the two main advertising assemblies, Microsoft.Advertising.Mobile.dll and Microsoft.Advertising.Mobile.UI.dll. These are installed with the SDK, typically in %ProgramFiles% \Microsoft SDKs\Advertising for Phone\Libraries\ (32-bit computers) or %ProgramFiles(x86)% \Microsoft SDKs\Advertising for Phone\Libraries\ (64-bit computers). Dropping the *AdControl* also adds the corresponding namespace to your XAML, as shown here:

```
xmlns:my="clr-namespace:Microsoft.Advertising.Mobile.UI;assembly=Microsoft.Advertising.Mobile.UI"
```

Finally, the declaration of the *AdControl* instance itself. Initially, you would use the specific test values for *ApplicationId* and *AdUnitId*; that is, “test_client” and “Image480_80”, respectively.

```
<my:AdControl
    Grid.Row="2"
    AdUnitId="Image480_80" ApplicationId="test_client" Height="80" HorizontalAlignment="Left"
    Name="adControl1" VerticalAlignment="Top" Width="480" />
```

You want the ads to be seen by your users, but you don’t want them to make the overall UI cluttered or difficult to navigate. For this reason, it is generally better to put the *AdControl* either at the top or the bottom of your page. Most developers prefer the bottom, but if you have an App Bar, it might be more pleasing to position it at the top. Also recall that the standard grid sizes on wizard-generated pages will give you a content panel of 456x535 pixels, which won’t work very well with an *AdControl* sized at 480x80. To avoid this, you would generally put the *AdControl* outside any such content panel, to ensure that it can display at its full 480x80-pixel size.

The code that follows shows how you can add the *AdControl* programmatically, instead of declaratively. Again, you would replace the test *AppId* and *AdUnitId* prior to publication.


```
private void InitializeAds()
{
    String AppId = "test_client";
    String AdUnitId = "Image480_80";
    AdControl ac = new AdControl(AppId, AdUnitId, true);

    ac.Width = 480;
    ac.Height = 80;
    ac.VerticalAlignment = VerticalAlignment.Top;
    ac.HorizontalAlignment = HorizontalAlignment.Left;

    Grid grid = (Grid)this.LayoutRoot;
    ac.SetValue(Grid.RowProperty, 2);
    grid.Children.Add(ac);
}
```

Figure 14-21 shows the standard wizard-generated data-bound app (the *DataBoundAppWithAds* solution in the sample code) with two instances of the *AdControl*, one on the main page, and one on the details page. For the main page, the title bar is displayed in the normal way, and the *AdControl* is positioned at the bottom. For the details page, there is a visible *AppBar*, so you need to position the *AdControl* at the top, while at the same time you must turn off the *SystemTray* and reduce the title panel to just the page name.

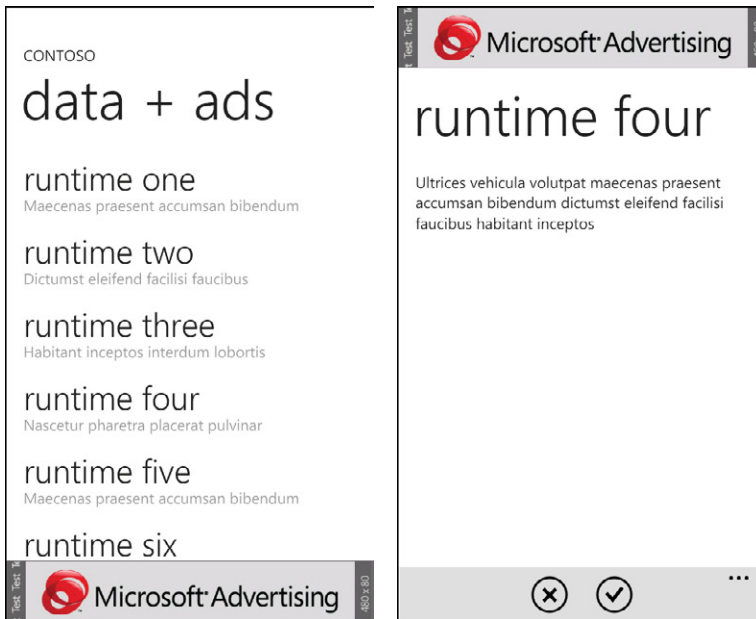


FIGURE 14-21 It's common to position the *AdControl* at either the bottom or top of a page.

Be aware that when you use the *AdControl* in your application, you must also specify the following capabilities:

```
<Capability Name="ID_CAP_IDENTITY_USER"/>
<Capability Name="ID_CAP_MEDIALIB"/>
<Capability Name="ID_CAP_NETWORKING"/>
<Capability Name="ID_CAP_PHONEDIALER"/>
<Capability Name="ID_CAP_WEBBROWSERCOMPONENT"/>
```

These will be identified by the Marketplace TestKit for version 7.1 projects. Unfortunately, they will not be identified by the Capabilities Detection tool, which is the only pre-publication capabilities tool you can use for a version 7 project. Networking is obviously required in order to request and retrieve the ads. The media library is used because of the way the control caches images. The phone dialer and web browser are used in click-to-call and click-to-web scenarios. Despite the potentially misleading capability, the *AdControl* doesn't actually make use of user identity; rather, it uses anonymized demographic data for your identity such as age and gender.



Note Looking forward, the ease with which you can incorporate advertising into your application will become increasingly important as the phone becomes available in regional markets, such as China and India, where users for the most part expect applications to be free to install.

Trial Mode

It is common to provide a trial mode for an application. The idea is to give users an opportunity to try your application for free, before they commit to buying it. In your code, you can detect whether the application is running in trial mode and restrict your features accordingly. It is entirely up to you what form this constraint takes. You might disable certain features, or hide certain pages, or you might enable ads in trial mode and turn them off in the paid version. Tying ads to trial mode is a common pattern, but be aware that these features are strictly unrelated, and it is not uncommon to implement ads in the paid version of your application as well as in the free version.

That having been said, the trial-with-ads approach is common, and easily implemented. You would typically need to detect which mode you're in across multiple pages. Also, fetching the trial mode status is relatively time consuming. For these reasons, it makes sense to cache the information, and to do so in the *App* class, perhaps with a static property, as shown in the code that follows. You can see this at work in the *DataBoundAppWithAds_TrialMode* solution in the sample code.

```
private static LicenseInformation license;
private static bool isTrial;
public static bool IsTrial
{
    get
    {
```

```

#if DEBUG
    return true;
#else
    if (license == null)
    {
        license = new LicenseInformation();
        isTrial = license.IsTrial();
    }
    return isTrial;
#endif
}
private set { }
}

```

On the emulator, the *LicenseInformation.IsTrial* method always returns false; hence the conditional debug statements. Then, in your page code, you can set up *AdControls* (or not), based on the trial mode status.

```

private void InitializeAds()
{
    if (App.IsTrial)
    {
        String AppId = "test_client";
        String AdUnitId = "Image480_80";
        AdControl ac = new AdControl(AppId, AdUnitId, true);

        ac.Width = 480;
        ac.Height = 80;
        ac.VerticalAlignment = VerticalAlignment.Top;
        ac.HorizontalAlignment = HorizontalAlignment.Left;

        Grid grid = (Grid)this.LayoutRoot;
        ac.SetValue(Grid.RowProperty, 2);
        grid.Children.Add(ac);
    }
}

```

This is one good reason for setting up your *AdControls* in code as opposed to declaratively in XAML. If your layout is complex enough to warrant setting up the *AdControl* in XAML, then instead of conditionally creating it, you could alternatively simply set its *Visibility* property so that it is dependent on the trial mode status. Also, observe how this is one scenario for which using the *Visibility* property is more appropriate than using the *Opacity* property.

To allow the user to switch from trial mode to the paid version of your application, you simply redirect him to the marketplace page for your application, from which he can choose to buy the paid version. To do this, you could provide him with a link, a button, or an App Bar button on one or more of your pages. Or, you could simply trigger the marketplace navigation when he selects a paid-only feature somewhere in your UI; for instance, by selecting an item from a list, or navigating to a given page. However, this last approach should be used very carefully, because it could confuse the user

by arbitrarily sending him to an entirely different UX in an unexpected way. The code to go to your application's marketplace page is trivial, as is demonstrated here:

```
MarketplaceDetailTask task = new MarketplaceDetailTask();
task.Show();
```

By default, if you don't set any properties on it, the *MarketplaceDetailTask* will navigate to the page for the calling application. You can't completely test this in an unpublished application, because this would always produce a marketplace exception (both on the emulator, and on a physical device). Also, when you switch to the full version, don't forget to remove the artifacts of the trial/full purchase UI.

```
private void InitializeAds()
{
    if (App.IsTrial)
    {
        // ... set up AdControl as before

        ApplicationBar.IsVisible = true;
    }
    else
    {
        ApplicationBar.IsVisible = false;
    }
}
```

Silverlight Analytics Framework

An early version of the Microsoft Silverlight Analytics Framework (MSAF) has been released as open-source on codeplex at <http://msaf.codeplex.com/>. This is an extensible framework for supporting analytics in Silverlight, Windows Presentation Foundation (WPF), and Windows Phone applications. The MSAF is itself based on the Managed Extensibility Framework (MEF), which is a composition-based, loosely coupled framework for supporting plug-ins. In this way, the MSAF can support an open-ended range of third-party vendors' analytics engines, including Google, Comscore, Webtrends, and so on. In essence, it allows you to apply tracking to any event in your application. The tracking data is sent to the analytics server(s) you choose, and you can then get reports from the corresponding service portal. Be aware that it is a marketplace requirement to list your application's privacy policy. You must also obtain user consent before sending any tracking information to a third-party service.

MEF (pronounced "mef") provides support for composing applications dynamically at runtime, as opposed to statically at compile-time. It is a common pattern for an application to have a composite model, where the total functionality is provided by a number of component parts. In some cases, these components are all known at design-time, so the composition is static. However, it is sometimes useful to be able to build the composition dynamically, where the set of components to be loaded is discovered only at runtime. MEF components (or "composable parts") do not directly depend on one another; instead, they depend on a contract. The idea is that the application specifies the contracts (interfaces) that it wants to consume, or "import," and at runtime it loads the assemblies that implement those interfaces (that is, assemblies that "export the contract types").

To get this all working in a phone application, you first need to add references to the MSAF assemblies, specifically:

- *ComponentModel*
- *ComponentModel.Initialization*
- *Microsoft.WebAnalytics*
- *Microsoft.WebAnalytics.Behaviors*
- *System.Windows.Interactivity*
- One or more analytics service MEF component assemblies that you want to work with, such as *Google.WebAnalytics* or *Webtrends.WebAnalytics.WP7*

You can see this at work in the *TestAnalytics* solution in the sample code. Next, create a class that implements *IApplicationService*, which is a standard Silverlight extensibility mechanism. This is used when you have a need for some global “service” in your application that you want the Silverlight runtime to invoke. The interface declares two methods: *StartService* and *StopService*. The Silverlight runtime will call *StartService* during application initialization, and it will call *StopService* just before the application terminates.

```
public class AnalyticsApplicationService : IApplicationService
{
    public void StartService(ApplicationServiceContext context)
    {
        CompositionHost.Initialize(
            new AssemblyCatalog(Application.Current.GetType().Assembly),
            new AssemblyCatalog(typeof(Microsoft.WebAnalytics.AnalyticsEvent).Assembly),
            new AssemblyCatalog(typeof(
                Microsoft.WebAnalytics.Behaviors.TrackAction).Assembly),
            new AssemblyCatalog(typeof(Google.WebAnalytics.GoogleAnalytics).Assembly));
    }

    public void StopService() { }
}
```

This code initializes a MEF *CompositionHost*, which acts like a container for each *AssemblyCatalog* in which you’re interested, and invokes the *Compose* method. This method matches up all the import requirements in the MSAF with the specific export implementations in the component assemblies and makes them available in the container.

To make use of this application service, you need to add a couple of namespaces to the App.xaml, for the *Microsoft.WebAnalytics* assembly, which acts as a bridge to third-party analytics services, plus one or more third-party services. This example uses the Google Analytics service. You also need to add an XML namespace declaration for the current assembly, where you’ll be defining the *AnalyticsApplicationService*.

```
xmlns:mwa="clr-namespace:Microsoft.WebAnalytics;assembly=Microsoft.WebAnalytics"
xmlns:ga="clr-namespace:Google.WebAnalytics;assembly=Google.WebAnalytics"
xmlns:local="clr-namespace:TestAnalytics"
```

Then, update the *ApplicationLifetimeObjects* section by adding the Google Analytics service to the collection of web analytics services that you want to use. The *WebPropertyId* must be set to a valid Google Analytics Account ID. To get an ID, you need to sign up with the Google Analytics service here at <http://www.google.com/analytics/> (this is a free service).

```
<Application.ApplicationLifetimeObjects>
  <shell:PhoneApplicationService
    Launching="Application_Launching" Closing="Application_Closing"
    Activated="Application_Activated" Deactivated="Application_Deactivated"/>
  <local:AnalyticsAppApplicationService/>
  <mwa:WebAnalyticsService>
    <mwa:WebAnalyticsService.Services>
      <ga:GoogleAnalytics WebPropertyId="UA-12345-1"/>
    </mwa:WebAnalyticsService.Services>
  </mwa:WebAnalyticsService>
</Application.ApplicationLifetimeObjects>
```

In this simple application, you'll put a *Button* on the main page, and then track when the user clicks this *Button*. To do this, in the *MainPage.xaml*, you need to add two further namespaces for the *System.Windows.Interactivity* and *Microsoft.WebAnalytics.Behaviors* assemblies. Using these, you can attach behaviors to your UI elements, specifically so that you can apply tracking to them.

```
xmlns:i="clr-namespace:System.Windows.Interactivity;assembly=System.Windows.Interactivity"
xmlns:mwb="clr-namespace:Microsoft.WebAnalytics.Behaviors;assembly=Microsoft.WebAnalytics.Behaviors"
```

With these namespaces in place, you can attach a *TrackAction* behavior to the button's *Click* event, assigning an arbitrary value for the *Category* and *Value* attributes that will be meaningful in the analytics report.

```
<i:Interaction.Behaviors>
  <mwb:ConsoleAnalytics/>
</i:Interaction.Behaviors>

<StackPanel x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
  <Button x:Name="helloButton" Content="hello">
    <i:Interaction.Triggers>
      <i:EventTrigger EventName="Click">
        <mwb:TrackAction Category="MainPage" Value="2"/>
      </i:EventTrigger>
    </i:Interaction.Triggers>
  </Button>
</StackPanel>
```

Now, when the user taps the *Button*, tracking information will be sent to the Google Analytics service, and the developer will be able to log on to the portal to retrieve the analytics reports. As of this writing, the MSAF is still under development, so although the concepts and design will continue to be valid, the exact steps you need to incorporate MSAF in your application can vary over time. If you cannot get MSAF to work out of the box, you always have the option to download the source code so that you can make any necessary modifications to get it to work in your scenario.

Also note that the ability to track your application's runtime behavior carries with it a responsibility regarding privacy. Specifically, you must be very careful not to collect any personally identifiable information, and most especially no personal user data, medical, financial, or other potentially sensitive information.

Summary

Before you start on a project, it's always a good idea to have an understanding of all the things you have to do before you finally ship the product. This chapter looked at the end-game of bringing your application to market: performance tuning, automated testing, the certification and publication process itself, and options for monetization. The Windows Phone marketplace is becoming increasingly more sophisticated with each update, introducing features such as trial mode and private beta testing, which give you even more options for publishing your application. Simple analytics are available for the data that the marketplace collects on download profiles and crash statistics, and more sophisticated tracking can be incorporated within your code by using third-party services such as Google Analytics.

PART IV

Version 7.5 Enhancements

CHAPTER 15	Multi-Tasking and Fast App Switching	553
CHAPTER 16	Enhanced Phone Services.	589
CHAPTER 17	Enhanced Connectivity Features.	627
CHAPTER 18	Data Support	667
CHAPTER 19	Framework Enhancements.	711
CHAPTER 20	Tooling Enhancements	745

The basic building blocks, application model, and extended phone services can be used by all versions of Windows Phone 7. This part examines the enhancements and additional features that were introduced with version 7.1, including multi-tasking, background agents, enhanced sensor support, camera manipulation, augmented reality, and local database support. It also covers the enhanced tooling support, including the application profiler.

Multi-Tasking and Fast App Switching

The first 14 chapters in this book have focused on the features that are common to both Windows Phone 7 and Windows Phone 7.1/7.5. From this point forward, the focus shifts to the new features and platform enhancements introduced in Windows Phone “Mango.” This encompasses product version 7.5, which consists of OS version 7.1 and the Windows Phone 7.1 SDK. In this chapter, you will start by looking at how the platform supports multi-tasking. There are two broad categories here: fast application switching, and the various multi-tasking features themselves. Fast application switching is more of an infrastructure enhancement, by which users can switch rapidly from one application to another. The new multi-tasking features provide a range of options for an application to divide work into multiple processes.

Fast Application Switching

This is an infrastructure improvement that enhances the application-switching behavior such that an application is less likely to be tombstoned and more likely to be able to resume more quickly. Under the covers, what the application platform actually implements is fast application *resume*. The user perceives this as fast application *switching*, because it allows her to switch rapidly between applications, and it’s the switching that a user cares about, not so much the resuming.

In Windows Phone 7, when the user is running an application and then taps the Start button, one of two things can happen: either the application is deactivated and then tombstoned, or it is merely deactivated and not tombstoned. Typically, the application is tombstoned, but if the user taps Back immediately after tapping Start, there is a chance that the application will avoid being tombstoned and will be reactivated immediately.

In version 7.1, from the user’s perspective, this behavior continues, with the modification that the tombstoning case is less common, and the fast deactivation/reactivation case is more common. The difference between the tombstone scenario and the fast application resume (non-tombstone) scenario is illustrated in Figures 15-1 and 15-2.



Note In version 7.1.1 (announced in February 2012), which specifically targets low-memory devices (that is, devices with no more than 256 MB of physical memory), the tombstoning case is likely to occur more often than in version 7, and is also likely to happen faster. This ensures the overall health of the phone, but at the cost of slower reactivation times.

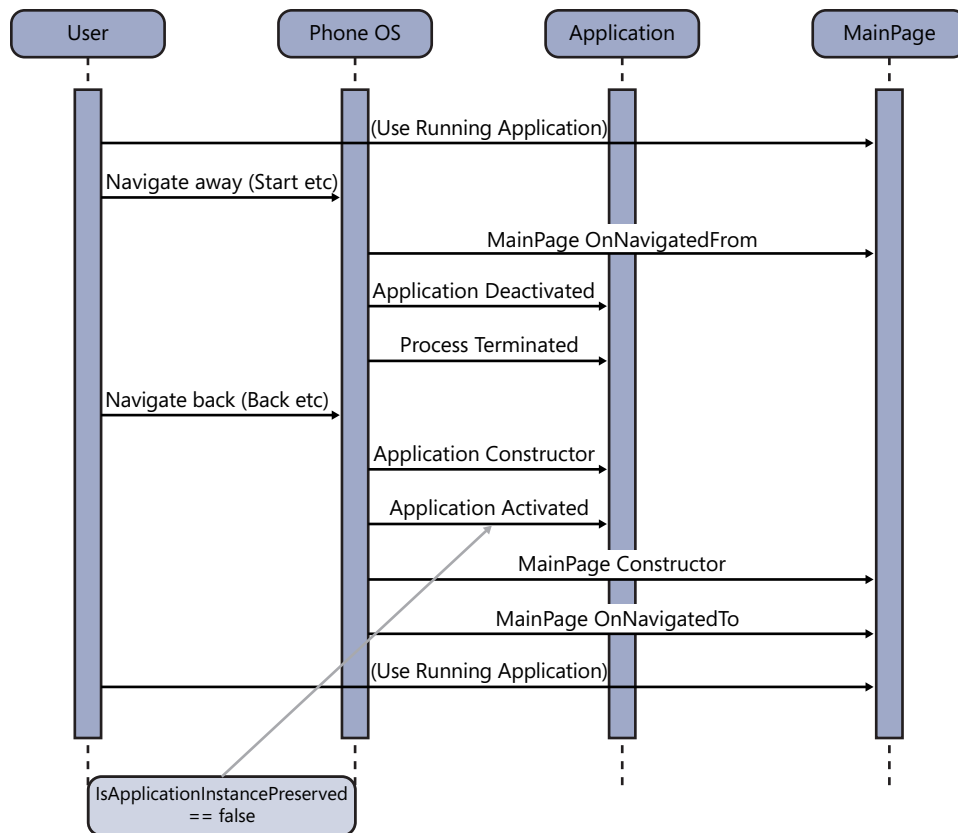


FIGURE 15-1 The tombstone case.

In the fast application resume case, the application process is not terminated; it is deactivated, but it remains in memory, taking up space. This is what allows the system to reactivate and resume the application quickly, if and when the user navigates back to the application.

To make it easier for developers to determine whether their application has been reactivated after dormancy (as opposed to after tombstoning), version 7.1 exposes a new property on the *ActivatedEventArgs* named *IsApplicationInstancePreserved*. If this is set to *true*, then the application doesn't need to restore state from persistent storage because state is still in memory.

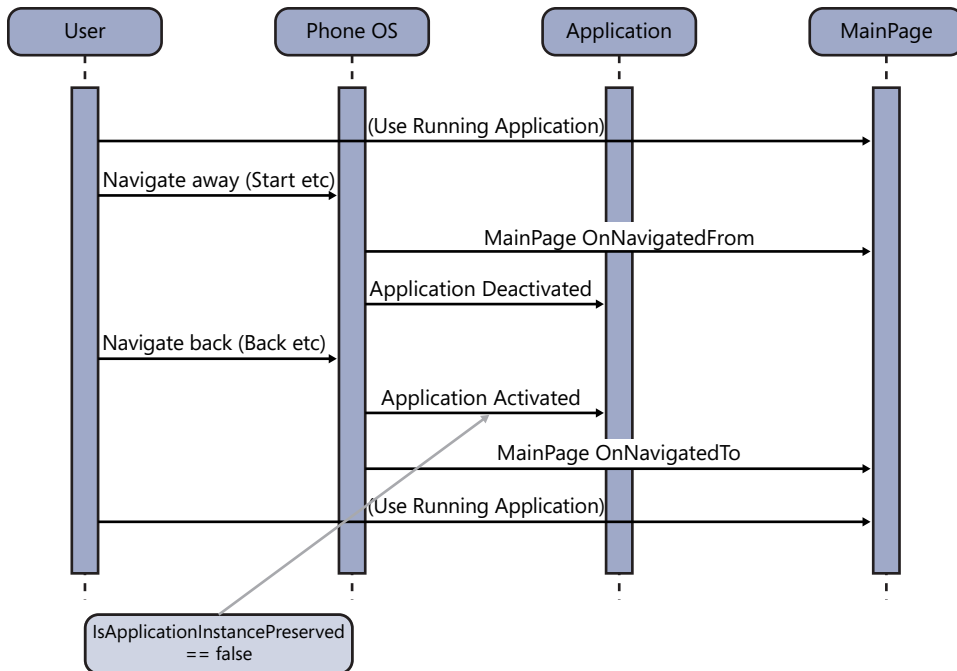


FIGURE 15-2 The fast application resume (non-tombstone) scenario.

As you can see from the sequence diagrams in Figures 15-1 and 15-2, there are two critical differences:

- In the fast application resume scenario, the application is maintained in memory, which means that the various application objects are not destroyed after deactivation, and there is therefore no need to run the *App*, *MainPage*, or other constructors upon subsequent activation. In the tombstone scenario, the application's memory is reclaimed by the system, so constructors must therefore be run again when the user switches back to the application.
- In the fast application resume scenario, the *IsApplicationInstancePreserved* property is *true* on *Application.Activated*, whereas in the tombstone scenario, the *IsApplicationInstancePreserved* property is *false* on *Application.Activated*.

If memory pressure increases to the point where the system needs to reclaim memory from dormant applications, it will first start tombstoning applications from the end of the backstack; that is, the least-recently used application is tombstoned first.

A further consideration is resource management. Figure 15-3 shows the sequence when an application becomes dormant. In this scenario, you don't want it consuming resources, especially hardware resources such as sensors, and most especially resources such as the camera, which can only be used by one application at a time. The standard *OnNavigatedFrom* and *Deactivated* events are the developer's opportunity to relinquish resources. However, if the developer does not proactively release resources, then the framework will do the job for you. When an application is deactivated, its resources are detached, and threads and timers are suspended. The application enters a dormant

state in which it cannot execute code, it cannot consume runtime resources, and it cannot consume any significant battery power. The sole exception to this is memory: the dormant application remains in memory.

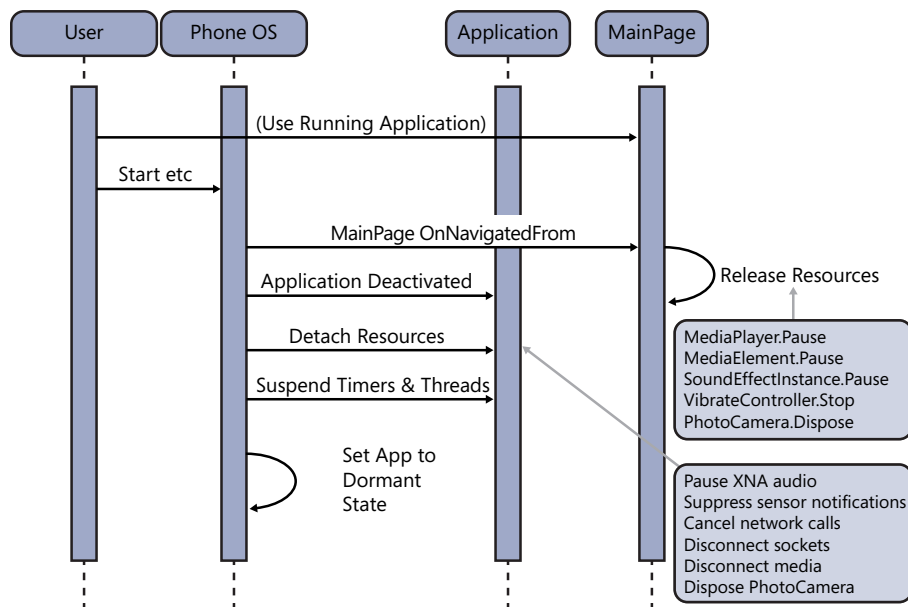


FIGURE 15-3 Bringing an application to the dormant state.

Conversely, when an application is reactivated from the dormant state, the framework resumes timers and threads, and reattaches some (but not all) resources that it previously detached (see Figure 15-4). The developer is responsible for reconnecting/resuming media playback, HTTP requests, sockets, and camera.

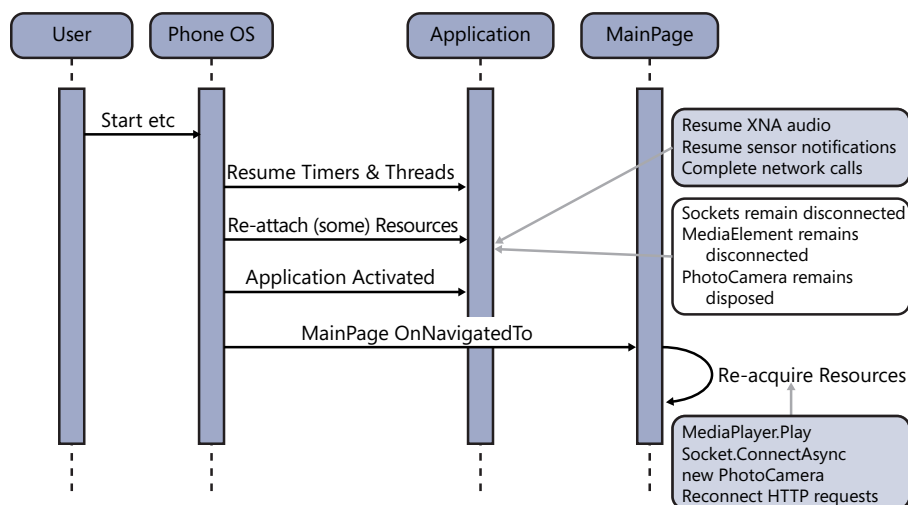


FIGURE 15-4 Resuming an application from the dormant state.

Multi-Tasking

Windows Phone 7 has technically been a multi-tasking system from the start, but it is perceived as a single-tasking system in the sense that there can only be one foreground application running at a time. Compared to a desktop application environment, the resources on the phone are significantly constrained. For this reason, the user experience (UX) is optimized by giving the foreground application a very large proportion of these resources, to the exclusion of other applications. For example, the limited amount of screen real estate on a phone means that it makes sense to run only one application in the foreground at a time. That said, certain other applications, notably system services such as incoming phone calls, SMS messages, or toasts, will continue to run even when an application is running in the foreground, and are allowed to impinge partially on the foreground user interface (UI).

In version 7.1, the following features were added to give you greater flexibility over when your code runs:

- **Generic Background Agents** Application developers can use this to run code in the background when their foreground application is not running.
- **Alarms and Reminders** A simple programmatic way to invoke one-off or periodic alerts that run independently of your main application.
- **Background Transfer Service** You can use this to direct an application to queue up file downloads and uploads in the background.
- **Background Audio** Developers can use this to have an application initiate audio playback and have it continue to run, even when the user navigates away from the application.

These features support two broad user scenarios:

- The scenario in which the user wants to start some application in the foreground, then switch to another application, yet have some features of the initial application continue to run in the background. An example of this is audio: you might start playing audio, and then switch to email, but you want the audio to continue running in the background.
- The scenario in which you set something up in the foreground, but it doesn't actually start running until some future point, at which time it starts running in the background, even though the user might be doing something else at that moment in the foreground. Examples of this are alarms and reminders.

To enable user applications to run code both in the foreground and in the background clearly runs an increased risk of resource contention. In designing the permitted behaviors, considerable thought went into achieving the best possible overall UX. Consider this possible scenario: application A runs in the foreground; at the same time, application B is running multiple resource-intensive operations in the background, to the point where the performance, responsiveness, and UX in application A is noticeably degraded. Application A might be playing by the rules and using resources conservatively, whereas application B is being reckless. This won't help the developer of application A because the user's perception will be that application A is performing poorly. Also consider that the phone is by nature resource-constrained. Compared to a desktop computer, it has significantly less memory,

persistent storage, CPU power, and connectivity. It also cannot rely on constant power. To mitigate these issues and maintain the health of the phone as well as the overall UX, there are a number of constraints in how you can set up and run background tasks. One critical constraint is that certain types of background features are actually prohibited from running while any foreground application is running. Further limitations for each class of background feature are described in the following sections.

Alarms and Reminders

Alarms and reminders are two forms of the same type of UX. In both cases, they trigger a notification to the user according to some schedule defined in your application. The notification is presented to the user in the form of a dialog box that pops up on the screen at a specified time. The dialog box can display some text that you determine and offer buttons with which the user can dismiss the notification or postpone it until later. If the user taps the notification, this launches your application. You can set up alarms and reminders to be either single or recurring events.

There are minor differences between alarms and reminders. For instance, with an alarm, you can specify a sound file to play when the notification is launched. On the other hand, with a reminder, you can specify a page in your application to go to when the user taps the reminder UI. Note that there is a limit of 50 alarms and/or reminders at a time, per application, and that they are accurate only to within 60 seconds.

Alarms

There has been an Alarms application built into the phone from version 7. Version 7.1 adds to that by providing a way for developers to build alarms and reminders programmatically. Under the covers, both custom alarms and the built-in Alarms application use the same scheduling system. From a user's perspective, custom alarms are similar to the built-in Alarms application in that the alarm alert UX is the same, the options to snooze or dismiss are the same, both adhere to user settings, and so on. Here's the basic usage: create an *Alarm* object, set its properties, and then add it to the *ScheduledActionService*, as demonstrated in the following:

```
Alarm alarm = new Alarm("Coffee");
alarm.BeginTime = DateTime.Now.AddMinutes(30);
alarm.Content = "Time for a break.";
ScheduledActionService.Add(alarm);
```

Figure 15-5 shows a simple example, which is an interval training tool (the *IntervalTraining* application in the sample code). The user enters each interval item into a list, and then starts the training session. At the start of the session, you create an *Alarm* for each item that starts at the end of the previous item.

CONTOSO

intervals

item: cool-down mins: 5

mins	item
10	warm-up
20	jog
2	half-pace
1	sprint
2	half-pace
10	jog
5	cool-down

start session

FIGURE 15-5 An interval training application that utilizes alarms.

The code is very simple. First, you declare a class to hold the session item data, including the *Alarm* for each item. Note in the following example that each *Alarm* has a unique name:

```
public class SessionItem
{
    public int Minutes { get; set; }
    public string Name { get; set; }
    public Alarm EndAlarm { get; set; }

    public SessionItem(int minutes, string name)
    {
        Minutes = minutes;
        Name = name;
        EndAlarm = new Alarm(Guid.NewGuid().ToString());
        EndAlarm.Content = name;
    }
}
```

On the main page, you set up a collection of session items in the constructor (this could also be done in the *OnNavigatedTo* override), and then data-bind these as the *ItemsSource* for a *ListBox*. You're also taking this opportunity to clean up any old alarms for this application that haven't already been triggered. Calling *ScheduledActionService.GetActions* will retrieve the scheduled actions of the specific type for this application only; there's no danger that you'd interfere with alarms/reminders for any other application. So far, you're only considering *Alarm* and *Reminder* types, but note that this will also get other scheduled actions for this application, such as periodic or resource-intensive background tasks (discussed later in this chapter).

When the user taps the “+” button, you’ll add a new session item to the list. Finally, when the user taps the Start Session button, you’ll walk through the collection and schedule an *Alarm* for the end of each item. There’s a conditional statement in there that affects the timing of each alarm. This is because, while testing, you can reduce the time of the alarm so that you don’t have to wait so long for it to fire. Note, however, that the alarm scheduling system does not work well with alarms scheduled within a very short time of each other. The finest granularity that is guaranteed is 60 seconds, and you can be pretty sure that alarms within ~30 seconds of each other might not be scheduled in the correct order.

```
public ObservableCollection<SessionItem> SessionItems { get; set; }

public MainPage()
{
    InitializeComponent();
    SessionItems = new ObservableCollection<SessionItem>();
    SessionItemsList.ItemsSource = SessionItems;

    IEnumerable<Alarm> oldAlarms = ScheduledActionService.GetActions<Alarm>();
    foreach (Alarm alarm in oldAlarms)
    {
        if (alarm.ExpirationTime <= DateTime.Now)
        {
            ScheduledActionService.Remove(alarm.Name);
        }
    }
}

private void AddButton_Click(object sender, RoutedEventArgs e)
{
    SessionItems.Add(new SessionItem(Int32.Parse(ItemMinutes.Text), ItemName.Text));
    ItemMinutes.Text = String.Empty;
    ItemName.Text = String.Empty;
}

private void StartButton_Click(object sender, RoutedEventArgs e)
{
    DateTime startTime = DateTime.Now;
    foreach (SessionItem item in SessionItems)
    {
        #if DEBUG
            startTime = item.EndAlarm.BeginTime = startTime.AddSeconds(item.Minutes * 10);
        #else
            startTime = item.EndAlarm.BeginTime = startTime.AddMinutes(item.Minutes);
        #endif
        ScheduledActionService.Add(item.EndAlarm);
    }
}
```

Reminders

Reminders are also very similar to the standard calendar reminders that are built into the phone: they behave like standard reminders, and will stack with all the other reminders in the system from the user's calendar. They are slightly richer than alarms, with more scope for providing additional data for the reminder. Standard calendar reminders provide for a navigation target; that is, when the user clicks on a reminder, it takes her to the corresponding item in the calendar. With custom reminders, you have the same behavior: you can specify to navigate to a page in your application when the user responds to the reminder. Figure 15-6 shows an example (the *TrailReminders* application in the sample code). This is a trail application that you can use to set a reminder for when you're planning to hike a given trail.

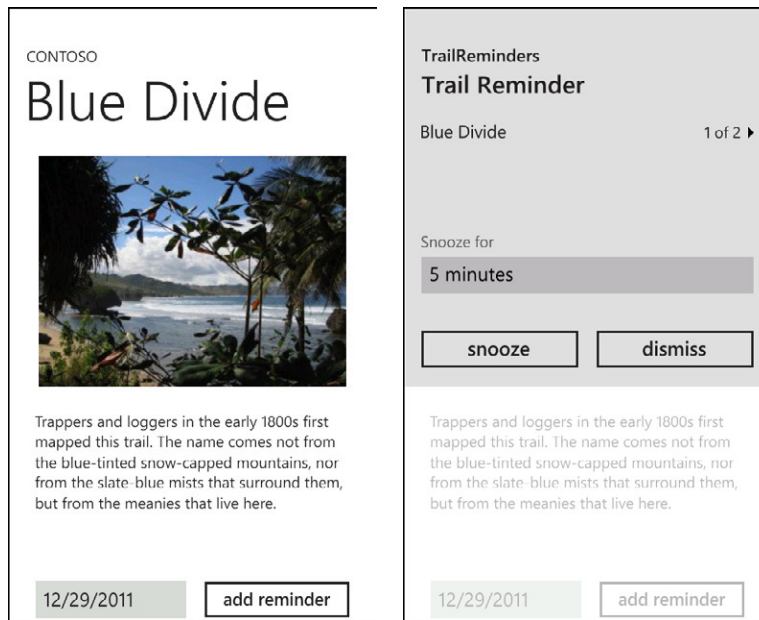


FIGURE 15-6 Using the *TrailReminders* application, the user can add a reminder for a selected trail, which later triggers a reminder alert.

To represent the trail data, there is a simple *Trail* class, as shown in the following:

```
public class Trail
{
    public string Title { get; set; }
    public string Photo { get; set; }
    public string Description { get; set; }

    public Trail(string title, string photo, string description)
    {
        Title = title;
        Photo = photo;
        Description = description;
    }
}
```

The *App* class holds the data (as a simple viewmodel) with a new method, *InitializeData*, which is called at the end of the constructor. In this method, you set up a few trail data items. For debugging, clear out any old reminders. Realistically, you might want to keep an independent list of the reminders for this application within the application itself, which you could then manage appropriately (removing expired reminders, providing a list to the user of all remaining reminders, and so on).

```
public static ObservableCollection<Trail> Trails { get; set; }

private void InitializeData()
{
    Trails = new ObservableCollection<Trail>();

    Trails.Add(new Trail("Frog Ridge", "images/FrogRidge.jpg", "This is a 16-mile roundtrip hike, set deep in the North Cascades forests. It boasts a wide range of wildlife, including black and grizzly bears, mountain lions, elk, deer and marmots - but no frogs."));
    Trails.Add(new Trail("Frost Creek", "images/FrostCreek.jpg", "Frost Creek is almost never free of snow and ice, as it is at 7000ft elevation in the Olympics, on the east-facing side, so it gets very little sun, and only for half an hour or so, once a year."));
    Trails.Add(new Trail("Blue Divide", "images/BlueDivide.jpg", "Trappers and loggers in the early 1800s first mapped this trail. The name comes not from the blue-tinted snow-capped mountains, nor from the slate-blue mists that surround them, but from the meanies that live here."));

    #if DEBUG
        IEnumerable<Reminder> oldReminders = ScheduledActionService.GetActions<Reminder>();
        foreach (Reminder r in oldReminders)
        {
            ScheduledActionService.Remove(r.Name);
        }
    #endif
}
```

The main page offers a *ListBox* populated with trail *Titles* via data binding, and implements the *SelectionChanged* handler to navigate to the corresponding individual trail page.

```
public MainPage()
{
    InitializeComponent();
    TrailList.ItemsSource = App.Trails;
}

private void TrailList_SelectionChanged(object sender, SelectionChangedEventArgs e)
{
    if (TrailList.SelectedIndex == -1)
    {
        return;
    }
    String navigationString =
        String.Format("/TrailPage.xaml?Title={0}", ((Trail)TrailList.SelectedItem).Title);
    NavigationService.Navigate(new Uri(navigationString, UriKind.Relative));
}
```

The only interesting work is done in the *TrailPage*. In the XAML, you must data-bind UI elements to the properties on the *Trail* class. Take notice of the *DatePicker* from the Silverlight Toolkit; this is how you allow the user to pick a date for his reminder.

```
<StackPanel x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <Image Source="{Binding Photo}" Height="300" Width="410"/>
    <Grid Height="30"/>
    <TextBlock
        Text="{Binding Description}" Height="210" Width="410"
        Style="{StaticResource PhoneTextNormalStyle}" TextWrapping="Wrap"/>
    <StackPanel Orientation="Horizontal">
        <toolkit:DatePicker x:Name="ReminderDate" Width="220" Margin="12,0,0,0"/>
        <Button
            x:Name="AddButton" Content="add reminder"
            Height="70" Width="212" Click="AddButton_Click"/>
    </StackPanel>
</StackPanel>
```

In the code-behind, override *OnNavigatedTo* to extract the trail *Title* from the navigation *QueryString*. You use this to find the corresponding *Trail* data from the viewmodel collection, and set it as the *DataContext* for the page:

```
private string thisTitle;
private Trail trail;
private string thisPageUri;

protected override void OnNavigatedTo(NavigationEventArgs e)
{
    thisTitle = NavigationContext.QueryString["Title"];
    thisPageUri = e.Uri.ToString();

    foreach (Trail t in App.Trails)
    {
        if (t.Title == thisTitle)
        {
            DataContext = trail = t;
            break;
        }
    }
}
```

Finally, implement the *Add Reminder Button* to create a new *Reminder* using data from the *Trail* object, and the user's selected *DateTime* from the *DatePicker*.

```
private void AddButton_Click(object sender, RoutedEventArgs e)
{
    Reminder reminder = new Reminder(Guid.NewGuid().ToString());
    reminder.BeginTime = (DateTime)ReminderDate.Value;
    reminder.Content = thisTitle;
    reminder.Title = "Trail Reminder";
    reminder.RecurrenceType = RecurrenceInterval.None;
    reminder.NavigationUri = new Uri(thisPageUri, UriKind.Relative);
    ScheduledActionService.Add(reminder);
}
```

Later, when the reminder is triggered, the user can tap it to navigate directly to the corresponding trail page, regardless of which application is running at the time. There are two caveats to this: first, if the phone is locked, nothing happens; second, if the application that owns the reminder is currently the active application, nothing happens.

Observe the seamless way in which custom alarms and reminders are integrated with built-in alarms and calendar reminders. They even survive reboots, just like the built-in versions.

Background Transfer Service

The standard approach to downloading files to (and uploading files from) the phone has been to use *WebClient* or *HttpWebRequest*, or custom web service proxies. You can carry on using these in version 7.1 projects, but you now have the additional option of using the Background Transfer Service. This is a very simple API to use: you make a *BackgroundTransferRequest*, specifying the source and destination URLs, and then add that request to the *BackgroundTransferService* queue. You can also subscribe to progress events on the operation. Then, the service takes over and executes your request asynchronously, continuing on to completion, even if your application terminates—and even in the face of reboots. There is also a built-in retry mechanism. In addition, you have the ability to stop and restart requests or delete them from the queue.

Behind the scenes, the *BackgroundTransferService* uses the same infrastructure as that used by the Xbox Live and Zune marketplace, and therefore, applies the same constraints; that is, file downloads are limited to 20 MB maximum for any one request over the cell network. You *can* transfer bigger files, but the request will be queued until the phone switches from cell network to WiFi. File uploads are limited to 5 MB.

Figure 15-7 illustrates a simple example (the *BackgroundTransferDemo* application in the sample code), wherein the UI has two *Button* controls: a *ProgressBar* and a *MediaElement*. The scenario is that the user taps the download button to initiate a background download of an arbitrary video file. While this is ongoing, you report progress in the *ProgressBar*. Then, when the download is complete, you enable the Play button so that the user can play the newly downloaded video. After initiating the download, the user is free to navigate away from the application, if she so wishes; the download will continue to run in the background.

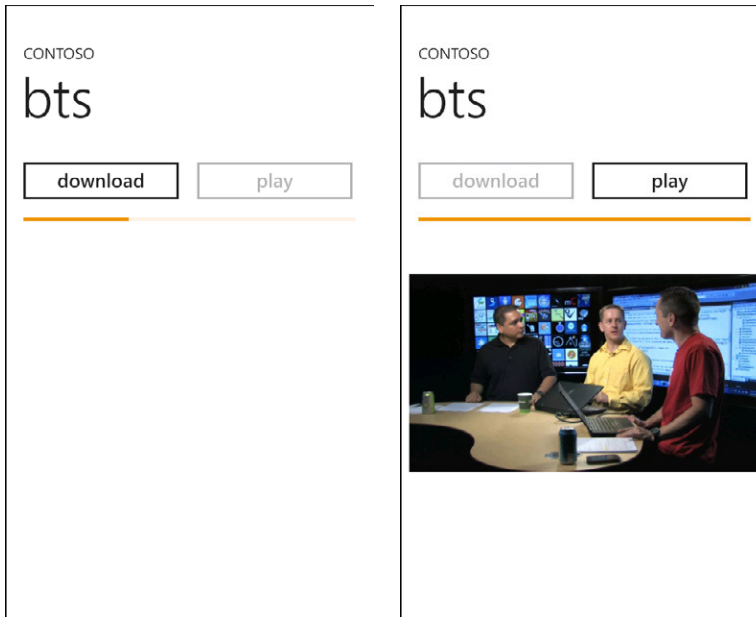


FIGURE 15-7 Starting a background transfer (on the left) and completing it (right).

In the code, you first set up some fields for the *BackgroundTransferRequest* and the target *IsolatedStorageFile*, and initialize the storage file in the page constructor. When you download a file, it must be stored in the application's isolated storage, and it must be in the shared/transfers folder (or a subfolder, thereof).

```
private static readonly Uri remoteFileUri =
    new Uri(
        @"http://media.ch9.ms/ch9/4b58/de9f7501-2a51-4875-8b2f-9f4d014b4b58/
IWP22PtorrBkgndAgnt_ch9.wmv",
        UriKind.Absolute);
private static readonly Uri localFileUri =
    new Uri("/shared/transfers/movie.wmv", UriKind.Relative);
private IsolatedStorageFile isf;
private BackgroundTransferRequest btr;

public MainPage()
{
    InitializeComponent();
    isf = IsolatedStorageFile.GetUserStoreForApplication();
}
```

When the user taps the *Download Button*, you create a new *BackgroundTransferRequest* and hook up the progress and status events. You'll get progress events while the transfer is ongoing, and status events when the status changes (for example, from ongoing transfer to completed, or error). The application goes a bit further and disables the download button to prevent the user from starting the transfer again while it is already in progress.

```
private void downloadButton_Click(object sender, RoutedEventArgs e)
{
    btr = new BackgroundTransferRequest(remoteFileUri, localFileUri);
    BackgroundTransferService.Add(btr);
    btr.TransferProgressChanged += btr_TransferProgressChanged;
    btr.TransferStatusChanged += btr_TransferStatusChanged;
    downloadButton.IsEnabled = false;
}
```

In the *TransferProgressChanged* event, you should ensure that the *ProgressBar* is updated to match its maximum value to the total file size, and its current value to the number of bytes received at that point.

```
private void btr_TransferProgressChanged(object sender, BackgroundTransferEventArgs e)
{
    Dispatcher.BeginInvoke(() =>
    {
        progressBar.Maximum = btr.TotalBytesToReceive;
        progressBar.Value = btr.BytesReceived;
    });
}
```

In the *TransferStatusChanged* event, check to see if the new status is *Completed*; if it is, open the downloaded file and attach it to the *MediaElement*. At the same time, ensure that the *ProgressBar* value is set to the number of bytes received on the download. This is necessary because if the user switches away from the application and then back again, if the download has completed by the time she switches back, you won't get any more transfer events (for which you would normally update the *ProgressBar*). For this example, you're going to ignore all other status events. In a more sophisticated application, you would probably handle other cases, as well.

```
private void btr_TransferStatusChanged(object sender, BackgroundTransferEventArgs e)
{
    if (btr.TransferStatus == TransferStatus.Completed)
    {
        UpdateUi();
    }
}

private void UpdateUi()
{
    Dispatcher.BeginInvoke(() =>
    {
        progressBar.Value = btr.BytesReceived;
        try
        {
            using (IsolatedStorageFileStream file = isf.OpenFile(
```



```

        btr.DownloadLocation.ToString(), FileMode.Open, FileAccess.Read))
    {
        mediaElement.SetSource(file);
    }
}
catch (Exception ex)
{
    MessageBox.Show(ex.ToString());
}
});
}

```

When you attach the file to the *MediaElement*, this triggers the *MediaOpened* event. When this event fires, you enable the *Play Button* and disable the *Download Button*. The user can then tap the *Play Button* to start the video. Observe how the *MediaOpened* event is hooked up in the XAML.

```

private void mediaElement_MediaOpened(object sender, RoutedEventArgs e)
{
    playButton.IsEnabled = true;
    downloadButton.IsEnabled = false;
}

private void playButton_Click(object sender, RoutedEventArgs e)
{
    mediaElement.Play();
}

```

Finally, you need to override *OnNavigatedTo*: check to see if you're coming back to the application with an existing transfer request. If so, this indicates that the user must have started the request and then navigated forward away from the application. In this scenario, you hook up the transfer status event handlers again, check to see if the transfer has already completed, and then update the UI if it has. You want to also set the *IsEnabled* state of the Download button according to whether there is a transfer in progress.

```

protected override void OnNavigatedTo(NavigationEventArgs e)
{
    btr = BackgroundTransferService.Requests.FirstOrDefault();
    if (btr != null)
    {
        downloadButton.IsEnabled = false;
        btr.TransferProgressChanged += btr_TransferProgressChanged;
        btr.TransferStatusChanged += btr_TransferStatusChanged;
        if (btr.TransferStatus == TransferStatus.Completed)
        {
            UpdateUi();
        }
    }
    else
    {
        downloadButton.IsEnabled = true;
    }
}

```

Background transfers add to the existing techniques for asynchronous web requests by providing for requests that can continue beyond the lifetime of the application that initiated them, and even beyond a phone reboot.

Generic Background Agents

Alarms, reminders, background transfers, and background audio are all specific types of background agent. There is also a class of agent known as generic background agents (GBAs). These are not limited to one specific set of functionality; rather, you can implement them to perform an open-ended range of functionality, per your specific domain requirements. Windows Phone 7.1 supports two types of GBA:

- **Periodic** Represented by the *PeriodicTask* class, this type of agent runs for a short time on a regular recurring interval. It is only allowed 25 seconds to run, roughly every 30 minutes (the exact timing is unpredictable, because the system will attempt to align the execution of periodic agents with other system activities; the interval will be in the range 20–40 minutes). There is a hard limit on the number of periodic agents that can be scheduled on the phone. This varies per device, but it can be as low as 6 and no more than 18.
- **Resource-intensive** Represented by the *ResourceIntensiveTask* class, this type of agent runs for a relatively long period of time and is only executed when a specific set of conditions are satisfied. A *ResourceIntensiveTask* can run for a maximum of 10 minutes at a time but only when all the following conditions are met (keep in mind that it is quite possible that these conditions will never be met on a given phone, so you must allow for the possibility that your agent never in fact has the opportunity to run):
 - The phone is on external power.
 - The phone has network connectivity over WiFi or through a PC.
 - Battery power is >90 percent.
 - The device screen must be locked.
 - There is no active phone call.

Here's how the platform schedules resource-intensive tasks: the scheduler runs them in a round-robin manner at 10 minutes apiece until the task calls *NotifyComplete*. So, suppose that you have 25 minutes' worth of work to do. At the appropriate time, the platform starts the task, and then terminates it after 10 minutes. The platform can then run other scheduled tasks and will eventually return to the task that was terminated. The platform then runs that task again. When another 10 minutes elapses, the task is again terminated so that the platform can run other scheduled tasks. Finally, the first task runs again, and after 5 more minutes of work, the task completes and calls *NotifyComplete*. At this point, the platform will no longer schedule this task until the original conditions are met again. For such a long-running task, it is clearly important that it is resilient to being terminated, and that it

performs checkpoints on the progress of work completed so that it can pick up and carry on the next time it is invoked.

Under the covers, the application platform treats both periodic and resource-intensive tasks as the same kind of background agent. The only distinction internally is that each task happens to follow a different schedule and have different resource constraints. The task type implicitly defines the schedule and resource set.

A GBA has a default expiry of 14 days, which also happens to be its maximum. That is, you cannot create a GBA with an expiration date beyond 14 days. However, you can renew the agent any time your application runs in the foreground. So, if the user keeps running your application, you can keep renewing your background agents indefinitely. Conversely, if the user doesn't run your application for a while, then they're probably also not interested in the application's background agents—hence, the limited expiry.

There are also memory limitations: a GBA cannot take up more than 6 MB of memory. Furthermore, there are restrictions as to the types of operation your background agent can perform. These restrictions are summarized in Table 15-1.

TABLE 15-1 Permitted and Prohibited Operations in GBAs

Permitted Operations	Prohibited Operations
Create and show Tiles and Toast	Display arbitrary UI
Use location functionality	Use of the XNA libraries
Access the network	Use sensors, including microphone and camera
Read/write isolated storage	Play audio (outside the <i>BackgroundAudioPlayer</i>)
Use sockets	Schedule alarms, reminders, or background transfers
Use most framework APIs	

Finally, version 7.1 also includes new user settings that inform the user as to which applications have registered background agents. The user can turn off/on the background agents for an application, individually.

To make use of GBAs in your application, you produce an XAP file that contains your main application assembly (or assemblies) and an additional assembly for your background agent. Dividing your code in this manner allows the phone system services to launch your background agent in a separate process, independent of your main application. This allows your background agents to run in the background when your main application is not running. An application is allowed to have only one background agent; this can be a *PeriodicTask*, a *ResourceIntensiveTask*, or both. The two parts of your application are associated via the main application's manifest, which defines the background agent(s), including the assembly and entrypoint type for your agent.

In addition, the application and agent can optionally be connected in two other ways:

- Your agent can create a *ShellToast* object and show it. At this point, the user can click the toast, which will navigate to the page in your main application that you specified when you created it.
- Both the application and agent have access to the same application isolated storage; therefore, they can share files.

The relationships are summarized in Figure 15-8.

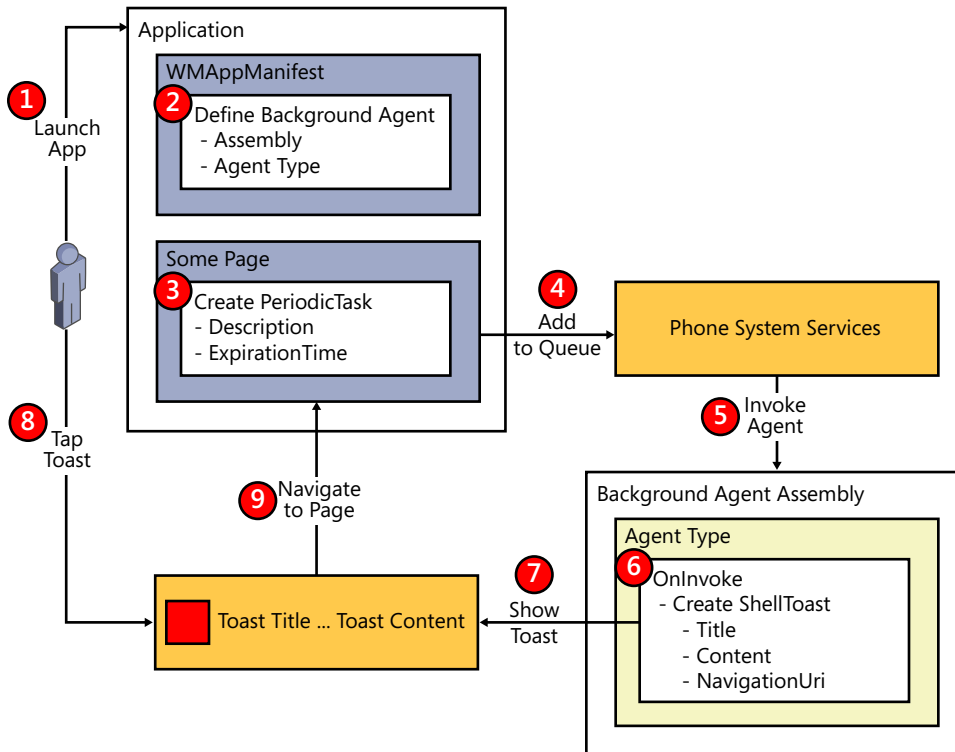


FIGURE 15-8 The relationships between background agent components.

The Windows Phone 7.1 SDK includes Microsoft Visual Studio template support for adding GBAs to your application. Figure 15-9 presents a simple “geo-fencing” example (the *BackgroundAgentDemo* solution in the sample code). The main application offers Start and Stop *Button* controls to start and stop its background agent. The middle of the screen is taken up with a *ListBox* that will be populated with time-stamped location information. This data will come from the background agent. Every time it wakes up on schedule, it will find the current location and add the information to a collection, persisted to isolated storage, and then stop. Each time the user enters the main application, it will fetch the location data from isolated storage, and then data-bind it to the *ListBox*.

The list of locations is not updated in real time in the UI, because this kind of synchronization is difficult to get right. Figure 15-9 (right) depicts the situation when the main application is not running. In this case, as there is no application in the foreground, the platform allows up to six background agents to run. In this example, the background agent pops a toast message with each new set of location data. The user can tap this, and it will launch the main application, which in turn will fetch the latest list of location data.

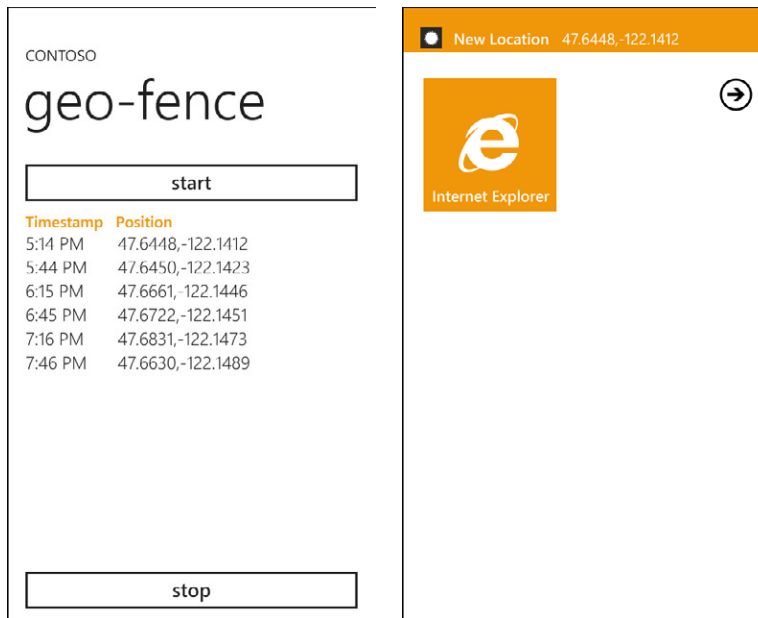


FIGURE 15-9 A foreground application (on the left) that controls a periodic background agent, and toast from the background agent (right).



Note If you want the UI to be updated by the background agent in real time while the foreground application is running, then you'd need to implement some kind of polling mechanism. For example, the foreground application could check every few seconds (or minutes, depending on the nature of the application) to fetch updated data from the file. Polling is always a difficult technique to get right: poll too frequently, and you're wasting time and resources (which is especially critical on a mobile device); poll too infrequently, and your UX suffers because the user is often looking at stale data.

In the example code that follows, the data is represented by a custom *PositionLite* class, which exposes *Timestamp* and *Location* properties. This class is defined in an independent class library, which is shared by both the main application and the agent. You'll be fetching the raw data by using a *GeoCoordinateWatcher*. This fetches data in the form of *GeoPosition<GeoCoordinate>* objects, which contain more information than you need. You want to extract only the specific data items that you need (latitude, longitude, and timestamp) and use them to initialize a *PositionLite* for each set. From the *PositionLite* objects, you'll be using the *Timestamp* and *Location* properties for data binding. However, you will serialize (and deserialize) only the raw *Latitude*, *Longitude*, and *Timestamp* values.

```

public class PositionLite
{
    public double Latitude;
    public double Longitude;

    private DateTime timestamp;
    public DateTime Timestamp
    {
        get { return timestamp; }
        set { timestamp = value; }
    }

    [XmlIgnore]
    public String Location
    {
        get
        {
            return String.Format("{0:N4},{1:N4}", Latitude, Longitude);
        }
    }
}

```

In the *MainPage* code-behind, initialize an *ObservableCollection* of these objects, fetching it from isolated storage in the *OnNavigatedTo* override. Of course, the first time the application is run, there will be no previous storage. At some point after that first run, the data will have been written out by the background agent.

```

public ObservableCollection<Utilities.PositionLite> Positions;
private string storageFile = "positions.xml";
private string agentId = "GeoAgent";

protected override void OnNavigatedTo(NavigationEventArgs e)
{
    if (Positions == null)
    {
        Positions = Utilities.StorageHelper.ReadFromStorage<
            ObservableCollection<Utilities.PositionLite>>(storageFile);
    }
    if (Positions == null)
    {
        Positions = new ObservableCollection<Utilities.PositionLite>();
    }
    PositionList.ItemsSource = Positions;
}

```

This code uses a helper library for reading and writing isolated storage. This same helper is used by both the main application and the background agent. The helper is genericized so that it can read and write objects of any serializable type. It is important to note that it also uses a named *Mutex* in order to ensure exclusive access to the file. This is required because both the foreground application and the background agent could attempt to read or write the same file at the same time.

```

public class StorageHelper
{
    private static Mutex StorageMutex = new Mutex(false, "StorageMutex");

    public static void SaveToStorage<T>(T data, string storageFile)
    {
        try
        {
            StorageMutex.WaitOne();
            using (IsolatedStorageFile isoFile =
                IsolatedStorageFile.GetUserStoreForApplication())
            {
                using (IsolatedStorageFileStream isoStream =
                    isoFile.OpenFile(storageFile, FileMode.OpenOrCreate))
                {
                    XmlSerializer xs = new XmlSerializer(typeof(T));
                    xs.Serialize(isoStream, data);
                }
            }
        }
        finally
        {
            StorageMutex.ReleaseMutex();
        }
    }

    public static T ReadFromStorage<T>(string storageFile)
    {
        T data = default(T);

        try
        {
            StorageMutex.WaitOne();
            using (IsolatedStorageFile isoFile =
                IsolatedStorageFile.GetUserStoreForApplication())
            {
                if (isoFile.FileExists(storageFile))
                {
                    using (IsolatedStorageFileStream isoStream =
                        isoFile.OpenFile(storageFile, FileMode.Open))
                    {
                        XmlSerializer xs = new XmlSerializer(typeof(T));
                        data = (T)xs.Deserialize(isoStream);
                    }
                }
            }
        }
        finally
        {
            StorageMutex.ReleaseMutex();
        }
        return data;
    }
}

```

The Start button creates a new *PeriodicTask* and adds it to the *ScheduledActionService*. You should use a suitable name, a suitable description, and (optionally) a time after which you no longer want the agent to be scheduled. It is generally considered best practice to set the *ExpirationTime* in those scenarios when you know the useful life of the agent. The description is required for periodic agents: this is the string that the user will see in the background services Settings page on the phone. The Stop button removes the task from the service. Notice that this code wisely puts the calls to *Add* and *Remove* in *try/catch* blocks: this protects you in the event that the user has disabled background agents for this application.

Also note the call to *LaunchForTest*: this is a test-only method. The idea is that you can cause the agent to be invoked faster and/or more frequently than it would be normally, just for testing purposes. This method must not be used in your published version; it will fail certification. In this example, you cause the agent to be invoked for the first time only 10 seconds after you set it up.

```
private void StartButton_Click(object sender, RoutedEventArgs e)
{
    PeriodicTask task = new PeriodicTask(agentId);
    task.Description = "Timestamped position data";
    task.ExpirationTime = DateTime.Now.AddDays(1);
    if (ScheduledActionService.Find(agentId) != null)
    {
        ScheduledActionService.Remove(agentId);
    }
    try
    {
        ScheduledActionService.Add(task);
    }
    #if DEBUG
        ScheduledActionService.LaunchForTest(agentId, TimeSpan.FromSeconds(10));
    #endif
    catch (InvalidOperationException ex)
    {
        if (ex.Message.Contains("BNS Error: The action is disabled"))
        {
            MessageBox.Show(
                "The user has disabled background agents for this application.");
        }
    }
}

private void StopButton_Click(object sender, RoutedEventArgs e)
{
    if (ScheduledActionService.Find(agentId) != null)
    {
        ScheduledActionService.Remove(agentId);
    }
}
```

The display makes use of a custom type converter, which is used during data-binding to convert the *Timestamp* value (which is a *DateTime*) to a simple string.


```

public class DateTimeConverter : IValueConverter
{
    public object Convert(
        object value, Type targetType, object parameter, CultureInfo culture)
    {
        DateTime dt = (DateTime)value;
        return dt.ToShortTimeString();
    }

    public object ConvertBack(
        object value, Type targetType, object parameter, CultureInfo culture)
    {
        throw new NotImplementedException();
    }
}

```

So much for the main application; now for the background agent. In Visual Studio, right-click the solution to add a new project. Choose a Windows Phone Scheduled Task Agent project, as shown in Figure 15-10.

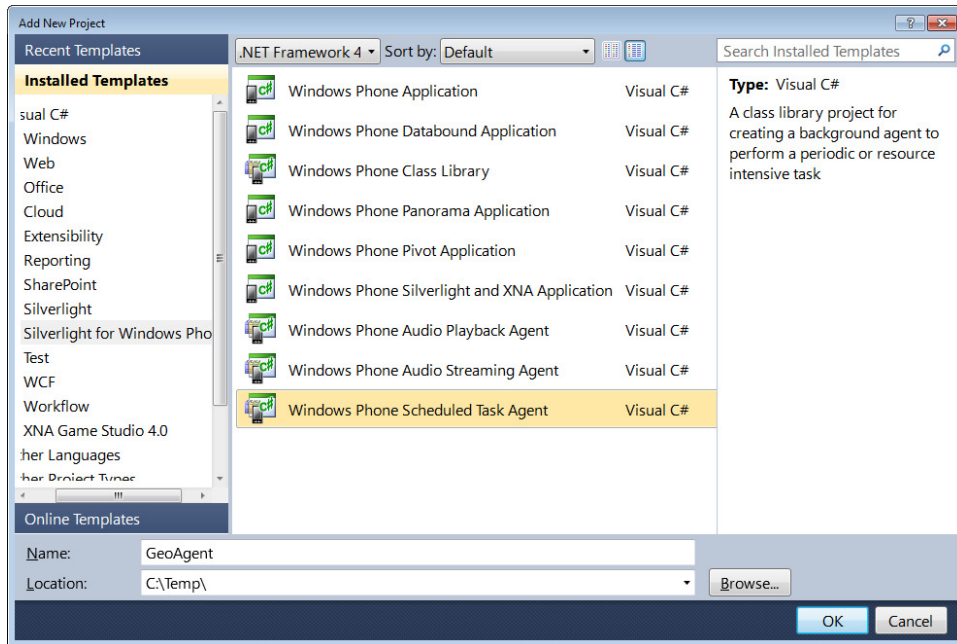


FIGURE 15-10 Adding a scheduled task agent project.

This will generate a class library project with a starter class derived from *ScheduledTaskAgent*. The constructor for this class hooks up an *UnhandledException* handler—the same behavior that you get in the standard *App* class for a phone application. Apart from that, the class has just one other method: a skeleton override of the base class *OnInvoke* method.

In this application, you implement the *OnInvoke* to start a *GeoCoordinateWatcher* for just long enough to gather location information. The agent uses an *ObservableCollection* of *PositionLite* objects, mirroring the data model in the main application. The agent will gather the data and save it to isolated storage. Clearly, both the agent and the main application are part of a single installation, and share the same isolated storage. As before, you have a debug-only call to *LaunchForTest*, which ensures that the agent is invoked again 10 seconds after the current invocation.

Notice the call to *NotifyComplete*: it is the developer's responsibility to do all work in the agent within 25 seconds and to notify the system when its work is complete. If you don't do this, the system will terminate the agent after 25 seconds, anyway. Furthermore, if it has to cancel your agent very often, the system might decide that your agent is badly behaved and might choose not to schedule it again. You must stop the *GeoCoordinateWatcher* as soon as you've obtained the information that you need. Be aware that when used in a background agent, the *GeoCoordinateWatcher* uses a cached location value instead of real-time data. This cache is primed right before the first agent runs, and then is updated every 10 minutes if agents are still running. Also keep in mind that when you add an agent to the schedule service, it will be invoked subsequently, according to the schedule or idle conditions, depending on the type of agent.

```
private GeoCoordinateWatcher coordWatcher;
public ObservableCollection<Utilities.PositionLite> Positions;
private string storageFile = "positions.xml";

protected override void OnInvoke(ScheduledTask task)
{
    if (Positions == null)
    {
        Positions = Utilities.StorageHelper.ReadFromStorage<
            ObservableCollection<Utilities.PositionLite>>(
                storageFile);
    }
    if (Positions == null)
    {
        Positions =
            new ObservableCollection<Utilities.PositionLite>();
    }
    if (coordWatcher == null)
    {
        coordWatcher = new GeoCoordinateWatcher();
    }

    coordWatcher.Start();
    GeoPosition<GeoCoordinate> pos = coordWatcher.Position;
    Utilities.PositionLite lastPosition = new Utilities.PositionLite
    {
        Latitude = pos.Location.Latitude,
        Longitude = pos.Location.Longitude,
        Timestamp = pos.Timestamp.DateTime
    };
    coordWatcher.Stop();
    Positions.Add(lastPosition);
}
```

```

        Utilities.StorageHelper.SaveToStorage<
            ObservableCollection<Utilities.PositionLite>>(Positions, storageFile);
    #if DEBUG
        ScheduledActionService.LaunchForTest(task.Name, TimeSpan.FromSeconds(10));
    #endif
    NotifyComplete();
}

```

The same starter code is used for either a *PeriodicTask* or a *ResourceIntensiveTask*. If you want both types of agent, you don't add multiple agent projects to your application; instead, you use the single class for both types. You can use the *ScheduledTask* parameter passed into the *OnInvoke* method and check its type to determine which type of agent is being invoked. Your code then takes the appropriate path.

If you want to notify the user when your background agent runs, and perhaps allow the user to link back easily to your main application, you can provide a toast at the end of the *OnInvoke*.

```

protected override void OnInvoke(ScheduledTask task)
{
    ... previously-listed code omitted for brevity.

    if (lastPosition != null)
    {
        ShellToast toast = new ShellToast();
        toast.Title = "New Location";
        toast.Content = lastPosition.Location;
        toast.NavigationUri = new Uri("/MainPage.xaml", UriKind.Relative);
        toast.Show();
    }

    NotifyComplete();
}

```

When you do add the agent project to your solution, the *WMAppManifest* is updated with the agent attributes. The key attributes are the *Source* assembly and the *Type* of the agent. Of course, if you change the name and/or namespace of your agent class, or the assembly name, then everything will break until you manually update this file.

```

<ExtendedTask Name="BackgroundTask">
    <BackgroundServiceAgent Specifier="ScheduledTaskAgent" Name="GeoAgent" Source="GeoAgent"
    Type="GeoAgent.ScheduledAgent" />
</ExtendedTask>

```

The agent assembly must be added as a reference in the main application project. This ensures that the two assemblies are packaged together in one installation XAP. It would be nice if Visual Studio were to add this reference at the point where you add the agent project to the solution, but this is not done. You should not actually use the agent assembly in your main application.



Note The Visual Studio debugger provides additional support for background agents. When you debug a solution which contains a main application and a background agent, Visual Studio allows you to step seamlessly between the two projects. You can also show the “Debug Location” toolbar which indicates whether you’re in the main application or the background agent.

Background Audio

As with GBAs, you can set up an agent to play audio in the background. You would typically start the audio playing from your foreground application, and then have the audio continue playing, even after the user navigates away from your main application. Background audio agents share some similarities with GBAs: you create a background audio agent in much the same way, and the architecture is very similar. In both cases, you build a main phone application, with a UI, and then add a background agent project to the solution so that it is referenced in the *WMAppManifest*. In the case of a GBA, your agent class is derived from *ScheduledTaskAgent*, but in the case of a background audio agent, it's derived from *AudioPlayerAgent*.

All media on Windows Phone is actually played through the Zune media queue (ZMQ). However, your application does not interact directly with the ZMQ; instead, it uses the *BackgroundAudioPlayer* class, which acts as a kind of proxy to the ZMQ. Typically, your main application would set up and maintain the playlist of tracks (including, optionally, both files in isolated storage as well as files at remote URLs) and save this playlist to isolated storage. Then you would provide suitable UI with which the user can play, pause, skip, fast-forward, rewind, and so on. You would implement the handlers for these UI elements to invoke the corresponding methods (Play, Pause, and so forth) on the *BackgroundAudioPlayer*.

However, this does not play the audio directly; rather, the *BackgroundAudioPlayer* negotiates with your background audio agent. Your agent fetches the playlist and confirms the action to be taken. The *BackgroundAudioPlayer* works with the ZMQ to actually play the audio tracks. The *BackgroundAudioPlayer* also feeds state change events back to both the main application (if it is running) and to the agent.

Your application does not directly launch the agent. This is done implicitly by the *BackgroundAudio Player* when your application makes calls to *Play*, *Pause*, and so on. The *BackgroundAudioPlayer* launches the correct agent, based on its association with your application defined in the application manifest.

If you start audio playing in your main application, and the user then navigates away, the audio will continue, under the control of your agent. While your application is not in the foreground (or even if it is), the user can use the Universal Volume Control (UVC) to control both the volume and the tracks. On a physical device, the UVC is invoked when the user taps the hardware volume controls. On the emulator, you can drop down the UVC by pressing F10. Figure 15-11 illustrates the relationships between these components. Note that a background audio agent is capped at 15 MB of memory, and no more than 10 percent of CPU time.

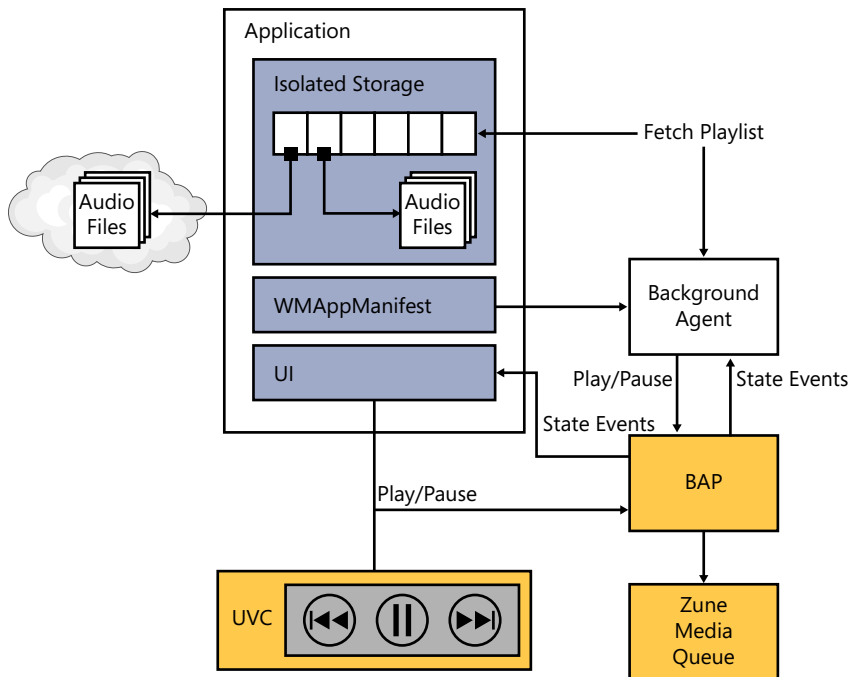


FIGURE 15-11 The relationships between background audio agent components.

The sequence of operations between these various components involves a fairly straightforward handshake. Figure 15-12 illustrates the sequence for playing tracks.

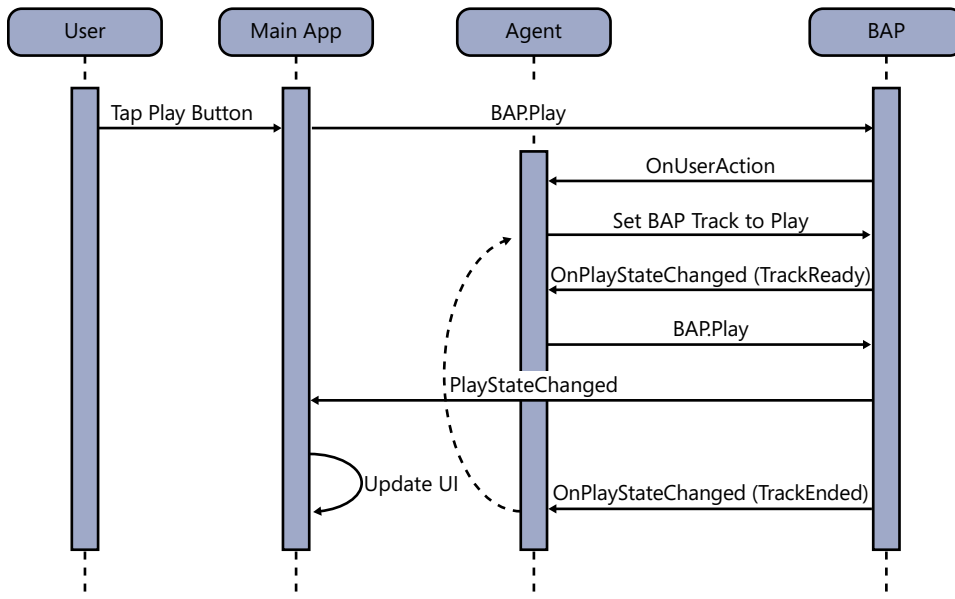


FIGURE 15-12 Playing audio tracks with the *BackgroundAudioPlayer*.

When you add an Audio Playback Agent project to your solution, the wizard-generated code in the class derived from *AudioPlayerAgent* includes almost everything you might need for playing, pausing, skipping, rewinding, and so on. The two main methods that are generated for you are the *OnUserAction* override and the *OnPlayStateChanged* override. *OnUserAction* is called as a result of user action, either from your main application, or from the UVC. You would typically implement this to invoke the *BackgroundAudioPlayer* method that corresponds to the required action. For example, if you get a *UserAction.Play*, you would typically invoke *BackgroundAudioPlayer.Play*. Alternatively, you can set the track to be played, which will cause an *OnPlayStateChanged* event to be raised. You can then handle this event by calling *BackgroundAudioPlayer.Play*.

If your main application is running, you'll also receive state change events; this is your opportunity to update your UI, typically to show the current track information. Figure 15-13 shows the pause sequence.

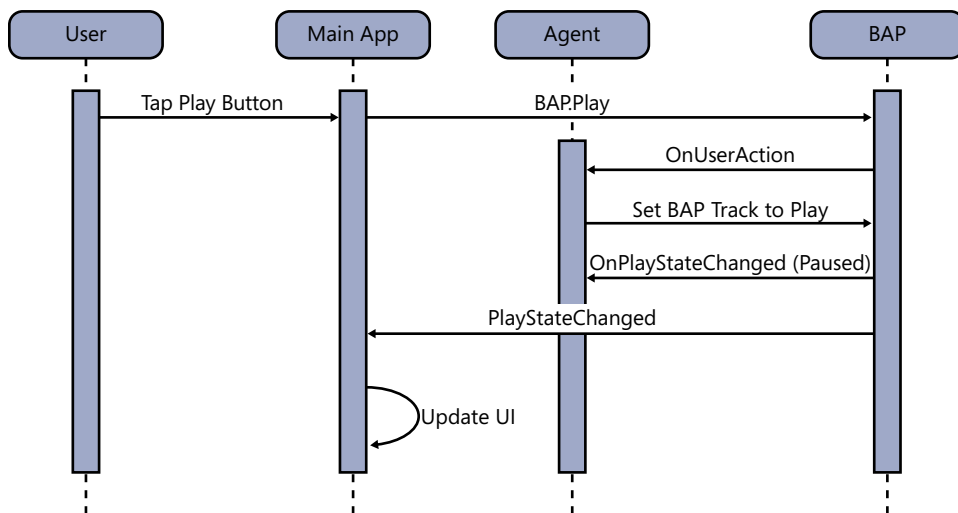


FIGURE 15-13 Pausing audio with the *BackgroundAudioPlayer*.

The pause sequence follows the same model as the play sequence: the user indicates that he wants to pause; you invoke the *BackgroundAudioPlayer.Pause*, which invokes your agent's *OnUserAction*, and so on. It is important to note that your foreground application should not directly update its UI based on user action, but should instead wait for the associated event from the *BackgroundAudioPlayer* to ensure everything proceeded as you planned.

Figure 15-14 shows a simple application (the *BapApp* solution in the sample code) that has an associated background audio agent. The screenshot on the left shows the main application running in the foreground; the other shows the UVC displayed when the user has navigated away from the application, and the background audio agent is running. The section that follows describes how to create this application.

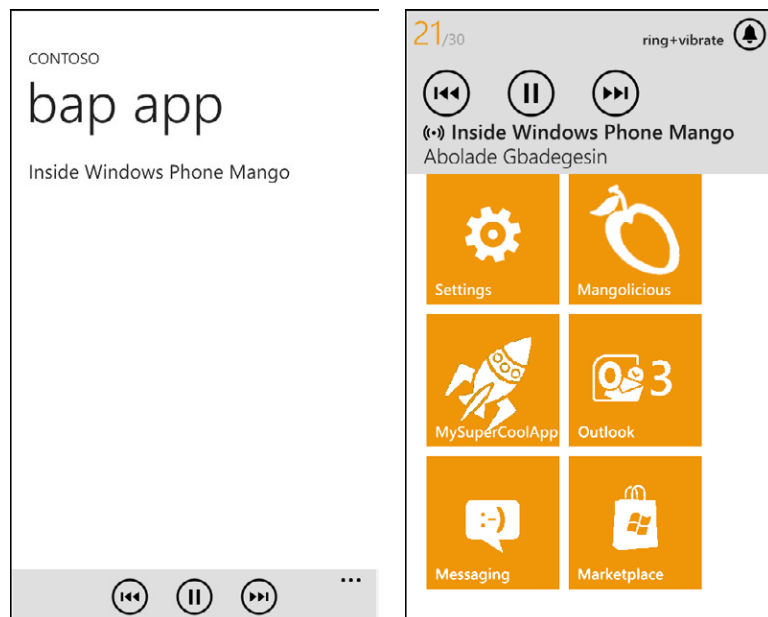


FIGURE 15-14 A foreground application (on the left) running background audio, and the background audio agent continuing to run in the background (right).

Here's a summary of the main tasks:

- Create a regular Windows Phone application.
 - Add one or more audio files as loose files to be deployed as part of the main application XAP. Also add code to copy these files from the application install folder to isolated storage.
 - Add UI elements such as *Buttons*, to allow the user to play audio. In the *Click* handlers, call into the *BackgroundAudioPlayer* methods to play, pause, and so on.
 - Respond to *BackgroundAudioPlayer* state change events and update the UI accordingly.
- Add a Windows Phone Audio Playback Agent project and reference this project in the main application.
 - Define a list of *AudioTrack* items as the playlist for this agent. The items in this list must be either audio files in the application's isolated storage or remote audio files specified by absolute URI, or a mixture of both. In this example, the playlist is not maintained by the main application.

Background Audio: The Main Application

The main application has one *TextBlock* on the page for displaying the current track, and three *AppBar* buttons, for skip-back, play, and skip-forward.

```
<Grid x:Name="LayoutRoot">
    <StackPanel x:Name="ContentPanel">
        <TextBlock x:Name="currentTrack"/>
    </StackPanel>
</Grid>

<phone:PhoneApplicationPage.ApplicationBar>
    <shell:ApplicationBar IsVisible="True">
        <shell:ApplicationBarIconButton
            x:Name="appBarPrev" IconUri="prev.png" Text="prev" Click="appBarPrev_Click"/>
        <shell:ApplicationBarIconButton
            x:Name="appBarPlay" IconUri="play.png" Text="play" Click="appBarPlay_Click"/>
        <shell:ApplicationBarIconButton
            x:Name="appBarNext" IconUri="next.png" Text="next" Click="appBarNext_Click"/>
    </shell:ApplicationBar>
</phone:PhoneApplicationPage.ApplicationBar>
```

You'll have just one local audio file and one remote audio file. In a more sophisticated application, you would allow the user to add and remove items from the playlist via the UI, perhaps downloading files to isolated storage from the Internet. For the purposes of this example, just copy any suitable audio file from the local PC into the project, and then set its Copy To Output Directory property to Copy If Newer. This will be deployed as a loose file in the application's install folder, and you then need to copy it to the application's isolated storage. For this simple sample, you can do this at the end of the *App* class constructor, but you would generally avoid doing this at this critical startup time in a real application.

```
using (IsolatedStorageFile storage = IsolatedStorageFile.GetUserStoreForApplication())
{
    String fileName = " MySampleAudioFile.mp3";
    if (!storage.FileExists(fileName))
    {
        StreamResourceInfo sourceFile =
            Application.GetResourceStream(new Uri(fileName, UriKind.Relative));
        using (IsolatedStorageFileStream targetFile = storage.CreateFile(fileName))
        {
            byte[] bytes = new byte[sourceFile.Stream.Length];
            sourceFile.Stream.Read(bytes, 0, bytes.Length);
            targetFile.Write(bytes, 0, bytes.Length);
        }
    }
}
```

Next, you can implement the App Bar button *Click* handlers in the *MainPage* code-behind. These are trivial to implement; you just need to call into the *BackgroundAudioPlayer* methods to *Play*, *Pause*, and so on. Notice that the *BackgroundAudioPlayer* is a singleton object.

```

private void appBarPlay_Click(object sender, EventArgs e)
{
    if (BackgroundAudioPlayer.Instance.PlayerState == PlayState.Playing)
    {
        BackgroundAudioPlayer.Instance.Pause();
    }
    else
    {
        BackgroundAudioPlayer.Instance.Play();
    }
}

private void appBarNext_Click(object sender, EventArgs e)
{
    BackgroundAudioPlayer.Instance.SkipNext();
}

private void appBarPrev_Click(object sender, EventArgs e)
{
    BackgroundAudioPlayer.Instance.SkipPrevious();
}

```

In the *MainPage* constructor, hook up the *PlayStateChanged* event for the *BackgroundAudio Player*—you'll get these events when the state changes from *Playing* to *Paused* or *Stopped*, and so forth. This is your opportunity to update the UI; you'll toggle the icon and text for the dual-purpose *Play/Pause* button and fetch the current track information to display in your one and only *TextBlock*.

```

private void Bap_PlayStateChanged(object sender, EventArgs e)
{
    switch (BackgroundAudioPlayer.Instance.PlayerState)
    {
        case PlayState.Playing:
            appBarPlay.IconUri = new Uri("pause.png", UriKind.Relative);
            appBarPlay.Text = "pause";
            break;

        case PlayState.Paused:
        case PlayState.Stopped:
            appBarPlay.IconUri = new Uri("play.png", UriKind.Relative);
            appBarPlay.Text = "play";
            break;
    }

    if (null != BackgroundAudioPlayer.Instance.Track)
    {
        currentTrack.Text = BackgroundAudioPlayer.Instance.Track.Title;
    }
    else
    {
        currentTrack.Text = String.Empty;
    }
}

```

You do need to initialize the *AppBar* button fields. Recall from Chapter 5, “Touch UI,” that these need to be initialized explicitly, and this kind of one-time setup should be done in the page constructor (not in the *OnNavigatedTo* override, which is called more frequently). You can add code to *OnNavigatedTo* to accommodate the user returning back to the application after having navigated away. In this scenario, you should check to see if background audio is still playing, and then update the UI accordingly.

```
public MainPage()
{
    InitializeComponent();
    BackgroundAudioPlayer.Instance.PlayStateChanged += new EventHandler(Bap_PlayStateChanged);

    appBarPrev = ApplicationBar.Buttons[0] as ApplicationBarIconButton;
    appBarPlay = ApplicationBar.Buttons[1] as ApplicationBarIconButton;
    appBarNext = ApplicationBar.Buttons[2] as ApplicationBarIconButton;
}

protected override void OnNavigatedTo(NavigationEventArgs e)
{
    switch (BackgroundAudioPlayer.Instance.PlayerState)
    {
        case PlayState.Playing:
            appBarPlay.IconUri = new Uri("pause.png", UriKind.Relative);
            appBarPlay.Text = "pause";
            currentTrack.Text = BackgroundAudioPlayer.Instance.Track.Title;
            break;

        case PlayState.Paused:
        case PlayState.Stopped:
            appBarPlay.IconUri = new Uri("play.png", UriKind.Relative);
            appBarPlay.Text = "play";
            currentTrack.Text = "";
            break;
    }
}
```

Background Audio: The Background Agent

That’s it for the main application. Now, for the agent. Add an audio playback agent project to the solution, and then reference this project in the main application. The wizard generates a great deal of placeholder code, including overrides of *OnPlayStateChanged* and *OnUserAction*. The main thing you need to do is to define the playlist. This simple example is not retrieving the playlist itself from isolated storage; instead, it hard-codes the list, which consists of just two *AudioTrack* items (one local file in isolated storage, and one remote URL).

```
private static List<AudioTrack> playList = new List<AudioTrack>
{
    new AudioTrack(new Uri("MySampleAudioFile.mp3", UriKind.Relative),
        "My Track Title",
        "Author's Name",
        "Album Title",
        null),
```

```

new AudioTrack(new Uri(
    "http://media.ch9.ms/ch9/6e9a/9d148183-4683-41c6-8b70-9f1001346e9a/
    AboladeGbadegesinInsideMango_ch9.mp3"),
    "Inside Windows Phone Mango",
    "Abolade Gbadegesin",
    "Channel 9",
    null)
};

```

You also need to make minor changes to the *GetNextTrack* and *GetPreviousTrack* methods, because the placeholder code simply returns null in each case.

```

static int currentTrack = 0;
private AudioTrack GetNextTrack()
{
    //AudioTrack track = null;
    //return track;
    if (++currentTrack >= playList.Count)
    {
        currentTrack = 0;
    }
    return playList[currentTrack];
}

private AudioTrack GetPreviousTrack()
{
    //AudioTrack track = null;
    //return track;
    if (--currentTrack < 0)
    {
        currentTrack = playList.Count - 1;
    }
    return playList[currentTrack];
}

```

Also, a minor modification to the *OnUserAction* method: the wizard-generated code for the *UserAction.Play* case calls the *Play* method; however, you're relying on the fact that every time you change the *Track* property, you'll get a *PlayStateChanged* event. That's where you actually invoke the *Play* method. So replace the two lines of code in this case to simply set the current track.

```

protected override void OnUserAction(
    BackgroundAudioPlayer player, AudioTrack track, UserAction action, object param)
{
    switch (action)
    {
        case UserAction.Play:
            //if (player.PlayerState != PlayState.Playing)
            //{
            //    player.Play();
            //}
            player.Track = playList[currentTrack];
            break;
        case UserAction.Stop:
            player.Stop();
            break;
    }
}

```

```

        case UserAction.Pause:
            player.Pause();
            break;
        case UserAction.FastForward:
            player.FastForward();
            break;
        case UserAction.Rewind:
            player.Rewind();
            break;
        case UserAction.Seek:
            player.Position = (TimeSpan)param;
            break;
        case UserAction.SkipNext:
            player.Track = GetNextTrack();
            break;
        case UserAction.SkipPrevious:
            player.Track = GetPreviousTrack();
            break;
    }

    NotifyComplete();
}

```

Finally, you need to make one correction to the *OnPlayStateChanged* method. The wizard generated code in the *PlayState.TrackEnded* case gets the previous track. However, the user would probably normally expect that when a track ends, the player moves on to play the *next* track, not the previous one.

```

case PlayState.TrackEnded:
    //player.Track = GetPreviousTrack();
    player.Track = GetNextTrack();
    break;

```

Summary

From Chapter 15 through the end of the book, the focus shifts from the common core features to the new features and platform enhancements introduced in Windows Phone 7.1. In this chapter, you started by looking at how the platform has been enhanced to support multi-tasking in a more seamless way than before. Fast application switching reduces the likelihood that an application will be tombstoned, and therefore allows it to be resumed much more quickly than before, giving the user a more traditional perception of a multi-tasking platform. The multi-tasking features provide a range of options for an application to divide work into multiple processes, with part of the functionality running in the background. Using generic background agents, you can write code that will wake up periodically and perform some operation on a schedule or opportunistically. You can use background transfers to initiate network uploads or downloads, and have them continue even when the user navigates away from your application. Finally, with background audio, you can take advantage of much the same model but with audio playback in place of network transfers. These are just some of the ways that Windows Phone 7.1 has introduced improvements and new features. The remaining chapters look at all the other new features.

Enhanced Phone Services

In addition to the infrastructure improvements to support fast application switching and the range of background task types, Windows Phone 7.1 also introduces many new and enhanced phone services. Sensor programmability is now richer and more flexible, the platform adds support for (optional) front-facing cameras, and camera preview frames are now exposed to the developer for the first time. The combination of sophisticated sensor APIs and access to the camera data brings considerable opportunities for building compelling augmented reality applications. Apart from these major improvements, version 7.1 also brings many additional Launchers and Choosers, and a replacement for *DeviceExtendedProperties*.

Sensor APIs

In Windows Phone 7, the only sensor exposed to marketplace developers was the accelerometer, which is discussed in Chapter 9, “Phone Services.” Windows Phone 7.1 additionally exposes the compass (magnetometer) and gyroscope. On top of that, version 7.1 introduced the *Motion* class, which is a logical fusion of multiple sensors.

The APIs for the three physical sensors and one logical sensor share a high degree of consistency, and they all derive from the *SensorBase<T>* base class, where *T* is a sensor reading type. All public sensor methods are actually methods defined in *SensorBase<T>*. This is because most of the functionality required by developers is actually common to all sensors: the ability to start and stop the sensor, and the *CurrentValueChanged* event, which is raised whenever the sensor data is changed.

Windows Phone 7.1 supports both the existing version 7 chassis (hardware) specification as well as a modified chassis specification designed to target the enhanced version 7.1 user experience (UX). In version 7, there were no optional sensors, but from version 7.1 onward, certain sensors are optional. As a developer, therefore, you should design your application to allow for the possibility that some of the sensors you expect to use might not in fact be present on a given device. This is summarized in Table 16-1.

TABLE 16-1 Sensor Availability in Versions 7 and 7.1

Sensor	Version 7	Version 7.1
Accelerometer	Yes.	Yes.
Compass	Sensor is required, but there is no API exposure.	Optional (but if you have a gyroscope, you must have a compass).
Gyroscope	No.	Optional.

Note that some sensors, such as the magnetometer, are susceptible to environmental conditions (in particular, electromagnetic interference from nearby objects or the phone itself), which introduce significant errors in the reported readings. As a result, the sensor needs recalibration from time to time, in order to report meaningful values. Depending on the sensor, this calibration might be handled automatically by the hardware/driver or it might involve user interaction. The compass is the only sensor in version 7.1 for which *user* calibration is required.

Accelerometer

The only sensor exposed programmatically in version 7 was the accelerometer, represented by the *Accelerometer* class, as discussed in Chapter 9. Code written for version 7 using this class will continue to work, of course, on both version 7 and version 7.1 devices. In addition, you can even continue to use the same code in version 7.1 projects, and it will still work. However, note that IntelliSense will intervene and you'll get compiler warnings if you do this. Specifically, the *ReadingChanged* event was superseded in version 7.1 by the *CurrentValueChanged* event. Behind the scenes, the *Accelerometer* class itself was changed significantly. In particular, it is now derived from the new *SensorBase<T>* class, whereas previously it was not derived from any base class. However, in the interest of maintaining backward compatibility, very little of the change is surfaced to the developer.

So, if you revisit the simple accelerometer application from Chapter 9—the *TestAccelerometer* solution in the sample code—to bring it up to date, you would make this event type change. You can see an updated version in the *MangoAccelerometer* solution in the sample code. At the same time, the version 7.1 *Accelerometer* class also has the option to specify the preferred time interval between reading updates; this is another property inherited from *SensorBase<T>*.

```
accelerometer = new Accelerometer();
//accelerometer.ReadingChanged +=
//    new EventHandler<AccelerometerReadingEventArgs>(
//        accelerometer_ReadingChanged);
accelerometer.CurrentValueChanged +=
    new EventHandler<SensorReadingEventArgs<AccelerometerReading>>(
        accelerometer_CurrentValueChanged);
accelerometer.TimeBetweenUpdates = TimeSpan.FromMilliseconds(33);
```


When you define event handlers, Microsoft Visual Studio generates very explicit code; in fact, more than is strictly necessary. The previous code snippet lists the full auto-completed event handler code so that you can compare it with the version 7 code that is commented out. However, you can safely replace this auto-completed code

```
accelerometer.CurrentValueChanged +=
    new EventHandler<SensorReadingEventArgs<AccelerometerReading>>(
        accelerometer_CurrentValueChanged);
```

with this:

```
accelerometer.CurrentValueChanged += accelerometer_CurrentValueChanged;
```

You should keep in mind that the specific sensor on any given device might not support the requested interval. The API layer for the sensor (that is, the *Accelerometer*, *Compass*, *Gyroscope*, and *Motion* classes) will round the input value to the closest value that is actually permitted on the device. If you want to see the actual value used, you can examine the *TimeBetweenUpdates* property after you set it. Typical values for shipping devices at the time of writing are given in Table 16-2. In the examples in this chapter, each application is handling the incoming sensor data to display some user interface (UI). For this reason, the *TimeBetweenUpdates* property is set consistently to 33 ms, to correspond with the optimum screen frame rate.

TABLE 16-2 *TimeBetweenUpdates* Interval Settings for All Sensors

Sensor	<i>TimeBetweenUpdates</i> interval (minimum and multiples, thereof)
Accelerometer	20 ms
Compass	25 ms
Gyroscope	5 ms
Motion	17 ms

You can also update the event handler accordingly. Specifically, because the *CurrentValueChanged* event is defined on the *SensorBase<T>* base class, it is more generic than the older *ReadingChanged* event. This means that the event arguments are also more generic, which means that you need to do a little more work in order to extract the specific accelerometer readings from them. Where previously you could get to the X, Y, and Z values directly off the event argument object, in version 7.1, you need to drill down a couple of levels to get to the properties that you want.

```
//private void accelerometer_ReadingChanged(
//    object sender, AccelerometerReadingEventArgs e)
//{
//    Dispatcher.BeginInvoke(() =>
//    {
//        statusText.Text = String.Format("X={0}, Y={1}, Z={2}",
//            e.X.ToString("0.00"), e.Y.ToString("0.00"), e.Z.ToString("0.00"));
//    });
//}
```

```
private void accelerometer_CurrentValueChanged(
    object sender, SensorReadingEventArgs<AccelerometerReading> e)
{
    Dispatcher.BeginInvoke(() =>
    {
        statusText.Text = String.Format("X={0}, Y={1}, Z={2}",
            e.SensorReading.Acceleration.X.ToString("0.00"),
            e.SensorReading.Acceleration.Y.ToString("0.00"),
            e.SensorReading.Acceleration.Z.ToString("0.00"));
    });
}
```



Note As with any of the sensor classes derived from *SensorBase<T>*, you need to include a reference to *Microsoft.Xna.Framework.dll*, because they all use XNA-defined types such as the three and four-dimensional matrices, *Vector3* and *Quaternion*.

The version 7.1 SDK includes an extension to the emulator for testing the accelerometer, as shown in Figure 16-1. You access this from the additional tools button on the main emulator, which brings up a window that includes an Accelerometer tab. To test your application's accelerometer code, drag the pink dot around the window. As you drag the dot, the X, Y, and Z coordinates are updated based on the rotation calculations, and those same values are passed into your application via the accelerometer readings.

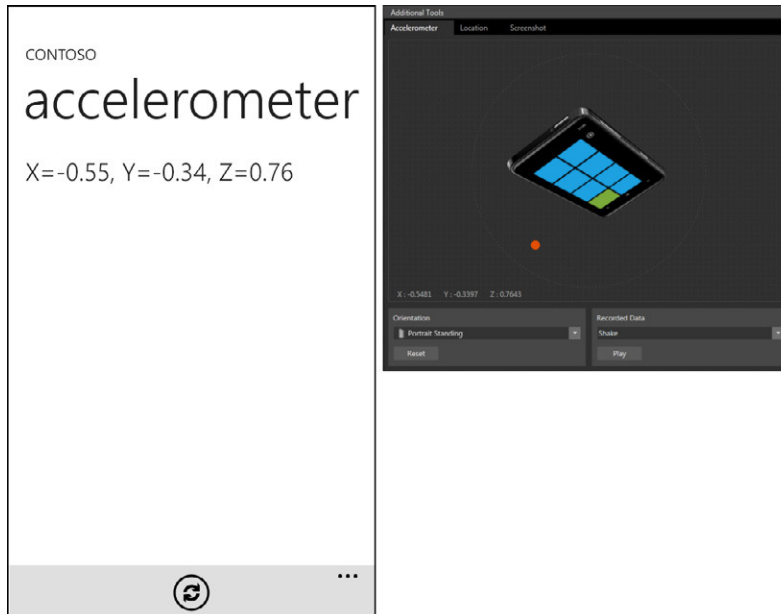


FIGURE 16-1 The accelerometer application and simulator.

Compass

The compass—that is, the magnetometer—in the phone contains a magnetic sensor component that interacts with the earth’s magnetic field that can be aligned to point to magnetic north. The device driver also incorporates the necessary code to compute the declination—the variation between magnetic north and true north. The magnetic declination is different at different points on the earth, and changes with time.

The *Compass* class provided in the application platform encapsulates the functionality of the magnetometer sensor and exposes properties and methods for determining both true north and magnetic north as well as the accuracy of the readings. The *Compass* class also exposes raw magnetometer readings, which can be used to detect magnetic forces around the device.

Figure 16-2 shows a simple compass application that uses the *Compass* class to gather magnetometer sensor readings (the *SimpleCompass* solution in the sample code). The application provides a button in the App Bar to start/stop the compass sensor, and a simple graphical display based on the true north readings. Note that the emulator does not support a compass; therefore, you must test compass applications on a physical device.

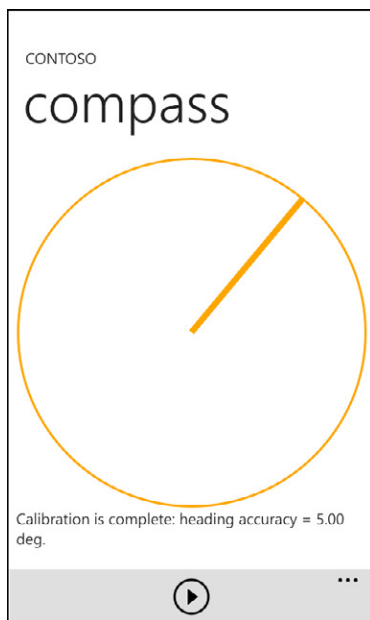


FIGURE 16-2 A simple compass application.

First, you add references to `Microsoft.Devices.Sensors.dll` and `Microsoft.Xna.Framework.dll`. In the XAML, define an inner *Grid* that contains an *Ellipse* for the compass border, and a *Line* for the compass needle. Observe that the application uses the current accent color for both the compass border and the needle. Below that is a *TextBlock* to display errors and other status messages.

```
<Grid x:Name="ContentPanel">
    <Grid.RowDefinitions>
        <RowDefinition Height="*" />
        <RowDefinition Height="80" />
    </Grid.RowDefinitions>

    <Grid Grid.Row="0">
        <Ellipse
            x:Name="compassBorder" Width="455" Height="455" StrokeThickness="3"
            Stroke="{StaticResource PhoneAccentBrush}" />
        <Line
            x:Name="trueNorth" X1="228" Y1="227" X2="228" Y2="0" StrokeThickness="8"
            Stroke="{StaticResource PhoneAccentBrush}" />
    </Grid>

    <TextBlock x:Name="status" Grid.Row="1" TextWrapping="Wrap" />
</Grid>
```

In the *MainPage* code-behind, you declare fields for the *Compass* object and for the center point of the compass needle. In the constructor, you first test to see if this device actually supports a compass. If not, hide the App Bar—and because everything else you do in the application follows on from the App Bar button interaction, this means that if there is no compass, then the application does nothing.

```
private Compass compass;
private double centerX;
private double centerY;

public MainPage()
{
    InitializeComponent();

    if (!Compass.IsSupported)
    {
        status.Text = "This device does not support a compass.";
        ApplicationBar.IsVisible = false;
    }
    else
    {
        appBarStopGo = ApplicationBar.Buttons[0] as ApplicationBarIconButton;
        Loaded += new EventHandler(MainPage_Loaded);
    }
}

private void MainPage_Loaded(object sender, EventArgs e)
{
    centerX = compassBorder.ActualWidth / 2.0;
    centerY = (compassBorder.ActualHeight - status.ActualHeight) / 2.0;
}
```

The page constructor is too early to calculate actual UI element sizes, so you defer to the *Loaded* event. In this handler, you can get the measurements you need to position the needle at the center of the grid (allowing for the status box at the bottom).

Assuming this device does support a compass, then when the user taps the App Bar button, you either start or stop the compass. There is no *IsStopped* or *IsStarted* property on the *Compass* type, but you can determine if the compass readings are ongoing by inspecting the *IsDataValid* property, which is inherited from the *SensorBase<T>* base class. If the compass is non-null and the *IsDataValid* property returns *true*, this must mean that the readings are ongoing, and therefore, that the user has tapped the App Bar to stop the compass. In this case, you stop acquiring sensor readings, and update the App Bar accordingly.

On the other hand, if *IsDataValid* is *false*, then it means that you are not currently receiving sensor readings. This could be because the user has never started the compass in this session or because she previously started it and then stopped it. You therefore instantiate the *Compass* object (if you haven't already done so) and specify the required time interval between data updates. Once this configuration is done, you can start the flow of compass readings.

```
private void appBarStopGo_Click(object sender, EventArgs e)
{
    if (compass != null && compass.IsDataValid)
    {
        compass.Stop();
        appBarStopGo.IconUri = new Uri("play.png", UriKind.Relative);
        appBarStopGo.Text = "go";
    }
    else
    {
        if (compass == null)
        {
            compass = new Compass();
            compass.TimeBetweenUpdates = TimeSpan.FromMilliseconds(33);
            compass.CurrentValueChanged += compass_CurrentValueChanged;
        }

        try
        {
            compass.Start();
            appBarStopGo.IconUri = new Uri("stop.png", UriKind.Relative);
            appBarStopGo.Text = "stop";
        }
        catch (InvalidOperationException)
        {
            status.Text = "Error starting compass.";
        }
    }
}
```

To examine the readings, you hook up the *CurrentValueChanged* event. In this handler, you're interested only in the *TrueHeading* (that is, the true north reading, not the magnetic north reading). You can extract this from the event arguments, and convert it to radians so that you can use it to calculate the new endpoint for the needle line. These events obviously come in on a background thread, so you need to marshal the UI changes to the UI thread with a *Dispatcher*.

```
private void compass_CurrentValueChanged(
    object sender, SensorReadingEventArgs<CompassReading> e)
{
    double trueHeading = e.SensorReading.TrueHeading;
    float headingRadians = MathHelper.ToRadians((float>trueHeading);

    Dispatcher.BeginInvoke(() =>
    {
        trueNorth.X2 = centerX - centerY * Math.Sin(headingRadians);
        trueNorth.Y2 = centerY - centerY * Math.Cos(headingRadians);
    });
}
```

The preceding code is enough to get and render compass readings; however, there's one piece missing: calibration. After each reboot, the compass will need recalibration. Accuracy also fluctuates over time, especially if the phone is moving considerable distances (perhaps the user is flying cross-country, for example). So, if the user has not calibrated the compass recently, then the readings will be inaccurate. In this case, this fact is surfaced to your code by the *Calibrate* event on the *Compass* object. So, before you can have confidence in the readings, you need to ensure that you respond to the *Calibrate* events. To do this, hook up the event when the *Compass* object itself is created.

```
if (compass == null)
{
    compass = new Compass();
    compass.TimeBetweenUpdates = TimeSpan.FromMilliseconds(20);
    compass.CurrentValueChanged += compass_CurrentValueChanged;

    // For calibration.
    compass.Calibrate += compass_Calibrate;
}
```

Also, when you get a reading, you cache the *HeadingAccuracy* property value, as follows:

```
private double headingAccuracy;

private void compass_CurrentValueChanged(
    object sender, SensorReadingEventArgs<CompassReading> e)
{
    double trueHeading = e.SensorReading.TrueHeading;
    float headingRadians = MathHelper.ToRadians((float>trueHeading);

    // For calibration.
    headingAccuracy = Math.Abs(e.SensorReading.HeadingAccuracy);
    ...previous code unchanged, and omitted for brevity.
}
```

To calibrate the sensor, you need to instruct the user to rotate the phone several times over a period of a few seconds so that you can gather a range of readings for the sensor driver to compute the declination. In the *Calibrate* event handler itself, you set up a *DispatcherTimer* to update every frame. So long as the current *HeadingAccuracy* is still not within our acceptable range (≤ 10 degrees), you continue to update the status message to encourage the user to keep moving the phone. As you're updating the UI, you could either use a standard *Timer* and then invoke the page *Dispatcher* to marshal to the UI thread, or simply use the combined *DispatcherTimer* class, which effectively performs both aspects of the task for you.

```
private DispatcherTimer timer;

private void compass_Calibrate(object sender, CalibrationEventArgs e)
{
    Dispatcher.BeginInvoke(() =>
    {
        timer = new DispatcherTimer();
        timer.Interval = TimeSpan.FromMilliseconds(30);
        timer.Tick += new EventHandler(timer_Tick);
        timer.Start();
    });
    compass.Calibrate -= compass_Calibrate;
}

private void timer_Tick(object sender, EventArgs e)
{
    if (headingAccuracy <= 10)
    {
        status.Text = String.Format(
            "Calibration is complete: heading accuracy = {0:00} deg.",
            headingAccuracy);
        timer.Stop();
    }
    else
    {
        status.Text = String.Format(
            "Rotate the phone to calibrate (heading accuracy is only {0:00} deg).",
            headingAccuracy);
    }
}
```

As you can see, it's fairly easy to use the *Compass* type to work with the compass sensor. As is often the case, all the significant work in a compass-based application is likely to be in creating a compelling UI rather than in the use of the sensor itself.

Gyroscope

A gyroscope sensor is used to determine the angular momentum of the device in each of the three primary axes. Compare this with the accelerometer, which measures acceleration in each of the three axes. So, with the accelerometer, the reading increases with the size of the rotation. With the gyroscope, on the other hand, the reading increases with the speed of the rotation. The gyroscope sensor in Windows Phone appliances is a Micro-Electromechanical Systems (MEMS) device, which uses vibration or resonance to generate the readings. The axes are illustrated in Figure 16-3.

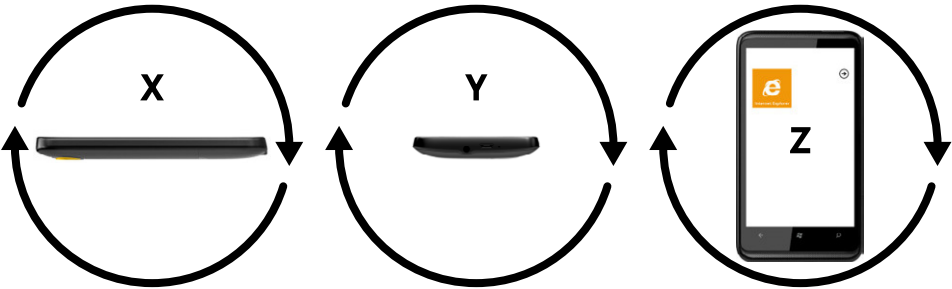


FIGURE 16-3 Rotation about the x, y, and z axes.

Sensors in Windows Phones use a 3D world coordinate system that is based on the system used in the XNA framework. Note that the sensor’s reference point deliberately does not auto-rotate as the screen auto-rotates. This is to allow for applications that wish to use sensors in combination with the orientation of the device as a whole in reference to the Earth, not in reference to the current viewport orientation. The rotations above are known as pitch, roll, and yaw (summarized in Table 16-3), along with their line representations in the sample application that follows.

TABLE 16-3 Descriptions of Pitch, Roll, and Yaw

Movement	Description	Sample Application Line
Pitch	Rotation around the device’s X axis	Red
Roll	Rotation around the device’s Y axis	Green
Yaw	Rotation around the device’s Z axis	Blue

A developer can use the values obtained from the gyroscope sensor to determine which way a device is facing. The rotational velocity is measured in units of radians per second. Because a gyroscope measures rotational velocity and not angle, it is susceptible to drift. Figure 16-4 shows a simple gyroscope application (the *SimpleGyro* solution in the sample code), with three colored bars, each bar representing the rotational velocity along one of the three axes. The X and Y bars are aligned with the X and Y axes on the phone; the Z bar is angled in a way that represents the Z axis on a 2D surface.

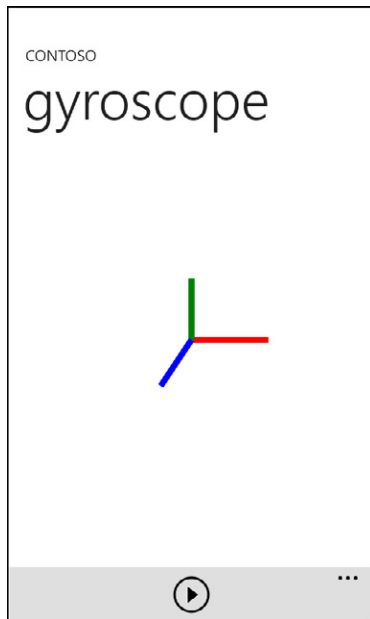


FIGURE 16-4 A gyroscope application with three axes of rotation.

As mentioned previously, all the sensor classes are now based on *SensorBase<T>*, which gives them a high degree of consistency. So, the code for creating a gyroscope-based application is very similar to the code for a compass or accelerometer application. For this display, you can set up the XAML in a very similar way to your previous compass application; this example has a *Grid* containing three *Lines*, and a *TextBlock* below that for status messages.

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto"/>
        <RowDefinition Height="80"/>
    </Grid.RowDefinitions>
    <Grid Height="440" Grid.Row="0">
        <Line x:Name="currentX" X1="228" Y1="240" X2="328" Y2="240" Stroke="Red"
StrokeThickness="8"/>
        <Line x:Name="currentY" X1="228" Y1="240" X2="228" Y2="160" Stroke="Green"
StrokeThickness="8"/>
        <Line x:Name="currentZ" X1="228" Y1="240" X2="188" Y2="300" Stroke="Blue"
StrokeThickness="8"/>
    </Grid>
    <TextBlock x:Name="status" Grid.Row="1" Margin="{StaticResource PhoneHorizontalMargin}"/>
</Grid>
```

In the *MainPage* constructor, test to verify that this device does in fact support a gyroscope sensor, and then set up the App Bar accordingly.

```
private Gyroscope gyroscope;
private double centerX;

public MainPage()
{
    InitializeComponent();

    if (!Gyroscope.IsSupported)
    {
        status.Text = "This device does not support a gyroscope.";
        ApplicationBar.IsVisible = false;
    }
    else
    {
        appBarStopGo = ApplicationBar.Buttons[0] as ApplicationBarIconButton;
    }
}
```

The code model in the App Bar button *Click* handler is also very similar. As before, you check to see if the sensor is currently running and providing valid readings. If it is, stop it, and then reset the App Bar button accordingly. Otherwise, instantiate the *Gyroscope* object (if you haven't already done so) and configure the time between updates. The same restrictions apply as for compass reading intervals, although the exact numbers will vary per device. As before, the critical operation is to hook up the *CurrentValueChanged* event.

```
private void appBarStopGo_Click(object sender, EventArgs e)
{
    if (gyroscope != null && gyroscope.IsDataValid)
    {
        gyroscope.Stop();
        appBarStopGo.IconUri = new Uri("play.png", UriKind.Relative);
        appBarStopGo.Text = "go";
    }
    else
    {
        if (gyroscope == null)
        {
            gyroscope = new Gyroscope();
            gyroscope.TimeBetweenUpdates = TimeSpan.FromMilliseconds(33);
            gyroscope.CurrentValueChanged += gyroscope_CurrentValueChanged;
        }

        try
        {
            gyroscope.Start();
            appBarStopGo.IconUri = new Uri("stop.png", UriKind.Relative);
            appBarStopGo.Text = "stop";
        }
    }
}
```

```

        catch (InvalidOperationException)
        {
            status.Text = "Error starting gyroscope.";
        }
    }
}

```

In the *CurrentValueChanged* event handler, you extract the current rotation rate from the event arguments. This value is in radians per second, so it is a simple calculation to render a line of a suitable size for each of the axes.

```

private void gyroscope_CurrentValueChanged(
    object sender, SensorReadingEventArgs<GyroscopeReading> e)
{
    Vector3 currentRotationRate = e.SensorReading.RotationRate;
    Dispatcher.BeginInvoke(() =>
    {
        currentX.X2 = currentX.X1 + currentRotationRate.X * 50;
        currentY.Y2 = currentY.Y1 - currentRotationRate.Y * 50;
        currentZ.X2 = currentZ.X1 - currentRotationRate.Z * 50;
        currentZ.Y2 = currentZ.Y1 + currentRotationRate.Z * 50;
    });
}

```

As with the compass, the emulator does not support a gyroscope, so you must test gyroscope applications on a physical device. Also, developers interested in knowing the attitude of the device (pitch, roll, yaw) can use the *Motion* class, instead.



Note If you're wondering why the *Compass* reports values in degrees, whereas the *Gyroscope* reports values in radians, this is purely for historical reasons. Most people are used to working with compasses in degrees. On the other hand, the *Gyroscope* and *Motion* types are dependent on classes in the XNA framework that internally use radians. For the same reason, the *Motion* class uses floats like the rest of the XNA framework rather than doubles, as with the rest of Silverlight.

Motion APIs

The managed sensor APIs all expose raw sensor data for those scenarios in which the developer needs a finer level of granularity. However, the raw data can be difficult to work with, and is often not what most applications want to use. In some cases, complex geometrical calculations are required in order to convert the low-level readings into the true orientation of the device.

Hardware sensors are susceptible to a variety of errors, including bias (for example, as a result of temperature fluctuations), drift (loss of accuracy between calibrations), and accuracy limitations (mostly arising from the constant electromagnetic interference from outside and within the phone).

Each sensor has unique strengths and weaknesses with respect to these factors. Gyroscopes, for example, are sensitive to electromagnetic interference and are prone to drift errors because they measure angular velocity as opposed to angle. Accelerometers, on the other hand, do not suffer from drift errors, but they produce poor readings while the device is in motion because the sensor is actually measuring linear acceleration along with gravitational pull.

By taking advantage of multiple sensors simultaneously, applications can compensate for these kinds of errors and produce more accurate readings than can be obtained through a single sensor alone. Although you could build your own processing logic to produce combined readings based on multiple raw sensor readings, the version 7.1 application platform makes this unnecessary. To use combined—or “fusion”—readings, you can use the *Motion* class, which internally takes readings across multiple sensors. The *Motion* class takes the raw sensor readings and surfaces a higher-level abstraction, specifically the device’s attitude (pitch, roll, and yaw), rotational acceleration, and linear acceleration. Augmented-reality applications benefit most from consuming this processed form of sensor data because they typically need to gather readings simultaneously across multiple sensors.

The screenshot in Figure 16-5 shows an application that uses the *Motion* type to represent device rotation rate in a similar way to the earlier gyroscope example as well as attitude (pitch, roll, yaw). This is the *SimpleMotion* solution in the sample code.

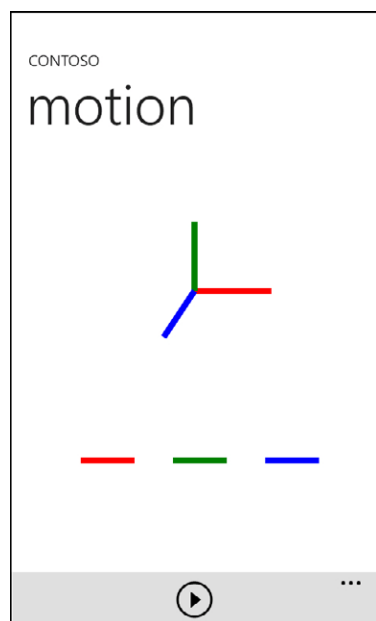


FIGURE 16-5 Using the Motion API.

As with the gyroscope application, you start off with three lines to represent the X, Y, and Z axes. Below that, you have three more lines; these represent the device’s attitude (pitch, roll, and yaw, respectively). For each of the attitude lines, you define a *RotateTransform* so that you can rotate these lines in response to the corresponding sensor readings.

```

<Grid Height="330">
  <Line x:Name="gyroscopeX" X1="228" Y1="170" X2="328" Y2="170"
    Stroke="Red" StrokeThickness="8"/>
  <Line x:Name="gyroscopeY" X1="228" Y1="170" X2="228" Y2="80"
    Stroke="Green" StrokeThickness="8"/>
  <Line x:Name="gyroscopeZ" X1="228" Y1="170" X2="188" Y2="230"
    Stroke="Blue" StrokeThickness="8"/>
</Grid>

<Grid Height="120" >
  <Line x:Name="currentPitch" X1="80" Y1="60" X2="150" Y2="60"
    Stroke="Red" StrokeThickness="8">
    <Line.RenderTransform>
      <RotateTransform CenterX="115" CenterY="60"/>
    </Line.RenderTransform>
  </Line>
  <Line x:Name="currentRoll" X1="200" Y1="60" X2="270" Y2="60"
    Stroke="Green" StrokeThickness="8">
    <Line.RenderTransform>
      <RotateTransform CenterX="235" CenterY="60"/>
    </Line.RenderTransform>
  </Line>
  <Line x:Name="currentYaw" X1="320" Y1="60" X2="390" Y2="60"
    Stroke="Blue" StrokeThickness="8">
    <Line.RenderTransform>
      <RotateTransform CenterX="355" CenterY="60"/>
    </Line.RenderTransform>
  </Line>
</Grid>

```

In the *MainPage* code-behind, declare a *Motion* field, and then implement the page constructor to test to see if the *Motion* API is supported on this device, configuring the App Bar as in previous examples. Be aware that the *Motion* class has two different sensor configurations:

- Normal Motion, which uses the compass and the accelerometer sensor.
- Enhanced Motion, which uses the compass, the accelerometer, and the gyroscope.

These modes are configured internally and are not exposed in the API. However, if your application requires the accuracy of Enhanced motion, you should check to verify that the device on which the application is running supports the gyroscope sensor. Here's an additional test for this:

```

private Motion motion;

public MainPage()
{
    InitializeComponent();

    if (!Motion.IsSupported)
    {
        status.Text = "This device does not support the Motion API.";
        ApplicationBar.IsVisible = false;
    }
}

```

```

else
{
    if (Gyroscope.IsSupported)
    {
        status.Text = "Enhanced Motion supported";
    }
    else
    {
        status.Text = "Normal Motion supported";
    }
    appBarStopGo = ApplicationBar.Buttons[0] as ApplicationBarIconButton;
}
}

```

In the App Bar button *Click* handler, you start or stop the *Motion* object, which under the covers will start/stop the two or three sensors that the *Motion* class is using. As with all *SensorBase<T>* types, you can set the *TimeBetweenUpdates* property and hook up the *CurrentValueChanged* event.

```

private void appBarStopGo_Click(object sender, EventArgs e)
{
    if (motion != null && motion.IsDataValid)
    {
        motion.Stop();
        appBarStopGo.IconUri = new Uri("play.png", UriKind.Relative);
        appBarStopGo.Text = "go";
    }
    else
    {
        if (motion == null)
        {
            motion = new Motion();
            motion.TimeBetweenUpdates = TimeSpan.FromMilliseconds(33);
            motion.CurrentValueChanged += motion_CurrentValueChanged;
        }

        try
        {
            motion.Start();
            appBarStopGo.IconUri = new Uri("stop.png", UriKind.Relative);
            appBarStopGo.Text = "stop";
        }
        catch (InvalidOperationException)
        {
            status.Text = "Error starting the Motion sensors.";
        }
    }
}

```

When you get a *CurrentValueChanged* event, you first extract the *DeviceRotationRate*, and then use the *X*, *Y*, and *Z* property values to determine the endpoints of the axis rotation lines. You then extract the *Attitude* property and set the angle of the *RenderTransform* for each of the three attitude lines to correspond to the *Pitch*, *Roll*, and *Yaw* property values.

```

private void motion_CurrentValueChanged(
    object sender, SensorReadingEventArgs<MotionReading> e)
{
    Vector3 rotationRate = e.SensorReading.DeviceRotationRate;
    AttitudeReading attitude = e.SensorReading.Attitude;
    Dispatcher.BeginInvoke(() =>
    {
        gyroscopeX.X2 = gyroscopeX.X1 + rotationRate.X * 50;
        gyroscopeY.Y2 = gyroscopeY.Y1 - rotationRate.Y * 50;
        gyroscopeZ.X2 = gyroscopeZ.X1 - rotationRate.Z * 50;
        gyroscopeZ.Y2 = gyroscopeZ.Y1 + rotationRate.Z * 50;

        ((RotateTransform)currentPitch.RenderTransform).Angle =
            MathHelper.ToDegrees(attitude.Pitch);
        ((RotateTransform)currentRoll.RenderTransform).Angle =
            MathHelper.ToDegrees(attitude.Roll);
        ((RotateTransform)currentYaw.RenderTransform).Angle =
            MathHelper.ToDegrees(attitude.Yaw);
    });
}

```

The *Motion* type not only exposes gyroscopic rotation rates and device attitude, it also exposes acceleration and gravity readings. So, you could add another couple of grids comprised of three lines, with the first set to represent the accelerometer readings, and the second set to represent the gravity readings. You can see this at work in the *Motion4* solution in the sample code.

```

<Grid x:Name="accelerometerGrid" Height="80" Grid.Row="2">
    <Line x:Name="accelerometerX" X1="225" Y1="20" X2="230" Y2="20"
        Stroke="Red" StrokeThickness="8"/>
    <Line x:Name="accelerometerY" X1="225" Y1="40" X2="230" Y2="40"
        Stroke="Green" StrokeThickness="8"/>
    <Line x:Name="accelerometerZ" X1="225" Y1="60" X2="230" Y2="60"
        Stroke="Blue" StrokeThickness="8"/>
</Grid>

<Grid Height="80" Grid.Row="3">
    <Line x:Name="gravityX" X1="225" Y1="20" X2="230" Y2="20" Stroke="Red" StrokeThickness="8"/>
    <Line x:Name="gravityY" X1="225" Y1="40" X2="230" Y2="40"
        Stroke="Green" StrokeThickness="8"/>
    <Line x:Name="gravityZ" X1="225" Y1="60" X2="230" Y2="60"
        Stroke="Blue" StrokeThickness="8"/>
</Grid>

```

You could add a center-point calculation, as in the earlier compass example. Then, in the *Current ValueChanged* handler, you could update the UI by redrawing these two sets of lines according to the *DeviceAcceleration* and *Gravity* properties of the *SensorReading*.

```

accelerometerX.X2 = centerX + acceleration.X * 200;
accelerometerY.X2 = centerX + acceleration.Y * 200;
accelerometerZ.X2 = centerX + acceleration.Z * 200;

gravityX.X2 = centerX + gravity.X * 200;
gravityY.X2 = centerX + gravity.Y * 200;
gravityZ.X2 = centerX + gravity.Z * 200;

```

Working with each of the *SensorBase<T>* types is very similar, and the *Motion* logical fusion sensor follows exactly the same model. The *Motion* type really comes into its own when used in an augmented reality scenario, as discussed in the upcoming self-named section.

Camera Pipeline

In Chapter 9, you looked at the core support for taking photos programmatically by using the *Camera CaptureTask* Chooser. Windows Phone 7.1 introduces support for working directly with the camera input. This gives you significantly greater flexibility for camera-based functionality within your application.

Figure 16-6 shows a simple application (the *SimpleCamera* solution in the sample code) that uses the *PhotoCamera* class to take photos and store them to the local media library on the phone. This application doesn't provide any more functionality than is already available to the user via the standard hardware camera button, but it does act as a good starting point for software-based camera operations.

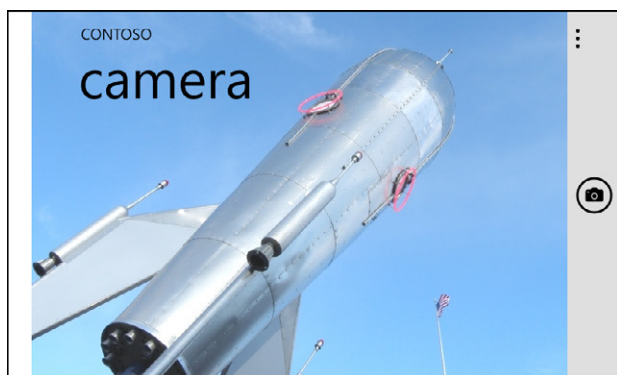


FIGURE 16-6 A simple camera application.

In the application, there is an added reference to the *Microsoft.Xna.Framework.dll*; this is because you're using the *MediaLibrary* type defined in that assembly. Notice also that all new projects created to target Windows Phone 7.1 include the camera capability in the *WMAppManifest.xml*, by default. However, if you're porting a project that originally targeted version 7, you'll need to add this line manually, as follows:

```
<Capability Name="ID_CAP_ISV_CAMERA"/>
```

In the *MainPage* XAML, you first set the *SupportedOrientations* and the *Orientation* attributes both to *Landscape*. This is because the *PhotoCamera* defaults to landscape viewing mode (and you don't want to do the work in this simple example to support portrait mode). Strictly speaking, this is true for rear-facing cameras, but not necessarily for front-facing cameras. If you wanted to support a front-facing camera, you'd want to look at the *Camera.Orientation* property to know how much (if at all) you need to rotate the preview window. You then define a *Canvas* and a *StackPanel* for the usual title

information. Be aware that the order in which these are defined is important; you want the title panel to overlay the canvas. On a realistic camera application, you would probably dispense with the titles altogether. The *Canvas* specifies a *VideoBrush* for the *Background* property. At runtime, you'll take the camera data stream as the source for this *VideoBrush*, thereby implementing a simple viewfinder.

```
<Grid x:Name="LayoutRoot" Background="Transparent">

    <Canvas>
        <Canvas.Background>
            <VideoBrush x:Name="viewfinderBrush"/>
        </Canvas.Background>
    </Canvas>

    <StackPanel x:Name="TitlePanel" Margin="12,17,0,28">
        <TextBlock x:Name="ApplicationTitle" Text="CONTOSO"
            Style="{StaticResource PhoneTextNormalStyle}"/>
        <TextBlock x:Name="PageTitle" Text="camera" Margin="9,-7,0,0"
            Style="{StaticResource PhoneTextTitle1Style}"/>
    </StackPanel>
</Grid>

<phone:PhoneApplicationPage.ApplicationBar>
    <shell:ApplicationBar IsVisible="True">
        <shell:ApplicationBarIconButton x:Name="appBarCamera" IconUri="camera.png" Text="go"
            Click="appBarCamera_Click"/>
    </shell:ApplicationBar>
</phone:PhoneApplicationPage.ApplicationBar>
```

You also define a single-button App Bar. The idea is that the user can tap this button to operate the camera shutter (to take a photo). In the code-behind, you define fields for the *PhotoCamera* and the *MediaLibrary*. At some suitable early point (the page constructor or the *OnNavigatedTo* override), you need to check whether this device actually supports a camera. If so, instantiate the *PhotoCamera* object, specifying the primary (backward-facing) camera device. Then, you can set the viewfinder *VideoBrush* source to this camera. You also hook up the *Initialized* and *CaptureImageAvailable* events. You only enable the Take A Photo UI (the App Bar button) after you get the *Initialized* event; this prevents the narrow window of opportunity in which the user might try to take a photo before the camera is ready. You'll get the *CaptureImageAvailable* event when the user actually takes a photo.

```
private PhotoCamera camera;
private MediaLibrary library = new MediaLibrary();

protected override void OnNavigatedTo(NavigationEventArgs e)
{
    if (PhotoCamera.IsCameraTypeSupported(CameraType.Primary))
    {
        appBarCamera = ApplicationBar.Buttons[0] as ApplicationBarIconButton;
        camera = new PhotoCamera(CameraType.Primary);
        camera.Initialized += camera_Initialized;
        camera.CaptureImageAvailable += camera_CaptureImageAvailable;
        viewfinderBrush.SetSource(camera);
    }
    else
    {

```

```

        ApplicationBar.IsVisible = false;
    }
}

private void camera_Initialized(object sender, CameraOperationCompletedEventArgs e)
{
    Dispatcher.BeginInvoke(() => {appBarCamera.IsEnabled = true; });
}

protected override void OnNavigatedFrom(NavigationEventArgs e)
{
    if (camera != null)
    {
        camera.CaptureImageAvailable -= camera_CaptureImageAvailable;
        camera.Dispose();
        camera = null;
    }
}

```

The camera is an exclusive-use device; that is, only one application can take control of the camera at a time, and it is imperative to release the camera as soon as possible. For this reason, you want to override the *OnNavigatedFrom* method to unhook the event handler, dispose the *PhotoCamera* object, and set the object reference to null. This allows garbage collection of the object, and relinquishes the underlying device resource. This also assists in terminating the application more quickly, and finally it serves to minimize power consumption, because the camera driver can shut down the device as soon as the reference is disposed.



Note Performing the camera clean-up work in the *OnNavigatedFrom* covers the possibility of the user navigating (either forward or backward) away from the application. For maximum robustness, you could also perform the same work in an override of the *OnRemovedFromJournal* method. This would be invoked in other cases than application navigation, such as programmatically clearing the backstack. For more details on backstack management, see Chapter 19, “Framework Enhancements.”

The *CaptureImageAvailable* event is raised when the capture sequence is complete and an image is available; in other words, when a photo has been taken. You implement this to save the photo to the local media library. You do this inside a using statement to ensure that the image stream is closed when you’re done.

```

private void camera_CaptureImageAvailable(object sender, ContentReadyEventArgs e)
{
    String fileName = DateTime.Now.ToLongTimeString() + ".jpg";

    using (e.ImageStream)
    {
        library.SavePictureToCameraRoll(fileName, e.ImageStream);
    }
}

```

That's all there is to it—a fully functioning camera application in about half a dozen lines of code. Of course, the camera supports a wider range of features, including flash mode, auto-focus, orientation, final and preview resolutions, and so on. All of these features are exposed through the *Photo Camera* class. For example, it's easy enough to add a second App Bar button to allow the user to cycle through the available flash mode options. You can see this at work in the *SimpleCamera+Flash* solution in the sample code.

```
private void appBarFlash_Click(object sender, EventArgs e)
{
    switch (camera.FlashMode)
    {
        case FlashMode.Off:
            if (camera.IsFlashModeSupported(FlashMode.On))
            {
                camera.FlashMode = FlashMode.On;
                appBarFlash.Text = "on";
            }
            break;
        case FlashMode.On:
            if (camera.IsFlashModeSupported(FlashMode.RedEyeReduction))
            {
                camera.FlashMode = FlashMode.RedEyeReduction;
                appBarFlash.Text = "redeye";
            }
            else if (camera.IsFlashModeSupported(FlashMode.Auto))
            {
                camera.FlashMode = FlashMode.Auto;
                appBarFlash.Text = "auto";
            }
            else
            {
                camera.FlashMode = FlashMode.Off;
                appBarFlash.Text = "off";
            }
            break;
        case FlashMode.RedEyeReduction:
            if (camera.IsFlashModeSupported(FlashMode.Auto))
            {
                camera.FlashMode = FlashMode.Auto;
                appBarFlash.Text = "auto";
            }
            else
            {
                camera.FlashMode = FlashMode.Off;
                appBarFlash.Text = "off";
            }
            break;
        case FlashMode.Auto:
            if (camera.IsFlashModeSupported(FlashMode.Off))
            {
                camera.FlashMode = FlashMode.Off;
                appBarFlash.Text = "off";
            }
            break;
    }
}
```

Apart from features formalized in the *PhotoCamera* API, the most critical innovation in version 7.1 is the ability for the developer to get hold of the raw camera data stream. This gives you considerable opportunity to manipulate the view and the photo image. In the next section, you'll look at how you can take this a step further, by combining the *Motion* API with the *PhotoCamera* API to build augmented reality applications.

Augmented Reality

An augmented reality (AR) application is one with which you enhance the user's view of the real world with some additional information. Typically, you would use the camera to allow the user to view the world, and then overlay onto the view some other data. For example, you could overlay data about nearby cafés and restaurants, or overlay a map onto the view. If you had face-recognition software, you could overlay contact details whenever the user views a person known to the user, and so on. Think of first-person shooter arcade games, in which target information and ordinance resources are usually overlaid onto the scene ahead.

Figure 16-7 shows an AR application (the *DirectionalViewfinder* solution in the sample code) in which directional information (compass readings) is overlaid on top of the camera viewfinder. As the user pivots around, you track which direction he's facing.



FIGURE 16-7 An AR application that combines compass sensor readings with the camera.

To build the UI for this application, define a *VideoBrush*-based camera viewfinder as you did with the earlier camera applications. Layered on top of that, there are four *TextBlocks*, one each for the major points of the compass (reading clockwise, North, East, South, and West).

```

<Canvas>
  <Canvas.Background>
    <VideoBrush x:Name="viewFinder"/>
  </Canvas.Background>
</Canvas>

<Grid Width="800" Height="480" >
  <TextBlock Text="North">
    <TextBlock.Projection>
      <PlaneProjection
        x:Name="northProjection" CenterOfRotationZ="400" LocalOffsetZ="-400"/>
    </TextBlock.Projection>
  </TextBlock>

  <TextBlock Text="East">
    <TextBlock.Projection>
      <PlaneProjection
        x:Name="eastProjection" CenterOfRotationZ="400" LocalOffsetZ="-400"/>
    </TextBlock.Projection>
  </TextBlock>

  <TextBlock Text="South">
    <TextBlock.Projection>
      <PlaneProjection
        x:Name="southProjection" CenterOfRotationZ="400" LocalOffsetZ="-400"/>
    </TextBlock.Projection>
  </TextBlock>

  <TextBlock Text="West">
    <TextBlock.Projection>
      <PlaneProjection
        x:Name="westProjection" CenterOfRotationZ="400" LocalOffsetZ="-400"/>
    </TextBlock.Projection>
  </TextBlock>
</Grid>

```

At runtime, you'll be rotating each *TextBlock* on its Y axis so that each one is positioned in space around the viewer. By default, the axes of rotation run directly through the center of an object, which would cause the object to rotate around its center, but that's not what you want. Here, you're moving the center of rotation to the outer edge of the object on the Z axis so that it rotates around that edge. You also set the *LocalOffsetZ* to a large negative number, to reposition the text further away from the viewpoint on the Z axis. In this way, when you rotate the text on the Y axis, it doesn't rotate in place; rather, it rotates at a distance away from the center point.

As part of the page resources, you define an implicit style (new in Microsoft Silverlight 4.0, included with Windows Phone SDK 7.1) for the *TextBlocks* in the application. You can use this to provide a default value for selected properties for all *TextBlocks* (unless manually overridden, subsequently). Specifically, set the *FontSize* to huge, the *Foreground* color to *Mango*, and the *Opacity* to 50 percent, as demonstrated in the following:

```
<phone:PhoneApplicationPage.Resources>
  <Style TargetType="TextBlock">
    <Setter Property="FontSize" Value="{StaticResource PhoneFontSizeHuge}"/>
    <Setter Property="Foreground" Value="#FFF09609"/>
    <Setter Property="Opacity" Value="0.5"/>
  </Style>
</phone:PhoneApplicationPage.Resources>
```

In the code-behind, declare a couple of fields for the *Motion* sensor and the *PhotoCamera* object, and then initialize these in the *OnNavigatedTo* override. As before, hook up a handler for the *CurrentValueChanged* event. You should also perform due diligence clean-up operations in the *OnNavigatedFrom* override.

```
private Motion motion;
private PhotoCamera camera;

protected override void OnNavigatedTo(NavigationEventArgs e)
{
    if (PhotoCamera.IsCameraTypeSupported(CameraType.Primary))
    {
        camera = new PhotoCamera(CameraType.Primary);
        camera = new PhotoCamera();
        viewFinder.SetSource(camera);
    }
    if (Motion.IsSupported)
    {
        motion = new Motion();
        motion.TimeBetweenUpdates = TimeSpan.FromMilliseconds(33);
        motion.CurrentValueChanged += motion_CurrentValueChanged;
        motion.Start();
    }
}

protected override void OnNavigatedFrom(NavigationEventArgs e)
{
    if (camera != null)
    {
        camera.Dispose();
    }
    if (motion != null)
    {
        motion.CurrentValueChanged -= motion_CurrentValueChanged;
        motion.Stop();
    }
}
```

All the interesting work is done in the handler for the *CurrentValueChanged* event. First, you get the matrix representation of the attitude reading data from the event arguments. You want to know the X,Y coordinates of the position in space that you're currently looking at, along the Z axis. You can get the values in the Z column of the matrix. The device coordinate system on the phone has the Z axis pointing out of the phone back toward the user, but what you want is the opposite direction; that is, looking through the viewfinder, your perspective relative to the device is negative on the Z axis, so you get the negative of these two values.

Next, use *Atan2* to establish the angle between the X axis and the coordinates, and then convert from radians to degrees. You're going to use the final value to rotate your *TextBlocks* on the Y axis. Silverlight assumes a clockwise rotation angle, but *Motion* measures rotation counter-clockwise. To fix this, set the value to negative.

This will give you the Y coordinate at the center of the position to which you want to set the North *TextBlock*. Finally, offset the East, South, and West *TextBlocks* by 90 degrees cumulatively, and then update the UI accordingly, via the *PlaneProjection*. (Many thanks to Mark Paley for explaining the math to me in simple terms.)

```
private void motion_CurrentValueChanged(
    object sender, SensorReadingEventArgs<MotionReading> e)
{
    Matrix rotation = e.SensorReading.Attitude.RotationMatrix;
    double heading = Math.Atan2(-rotation.M13, -rotation.M23);
    heading = -(MathHelper.ToDegrees((float)heading)) % 360;

    Dispatcher.BeginInvoke(() =>
    {
        northProjection.RotationY = heading;
        eastProjection.RotationY = heading + 90;
        southProjection.RotationY = heading + 180;
        westProjection.RotationY = heading + 270;
    });
}
```

Building an AR experience involves two fairly easy techniques, and two potentially very complex issues. The easy techniques are presenting a camera viewfinder to the user, and overlaying data onto that view. The potentially complex issues are identifying which items of data to surface and determining where in the view to position each item. A sophisticated AR application might also add and remove data items dynamically as the user moves through the world. For example, adding details about nearby cafés as the user walks or drives down the street.

The Geo Augmented Reality Toolkit

Some of the more complex issues that arise with AR applications can be alleviated by using a dedicated AR toolkit. One such toolkit, the Geo Augmented Reality Toolkit (GART) has been released under the Microsoft Limited Permissive License (MS-LPL) on codeplex at <http://gart.codeplex.com>. The screenshot in Figure 16-8 shows an application built with the GART. This is the *TestGart* solution in the sample code.

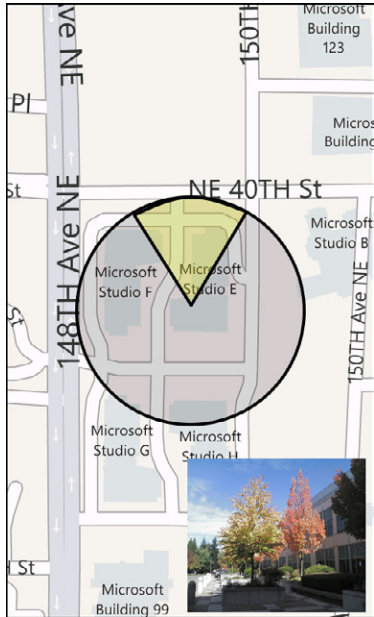


FIGURE 16-8 A GART-based application.

To build a GART-based application, you first need to add a reference to `GART.dll`. Next, add a namespace declaration in your page XAML for the GART assembly, and then add an `ARDisplay` control to your page. This control contains the core AR engine in the toolkit. You can add other GART controls as children to your `ARDisplay`. This example application adds *OverheadMap*, *HeadingIndicator*, and *VideoPreview* controls. In the screenshot, the map takes up the whole screen, the *HeadingIndicator* (a primitive compass display) is overlaid on top of the map, and the *VideoPreview* is the top layer, sized to fit into the lower-right corner (offset from the edge by 12 pixels).

```
xmlns:gart="clr-namespace:GART.Controls;assembly=GART"

<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
  <gart:ARDisplay x:Name="gartDisplay">
    <gart:OverheadMap
      ZoomLevel=".8" CredentialsProvider="{StaticResource BingCredentials}"/>
    <gart:HeadingIndicator/>
    <gart:VideoPreview
      Width="250" Height="200" Margin="12"
```



```

        HorizontalAlignment="Right" VerticalAlignment="Bottom"/>
    </gart:ARDisplay>
</Grid>

```

In this example, the map *ZoomLevel* on the *OverheadMap* is set to something a little wider than street level (0.8); accept all the defaults for the *HeadingIndicator*, and then set the *VideoPreview* to a smaller rectangle in the lower-right corner of the page. The *OverheadMap* control uses Bing maps under the covers; therefore, need to add your Bing developer credentials to the application in order to use the map service. (This is described in Chapter 11, “Web and Cloud.”) One way to go about doing this is to add an *ApplicationCredentialsProvider* to your App.xaml. To do that, you need to add a reference to the *Microsoft.Phone.Controls.Maps* assembly in your project, and then add an XML namespace for this in your App.xaml. Using this approach, you can set the named *CredentialsProvider* to the corresponding property in the *OverheadMap* declaration.

```

xmlns:maps="clr-namespace:Microsoft.Phone.Controls.Maps;assembly=Microsoft.Phone.Controls.Maps"
<Application.Resources>
    <maps:ApplicationIdCredentialsProvider x:Key="BingCredentials"
        ApplicationId="<< YOUR BING MAPS CREDENTIALS >>" />
</Application.Resources>

```

Then, all you have to do is to start and stop the GART services, typically in the *OnNavigatedTo* and *OnNavigatedFrom* overrides, respectively.

```

protected override void OnNavigatedTo(NavigationEventArgs e)
{
    gartDisplay.StartServices();
}

protected override void OnNavigatedFrom(NavigationEventArgs e)
{
    gartDisplay.StopServices();
}

```

The GART includes other views that you can layer onto the page, and you can also create your own views by writing custom controls that implement the *IARView* interface. By implementing *IARView*, you can indicate where the user is located, or the location for which he’s searching. If you also want to create a view that shows data for items in the immediate surroundings of the user (buildings, shops, restaurants, people, and so on), then you need to implement the *IARItemsView* interface. The download on codeplex includes sample applications to get you started. Although as of this writing the GART is a very early release and has some rough edges, it’s a great way to build simple geocoordinate-based AR applications, quickly.



More Info There’s another good AR toolkit on codeplex that you can use in both desktop Silverlight and Windows Phone applications. It’s called the SLARToolkit, and you can download it from <http://slartoolkit.codeplex.com/>. In addition to supporting the phone’s *PhotoCamera* model, this toolkit has support for Silverlight’s *Webcam API* and *CaptureSource* model.

New Photo Extensibility

In Chapter 9, you looked at how you could add your application to the Extras menu in the standard photo/picture library on the phone. The concept being that the user selects a picture from the library, and then selects your application from the menu to perform some operation on that picture. This involved adding an extras.xml file to your project and overriding *OnNavigatedTo* to see if the *NavigationContext* query string includes the key "token"; this is how you know your application has been launched via the Extras menu, as opposed to the normal Start menu.

Windows Phone SDK 7.1 introduces an alternative mechanism to extend the picture functionality on the phone. This feature supersedes the previous Extras mechanism. It should be used instead for version 7.1 projects. In version 7.1, the Extras menu itself has been replaced with an Apps menu. This Apps menu leads to a list of installed applications that extend the picture library. In fact, there are now three jumping off points in the phone for picture extensions, which are described in Table 16-4. In each case, to implement the extension, you must provide an Extensions section in your WMAppManifest.xml as a child of the *App* element, following this format:

```
<Extensions>
  <Extension ExtensionName="XXX" ConsumerID="{5B04B775-356B-4AA0-AAF8-6491FFE5632}"
    TaskID="_default" />
</Extensions>
```

Note that "XXX" is a placeholder for the specific type of picture extension, listed in the table. All other entries are identical across all three types of extension.

TABLE 16-4 Three Methods for Adding Picture Extension Applications

Launch Point in the UI	Description	Application Behavior	Manifest ExtensionName
The apps menu in the viewer for an individual picture in the library	This is the version 7.1 replacement for the old Extras menu, and closely follows the version 7 UX.	Your application is launched, and then a parameter named <i>token</i> is passed in, corresponding to the picture the user is viewing.	<i>Photos_Extra_Viewer</i>
The new apps pivot on the pictures hub.	Note that this will not be visible unless the user has at least one picture extension application installed.	Your application is launched in the same way as from the Start page; no additional parameters are passed in.	<i>Photos_Extra_Hub</i>
The share link in the individual picture viewer.	Intended for your application to share the selected picture via a custom web service.	Your application is launched, and a parameter named <i>FileId</i> is passed in, corresponding to the picture the user is viewing.	<i>Photos_Extra_Share</i>

You can also specify multiple extension points. For example, you could have your application launched from both the picture viewer and the picture hub, or even from all three launch points.

The following sample application (the *MyPictureExtension* solution in the sample code), which is shown in Figure 16-9, closely follows the behavior of the photo extras sample in Chapter 9. The extension is set up to be listed in the picture viewer menu. When the user navigates to the application, you simply fetch the selected picture and display it in the application UI.

CONTOSO

extension

this is the picture you selected



FIGURE 16-9 A pictures extension application.

To create this application, you first need to add the appropriate extension to the `WMAppManifest.xml`, as shown here:

```
<Extensions>
  <Extension ExtensionName="Photos_Extra_Viewer"
    ConsumerID="{5B04B775-356B-4AA0-AAF8-6491FFE5632}" TaskID="_default" />
</Extensions>
```

Next, in the `MainPage.xaml`, declare a `TextBlock` for a message, and an `Image` control for the selected picture.

```
<StackPanel x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
  <TextBlock Text="this is the picture you selected"
    Margin="{StaticResource PhoneHorizontalMargin}"
    FontSize="{StaticResource PhoneFontSizeLarge}" />
  <Image Height="350" HorizontalAlignment="Left"
    Name="selectedPicture" VerticalAlignment="Top" Width="450" />
</StackPanel>
```

Override `OnNavigatedTo` to examine the `NavigationContext` for the “token” query string and use that value to fetch the corresponding picture from the media library. As always, this requires adding a reference to `Microsoft.Xna.Framework`, it won’t work on the emulator, and it won’t work while the device is tethered and connected to Zune. So, you need to deploy the application to an attached device, and then detach it before testing the application. Alternatively, use the `WPConnect` tool, as described in Chapter 8, “Diagnostics and Debugging.”

```

private bool launchedFromStart;
private PhotoChooserTask chooser;

public MainPage()
{
    InitializeComponent();
    launchedFromStart = true;
    chooser = new PhotoChooserTask();
    chooser.Completed +=
        new System.EventHandler<PhotoResult>(chooser_Completed);
}

protected override void OnNavigatedTo(NavigationEventArgs e)
{
    if (NavigationContext.QueryString.ContainsKey("token"))
    {
        launchedFromStart = false;
        MediaLibrary library = new MediaLibrary();
        Picture picture = library.GetPictureFromToken(
            NavigationContext.QueryString["token"]);
        DoSomethingWithSelectedImage(picture.GetImage());
    }
    else
    {
        if (!launchedFromStart)
        {
            launchedFromStart = false;
            chooser.Show();
        }
    }
}

private void DoSomethingWithSelectedImage(Stream imageStream)
{
    BitmapImage bitmap = new BitmapImage();
    bitmap.SetSource(imageStream);
    selectedPicture.Source = bitmap;
}

private void chooser_Completed(object sender, PhotoResult e)
{
    DoSomethingWithSelectedImage(e.ChosenPhoto);
}

```

As with the previous example, you also add code to allow for the possibility that the user has launched the application directly from the Start menu. To test the application, go to the pictures hub, navigate to a picture in the viewer, and then bring up the viewer menu. From the Apps menu, select your application. Also test the alternative code path, whereby the user launches the application from Start. The behavior so far should be identical to the version in Chapter 9.

Now it's time to add support for launching from the pictures hub. All you need to do is to add a second extension entry in the WMAppManifest.xml.

```

<Extensions>
  <Extension ExtensionName="Photos_Extra_Viewer"

```

```

        ConsumerID="{5B04B775-356B-4AA0-AAF8-6491FFE5632}" TaskID="_default" />
<Extension ExtensionName="Photos_Extra_Hub"
        ConsumerID="{5B04B775-356B-4AA0-AAF8-6491FFE5632}" TaskID="_default" />
</Extensions>

```

With this line, the application should now also be available from the apps pivot in the pictures hub.

Launcher and Chooser Enhancements

Windows Phone 7.1 adds a range of enhancements to the original set of Launchers and Choosers. These include:

- Choosing contact information (phone, email, or address) now also returns the contact name.
- The *EmailComposeTask* has additional properties, *Bcc* and *CodePage*, for the BCC recipients and message character set, respectively.
- The *MediaPlayerLauncher* has an additional *Orientation* property, which you can set to specify the orientation of the player when it is launched.
- The *WebBrowserTask* now has an additional *Uri* property (also named *Uri*). This should be used instead of the deprecated *URL* property, which was a simple string.
- The *PhotoResult* object that is returned by the *PhotoChooserTask* now returns the real file name of the selected photo, not just a GUID.

In addition, version 7.1 introduces several completely new Launchers and Choosers, as described in Table 16-5.

TABLE 16-5 New Version 7.1 Launchers and Choosers.

Type	Task	Description
Launchers	<i>BingMapsDirectionsTask</i>	Launches the Bing Maps application, specifying a starting and/or ending location, for which driving or walking directions are displayed.
	<i>BingMapsTask</i>	Launches the Bing Maps application centered at the specified or current location.
	<i>ConnectionSettingsTask</i>	Launches a settings dialog with which the user can change the device's connection settings.
	<i>ShareLinkTask</i>	Launches a dialog with which the user can share a link on the social networks of their choice. If the user does not have any social networks setup, the launcher silently fails.
	<i>ShareStatusTask</i>	Launches a dialog with which the user can share a status message on the social networks of their choice. If the user does not have any social networks setup, the launcher silently fails.
Choosers	<i>AddressChooserTask</i>	Launches the Contacts application with which the user can find an address.
	<i>GameInviteTask</i>	Shows the game invite screen with which the user can invite players to a multiplayer game session.
	<i>SaveContactTask</i>	Launches the contacts application with which the user can save a contact.
	<i>SaveRingtoneTask</i>	Launches the ringtones application with which the user can save a ringtone from your application to the system ringtones list.

Figure 16-10 demonstrates the new *BingMapsDirectionsTask*. You can see this at work in the *NewBingMaps* solution in the sample code.

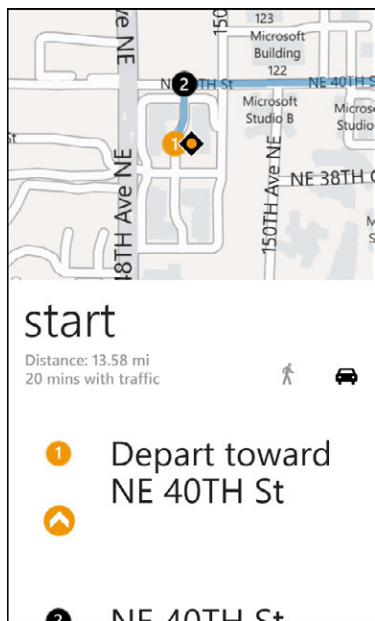


FIGURE 16-10 Using the *BingMapsDirectionsTask* Launcher.

Using this in your application is about as simple as it gets. The first thing to note is that you do not need to set up an *ApplicationIdCredentialsProvider*; in other words, you do not need a Bing Maps development account in order to use the *BingMapsTask* and *BingMapsDirectionsTask*. Next, provide a *Button* or similar mechanism with which the user can trigger the use of the *BingMapsDirectionsTask*. An example implementation is shown in the code that follows. You simply instantiate the task object, and then specify a label and/or *GeoCoordinates* for the target location (the end point). If you supply a location, then the label is used as a descriptive label. If you don't supply a location, then the label is used as a search string. You can optionally also specify a location start point, but if you don't, then the current location is used as the default starting point. Finally, invoke the *Show* method to execute the Launcher.

```
private void getDirections_Click(object sender, RoutedEventArgs e)
{
    BingMapsDirectionsTask bingTask = new BingMapsDirectionsTask();
    bingTask.End = new LabeledMapLocation("Smith Tower", null);
    bingTask.Show();
}
```

For more information about the new Launchers and Choosers that were introduced in version 7.1 for integrating with contacts and the calendar, see Chapter 18, "Data Support."

The *DeviceStatus* and *DeviceNetworkInformation* classes

Recall from Chapter 8, that you used the *DeviceExtendedProperties* class to report information about the device and current memory usage. From version 7.1, you can now also use the new *DeviceStatus* class to report almost the same information. The only difference in the data reported is that *DeviceStatus* provides additional information for the power source and hardware keyboard. Figure 16-11 depicts an application that uses *DeviceStatus* (the *NewDeviceInfo* solution in the sample code). This application also uses the new *DeviceNetworkInformation* class to retrieve information about the current network configuration and availability.

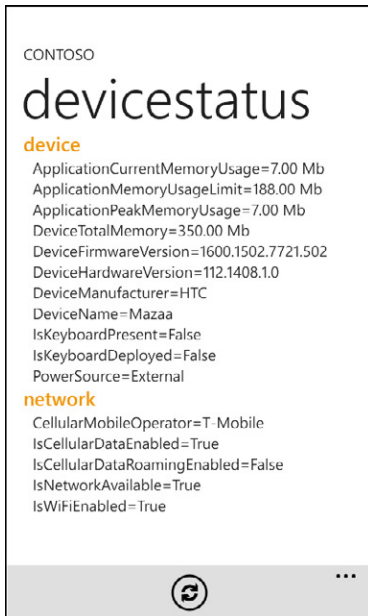


FIGURE 16-11 Using the new *DeviceStatus* class.

In addition to the set of properties, *DeviceStatus* exposes two events, *KeyboardDeployedChanged* and *PowerSourceChanged*, which are raised when the physical keyboard is either opened or closed and when the power source of the device changes, respectively. Similarly, *DeviceNetworkInformation* exposes a *NetworkAvailabilityChanged* event when anything changes, including connection/disconnection for the cellular and WiFi networks, airplane mode, data roaming, and so on.



Important Do not assume that “external power equates to infinite power.” If the device is attached to a low-power USB hub, a low-performing USB charger, car cigarette lighter, or similar low-output device, then the phone might still be discharging even though it’s on external power, because it is using more power than is being supplied by the charger.

In the sample application, you build a collection of the *DeviceStatus* values either when the user taps the App Bar button, or when either of the two events is raised. You do the same for *DeviceNetworkInformation*.

```
private ObservableCollection<String> deviceItems;
private ObservableCollection<String> networkItems;

public MainPage()
{
    InitializeComponent();

    deviceItems = new ObservableCollection<string>();
    deviceList.ItemsSource = deviceItems;
    networkItems = new ObservableCollection<string>();
    networkList.ItemsSource = networkItems;

    DeviceStatus.KeyboardDeployedChanged += DeviceStatus_KeyboardDeployedChanged;
    DeviceStatus.PowerSourceChanged += DeviceStatus_PowerSourceChanged;

    DeviceNetworkInformation.NetworkAvailabilityChanged +=
        DeviceNetworkInformation_NetworkAvailabilityChanged;
}

private void DeviceStatus_PowerSourceChanged(object sender, EventArgs e)
{
    RefreshList();
}

private void DeviceStatus_KeyboardDeployedChanged(object sender, EventArgs e)
{
    RefreshList();
}

private void DeviceNetworkInformation_NetworkAvailabilityChanged(
    object sender, NetworkNotificationEventArgs e)
{
    RefreshList();
}

private void appBarRefresh_Click(object sender, EventArgs e)
{
    RefreshList();
}
```

The *RefreshList* helper method simply retrieves the current value of each of the *DeviceStatus* and *DeviceNetworkInformation* properties, and then adds a corresponding string to the data-bound collection.

```
private void RefreshList()
{
    deviceItems.Clear();
    networkItems.Clear();

    deviceItems.Add(String.Format("ApplicationCurrentMemoryUsage={0:N} Mb",
        DeviceStatus.ApplicationCurrentMemoryUsage / 1024 / 1024));
```



```

deviceItems.Add(String.Format("ApplicationMemoryUsageLimit={0:N} Mb",
    DeviceStatus.ApplicationMemoryUsageLimit / 1024 / 1024));
deviceItems.Add(String.Format("ApplicationPeakMemoryUsage={0:N} Mb",
    DeviceStatus.ApplicationPeakMemoryUsage / 1024 / 1024));
deviceItems.Add(String.Format("DeviceTotalMemory={0:N} Mb",
    DeviceStatus.DeviceTotalMemory / 1024 / 1024));
deviceItems.Add(String.Format("DeviceFirmwareVersion={0}",
    DeviceStatus.DeviceFirmwareVersion));
deviceItems.Add(String.Format("DeviceHardwareVersion={0}",
    DeviceStatus.DeviceHardwareVersion));
deviceItems.Add(String.Format("DeviceManufacturer={0}",
    DeviceStatus.DeviceManufacturer));
deviceItems.Add(String.Format("DeviceName={0}",
    DeviceStatus.DeviceName));
deviceItems.Add(String.Format("IsKeyboardPresent={0}",
    DeviceStatus.IsKeyboardPresent));
deviceItems.Add(String.Format("IsKeyboardDeployed={0}",
    DeviceStatus.IsKeyboardDeployed));
deviceItems.Add(String.Format("PowerSource={0}",
    DeviceStatus.PowerSource));

networkItems.Add(String.Format("CellularMobileOperator={0}",
    DeviceNetworkInformation.CellularMobileOperator));
networkItems.Add(String.Format("IsCellularDataEnabled={0}",
    DeviceNetworkInformation.IsCellularDataEnabled));
networkItems.Add(String.Format("IsCellularDataRoamingEnabled={0}",
    DeviceNetworkInformation.IsCellularDataRoamingEnabled));
networkItems.Add(String.Format("IsNetworkAvailable={0}",
    DeviceNetworkInformation.IsNetworkAvailable));
networkItems.Add(String.Format("IsWiFiEnabled={0}",
    DeviceNetworkInformation.IsWiFiEnabled));
}

```

For new projects, you are encouraged to use *DeviceStatus* rather than *DeviceExtendedProperties*. Indeed, the older mechanism is likely to be deprecated over time.

Version 7.1.1

Version 7.1.1 was announced in February 2012. This is a minor release, based on the 7.1 release, designed specifically to support low-memory devices (devices with only 256 MB of RAM). There is one major change to the platform in this release, and two programmability features. The major change is system paging. The programmability additions consist of a new memory-related property on the *DeviceExtendedProperties* API and a new manifest element.

In Windows Phone 7, you can ascertain the total amount of memory on the device and the amount that your application is currently using by checking the *DeviceTotalMemory* and *ApplicationCurrentMemoryUsage* properties of the *DeviceExtendedProperties* API. These are superseded in version 7.1 by the same properties on the *DeviceStatus* API. Note that *DeviceTotalMemory* is not really useful; you should certainly not use it for performing any calculations about your application and memory. Windows Phone 7.1 introduces the *ApplicationMemoryUsageLimit* property, which indicates

the maximum amount of memory that your application can allocate. Subtracting *ApplicationCurrentMemoryUsage* from *ApplicationMemoryUsageLimit* tells you how much more space you have to grow.

The version 7.1.1 release introduces one new property on the *DeviceExtendedProperties* API, named *ApplicationWorkingSetLimit*. You check this property to see how much physical memory you have available to your application. On Windows Phone 7.0 and 7.1, the working set is always ≥ 90 MB, but on a version 7.1.1 device, it will be some value < 90 MB. You can check the value and then make an informed decision about how to tune your functionality to fit within the available memory.

The key point is that the version 7.1.1 release introduces memory paging. Memory paging is familiar in the desktop PC context, where the amount of physical memory is augmented by a disk-based page file. The operating system can page-out sections of memory to disk to free up real memory, and then bring those pages back into memory when required. This increases the total amount of virtual memory on the system. In versions 7 and 7.1, an application's commit memory is always the same as its working set memory (and always ≥ 90 MB) because there is no paging in those systems. However, in version 7.1.1, the commit memory on a 256 MB device will be ≥ 90 MB (typically ~ 110 MB), but this will always be greater than the working set. The working set will always be < 90 MB, and typically around 50–60 MB, with the gap made up by virtual (paged) memory.

The second developer-related addition in the version 7.1.1 release is the introduction of a new marketplace manifest element. This is optional, and its purpose is to give the developer the ability to opt out of low-memory devices. If you add the new *ID_REQ_MEMORY_90* element to your manifest, two things happen: first, the marketplace certification process will not test your application on 256 MB devices; second, when a user with a 256 MB device attempts to download your application from the marketplace, she will be given a warning to the effect that this application will not work correctly on her phone, and the download will be blocked. The UX is similar to what happens if she attempts to download an application that requires a sensor that is not present on her phone.

```
<Deployment>
  <App>
    ... unchanged elements omitted for brevity.
  </App>
  <Requirements>
    <Requirement Name="ID_REQ_MEMORY_90"/>
  </Requirements>
</Deployment>
```

So, from a developer's perspective, the idea is that you would add this requirement in the following circumstances:

- You application actually requires ≥ 90 MB of memory to function correctly.
- Your application fits within 90 MB of memory, but the 90 MB of (virtual) memory provided on a low-memory device is not reliable enough to provide the expected UX.
- Your application uses a lot of graphics memory (textures). That memory is not tracked as part of your working set, so you could (for example) use only 50 MB of tracked working set memory but perhaps another 50 MB of GPU memory, and thus fail to work correctly on a low-memory device.

The last point bears a little explanation. The problem is that reading from and writing to disk is very time-consuming, so there is a significant performance penalty to paging. In version 7.1.1, when the memory available to applications falls below a certain critical threshold, the system starts paging. This can enable your application to function within the logical 90 MB, but at the price of degraded performance. So, it might be that even though your application fits in 90 MB, the performance degrades so badly under paging that you would prefer for it not to be available on low-memory devices. A badly performing application can negatively affect the user's perception of your application quality, which can lead to bad reviews and propagation of a bad public image. In this scenario, you might be better off restricting your application to devices with ≥ 512 MB memory, where performance is acceptable.

Figure 16-12 shows the *TangoTest* solution in the sample code, running first on a version 7.1 device, and then on a version 7.1.1 device.



FIGURE 16-12 You can check to determine if the application is running on a low-memory device.

The significant code for this is shown below. If your check reveals that you are running on a low-memory device—that is, where working set <90 MB—you would take some mitigating action such as constraining or disabling some of your more memory-intensive functionality. The *ApplicationWorkingSetLimit* will be a long value of less than 90 MB (94371840) if the device is a 256 MB device.

```
private void appBarRefresh_Click(object sender, EventArgs e)
{
    deviceItems.Clear();

    deviceItems.Add(String.Format(
        "OS Version={0}", System.Environment.OSVersion));
    deviceItems.Add(String.Format("ApplicationCurrentMemoryUsage={0:N} Mb",
```

```

        DeviceStatus.ApplicationCurrentMemoryUsage / 1024 / 1024));
deviceItems.Add(String.Format("ApplicationMemoryUsageLimit={0:N} Mb",
    DeviceStatus.ApplicationMemoryUsageLimit / 1024 / 1024));
deviceItems.Add(String.Format("ApplicationPeakMemoryUsage={0:N} Mb",
    DeviceStatus.ApplicationPeakMemoryUsage / 1024 / 1024));
deviceItems.Add(String.Format("DeviceTotalMemory={0:N} Mb",
    DeviceStatus.DeviceTotalMemory / 1024 / 1024));

long workingSetLimit;

try
{
    workingSetLimit = Convert.ToInt64(
        DeviceExtendedProperties.GetValue("ApplicationWorkingSetLimit"));
    deviceItems.Add(String.Format("ApplicationWorkingSetLimit={0:N} Mb",
        workingSetLimit / 1024 / 1024));
}
catch (ArgumentOutOfRangeException)
{
    workingSetLimit = DeviceStatus.ApplicationMemoryUsageLimit;
    deviceItems.Add("Commit memory == Working set");
}

if (workingSetLimit < 94371840)
{
    deviceItems.Add("Low-memory device: now reducing functionality.");
}
}

```

The version 7.1.1 SDK provides a 256 MB version of the emulator, and you can choose which version of the emulator to target from the target device drop-down in Visual Studio.

Note that generic background agents are not supported on version 7.1.1. If you implement agents in your application, then at the point where you attempt to add an agent to the schedule, this will throw an *InvalidOperationException*. This is the same exception you get if you attempt to add an agent on any device that has already reached the limit of the number of allowed agents (15). This means that if your application uses background agents, then it should already be handling this error case. Apart from this, all the performance and memory-optimization techniques discussed in Chapter 14, “Go to Market,” assume greater importance if you target 256 MB devices.

Summary

Windows Phone 7.1 adds to the already rich support for working with built-in applications, with a large number of additional Launchers and Choosers. All of these are almost trivial to use, and help your applications to integrate seamlessly with standard features and services on the phone. However, the most significant improvement in this area is the extensive new support for sensors, both in terms of the number of sensors available, and the provided APIs with which your application can interact. Exposing the raw camera data stream to marketplace application developers opens up a wide range of application scenarios, not least of which is the ability to build compelling augmented-reality applications by using both sensors and camera.

Enhanced Connectivity Features

Windows Phone 7.1 builds on the already rich internet capabilities of its predecessor with significant new connectivity features. There are new tile features, for both local use of tiles and within push notification scenarios. The most exciting addition is the new support for sockets, by which you can build applications that communicate remotely, outside the standard web client or web service models. The tooling support for OData web services is greatly improved, and the use of Bing search is enhanced by providing extensibility points. This last is another example of how the Windows Phone user experience (UX) is very deliberately designed to be integrated, encouraging the model, wherein marketplace applications work seamlessly with the phone's built-in features to give the user the clear impression that the phone is an ecosystem rather than a loose collection of independent applications.

Push, Tile, and Toast Enhancements

In Chapter 12, "Push Notifications," you saw how you can build applications that use push notifications to keep your application data fresh and relevant to the user by using tiles, toasts, and raw notifications. Windows Phone 7.1 adds new push-related features as well as support for local tiles that don't use the push system. Briefly, these include the following:

- You can create multiple tiles per application, each of which can optionally navigate to a different page. Previously, this feature was only available to built-in applications.
- You can create and update tiles locally without making a network round-trip through the push system.
- Tiles now have a back as well as a front side, and automatically flip between these every six seconds or so (the timing is slightly randomized to ensure that the entire screen doesn't flip all at once). Your application can control the back-of-tile images and text.
- You can deep-link to a specific page in toast push notifications.
- You can create toast notifications locally, without a round-trip through the push system; however, this only works from a background agent.
- Miscellaneous reliability and performance improvements.

Local Tiles

With version 7.1, you can create and update one or more tiles without using the push notification system at all. Using this approach, you don't get the benefits of real-time data pushes, but on the other hand, there's no need for a remote server, no network usage, and no unpredictability due to lost updates. Your tiles are purely local: created, updated, and deleted by your client application on the phone. The screenshots in Figure 17-1 show the *LiveTiles* application in the sample code. This has two pages; the *MainPage* simply shows a *HyperlinkButton* that goes to *Page2*. The only purpose of this is to have two pages in the application so that you can see the tile deep-linking at work. *Page2* has a *TextBlock* and three *Button* controls. The *TextBlock* is set to a string that indicates whether the user navigated to this page via a pinned tile on the Start menu or via the *HyperlinkButton* on the *MainPage*.

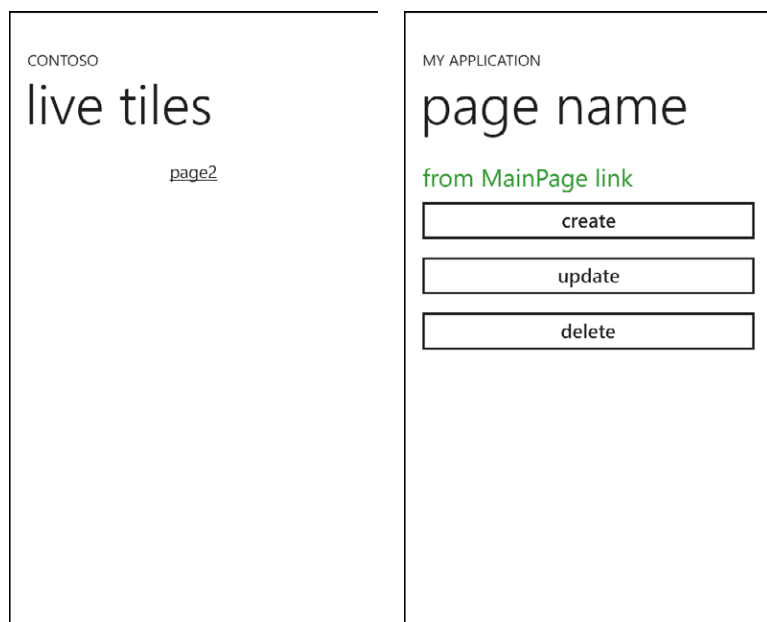


FIGURE 17-1 The *MainPage* (on the left) and *Page2* (right) of the *LiveTiles* sample application.

When the user taps Create, the application generates a secondary tile. When he taps the Update button, the application finds the secondary tile, and updates some of the properties. The Delete button is used to delete the secondary tile, if it exists.



Note This sample illustrates how to use the APIs for creating, updating, and deleting tiles, but note that a realistic application would likely have a more sophisticated user interface (UI) for triggering these operations through user action rather than just simple buttons. Normally, the user would opt to pin a tile to the Start screen that corresponds to some item in your application. Then, the application would update the tile—typically without user interaction—to keep the tile fresh. Finally, deleting the tile would normally not be done in the application code; instead, it would be triggered by the user explicitly unpinning it from the Start screen.

All the interesting work is in the *Page2* link *Click* handlers. You create a tile with the *ShellTile.Create* method, which takes a *StandardTileData* parameter. This is a simple class through which you set the properties of the tile, including images for the front and back, title text for the front and back, a count value for the front, and a longer content string for the back. You must use JPG or PNG images, sized at 173x173 pixels for both front and back. If you use PNG, the images can include transparency. When you create the tile, you must use local images; if you don't, your tile will disappear on a reboot. After the tile is created, for any subsequent updates, you can use either local or remote images.



More Info A common pattern is to generate images dynamically according to some run-time context, and then persist the images by using the *WriteableBitmap* class. The *System.Windows.Media.Imaging* namespace includes extension methods for this class to save and load JPG format images, but not PNG format. If you want to generate images on the fly, and you want to include transparency, you can consider using the *WriteableBitmapEx* third-party library, available on codeplex at <http://writeablebitmapex.codeplex.com/>.

```
private void createTile_Click(object sender, RoutedEventArgs e)
{
    StandardTileData tileData = new StandardTileData
    {
        BackgroundImage = new Uri("bananas_173x173.png", UriKind.Relative),
        Title = "Monkey Tile",
        Count = 1,
        BackTitle = "Back of Tile",
        BackContent = "Bananas today!",
        BackBackgroundImage = new Uri("monkey_173x173.png", UriKind.Relative)
    };

    ShellTile.Create(new Uri("/Page2.xaml?ID=MonkeyTile", UriKind.Relative), tileData);
    UpdateButtons();
}
```

Figure 17-2 shows the front and back of this secondary tile just after it's first created (that is, before updating the tile).

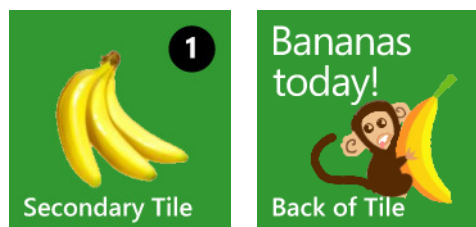


FIGURE 17-2 Front (on the left) and back (right) of the secondary tile when first created.

In addition to the *StandardTileData*, you also specify the *NavigationUri* for this tile. This must include the page to which to navigate within this application, plus optionally a query string with whatever parameters you want. In this example, you're simply setting one *ID* parameter, which you can use subsequently to determine which tile this is. When you call the *ShellTile.Create* method, the tile is created with the specified properties and pinned to the Start page on the phone. The Start page is an application that is part of the system shell, so this action causes a navigation away from your application, which is therefore deactivated. The reason for this is to avoid spamming the Start page: when you create a tile, the system makes it very obvious to the user that the tile has been created, and the user is shown where the tile is. He can then immediately interact with it, perhaps by moving it around, or deleting it if he doesn't want it.

Having created the tile, you also update the buttons. This ensures that in this application the user cannot create the same tile more than once, and that the buttons for updating and deleting the tile are only enabled if there is in fact a tile to update or delete. There's only one secondary tile in this application, so you can cache this as a *ShellTile* field in the class. If there were more tiles, it would make sense to use a collection, instead.

```
private ShellTile tile;

private void UpdateButtons()
{
    if (tile == null)
    {
        createTile.IsEnabled = true;
        updateTile.IsEnabled = false;
        deleteTile.IsEnabled = false;
    }
    else
    {
        createTile.IsEnabled = false;
        updateTile.IsEnabled = true;
        deleteTile.IsEnabled = true;
    }
}
```

There are two ways to get to *Page2* in the application: through normal navigation via the hyperlink on the main page of the application, or via a pinned tile on the Start page. So that you can determine

which route was taken, you need to override the *OnNavigatedTo* method. This is where the *ID* parameter comes into play; the application can examine the query string to see which tile the user tapped to get to this page. This is also where you cache the *ShellTile* object. Finally, you can update the buttons so that they always correctly reflect the state of the secondary tile whenever the user navigates to this page.

```
protected override void OnNavigatedTo(NavigationEventArgs e)
{
    String tmp;
    if (NavigationContext.QueryString.TryGetValue("ID", out tmp))
    {
        navigation.Text = String.Format("from Start ({0})", tmp);
    }
    else
    {
        navigation.Text = "from MainPage link";
    }

    tile = ShellTile.ActiveTiles.FirstOrDefault(
        x => x.NavigationUri.ToString().Contains("ID=MonkeyTile"));
    UpdateButtons();
}
```

In this example, if the query string indicates that the user arrived at *Page2* via a pinned tile, you simply extract the parameter value and display it in a *TextBlock*. In a more sophisticated application, you would use this identifier to govern some business logic in your solution.

Updating a tile's properties and deleting a tile are both very straightforward. To update a tile, you simply find that tile and invoke the *Update* method, passing in a replacement set of data by using the *StandardTileData* class, as before. You don't have to provide values for all the properties, because any properties for which you don't provide values will simply retain their previous value. If you want to clear a value, you can simply provide an empty string or a Uri with an empty string, depending on the item to be cleared. To delete a tile, simply find the tile and invoke the *Delete* method, but remember that in a real application, you should normally leave tile deletion to the user and avoid doing this programmatically. Note that neither updating nor deleting need to send the user to the Start page, so there is no navigation away from the application in these cases.

```
private void updateTile_Click(object sender, RoutedEventArgs e)
{
    ShellTile tile = ShellTile.ActiveTiles.FirstOrDefault(
        x => x.NavigationUri.ToString().Contains("ID=MonkeyTile"));
    if (tile != null)
    {
        StandardTileData NewTileData = new StandardTileData
        {
            Count = DateTime.Now.Second,
            BackContent = DateTime.Now.ToLongTimeString(),
        };

        tile.Update(NewTileData);
    }
}
```

```
private void deleteTile_Click(object sender, RoutedEventArgs e)
{
    ShellTile tile = ShellTile.ActiveTiles.FirstOrDefault(
        x => x.NavigationUri.ToString().Contains("ID=MonkeyTile"));
    if (tile != null)
    {
        tile.Delete();
    }
}
```

Pinning Tiles

The ability to pin local tiles to the Start page becomes more interesting when the user can choose from multiple possible tiles to pin within an application. Consider a typical weather application; the user can mark individual locations as favorites, and can then pin zero or more of these favorites to the Start page.

The screenshots in Figure 17-3 shows the *PinTiles* solution from the sample code. This is based on the standard Microsoft Visual Studio *Databound Application* project type.



FIGURE 17-3 The *MainPage* (on the left) and *DetailsPage* (right) of the *PinTiles* sample application.

The idea here is that the application presents a list of items on the *MainPage*, and when the user taps one of these items, it navigates to the *DetailsPage*, which is data-bound to that item's viewmodel. In addition, the UI presents an App Bar *Button* control which displays a "pin" image. When the user taps this control, the application creates a local tile and pins it to the Start page. In this way, the user can tap multiple items and pin them all to Start, as shown in Figure 17-4. When the user taps one of

the pinned tiles, this launches the application and navigates her to that page, with the corresponding data loaded.



Note Some applications in the marketplace adopt the practice of providing an “unpin” feature in addition to the pin feature. As you’ve seen from the previous example, the platform API does expose methods for programmatically deleting (and therefore, unpinning) secondary tiles. However, this is not strictly compliant with Metro guidelines. This is one of those guidelines that is not rigidly enforced; you can provide this kind of behavior, but it is not really in the spirit of Metro. In the Metro world, the UX is always very predictable and very simple. The user knows that she can always unpin any tile that she’s pinned to the Start page, and that this is the standard approach for doing this. While an application could provide an alternative mechanism for unpinning tiles, there’s really no need.

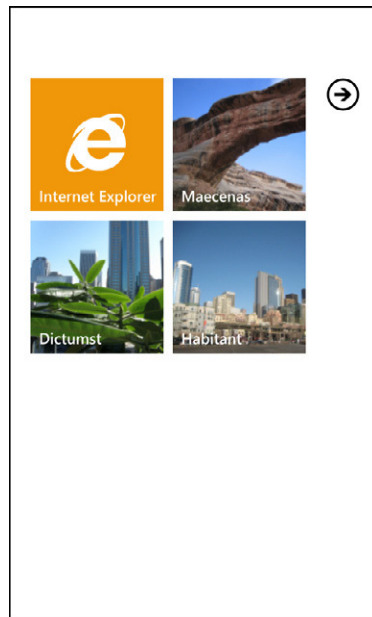


FIGURE 17-4 Pinning multiple tiles from the same application.

The viewmodel for each item is very simple: just a URI for the image, and two strings.

```
public class ItemViewModel
{
    public Uri Photo { get; set; }
    public String Title { get; set; }
    public String Details { get; set; }
}
```

The *MainViewModel* is exposed as a static property of the *App* class, and some dummy data is loaded when this property is first accessed. See Chapter 4, “Data Binding,” (or examine the sample code) for details of this design. The *ListBox.SelectionChanged* handler in the *MainPage* navigates to the *DetailsPage*, and then passes in a query string which includes an identifier for the selected item.

```
private void PhotoList_SelectionChanged(object sender, SelectionChangedEventArgs e)
{
    if (PhotoList.SelectedIndex != -1)
    {
        String targetUrl = String.Format(
            "/DetailsPage.xaml?Title={0}",
            ((ItemViewModel)PhotoList.SelectedItem).Title);
        NavigationService.Navigate(new System.Uri(targetUrl, UriKind.Relative));
        PhotoList.SelectedIndex = -1;
    }
}
```

All the interesting work is in the *DetailsPage*. In the XAML, you put an *Image* control for the item photo and a *TextBlock* inside a *ScrollView* (to allow for large amounts of text in the item’s *Details* property). The *Image* and the *TextBlock* are data-bound to the item properties. The App Bar has one button that displays a “pin” image. This will be conditionally enabled, depending on whether the user has already pinned this item to the Start page.

```
<Grid x:Name="LayoutRoot" Background="Transparent" d:DataContext="{Binding Items[0]}">
...

    <ScrollView
        Grid.Row="1" Margin="12,0,12,0"
        VerticalScrollBarVisibility="Auto" ManipulationMode="Control">
        <StackPanel >
            <Image
                Height="300" Source="{Binding Photo}"
                Stretch="UniformToFill" Margin="12,0,0,0"/>
            <Grid Height="12"/>
            <TextBlock
                Text="{Binding Details}" TextWrapping="Wrap"
                Margin="{StaticResource PhoneHorizontalMargin}" />
        </StackPanel>
    </ScrollView>
</Grid>

<phone:PhoneApplicationPage.ApplicationBar>
    <shell:ApplicationBar IsVisible="True" Opacity="0.8">
        <shell:ApplicationBarIconButton
            x:Name="appBarPin" IconUri="/Images/Pin.png" Text="pin to start"
            Click="appBarPin_Click"/>
    </shell:ApplicationBar>
</phone:PhoneApplicationPage.ApplicationBar>
```

To accommodate this pinning behavior, you override the *OnNavigatedTo* method. First, you need to figure out to which item to data-bind, based on the query string parameters. In the process, you formulate a string that you can use later as the URI for this page—this is cached in a field object so

that you can use it across methods. You also need to determine if there's an active tile for this item. If so, disable the App Bar button; otherwise, you need to enable it.

```
protected override void OnNavigatedTo(NavigationEventArgs e)
{
    itemTitle = NavigationContext.QueryString["Title"];
    var pinnedItem = App.ViewModel.Items.FirstOrDefault(x => x.Title == itemTitle);
    if (pinnedItem != null)
    {
        DataContext = thisItem = pinnedItem;
    }

    thisPageUri = String.Format("/DetailsPage.xaml?Title={0}", itemTitle);
    tile = ShellTile.ActiveTiles.FirstOrDefault(
        x => x.NavigationUri.ToString().Contains(thisPageUri));

    appBarPin = ApplicationBar.Buttons[0] as ApplicationBarIconButton;
    if (tile != null)
    {
        appBarPin.IsEnabled = false;
    }
    else
    {
        appBarPin.IsEnabled = true;
    }
}
```

In the *Click* handler for the Pin button, create the tile. In this method, you can rely on the fact that you've already performed the search for the tile in the *OnNavigatedTo* override, and cached the tile if it was found in the collection. If the cached tile is null (that is, it doesn't exist in the collection), go ahead and create it now by using the current item's *Photo* property and *Title* property as the *BackgroundImage* and *Title* for the tile.

```
private void appBarPin_Click(object sender, EventArgs e)
{
    StandardTileData tileData = new StandardTileData
    {
        BackgroundImage = thisItem.Photo,
        Title = thisItem.Title,
        BackTitle = "Lorem Ipsum!",
        BackBackgroundImage =
            new Uri("Images/monkey_173x173.png", UriKind.Relative)
    };
    ShellTile.Create(
        new Uri(thisPageUri, UriKind.Relative), tileData);
}
```

By doing this, the user can pin multiple tiles, with individual control over each tile and appropriate UI feedback (the pin button is conditionally enabled) so that it's clear what the pinned state of each item is. When the user taps a pinned tile, the application launches and navigates to the item page specified in the tile's *NavigationUri*. This does not go through the *MainPage*; therefore, if the user then taps the Back button, there are no more pages for this application in the navigation backstack, so the application will terminate.

There is an alternative UX model whereby the user can always return to the main page—and therefore, to the rest of the application—regardless of whether he launched the application from Start in the normal way or from a pinned tile. This model uses a “home” button, but you should use it with care because it varies from the normal expected behavior. Normally, the user model of a home button is not commonly employed, because it can result in a confusing navigation experience. However, the pinned tile technique gives the user two different ways to start the application. It can justify the decision, ensuring that he can always navigate from the individual item page back to the rest of the application.



Note The practice of providing a “home” feature as demonstrated in this sample is not strictly compliant with Metro guidelines. In general, you should avoid this usage pattern and assume by default that you do not need a Home button, unless you can prove to yourself otherwise. A good rule of thumb is to use the built-in applications as your inspiration. For example, with the People Hub, you can pin individual people to the Start page, but it does not provide a home button. The same is true of Music & Videos, and so on.

To implement this behavior, you can add a second App Bar button to the *DetailsPage*, as shown in Figure 17-5.



FIGURE 17-5 Adding a Home button.

You implement the *Click* handler for the Home button to navigate to the main page.

```
private void appBarHome_Click(object sender, EventArgs e)
{
    NavigationService.Navigate(new System.Uri("/MainPage.xaml", UriKind.Relative));
}
```

This ensures that the user can always consistently navigate from an item page back to the main page. However, it now introduces a different problem. Consider this scenario: the user navigates from the Start page, through a pinned tile to an item page, and then on to the main page. This means that pressing Back from the main page will go back to the item page. This is not what the user normally expects: her normal expectation is that pressing Back from the application's main page always exits the application. Fortunately, it is very easy to fix this. All you need do is to ensure that the main page is always the top of the page stack for this application by overriding the *OnNavigatedTo* in the main page to clear the in-application page navigation stack, as shown in the following:

```
protected override void OnNavigatedTo(NavigationEventArgs e)
{
    while (NavigationService.CanGoBack)
    {
        NavigationService.RemoveBackEntry();
    }
}
```

In addition to cleaning up the backstack, you should also ensure that the Home button is not available unless the user has navigated to the page via the corresponding pinned tile on the Start page. Using normal navigation within the application, the Home button is unnecessary. Enforcing this behavior helps to maintain the UX, where home navigation is a recognizable exception to the normal navigation behavior, and that it applies only in the specific scenario of a pinned tile. Although you can programmatically set the *IsEnabled* state of an *AppBarIconButton*, this class does not expose a *Visibility* property, as do regular controls. Disabling the button is not really good enough—under normal navigation, you should not make this button available at all, not even in a disabled state. This means that you need to implement this additional button programmatically, and not in XAML. To do this, you can update the *OnNavigatedTo* method. First, when creating the tile, enhance the page Uri to include an additional parameter that indicates that the user navigated to this page from a pinned tile. Then, you can check if the current *NavigationContext* does include this parameter in the query string. If so, you can create the additional Home button, and add it to the App Bar.

```
private AppBarIconButton appBarHome;

protected override void OnNavigatedTo(NavigationEventArgs e)
{
    itemTitle = NavigationContext.QueryString["Title"];
    var pinnedItem = App.ViewModel.Items.FirstOrDefault(x => x.Title == itemTitle);
    if (pinnedItem != null)
    {
        DataContext = thisItem = pinnedItem;
    }
}
```

```

thisPageUri = String.Format("/DetailsPage.xaml?Title={0}&Nav=FromPinned", itemTitle);
tile = ShellTile.ActiveTiles.FirstOrDefault(
    x => x.NavigationUri.ToString().Contains(thisPageUri));

appBarPin = ApplicationBar.Buttons[0] as ApplicationBarIconButton;
if (tile != null)
{
    appBarPin.IsEnabled = false;
}
else
{
    appBarPin.IsEnabled = true;
}

// Did the user get here from a pinned tile?
if (NavigationContext.QueryString.ContainsKey("Nav"))
{
    appBarHome = new ApplicationBarIconButton();
    appBarHome.Text = "home";
    appBarHome.IconUri = new Uri("/Images/Home.png", UriKind.Relative);
    appBarHome.Click += appBarHome_Click;
    ApplicationBar.Buttons.Add(appBarHome);
}
}

```

Finally, note that this is one scenario in which it can be useful during debugging to change your WMAppManifest file to have the application launched with a specific page and query string, as opposed to the default page. This is a debugging technique, and you must remember to remove the fake navigation before submitting your application to the marketplace.

```

<Tasks>
  <!--<DefaultTask Name="_default" NavigationPage="MainPage.xaml" />-->
  <DefaultTask Name="_default" NavigationPage="DetailsPage.xaml?Title=Dictumst&
    Nav=FromPinned"/>
</Tasks>

```

Push Enhancements

In Chapter 12, you saw how to build both the client and server applications for a push notification solution in order to send and receive raw, toast, and tile notifications. The same back-of-tile features used in local tiles can also be used in tile notifications. In addition, toast notifications include a new field: the URL for the target page. With a secondary local tile, when the user taps the tile, it takes him to the corresponding page. In the same way, when the user receives a toast notification that includes a deep-link URL, when he taps the toast, it takes him to that page and not to the application's default page, as it does in version 7.

The *PnServer_Mango* and *PnClient_Mango* solutions in the sample code are an adaptation of the same samples in Chapter 12, with the additional version 7.1 features added. The client application also incorporates most of the UI functionality of the earlier *PinTiles* sample in this chapter. Figure 17-6 presents the enhanced server application.

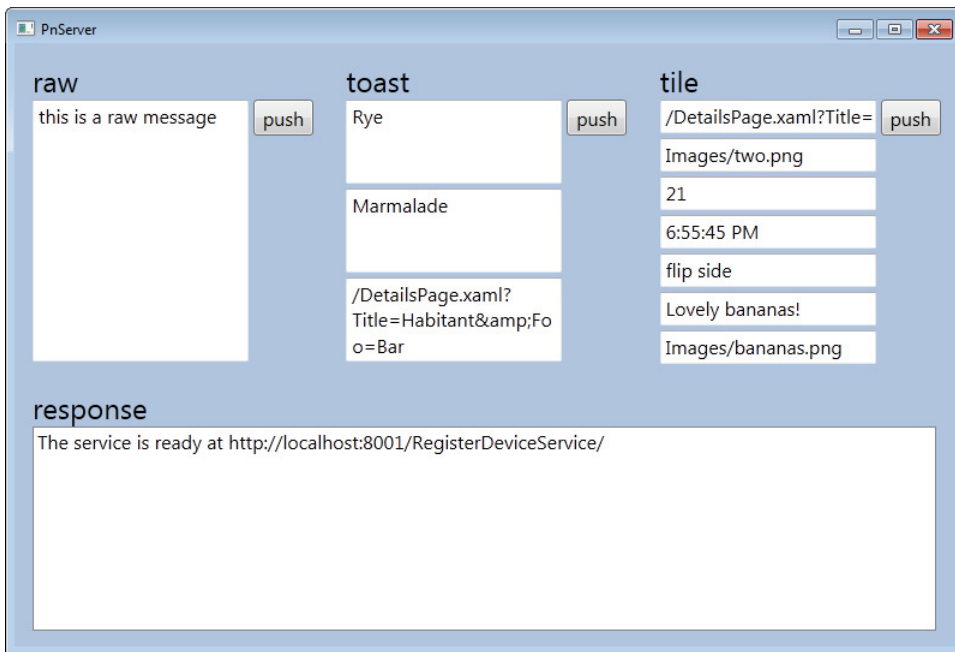


FIGURE 17-6 The push notification server with version 7.1 features.

Figure 17-6 demonstrates that the toast notification includes an extra field for the target URL. Figure 17-7 shows the client application running. Here, the incoming toast notification is reported in a *ListBox* at the bottom of the main page. Contrast this with Figure 17-8, which shows the Start page when a toast notification is received. This looks exactly the same as in version 7, displaying the application's icon and the incoming toast *Title* and *Message* values. The difference is that when the user taps the toast, this will navigate to the target URL specified in the toast notification payload. Any additional parameters in the query string will be passed in to the page in the *NavigationContext*.

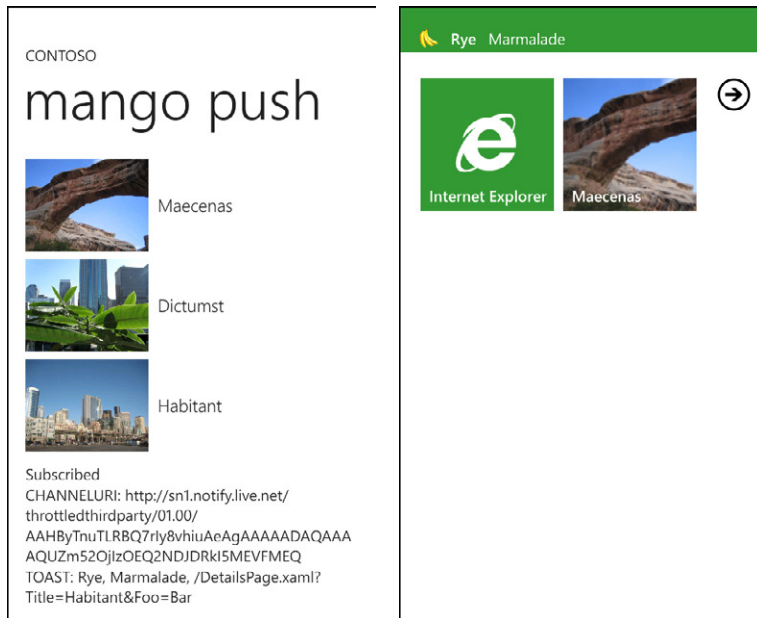


FIGURE 17-7 The client application receiving a toast notification while running (on the left), and the phone receiving a toast while the client application is not running (right).

Looking at the server-side code, the first change is in the toast notification string template. The additional field is the `<wp:Param/>` element. This would be set to the target URL, including query string parameters, if required, such as in the following:

```
"/DetailsPage.xaml?Title=Habitant&Foo=Bar"
```

You can pass whatever parameter key-value pairs make sense in your application. Keep in mind that you must escape the ampersand parameter delimiter, and specify it as `"&";`. The custom helper method to send the toast is updated to include this additional field. As before, this uses a custom `SendNotification` method, which is unchanged from the 7 version.

```
const String toastMessageFormat =
    "<?xml version=\"1.0\" encoding=\"utf-8\"?>" +
    "<wp:Notification xmlns:wp=\"WPNotification\">" +
    "  <wp:Toast>" +
    "    <wp:Text1>{0}</wp:Text1>" +
    "    <wp:Text2>{1}</wp:Text2>" +
    "    <wp:Param>{2}</wp:Param>" +
    "  </wp:Toast>" +
    "</wp:Notification>";

private void sendToast_Click(object sender, RoutedEventArgs e)
{
    String message = String.Format(
        toastMessageFormat, toastTitle.Text, toastMessage.Text,
        toastUrl.Text);
    ShowStatus(SendNotification(message, 2));
}
```

To be clear, when the user taps the toast, given the preceding URL value, she will navigate to the *DetailsPage* with the *Habitant* item loaded. The fact that there is a pinned tile for the *DetailsPage* with the *Maecenas* item is irrelevant at this point. On the client side, the *DetailsPage* code is exactly the same as in the earlier *PinTiles* sample. The existing *OnNavigatedTo* override, which was designed to meet the needs of navigation via local tile, also meets the needs of navigation via URL-targeted toast notification—and of course, navigation via a tile that might or might not have been updated via a push notification. The only update on the client side for this toast is to extract the additional query string values and display them in the UI, for circumstances in which the toast is received when the application is actually running.

Back in the server code, the tile notification template string is also updated for version 7.1. It is consistent with the additional back-of-tile and *ID* fields that are used for local tiles. Be aware that the *ID* is not an independent element; rather, it is an attribute of the *Tile* element.

```
const String tileMessageFormat =
    "<?xml version=\"1.0\" encoding=\"utf-8\"?>" +
    "<wp:Notification xmlns:wp=\"WPNotification\">" +
        "<wp:Tile ID=\"{0}\">" +
            "<wp:BackgroundImage>{1}</wp:BackgroundImage>" +
            "<wp:Count>{2}</wp:Count>" +
            "<wp:Title>{3}</wp:Title>" +
            "<wp:BackTitle>{4}</wp:BackTitle>" +
            "<wp:BackContent>{5}</wp:BackContent>" +
            "<wp:BackBackgroundImage>{6}</wp:BackBackgroundImage>" +
        "</wp:Tile>" +
    "</wp:Notification>";

private void sendTile_Click(object sender, RoutedEventArgs e)
{
    tileTitle.Text = DateTime.Now.ToLongTimeString();
    String message = String.Format(
        tileMessageFormat,
        tileId.Text,
        tileBackground.Text, tileCount.Text, tileTitle.Text,
        tileBackTitle.Text, tileBackContent.Text, tileBackBackground.Text);
    ShowStatus(SendNotification(message, 1));
}
```

Tile notifications are never handled explicitly by your client phone application, so there is no additional client code needed to handle the version 7.1 additions. Figure 17-8 shows what happens when the enhanced tile notification is received. In this scenario, the user has pinned two tiles: one for the *Maecenas* item on the *DetailsPage*, and one for the application itself. You have received tile notifications for both tiles.

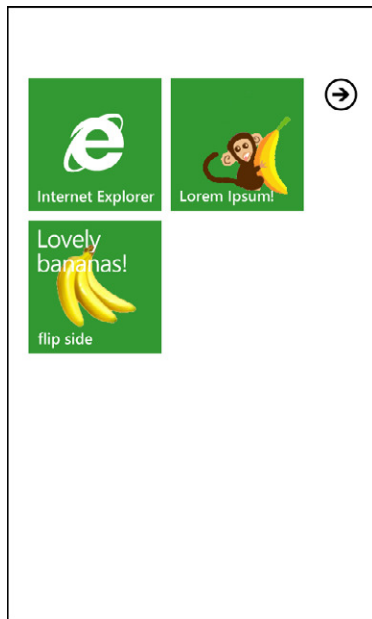


FIGURE 17-8 Receiving enhanced tile notifications.

In this example, you have a *DetailsPage* tile pinned for the *Maecenas* item. If you send a tile notification that includes this corresponding tile *ID*—that is `"/DetailsPage.xaml?Title=Maecenas"`—then the tile information will be used to update that specific tile. If the tile is not pinned, then the notification is simply suppressed in exactly the same way that a tile notification is suppressed when sent to a version 7 application that is not pinned to Start. If you send a tile notification and omit the *ID*, this will update the default tile—again, assuming that tile is pinned to Start.

Sockets

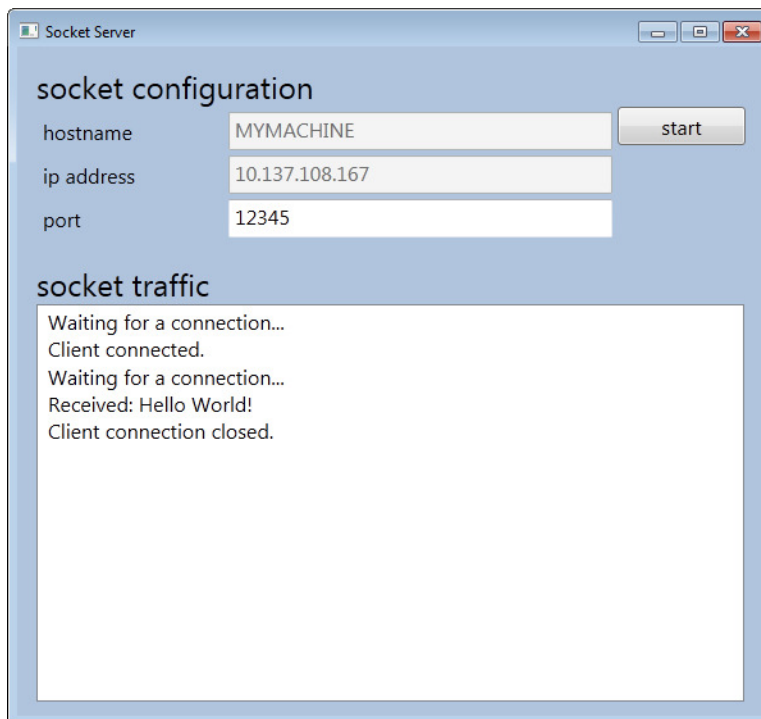
Windows Phone 7.1 introduces support for sockets programming. Using sockets, you can build applications that communicate remotely to other socket-enabled devices, including servers, desktop computers, and other mobile devices. The classic example of a socket-based application on the desktop is the remote peer-to-peer chat application. However, this would be difficult to build for the phone, because the phone can't really act as a server. Another classic example that is more relevant is a multi-player game. Email, streaming multimedia, and file transfer applications also often use sockets. You can even use sockets to build applications that convert your phone into a remote control for your computer. Both Transmission Control Protocol (TCP) and User Datagram Protocol (UDP) sockets are supported, and the differences are summarized in Table 17-1.

TABLE 17-1 A Comparison of TCP and UDP Sockets

Item	TCP	UDP
Connection type	<i>Connection-oriented</i> You must establish a connection before transmitting data.	<i>Connectionless</i> You do not set up a connection before transmitting data.
Transmission type	<i>Stream-based</i> Data is guaranteed to be received in the order it was sent.	<i>Datagram-based</i> Individual packets of data can arrive in any order.
Reliability	<i>Reliable</i> This doesn't mean that all data sent is guaranteed to be received. This is because the recipient device could lose power or network connectivity, which could result in lost packets. Rather, it means that you get ACKs when the data is received, so you can attempt to resend any that were lost.	<i>Unreliable</i> Some packets might be lost in transmission, and there is no built-in way to identify that this has happened.
Suitable applications	Email, file transfer, remote control.	Streaming media, online gaming, VoIP.
Modes	Unicast only (specific client and server endpoints must be configured for each connection).	Unicast or Multicast (where multiple endpoints can join a multicast group).

TCP Sockets

The *SimpleSockets* solutions (*SocketServer* and *SocketClient*) in the sample code demonstrate the most clear-cut implementation of a TCP unicast socket system. The server is a Windows Presentation Foundation (WPF) application that starts a socket listening, and the client is a Windows Phone application that sends text messages to the server. When the server receives a message, it logs it in the UI. First, you will examine the server, which is shown in Figure 17-9.

**FIGURE 17-9** A simple socket server.

In the *MainWindow* constructor, you first create and configure a socket. In a more sophisticated application, you might want to do this lazily and in a background thread rather than in the constructor. However, this is less critical for the server application, which is running on a desktop or server computer. Windows Phone only supports IPv4 sockets, but resolving the host name to an *IPHostEntry* object gives you a collection of addresses for this computer (multiple addresses, including IPv6 addresses), so you need to enumerate the collection to find the first IPv4 value. You're also using an arbitrary port number here. Note that the host name for the current computer is obtained by using *Dns.GetHostName*, and you derive the IP address from this, so these fields in the UI are read-only. On the other hand, the port number is user-editable. You then display the socket configuration properties in the UI. This is useful for testing purposes, especially because the client will need to know the endpoint that you're using. Realistically, of course, there would be some other way for the client to get this information (hard-coded, retrieved from a web service, and so on).

```
private Socket listener;
private IPAddress ipAddress;
private IPEndPoint endPoint;
private static AutoResetEvent done = new AutoResetEvent(false);

public MainWindow()
{
    InitializeComponent();

    listener = new Socket(
        AddressFamily.InterNetwork, SocketType.Stream, ProtocolType.Tcp);
    String hostName = Dns.GetHostName();
    IPHostEntry hostEntry = Dns.GetHostEntry(hostName);
    ipAddress = hostEntry.AddressList.First(
        (ip) => ip.AddressFamily == AddressFamily.InterNetwork);

    hostnameText.Text = hostName;
    ipAddressText.Text = ipAddress.ToString();
    portText.Text = 12345.ToString();
}
```

When the user clicks the Start button, you spin up a thread from the system threadpool to do the work, thus ensuring that you don't block the UI. Note that even though Chapter 14, "Go to Market," discusses threading in the context of Windows Phone applications, most of the details are equally relevant to WPF. The *StartListening* worker method first binds the socket to the endpoint, and puts the socket into a listening state. The parameter to the *Listen* method specifies the number of incoming connections that you will accept—in this case, just one.

You're using a connection-oriented TCP channel, so use the *BeginAccept* method to process incoming connection attempts asynchronously. Doing this asynchronously means that the operations to send and receive data are carried out on a separate thread. You must create a callback method that implements the *AsyncCallback* delegate and pass its name to the *BeginAccept* method, and you must also pass the listening socket object itself.

Calling *WaitOne* on the *AutoResetEvent* will block this worker thread until the event is signaled, which you will do later when you accept a connection. So, the effect here is that the thread will sit there waiting for connections until it gets one. As you go through, you report the received data and any interesting status changes or error conditions by adding a string to the *ListBox*.

```
private void start_Click(object sender, RoutedEventArgs e)
{
    endPoint = new IPEndPoint(ipAddress, Int32.Parse(portText.Text));
    ThreadPool.QueueUserWorkItem(StartListening);
    start.IsEnabled = false;
}

private void StartListening(object stateInfo)
{
    try
    {
        listener.Bind(endPoint);
        listener.Listen(1);

        while (true)
        {
            ShowStatus("Waiting for a connection...");
            listener.BeginAccept(ConnectionReceived, listener);
            done.WaitOne();
        }
    }
    catch (Exception ex)
    {
        ShowStatus(ex.ToString());
    }
}

private void ShowStatus(String message)
{
    Dispatcher.BeginInvoke((Action)(() => trafficList.Items.Add(message)));
}
```

The asynchronous callback extracts the *Socket* object from the result and invokes the *EndAccept* method. When you call *BeginAccept*, the system uses another thread to execute the specified callback method, and blocks on *EndAccept* until a pending connection is retrieved. *EndAccept* will return a new *Socket* object that you can use to send and receive data on the channel.

At this point, you signal the *AutoResetEvent* so that you can start listening again for another connection. Meanwhile, on this connection, you start receiving the sent message bytes—again, this is done asynchronously in the specified callback method. Observe that you’re passing a custom *StateObject* parameter: this neatly encapsulates all the properties you want to be able to access in the callback method.

```

private void ConnectionReceived(IAsyncResult result)
{
    Socket listener = (Socket)result.AsyncState;
    Socket socket = listener.EndAccept(result);
    ShowStatus("Client connected.");
    done.Set();
    StateObject state = new StateObject();
    state.Socket = socket;
    socket.BeginReceive(
        state.Buffer, 0, state.Buffer.Length,
        SocketFlags.None, DataReceived, state);
}

public class StateObject
{
    public Socket Socket;
    public byte[] Buffer = new byte[32];
    public StringBuilder Data = new StringBuilder();
}

```

You finally process the received message, adding the text to the state object's buffer. This server implements a primitive mechanism for checking to see if you've received all the bytes sent (you simply test for a dummy end-of-file marker [^Z]). If you haven't yet received this marker, you need to receive more bytes, so invoke the asynchronous callback again. When you do finally receive all the bytes sent, shutdown the connection and close the socket.

```

private void DataReceived(IAsyncResult result)
{
    StateObject state = (StateObject)result.AsyncState;
    Socket socket = state.Socket;

    int bytesRead = socket.EndReceive(result);
    if (bytesRead > 0)
    {
        state.Data.Append(Encoding.UTF8.GetString(state.Buffer, 0, bytesRead));
        String dataString = state.Data.ToString();
        if (dataString.IndexOf("^Z") <= -1)
        {
            socket.BeginReceive(
                state.Buffer, 0, state.Buffer.Length,
                SocketFlags.None, DataReceived, state);
        }
        else
        {
            ShowStatus(String.Format("Received: {0}", dataString.Replace("^Z", "")));
            socket.Shutdown(SocketShutdown.Both);
            socket.Close();
            ShowStatus("Client connection closed.");
        }
    }
}

```

Figure 17-10 shows the client side of this communication.

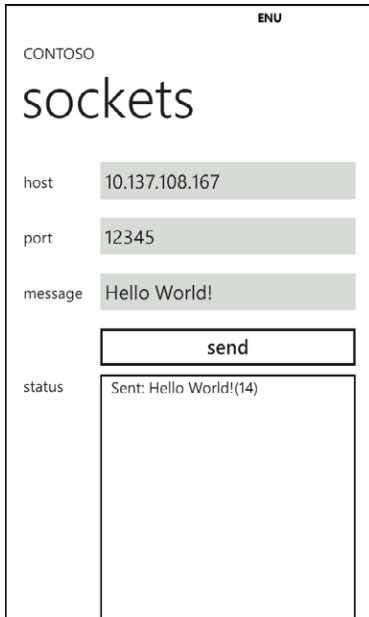


FIGURE 17-10 A simple socket client.

In the client application, you set up a collection of strings to data-bind to the *ListBox*, and a custom *ShowStatus* method to add status and error messages to this collection. You also define a *CloseSocket* method to perform due diligence clean-up operations.

```
private ObservableCollection<String> statusItems = new ObservableCollection<string>();
public ObservableCollection<String> StatusItems
{
    get { return statusItems; }
    private set { }
}

private void ShowStatus(String message)
{
    Dispatcher.BeginInvoke((Action)(() => StatusItems.Add(message)));
}

public MainPage()
{
    InitializeComponent();
    statusList.ItemsSource = StatusItems;
}

private void CloseSocket(SocketAsyncEventArgs e)
{
    if (e.ConnectSocket != null)
    {
        e.ConnectSocket.Shutdown(SocketShutdown.Both);
        e.ConnectSocket.Close();
    }
}
```

When the user taps the Send button, you extract the host and message values from the *TextBox* controls. Next, add the primitive message delimiter (^Z) to mark the end of the message, and then convert to a UTF8 byte array, ready for sending. The *SocketAsyncEventArgs* class is critical to the asynchronous pattern used by socket applications. The basic pattern is as follows:

- Create a new *SocketAsyncEventArgs* object and set its properties, including hooking up a handler for the asynchronous *Completed* event and assigning the message buffer contents.
- Call the specific socket method you want for the current operation; that is, *ConnectAsync*, *ReceiveAsync*, or *SendAsync* for connect, receive, or send operations, respectively.
- If the *XXXAsync* method returns *true*, the I/O operation is now pending, and the *Completed* event will be raised when the operation completes.
- On the other hand, if the *XXXAsync* method returns *false*, this means that the operation completed synchronously.

So, in the first instance, you create a *Socket* object, configured appropriately to match the server's socket, and then invoke *ConnectAsync*. If the operation completed synchronously, you simply go ahead and invoke your asynchronous handler anyway—this way, the same method is called, regardless of whether the operation completes synchronously or asynchronously.

```
private void send_Click(object sender, RoutedEventArgs e)
{
    IPAddress ipAddress = IPAddress.Parse(hostText.Text);
    IPEndPoint endPoint = new IPEndPoint(ipAddress, Int32.Parse(portText.Text));
    String message = String.Format("{0}^Z", messageText.Text);
    byte[] buffer = Encoding.UTF8.GetBytes(message);

    SocketAsyncEventArgs socketArgs = new SocketAsyncEventArgs();
    socketArgs.RemoteEndPoint = endPoint;
    socketArgs.Completed += SocketAsyncEventArgs_Completed;
    socketArgs.SetBuffer(buffer, 0, buffer.Length);

    Socket socket = new Socket(
        AddressFamily.InterNetwork, SocketType.Stream, ProtocolType.Tcp);
    bool isAsync = socket.ConnectAsync(socketArgs);
    if (!isAsync)
    {
        SocketAsyncEventArgs_Completed(socketArgs.ConnectSocket, socketArgs);
    }
}
```

The *Completed* event handler will be invoked for any of the three possible operations (connect, send, receive). In this example, you're not expecting to receive anything from the server, so you only handle the connect and send cases. If it's the case that a connection request has completed, go ahead and send a message on that connection by using *SendAsync*. This will use the same *SocketAsyncEventArgs* that you originally populated with the message data. When you're notified that a send operation has completed, simply report the relevant information by adding a suitable string to the *ListBox*.

```

private void SocketAsyncEventArgs_Completed(object sender, SocketAsyncEventArgs e)
{
    if (e.SocketError != SocketError.Success)
    {
        ShowStatus(e.SocketError.ToString());
        CloseSocket(e);
    }
    else
    {
        switch (e.LastOperation)
        {
            case SocketAsyncOperation.Connect:
                ConnectCompleted(e);
                break;
            case SocketAsyncOperation.Send:
                SendCompleted(e);
                break;
        }
    }
}

private void ConnectCompleted(SocketAsyncEventArgs e)
{
    if (e.ConnectSocket != null)
    {
        {
            bool isAsync = e.ConnectSocket.SendAsync(e);
            if (!isAsync)
            {
                SocketAsyncEventArgs_Completed(e.ConnectSocket, e);
            }
        }
    }
}

private void SendCompleted(SocketAsyncEventArgs e)
{
    String s = UTF8Encoding.UTF8.GetString(e.Buffer, 0, e.Buffer.Length);
    ShowStatus(String.Format("Sent: {0} ({1})", s.Replace("^Z", ""), e.BytesTransferred));
    CloseSocket(e);
}

```

With Windows Phone socket support, you can both send and receive data between the phone and a remote server. It is a small step from sending a text message that the server displays in a window, to sending a text message that the server interprets as a command to execute arbitrary functionality. By doing so, you can build an application that converts the phone into a remote control.



Note It's worth mentioning here that if you implement this kind of system, you'd also want to consider security carefully. In particular, you shouldn't allow your server to blindly accept commands from unauthorized clients.

OData Client

In Chapter 11, “Web and Cloud,” you saw how to build an OData client application to connect to a WCF data service. In Windows Phone 7, there was no Visual Studio support for generating client proxies for WCF data services, so the only option was to use the DataSvcUtil tool from the command line. The version 7.1 SDK does include Visual Studio support with which you can use the Add Service Reference Wizard in Windows Phone projects.

To highlight the differences, the following section will walk you through creating a version 7.1 application to mirror the version 7 application for connecting to the *CustomerWebApp* WCF data service from Chapter 11. First, run the server application and connect to the service URL in a browser window to verify that it’s functioning correctly. For example, navigate to <http://localhost:8001/CustomerWebApp.svc/Customers>. Assuming that you have Feed Reading View turned off (to see how to do this, go to Chapter 11) this should produce an XML document containing all of the Customer records from the *AdventureWorks* database.

Next, take a copy of the *DataServiceClient* phone application from Chapter 11, and then strip out the data service proxies. Specifically, remove the *CustomerData.cs* file altogether. Also remove the reference to the old *System.Data.Services.Client.dll*. Save the project. This was originally a version 7 project, so you still will not be able to use the Add Service Reference Wizard for a WCF data service. To fix this, change the project properties to target Windows Phone OS version 7.1. Note that this is a one-way operation—you cannot revert a version 7.1 project to a version 7 project (unless you’re willing to edit the .csproj file manually). After that’s done, you can use the Add Service Reference Wizard option to generate the proxies, as shown in Figure 17-11.



Note Using DataSvcUtil from the command line does not include an option to specify the namespace; you previously had to accept the default namespace (which in this example, was *AdventureWorksLT2008R2Model*). Conversely, the Add Service Reference Wizard does give you this option. In Figure 17-12, this has been set to *AdventureWorks*.

Don’t worry about the message about operations: the Add Service Reference Wizard cannot generate operation (method) proxies, but this is academic because this is a data service that doesn’t define any explicit service contract operations anyway. When you click OK, in addition to generating the proxies, the wizard automatically adds a reference to the updated *System.Data.Services.Client.dll* for you.

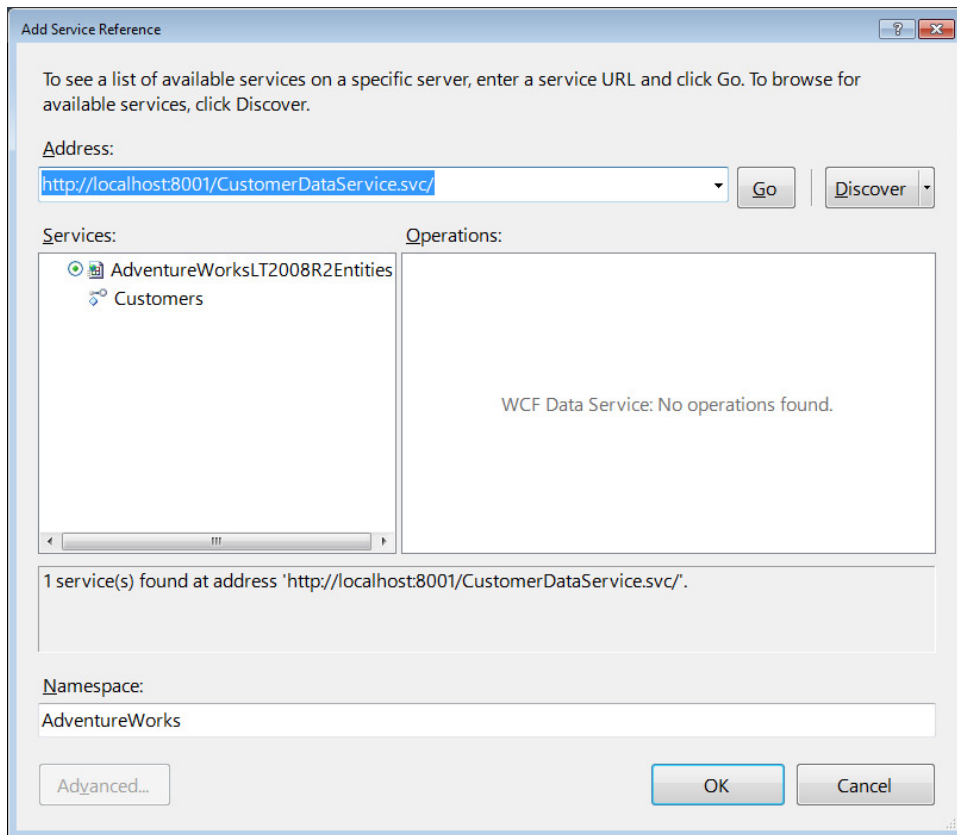


FIGURE 17-11 Add a WCF data service reference to generate proxies.

There is very little difference in the proxy code generated for version 7 as compared with version 7.1. As before, the new proxies include an entities class derived from *DataServiceContext*—this loosely represents the set of tables in the database. You get a class to represent each table that simply implements *INotifyPropertyChanged*. Then, the entities class encapsulates a property of type *DataServiceQuery<T>* for each table, where *T* is the table class type. So, in this example, there will be a *Customer* table class, which defines a property for each table column. The one slight difference is that each column property is additionally attributed with the *DataMemberAttribute*. This simply specifies that the property is part of a data contract and is serializable via a *DataContractSerializer*.

With the appropriate change to the using statement to accommodate the new namespace, there are no further changes required in the solution; just press F5 to run it, and you should experience exactly the same behavior as in the 7 version, as shown in Figure 17-12.

CONTOSO

datasvc client

A Bike Store	245-555-0173
Progressive Sports	170-555-0127
Advanced Bike Component	279-555-0130
Modular Cycle Systems	710-555-0173
Metropolitan Sports Supply	828-555-0186
Aerobic Exercise Company	244-555-0112
Associated Bikes	192-555-0173
Rural Cycle Emporium	150-555-0127
Sharp Bikes	926-555-0159
Bikes and Motorbikes	112-555-0191
Bulk Discount Store	1 (11) 500 555-0132
Catalog Store	440-555-0132
Center Cycle Shop	521-555-0195
Central Discount Store	582-555-0113
Chic Department Stores	928-555-0116
Travel Systems	121-555-0121
Bike World	216-555-0122
Eastside Department Store	926-555-0164
Coalition Bike Company	357-555-0161
Commuter Bicycle Store	972-555-0163

FIGURE 17-12 The WCF data service client application.

This version of the application uses the exact same client code as the 7 version, the critical elements of which are listed in the following for convenience:

```
private AdventureWorksLT2008R2Entities entities =
    new AdventureWorksLT2008R2Entities(
        new Uri("http://localhost:8001/CustomerDataService.svc"));

protected override void OnNavigatedTo(NavigationEventArgs e)
{
    DataServiceQuery<Customer> query = entities.Customers;
    query.BeginExecute(Customers_Callback, query);
}

private void Customers_Callback(IAsyncResult result)
{
    DataServiceQuery<Customer> query =
        (DataServiceQuery<Customer>)result.AsyncState;

    var customers = query.EndExecute(result);
    Dispatcher.BeginInvoke(() => { customerList.ItemsSource = customers; });
}
```

Rather than using a *DataServiceQuery<T>* as you did previously, you could use a *DataServiceCollection<T>*, instead (see the *DataServiceClient_Collection* solution in the sample code). This provides greater flexibility for composing complex queries. It also uses a more formalized *Load Async/Load Completed* model, rather than the generic *IAsyncResult* callback of the *DataServiceQuery* approach.

```

private AdventureWorksLT2008R2Entities entities =
    new AdventureWorksLT2008R2Entities(
        new Uri("http://localhost:8001/CustomerDataService.svc"));
private DataServiceCollection<Customer> customers;

protected override void OnNavigatedTo(NavigationEventArgs e)
{
    //DataServiceQuery<Customer> query = entities.Customers;
    //query.BeginExecute(Customers_Callback, query);

    customers = new DataServiceCollection<Customer>(entities);
    customers.LoadCompleted += customers_LoadCompleted;
    var query = from c in entities.Customers select c;
    customers.LoadAsync(query);
}

//private void Customers_Callback(IAsyncResult result)
//{
//    DataServiceQuery<Customer> query =
//        (DataServiceQuery<Customer> )result.AsyncState;

//    var customers = query.EndExecute(result);
//    Dispatcher.BeginInvoke(() => { customerList.ItemsSource = customers; });
//}

private void customers_LoadCompleted(object sender, LoadCompletedEventArgs e)
{
    if (e.Error == null)
    {
        customerList.ItemsSource = customers;
    }
}

```

The *LoadCompleted* event is raised on the UI thread, so you no longer need to marshal UI operations via a *Dispatcher* object. The preceding changes to the code should produce exactly the same results as before. Another optimization that might be worth making is to support paging of the data. This requires changes on both the server (to send the data in pages) and on the client (to handle paged data coming in). First, on the server, add the following line to the end of the *InitializeService* method in the *CustomerDataService.svc.cs* (this specifies a 20-record page size for the *Customers* table):

```
config.SetEntitySetPageSize("Customers", 20);
```

Next, in the client application, in the *MainPage.xaml*, insert a *TextBlock* below the title panel, and above the *ListBox*. This will report the current count of data records received, updated dynamically.

```
<TextBlock Grid.Row="1" x:Name="customerCount" Text="count = " Margin="24,0,12,0"/>
```

Finally, in the `MainPage.xaml.cs`, move the initialization of the `ListBox.ItemsSource` from the `LoadCompleted` handler to the `OnNavigatedTo` override, and then change the `LoadCompleted` handler to update the count of records as they come in from the service:

```
protected override void OnNavigatedTo(NavigationEventArgs e)
{
    customers = new DataServiceCollection<Customer>(entities);
    customers.LoadCompleted += customers_LoadCompleted;
    var query = from c in entities.Customers select c;
    customers.LoadAsync(query);
    customerList.ItemsSource = customers;
}

private void customers_LoadCompleted(object sender, LoadCompletedEventArgs e)
{
    if (e.Error == null)
    {
        if (customers.Continuation != null)
        {
            customers.LoadNextPartialSetAsync();
            customerCount.Text =
                String.Format("count = {0}", customerList.Items.Count);
        }
    }
}
```

By using paged data, the application is more responsive to user interaction; for example, the user can start working with the data, including scrolling through it, before it has all arrived from the server. Another optimization that you can use is to avoid loading the data on every navigation, because the current application does. To achieve this, you can serialize the data in the application or page state dictionaries (see the `DataServiceClient_State` solution in the sample code). The key to this is the `Serialize` and `Deserialize` methods exposed by the `DataServiceState` class. To add this functionality to the application, add a couple of *bool* fields in the page class to track when data is loaded. Set these to *true* in the `LoadCompleted` handler when all pages of the data are received.

```
private bool isAllDataReceived;
private bool isDataLoaded;

private void customers_LoadCompleted(object sender, LoadCompletedEventArgs e)
{
    if (e.Error == null)
    {
        if (customers.Continuation != null)
        {
            customers.LoadNextPartialSetAsync();
            customerCount.Text =
                String.Format("count = {0}", customerList.Items.Count);
        }
    }
}
```



```

    }
    else
    {
        isAllDataReceived = true;
        isDataLoaded = true;
    }
}
}

```

When the user navigates away from the current page (or the current application), the application persists the data received from the data service to page state. The *DataServiceState.Serialize* method takes an entities (that is, a *DataServiceContext*) object, and a dictionary of all the table class object data. Note that there are two overloads of this method, which you can use to serialize either the *DataServiceContext* object alone or the object with all its *DataServiceCollections*. If you choose the second option, you must serialize *all* the collections. In this example, there is only one: the customers collection.

There's an additional complication if the data is paged. It is not possible to know how many rows of data the server is going to send you at any given time. The only way to determine if all the data is received is if the *Continuation* object is null in the *LoadCompleted* handler. When the application persists the data received, it's simpler if this is only done if all the data has been received. So, in the *OnNavigatedFrom* override, you need to verify that all of the data is received before trying to persist it. If not, you should remove the corresponding state dictionary so that the application does not attempt to read in partial data when the user navigates back to the page.

```

protected override void OnNavigatedFrom(NavigationEventArgs e)
{
    if (isAllDataReceived)
    {
        Dictionary<String, Object> data = new Dictionary<String, Object>();
        data.Add("Customers", customers);
        State["DataServiceState"] = DataServiceState.Serialize(entities, data);
    }
    else
    {
        State.Remove("DataServiceState");
    }
}

```

The *OnNavigatedTo* override needs updating to match this behavior. If you haven't yet loaded the data, first try to fetch it from page state, and if that fails, go out and get it from the remote service. So, you test first to see if data is loaded, and then use *State.TryGetValue* to try to get the data from storage. If that is successful, you can deserialize the *DataServiceState* object to memory, extract the persisted entities and customer data, and then load the data into the UI.

```
protected override void OnNavigatedTo(NavigationEventArgs e)
{
    if (!isDataLoaded)
    {
        object tmp;
        if (State.TryGetValue("DataServiceState", out tmp))
        {
            DataServiceState state = DataServiceState.Deserialize(tmp as String);
            var stateEntities =
                (AdventureWorksLT2008R2Entities)state.Context;
            var stateCustomers =
                (DataServiceCollection<Customer>)state.RootCollections["Customers"];
            LoadDataFromPageState(stateEntities, stateCustomers);
        }
        else
        {
            LoadDataFromService();
        }
    }
}
```

There are two custom methods: one that takes in the data retrieved from storage, and the other that goes back to the remote service to get the data.

```
public void LoadDataFromPageState(
    AdventureWorksLT2008R2Entities stateEntities,
    DataServiceCollection<Customer> stateCustomers)
{
    entities = stateEntities;
    customers = stateCustomers;
    isDataLoaded = true;
}

private void LoadDataFromService()
{
    entities = new AdventureWorksLT2008R2Entities(
        new Uri("http://localhost:8001/CustomerDataService.svc"));
    customers = new DataServiceCollection<Customer>(entities);
    customers.LoadCompleted += customers_LoadCompleted;
    var query = from c in entities.Customers select c;
    customers.LoadAsync(query);
    customerList.ItemsSource = customers;
}
```

There's one final caveat with this approach. It is entirely possible that the data set returned from the server is just too big to persist to state storage. Recall from Chapter 7, "Navigation State and Storage," that no single page is allowed to store more than 2 MB of data, and the application overall is not allowed to store more than 4 MB. The simplest way to test this is to serialize representative (and boundary) volumes of data, and then just measure the length. If the data is too big for state storage, the alternative is to use isolated storage, instead, for which there is no enforced cap. This also assumes that you have already trimmed the query to return only the data in which you're interested, so that you're not retrieving or storing redundant data.

Search Extensibility

Another connectivity enhancement in version 7.1 is the ability to extend the Bing search experience with custom behavior, integrating your application with the search results. There are two ways that you can extend the Bing search behavior in your applications: App Connect, and App Instant Answer. With both features, you can set up your application so that it shows up in the Bing search results when the user taps the hardware Search button. The differences are summarized in Table 17-2.

TABLE 17-2 Bing Search Extensibility

Requirement	App Connect	App Instant Answer
WAppManifest.xml	Requires Extensions entries for each Bing category that you want to extend.	No specific changes required.
Extras.xml	Required: Specifies captions to be used in the search results apps pivot item.	Not used.
UriMapper	Recommended: Allows you to reroute to a specific page on application startup.	Not required.
Target page	You can reroute to multiple different pages, depending on the search item, if you want.	No option: Your application is launched as normal, with its default startup page.
Query string	You should parse the incoming query string for categories and item names for which you want to provide extensions.	You should parse the incoming query string for the <i>bing_query</i> value.
Search connection	Bing includes your application in the search results when the user's search matches the categories for which you registered extensions.	Bing includes your application in the search results if your application name exactly matches the search string.

In both cases, your application is launched with a particular query string, and you are responsible for parsing that query string to get the search context. It is then up to the application to decide what behavior to execute, based on this search context. Each of these two approaches is discussed in more detail in the following sections.

App Connect

The App Connect approach is the more complex of the two extensibility models. You register your application in a way that gives you more fine-grained control over the conditions that Bing will use to identify it as a suitable extension. You register your application for the search categories, or search “extensions,” for which you believe your application has relevance.



Note It is important to choose the extensions carefully, and to avoid spamming the system by registering for unrelated extensions. Applications that register excessive unrelated extensions will be removed from the marketplace.

If and when the user chooses to launch your application from the search results list, your application is given a richer query string, with which you can fine-tune the consequent behavior. The overall model for App Connect is illustrated in Figure 17-13. In summary, the user initiates a Bing search, and then taps one of the items in the search results. This navigates to a Quick Card, which is a pivot page

that offers an “about” pivot with basic information, a “reviews” pivot, and an “applications” pivot. Your application can be listed in the applications pivot.

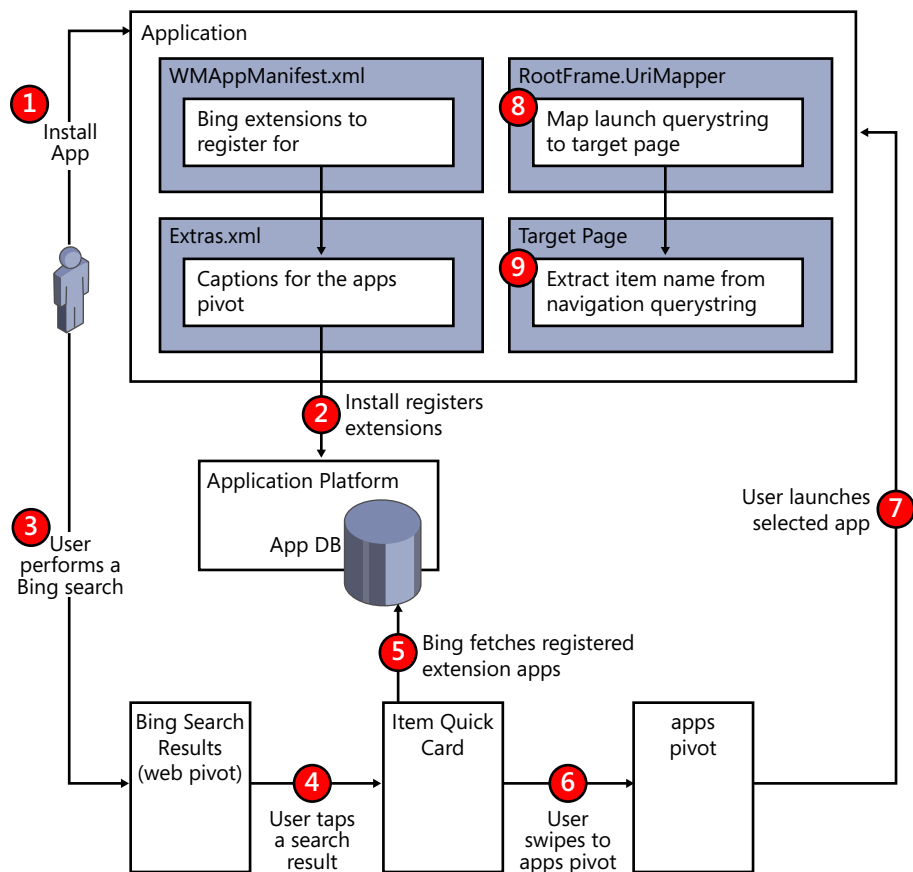


FIGURE 17-13 The App Connect extensibility model.

The following steps walk through how to create an App Connect search extension. A completed example is in the *SimplestAppConnect* solution in the sample code.

First, create a new Windows Phone application project. Add a second page to the project, named *MyTargetPage*. This will be the target page to which to navigate when the application is launched via App Connect. In the XAML for this page, add a simple *TextBlock*; the application will set the text for this dynamically, from the item information in the Bing search results.

```
<TextBlock x:Name="Target" TextWrapping="Wrap" Margin="{StaticResource PhoneHorizontalMargin}"/>
```

Then, add an *Extensions* section to your *WMApManifest.xml*, within the *App* element, after the *Tokens* section. Note that the *Extension* entry must specify one of the defined extension identifiers, as listed in the Search Registration and Launch Reference for Windows Phone, which you can find at [http://msdn.microsoft.com/en-us/library/hh202958\(VS.92\).aspx](http://msdn.microsoft.com/en-us/library/hh202958(VS.92).aspx). In this example, the application registers for just one extension: *Bing_Products_Gourmet_Food_and_Chocolate*. The *ConsumerID* is always the same: "{5B04B775-356B-4AA0-AAF8-6491FFEA5661}". This specifies that this is an extension to Bing search. The *TaskID* is always "_default", and the *ExtraFile* must be a relative path to the *Extras.xml* file in your project.

```
<Extensions>
  <Extension
    ExtensionName="Bing_Products_Gourmet_Food_and_Chocolate"
    ConsumerID="{5B04B775-356B-4AA0-AAF8-6491FFEA5661}"
    TaskID="_default"
    ExtraFile="Extensions\\Extras.xml" />
</Extensions>
```

This file doesn't exist yet, so you should next add a new XML file to the project and name it *Extras.xml*. The build action for *Extras.xml* should be set to *Content*. The *Extras.xml* must be in a folder in your project named *Extensions*. However, the path you specify in the *ExtraFile* attribute can omit the *Extensions* root path, and just specify "Extras.xml"—either will work, so long as the file itself is in the right place. This file is where you specify the strings to be used in the Bing search results list for your application.

```
<?xml version="1.0" encoding="utf-8" ?>
<ExtrasInfo>
  <AppTitle>
    <default>My Fab Search Extension</default>
  </AppTitle>
  <Consumer ConsumerID="{5B04B775-356B-4AA0-AAF8-6491FFEA5661}">
    <ExtensionInfo>
      <Extensions>
        <ExtensionName>Bing_Products_Gourmet_Food_and_Chocolate</ExtensionName>
      </Extensions>
      <CaptionString>
        <default>All you need to know about coffee</default>
      </CaptionString>
    </ExtensionInfo>
  </Consumer>
</ExtrasInfo>
```

You must supply an *AppTitle* with at least one string for the default language. In the *Consumer* section, the *ConsumerID* is again the same Bing search ID. Under this, you list all the *Extensions* that you want to register for Bing search. Again, you must provide at least one default *CaptionString*. These two strings are used in the search results, which you can see in Figure 17-14. The specific search item is displayed as the page title, and the application's title string and caption strings are used on the apps pivot item, alongside the application's icon. The application's icon (the 62x62-pixel image) for an extension application should not use transparency.

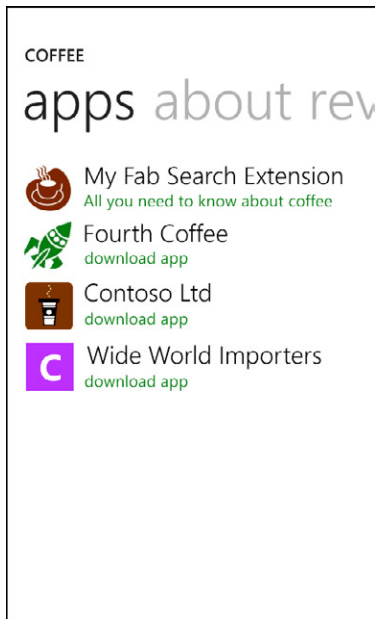


FIGURE 17-14 App title and caption strings in the Bing search results.

The next item you need is a custom URI mapper. Add a new class file to your project, and then change the code to derive your class from *UriMapperBase*. The purpose of the custom URI mapper is to map the navigation query string that Bing search passes to your application to the correct target page. For example, map this incoming URI:

```
/SearchExtras?ProductName=coffee&Category=Bing_Products_Gourmet_Food_and_Chocolate
```

to this target page, including the original query string parameters:

```
/MyTargetPage.xaml?ProductName=coffee&Category=Bing_Products_Gourmet_Food_and_Chocolate.
```

In addition to determining the correct target page, you're free to modify the parameter list according to your requirements before passing it on, if you need to. When the application has been launched via Bing search App Connect, the URI will include the `"/SearchExtras"` substring. So, if you examine the URI and find that it does not include this substring, you should immediately return, because this means that the application has been launched normally, not via Bing search.

```
public class MyUriMapper : UriMapperBase
{
    public override Uri MapUri(Uri uri)
    {
        String inputUri = uri.ToString();
        if (inputUri.Contains("/SearchExtras"))
        {
            if (inputUri.Contains("Bing_Products_Gourmet_Food_and_Chocolate"))
            {

```

```

        String outputUri = inputUri.Replace(
            "/SearchExtras", "/MyTargetPage.xaml");
        return new Uri(outputUri, UriKind.Relative);
    }
}
return uri;
}
}

```

In the code, the one and only public *MapUri* method takes in the search URI, as supplied by Bing. So, the *MapUri* method parses the URI, and first looks for the *"/SearchExtras"* substring. If this is found, it then goes on to look for the Products category (that is, *"Bing_Products_Gourmet_Food_and_Chocolate"*).

In this example, you have only one target page for all search requests. In a sophisticated application, you might have multiple pages; therefore, you would need to implement a more complex decision tree to determine which page to return from the URI mapper. You would also typically search for more than one category and one product.

To use the URI mapper in your application, you create an object of this type, and then set this as the value of the *UriMapper* property in the *RootFrame*. The best place to do this is at the end of the *InitializePhoneApplication* method in the *App* class:

```
RootFrame.UriMapper = new MyUriMapper();
```

This causes the system to load the specified target page and pass in the navigation URI, including the query string, as part of the *NavigationContext*. This comes in to the page in the form of a dictionary of key-value pairs.

In the target page, override the *OnNavigatedTo* method to examine the incoming query string. Look for an incoming *ProductName*. Having found the corresponding value for the key-value pair—in this case, *"coffee"*—you then make a decision as to whether you know anything about this specific product. If so, you can then go on to do whatever domain logic you want, based on this value. In this example, you simply indicate that this is a known product by setting the *TextBlock.Text* value.

```

protected override void OnNavigatedTo(NavigationEventArgs e)
{
    String tmp;
    if (NavigationContext.QueryString.TryGetValue("ProductName", out tmp))
    {
        String product = (String)tmp;
        if (product.ToLowerInvariant().Contains("coffee"))
        {
            Target.Text = String.Format("We know about {0}.", product);
        }
        else
        {
            Target.Text = String.Format("We don't know about {0}.", product);
        }
    }
}

```

With the code complete, you can test this, either in the emulator or on a physical device. Tap the Bing search button, and then enter **coffee**. In the primary search results, swipe over to the web pivot item, if it's not already selected. Then, scroll down to find the products list. If necessary, tap the Next Product Results link to get to a coffee product that contains the string "coffee". Tap any such item: this takes you to the Quick Card for that item. In the Quick Card, swipe over to the apps pivot item; your application should be listed there. When you tap the application to launch it, the URI mapper is invoked, and the application navigates to the target page and updates the text with the Bing search information.

If you want to test the functionality of the application in a more deterministic way, you can provide a fake launch query string in the WMAppManifest.xml file. For example, replace the *DefaultTask* entry with an entry that specifies a *SearchExtras* query string (without a leading slash). You can also test the negative case by providing a query string that should not result in listing your application in the search results. Be aware that this technique must only be used for testing—and the manifest submitted to marketplace for publication must use your default page, without additional parameters.

```
<Tasks>
  <!--<DefaultTask Name="_default" NavigationPage="MainPage.xaml"/>-->
  <DefaultTask Name="_default" NavigationPage="SearchExtras?ProductName=coffee&
    Category=Bing_Products_Gourmet_Food_and_Chocolate"/>
  <!--<DefaultTask Name="_default" NavigationPage="SearchExtras?ProductName=bananas&
    Category=Bing_Products_Gourmet_Food_and_Chocolate"/>-->
</Tasks>
```

Suppose that you want to support more than one extension category, and perhaps provide different caption strings for some or all of these? Or, suppose that you want to map the launch URI to one of several different target pages, according to some part of the query string? All of these behaviors are possible (see the *SimpleAppConnect* solution in the sample code). Consider first, the requirement to support multiple extension categories. To do this, simply add each additional category in the Extensions section in your WMAppManifest.xml file. In the listing that follows, the application supports one of each of the three major categories: Products, Places, and Movies.

```
<Extensions>
  <Extension ExtensionName="Bing_Products_Gourmet_Food_and_Chocolate"
    ConsumerID="{5B04B775-356B-4AA0-AAF8-6491FEEA5661}" TaskID="_default"
    ExtraFile="Extensions\Extras.xml" />
  <Extension ExtensionName="Bing_Places_Food_and_Dining"
    ConsumerID="{5B04B775-356B-4AA0-AAF8-6491FEEA5661}" TaskID="_default"
    ExtraFile="Extensions\Extras.xml" />
  <Extension ExtensionName="Bing_Movies" ConsumerID="{5B04B775-356B-4AA0-AAF8-6491FEEA5661}"
    TaskID="_default" ExtraFile="Extensions\Extras.xml" />
</Extensions>
```

Note that there's a difference between *Bing_Movies* and *Bing_Products_Movies*—the latter will include results for movies that are not found to be showing in theatres; for example, if the user is searching for movies to buy on DVD. In the Extras.xml, you could group multiple *ExtensionName* entries to share the same caption strings. You can also divide your supported categories into groups, each with its own caption strings.


```

<ExtensionInfo>
  <Extensions>
    <ExtensionName>Bing_Products_Gourmet_Food_and_Chocolate</ExtensionName>
    <ExtensionName>Bing_Places_Food_and_Dining</ExtensionName>
  </Extensions>
  <CaptionString>
    <default>All you need to know about coffee</default>
  </CaptionString>
</ExtensionInfo>

<ExtensionInfo>
  <Extensions>
    <ExtensionName>Bing_Movies</ExtensionName>
  </Extensions>
  <CaptionString>
    <default>Coffee in movies</default>
  </CaptionString>
</ExtensionInfo>

```

You could also enhance your URI mapper to target different pages for the different categories.

```

public override Uri MapUri(Uri uri)
{
    String inputUri = uri.ToString();
    if (inputUri.Contains("/SearchExtras"))
    {
        String targetPageName = "/MainPage.xaml";
        if (inputUri.Contains("Bing_Products"))
        {
            targetPageName = "/ProductTargetPage.xaml";
        }
        else if (inputUri.Contains("Bing_Places"))
        {
            targetPageName = "/PlaceTargetPage.xaml";
        }
        else if (inputUri.Contains("Bing_Movies"))
        {
            targetPageName = "/MovieTargetPage.xaml";
        }

        String outputUri = inputUri.Replace("/SearchExtras", targetPageName);
        return new Uri(outputUri, UriKind.Relative);
    }
    return uri;
}

```

Clearly, you could take this a step further by pivoting your decisions off any of the elements of the query string. To ensure robustness, you should also decode the incoming URI (typically, with *HttpUtility.UrlDecode*) before processing it, and then re-encode it (with *HttpUtility.UrlEncode*) before returning from your *MapUri* method.

App Instant Answer

The second search extensibility model is simpler. It requires no special manifest entries, no Extras.xml, and no URI mapper. You have no choice about which page to launch based on the search results, and Bing will always launch your application using its default page. The way Bing identifies your application as being suitable for listing in the search results is internal to Bing. As of this writing, the determining factor is the application name; however, keep in mind that this could change at any time. The overall model for App Instant Answer is summarized in Figure 17-15.

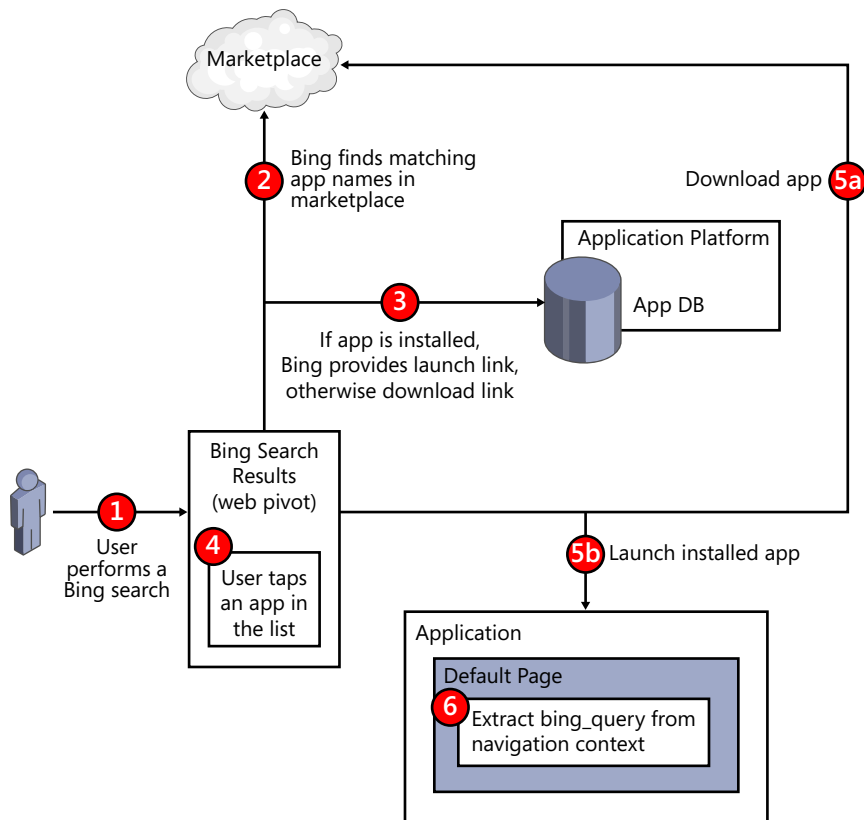


FIGURE 17-15 The App Instant Answer extensibility model.

To create an App Instant Answer application, create a Windows Phone application as normal. An example of this is in the *SimpleAppInstantAnswer* solution in the sample code. When the user performs a Bing search, it might include applications in the web pivot. Currently, it will do so only if your application name starts with the whole of the user's search term, ignoring case. For example, if the user searches for "banana", and your application name is "Banana Instant Answer", then this will match, and Bing will potentially add your application to the results list. On the other hand, if your application name is "Banoffee Instant Answer", then the match fails. To set your application name, you set the *Title* attribute of the *App* element in your *WMAppManifest.xml*. Typically you set this in

the project properties page in Visual Studio, although you can also edit the manifest manually, if you prefer, as follows:

```
Title="Banana Instant Answer"
```

Recall from Chapter 1, “Vision and Architecture,” that there are two *Title* entries in the manifest: one is an attribute of the application element, the other is a subelement of the *Tokens* element. The application element’s *Title* attribute is the one that you want here. Also note that even if your application name exactly matches the user’s search term, there’s no guarantee that your application will be listed in the search results.

If you want to allow for the possibility that you’ll be included in search results for App Instant Answers, you should test for the *bing_query* parameter in the navigation query string.

```
protected override void OnNavigatedTo(NavigationEventArgs e)
{
    String query;
    if (NavigationContext.QueryString.TryGetValue("bing_query", out query))
    {
        // Do something useful with this information.
    }
}
```

As with App Connect, you can test an App Instant Answer application by providing a fake launch URI in your WMAAppManifest.xml file. And also just as before, this must be removed before submitting your application to marketplace.

```
<Tasks>
  <!--<DefaultTask Name="_default" NavigationPage="MainPage.xaml"/>-->
  <DefaultTask Name="_default" NavigationPage="MainPage.xaml?bing_query=Banana" />
</Tasks>
```

Finally, be aware that this is the only way that you can test your application, because even on the emulator, Bing will only include App Instant Answer applications from the published catalog in marketplace. If it finds an application in marketplace that is already installed on the phone, it will change the link from a marketplace download link to an installed application link, but it will not include an installed application unless it first finds a match for the application in the marketplace.

Summary

In this chapter, you saw how the Metro tiles feature has been extended and made available to third-party developers so that you can build applications that align with the system features and give the user a sense of an integrated ecosystem. The tile enhancements, along with toast enhancements, are also used by the push notification system, allowing your server to target multiple secondary tiles, both back and front. Sockets is the major new connectivity feature, which opens up a wide range of possibilities for building data transfer, remote control, and multi-party remote connected experiences. Finally, Bing search extensibility does very much the same thing, allowing you to make your application a seamless part of the user’s day-to-day experience.

Data Support

In Windows Phone 7, you can store small amounts of application and page state in the standard state dictionaries. You can also store large amounts of data in your application's isolated storage. Isolated storage is based on a simple concept of folders and files, but there is no structured storage support in version 7. On the other hand, Windows Phone 7.1 introduces support for local databases. These provide a rich set of database capabilities, both in terms of data structure and that of Language-Integrated Query (LINQ). You can also encrypt your database (either some or all of the data stored there), and, most important, you can encrypt the user credentials that secure any application data on the phone. In keeping with the user experience (UX) paradigm of encouraging seamless integration with standard features, you now also have the ability to integrate with the user's aggregated contact and calendar data. Finally, Microsoft has released an early version of the Sync Framework, by which phone applications (among others) can establish synchronization contexts with remote data servers.

Local Database and LINQ-to-SQL

From Windows Phone 7.1, your application can create a database in isolated storage and perform standard create/read/update/delete (CRUD) operations on it via LINQ-to-SQL statements. The database file can be either in your application install folder or in the application's isolated storage. Loose files that are packaged as content in your XAP—potentially including a local database file—all end up in the application install folder. Your application can access this location, but only for read operations. On the other hand, your application has full read/write access to its isolated storage. Each local database is local to its application; it is not accessible outside that application. This contrasts with the traditional desktop or server database model, in which there is typically a database service running continuously, with access to all attached databases.

You can create a local database as part of your application code, and you can also prepare a database file in advance and deploy it with your application. Under the covers, the database will effectively be a SQL Compact Edition (SQL-CE) database file (.sdf). This means that you can use external tools such as the Isolated Storage Explorer Tool (which ships with the version 7.1 SDK) and SQL Server Management Studio (SSMS) to work with the database independently of the application. That having been said, developers are encouraged to think in terms of local databases, not in terms of SQL-CE. Exactly what this means will be discussed in the following sections. Figure 18-1 illustrates the development model at a high level.

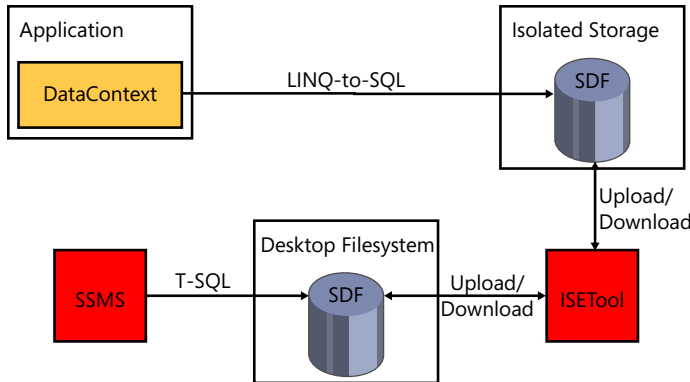


FIGURE 18-1 The local database development model.

From a code perspective, you use the *DataContext* class to represent your database, and LINQ mapping types such as *Table* and *Column* to represent the database schema. Access in code is via LINQ-to-SQL only; for example, you cannot use Microsoft ADO.NET or T-SQL in your code. You derive a custom class from *System.Data.Linq.DataContext*. This is completely different from the *Framework Element.DataContext* property (which you recall is typed as an object). The general model is shown in Figure 18-2, using *Employees* and *Customers* as examples of tables in the database, and *Name* and *Salary* as examples of columns in the *Employees* table.

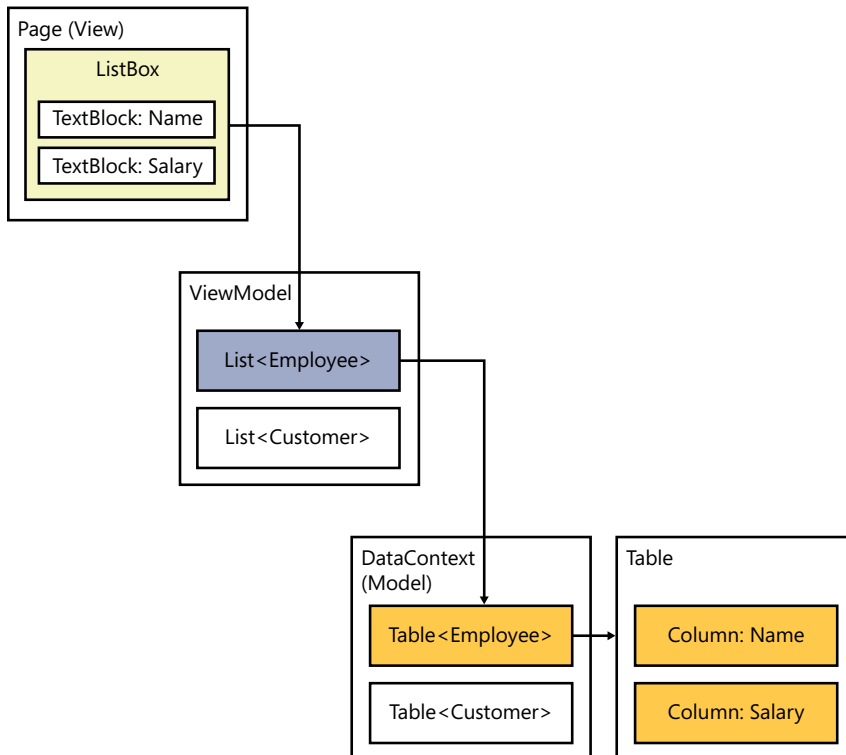


FIGURE 18-2 Local database support: code model.

To use a local database in your application, the minimum requirement is to have a custom *Data Context* with at least one *Table*, where that *Table* defines at least one *Column*. You would typically have multiple *Column* objects per *Table*, and often multiple *Table* objects. If you're using some form of Model-View-ViewModel (MVVM), you would also typically have a viewmodel bridging your page user interface (UI) and your *DataContext*, which represents your overall model.

Create and Read

Figure 18-3 is a screenshot of the *ShoppingList_CR* solution in the sample code. This provides a UI in which the user can add new items to build a shopping list. This list is displayed in a *ListBox* at the bottom of the page. As she adds each item, the application adds the item to a local database. The *ListBox* is data-bound to a viewmodel. The viewmodel sends and receives the data to and from the local database via a *DataContext*. In this example, there is only one *Table*, and the application displays only one *Column* (although the schema for the database table includes a second column for row identity, which is not displayed in the UI).

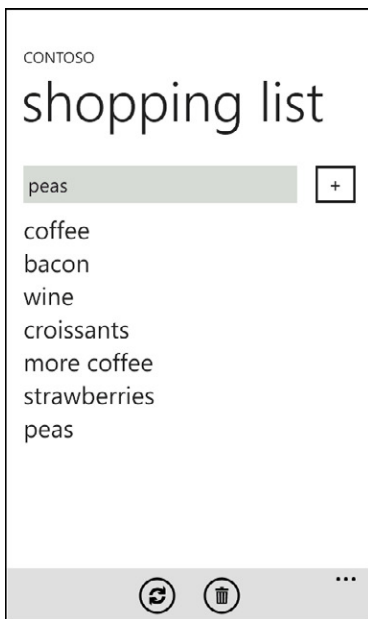


FIGURE 18-3 A simple shopping list application.

To build this application, you would typically start with the model. Define a class for each database table, specifying the columns with which you're going to work. For the shopping list application, you can define a *ShoppingItem* class, decorated with the *Table* attribute, which is defined in the *System.Data.Linq.Mapping* namespace. For each column, you define a public property with the *Column* attribute. In this attribute, you specify properties that map to the database schema properties for this column, including the data type and size, whether the column is nullable, whether it is a primary key, whether it is auto-generated by the database engine, and so on. Each shopping list has a simple string column for the *Name*, and an integer column for the *Id*. The *Id* column is an auto-generated identity

column in the database, which will not be used in the UI. This is not essential, but it is very common and serves to illustrate some of the typical schema options that are available to you. The class implements *INotifyPropertyChanged* so that it can take part in a data-binding chain that runs between the UI and the database.

```
[Table]
public class ShoppingItem : INotifyPropertyChanged
{
    private int id;

    [Column(
        IsPrimaryKey = true, IsDbGenerated = true,
        DbType = "INT NOT NULL Identity", CanBeNull = false,
        AutoSync = AutoSync.OnInsert)]
    public int Id
    {
        get { return id; }
        set
        {
            if (id != value)
            {
                id = value;
                NotifyPropertyChanged("Id");
            }
        }
    }

    private String name;

    [Column]
    public String Name
    {
        get { return name; }
        set
        {
            if (name != value)
            {
                name = value;
                NotifyPropertyChanged("Name");
            }
        }
    }

    public event PropertyChangedEventHandler PropertyChanged;
    private void NotifyPropertyChanged(string propertyName)
    {
        PropertyChangedEventHandler handler = PropertyChanged;
        if (handler != null)
        {
            handler(
                this, new PropertyChangedEventArgs(propertyName));
        }
    }
}
```


Next, you must define a custom *DataContext* class to hold your *Table* objects. This is very simple: it must derive from *DataContext*, it must initialize the base class with the data source properties (effectively, a connection string), and it must contain one or more *Table<T>* collections, for which each one represents the collection of database rows for a given *Table* type. In this example, the database is in isolated storage—this is the normal location, which gives you full CRUD capabilities on the database file.

```
public class ShoppingDataContext : DataContext
{
    public ShoppingDataContext()
        : base("Data Source=isostore:/ShoppingList.sdf")
    {
    }

    public Table<ShoppingItem> ShoppingItems;
}
```

Moving up a level, although a viewmodel is not required, it is common practice for all the reasons discussed in Chapter 4, “Data Binding.” To define a suitable viewmodel, your class should include a field for the custom *DataContext* object, and you initialize this in the viewmodel constructor. If the database doesn’t already exist, you can create it at this time by using *CreateDatabase*. The viewmodel also exposes a data-bindable public property for the collection of shopping items.

```
public class MainViewModel
{
    private ShoppingDataContext shoppingDb;
    public ObservableCollection<ShoppingItem> Items { get; private set; }

    public MainViewModel()
    {
        shoppingDb = new ShoppingDataContext();
        if (!shoppingDb.DatabaseExists())
        {
            shoppingDb.CreateDatabase();
        }
    }

    public void LoadData()
    {
        if (Items == null)
        {
            IQueryable<ShoppingItem> shoppingQuery =
                from ShoppingItem item in shoppingDb.ShoppingItems select item;
            Items = new ObservableCollection<ShoppingItem>(shoppingQuery);
        }
    }

    internal void AddItem(string name)
    {
        ShoppingItem item = new ShoppingItem() { Name = name };
        Items.Add(item);
        shoppingDb.ShoppingItems.InsertOnSubmit(item);
    }

    internal void SaveChanges()
```

```

    {
        shoppingDb.SubmitChanges();
    }

    internal void DisposeDb()
    {
        shoppingDb.Dispose();
    }
}

```

Take a look at the last two methods in the viewmodel. These both wrap calls to the underlying *DataContext*: one for persisting any changes to the database, and the other for disconnecting the database. The wrappers are exposed so that other parts of the system can access them—specifically, in this example, the view classes—without directly exposing the *DataContext* object.



Note Unlike earlier database technologies in Microsoft .NET, the *DataContext* is actually very aggressive about releasing the underlying database connection, so disposing of it is not normally necessary. You should examine your use of the *DataContext* to see if you need to call *Dispose* at any time.

The user will be adding items via the UI, but the UI of course should not work directly with the database. Instead, the UI will make calls into the viewmodel—specifically, to the *AddItem* and *SaveChanges* methods—which internally add the new item to the data-bound collection and also to the underlying database. The most interesting method is the custom *LoadData* method: this performs a LINQ-to-SQL query to extract the selected data rows from the database and populate the data-bound collection property. In this example, the query simply fetches all rows from the *ShoppingItems* table, but you could provide any suitable LINQ query to filter data or combine data from multiple tables. The viewmodel is initialized and exposed in the standard fashion as a public property of the *App* class.

```

private static readonly object myLock = new object();
private static MainViewModel viewModel = null;
public static MainViewModel ViewModel
{
    get
    {
        lock (myLock)
        {
            if (viewModel == null)
            {
                viewModel = new MainViewModel();
                //viewModel.LoadData();
            }
        }
        return viewModel;
    }
}

```

You could load the data when the viewmodel is created, but this could pose a performance issue if there's a lot of data to read from the database. In this scenario, you should consider deferring the data-loading operation to a separate *LoadData* method, or perhaps a property getter such as a *GetViewModel* method. You should also consider whether the potential performance penalty is severe enough to warrant making this operation asynchronous, and moving it off to another thread, so as to avoid the risk of negatively impacting the behavior of the UI.

Up at the top of the stack, the view defines a *TextBox* and a *Button* for the user to add a new item, and a *ListBox* whose items are data-bound to the *Name* property exposed from the viewmodel.

```
<StackPanel Grid.Row="1" Margin="12,0,12,0">
    <Grid>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="380" />
            <ColumnDefinition Width="*" />
        </Grid.ColumnDefinitions>
        <TextBox Grid.Column="0" x:Name="newItem" />
        <Button Grid.Column="1" x:Name="addItem" Content="+" Click="addItem_Click"/>
    </Grid>

    <ListBox x:Name="shoppingList" ItemsSource="{Binding Items}" >
        <ListBox.ItemTemplate>
            <DataTemplate>
                <TextBlock Text="{Binding Name}"/>
            </DataTemplate>
        </ListBox.ItemTemplate>
    </ListBox>
</StackPanel>
```

When the user adds a new item, the page code-behind responds by invoking the *AddItem* method on the viewmodel. It is also common practice to save and load the data when the user navigates to and from the page.

```
private void addItem_Click(object sender, RoutedEventArgs e)
{
    App.ViewModel.AddItem(newItem.Text);
}

protected override void OnNavigatedTo(NavigationEventArgs e)
{
    App.ViewModel.LoadData();
    shoppingList.ItemsSource = App.ViewModel.Items;
}

protected override void OnNavigatedFrom(NavigationEventArgs e)
{
    App.ViewModel.SaveChanges();
}
```

That's all the functionality you need for basic use of a local database. With this code, you can create the database, write to it, and read from it with suitably loosely coupled data-binding to the UI. As a developer convenience during testing, it is often useful to have dummy data to work with. There are two simple ways to build this in. First, you can simulate the user adding multiple new items. In this example, there is an App Bar, which is conditionally hidden.

```
public MainPage()
{
    InitializeComponent();
    DataContext = App.ViewModel;
#if TEST_DATA
    ApplicationBar.IsVisible = true;
#endif
}

private void appBarClear_Click(object sender, System.EventArgs e)
{
    App.ViewModel.DeleteAll();
}

private void appBarLoad_Click(object sender, System.EventArgs e)
{
    App.ViewModel.LoadTestData();
}
```

The test-only view methods call down to invoke test-only *DeleteAll* and *LoadTestData* methods in the viewmodel, which delete all rows in the table or add multiple new items, respectively. When you want to delete items in the database—either a single item, or a collection—you must pass the item(s) to the corresponding delete method. Under the covers, this is executing the equivalent of a T-SQL *DELETE WHERE* statement.

```
internal void DeleteAll()
{
    Items.Clear();
    shoppingDb.ShoppingItems.DeleteAllOnSubmit(shoppingDb.ShoppingItems);
    SaveChanges();
}

internal void LoadTestData()
{
    Items.Add(new ShoppingItem() { Name = "coffee" });
    Items.Add(new ShoppingItem() { Name = "bacon" });
    Items.Add(new ShoppingItem() { Name = "wine" });
    Items.Add(new ShoppingItem() { Name = "croissants" });
    Items.Add(new ShoppingItem() { Name = "more coffee" });
    Items.Add(new ShoppingItem() { Name = "strawberries" });
    shoppingDb.ShoppingItems.InsertAllOnSubmit(Items);
    SaveChanges();
}
```

A second developer convenience is to provide a designer-only set of dummy data. The project includes an XML file called `MainViewModelSampleData.xaml`, which defines an arbitrary set of dummy data, based on the *ShoppingItem* type.

```
<local:MainViewModel
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="clr-namespace:ShoppingList" >

    <local:MainViewModel.Items>
        <local:ShoppingItem Name="design one" />
        <local:ShoppingItem Name="design two" />
        <local:ShoppingItem Name="design three" />
        <local:ShoppingItem Name="design four" />
        <local:ShoppingItem Name="design five" />
        <local:ShoppingItem Name="design six" />
    </local:MainViewModel.Items>

</local:MainViewModel>
```

This is referenced in the page definition in the `MainPage.xaml`. Finally, this allows you to declare the *ItemsSource* for the *ListBox* in XAML, which in turn is what allows the UI designer to display data in the emulator design surface.

```
<phone:PhoneApplicationPage
    x:Class="ShoppingList.MainPage"
    d:DataContext="{d:DesignData Models/MainViewModelSampleData.xaml}"

...unchanged code omitted for brevity

<ListBox x:Name="shoppingList" ItemsSource="{Binding Items}" >
```

Update and Delete

So far, you've seen how to create the database, create new rows, write to the database, and read from it; or put another way, you've learned about the CR in CRUD. To add update and delete (UD) support is trivial once the core behavior is in place. Figure 18-4 shows the *ShoppingList_CRUD* solution in the sample code, which includes the ability for the user to edit or delete existing items.

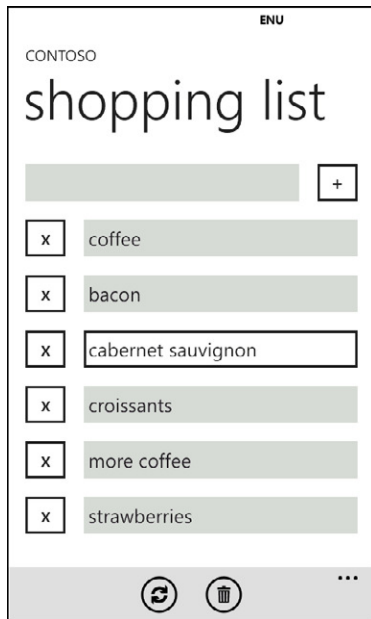


FIGURE 18-4 The shopping list application with update and delete capabilities.

To add this support, first update the view by replacing the single *TextBlock* with a combined *Button* and *TextBox*. The user can tap the button to delete the current item. Now that he has a *TextBox*, he can edit the contents of the item. You need to change the binding to specify *Mode=TwoWay*, which will automatically propagate the user's edits back to the viewmodel. You can update the viewmodel to call *SubmitChanges* to propagate the changes back through the *DataContext* to the database.

Implement the *Click* handler for the delete *Button* to call into the viewmodel's *DeleteItem* method. This method doesn't exist yet, so when you type this in, Visual Studio will ask you if you want to create a stub method for this in the viewmodel. You should accept this suggestion. Keep in mind that you need to get the *DataContext* from the *Button* object. This is slightly confusing, because what you want here is the *data-binding DataContext* object, which in this case maps to a *Table* item. This is not the custom *DataContext* type.

```
private void deleteItem_Click(object sender, RoutedEventArgs e)
{
    Button b = sender as Button;
    if (b != null)
    {
        ShoppingItem item = b.DataContext as ShoppingItem;
        App.ViewModel.DeleteItem(item);
    }
}
```

You can then implement the new *DeleteItem* method in the viewmodel to remove the specified item from the collection and also remove it from the underlying database.

```
internal void DeleteItem(ShoppingItem item)
{
    Items.Remove(item);
    shoppingDb.ShoppingItems.DeleteOnSubmit(item);
    SaveChanges();
}
```

As you can see, supporting update and delete operations on the database data is simple, once the basic connection is in place. However, as you'll learn in the following section, supporting updates to the database schema is a little more involved.

Schema Updates

As you improve your application over time and release updated versions, sooner or later you'll probably need to update the database schema. This presents an issue in circumstances for which your application is creating new data dynamically, as in the previous examples. When you deploy your new version, you don't want to wipe out the user's existing data. Consequently, you must accommodate the fact that her old data might not match the new schema. If your schema changes are purely additive, and where any new columns are all nullable, then you can simply ignore the gaps in the old data. Otherwise, you can proactively update the user's existing data to match the new schema, and this is typically done fairly early in the process, upon the first time the user runs the new version of the application.

Figure 18-5 shows an updated version of the shopping list application (this is the *ShoppingList_v1* solution in the sample code), with which the user can specify a quantity for each item.



FIGURE 18-5 The shopping application, using an updated schema.

If you work through the previous version of the application and then continue, working on the updated functionality in this section, you run the risk of missing one of the real-world dimensions to application development. To be specific, this is the historical dimension, wherein the user has been using your application in the wild and has saved some number of items to the local database on her phone. Recall that when you do a clean/rebuild cycle in Visual Studio, this translates to an uninstall/install cycle on the emulator or device. So, you might miss the opportunity to test the update scenario. One way to mitigate this is to take the following steps:

1. Rebuild your new version and deploy it to the emulator or device to ensure that all the new functionality works in isolation; that is, it takes the update dimension out of the picture. Doing this first also means that you have a built application that you can later deploy without uninstalling any existing version's data.
2. Clean/rebuild and run your older version. Use it to generate a representative set of data. This will uninstall the new version and remove any isolated storage.
3. Run (but don't rebuild) your new version. This will deploy the new version without removing the old isolated storage. This faithfully replicates the real-world update scenario.

Alternatively, for this step, you could use the ISETool to copy the old isolated storage contents to your computer. Next, build/deploy your new version, and then use ISETool to copy the old isolated storage contents back from your computer to the emulator/device. The ISETool is discussed later in this chapter.

For the new version, the changes to the application itself are straightforward. First, add a nullable property to the *ShoppingItem* table type. This needs to be nullable to allow for the fact that any existing rows in the database will not have a value for this column. Note that because this is a value type, it is declared as explicitly nullable. If it were instead a reference type, such as a string, then you would not need to make it explicitly nullable (because, of course, all reference types are implicitly nullable).

```
private int? quantity;

[Column(CanBeNull = true)]
public int? Quantity
{
    get { return quantity; }
    set
    {
        if (quantity != value)
        {
            quantity = value;
            NotifyPropertyChanged("Quantity");
        }
    }
}
```


The viewmodel's *AddItem* method is updated accordingly.

```
internal void AddItem(string name, int qty)
{
    //ShoppingItem item = new ShoppingItem() { Name = name };
    ShoppingItem item = new ShoppingItem() { Name = name, Quantity = qty };

    Items.Add(item);
    shoppingDb.ShoppingItems.InsertOnSubmit(item);
}
```

And, so is the view's button *Click* handler. The XAML declares a second two-way, data-bound *TextBox* for the quantity field to go with this.

```
private void addItem_Click(object sender, RoutedEventArgs e)
{
    //App.ViewModel.AddItem(newItem.Text);
    App.ViewModel.AddItem(newItem.Text, Int32.Parse(quantity.Text));
}
```

These updates are sufficient for the application to provide the required new functionality. However, there is additional work required to support the upgrade deployment scenario. So, you should first test the new version in isolation. Then, you should follow the aforementioned upgrade steps to test the upgrade path. Before you do that, you should add code to handle the upgrade situation, as described in the following sections.

It's quite possible that your old version did not include any notion of version, apart from the standard .NET *AssemblyVersion* attribute in the *AssemblyInfo.cs*. You could use this assembly version value to determine the schema version, but they're not really directly related; there might be all kinds of reasons why the assembly version changes, even though the schema version does not. It's better to define a specific schema version numbering scheme, and this can be as simple as an integer value. The logical place to declare this is in your *DataContext* class.

```
public class ShoppingDataContext : DataContext
{
    private const int version = 1;
    public int Version
    {
        get { return version; }
        private set { }
    }

    public ShoppingDataContext()
        : base("Data Source=isostore:/ShoppingList.sdf")
    {
    }

    public Table<ShoppingItem> ShoppingItems;
}
```

You can then test for the version number in your viewmodel class when you set up the database connection. To do this, use the *DatabaseSchemaUpdater* class, which is specific to the phone and defined in the *Microsoft.Phone.Data.Linq* namespace. When you attempt to connect to the database, if it doesn't already exist, you can simply go ahead and create it and set its version number to the current version number in your *DataContext* class. On the other hand, if the database does already exist, then this could be an upgrade scenario. In this case, you must test the version number of the existing database. If this is the first version of the database, then the *DatabaseSchemaVersion* property will be set to 0 by default. For any other version, the property is set to whatever value you set it to when you update it.

So, in the shopping list application, if you're updating from the old version, you need to add the new *Quantity* column (as defined in the new *ShoppingItem* table type), update the version number, and then call the *Execute* method to persist the changes.

```
public MainViewModel()
{
    shoppingDb = new ShoppingDataContext();
    if (!shoppingDb.DatabaseExists())
    {
        shoppingDb.CreateDatabase();

        DatabaseSchemaUpdater schemaUpdater = shoppingDb.CreateDatabaseSchemaUpdater();
        schemaUpdater.DatabaseSchemaVersion = shoppingDb.Version;
        schemaUpdater.Execute();
    }
    else
    {
        DatabaseSchemaUpdater schemaUpdater = shoppingDb.CreateDatabaseSchemaUpdater();
        if (schemaUpdater.DatabaseSchemaVersion < shoppingDb.Version)
        {
            schemaUpdater.AddColumn<ShoppingItem>("Quantity");
            schemaUpdater.DatabaseSchemaVersion = shoppingDb.Version;
            schemaUpdater.Execute();
        }
    }
}
```

If the new column is not nullable, then when you update the schema, you must also set a value for this column for each existing row in the database. It's up to you to decide what value to set. In the shopping list application, it would be reasonable to set the value to 1, but the specifics will vary according to your business logic requirements. Note also that only additive schema changes are supported right now; it is not possible to remove columns or rows from an existing database. In a new version of your application, you might choose not to use some of the old columns/tables, but they will remain physically in the database. Also, if they were not nullable, then whenever you add new rows, you'll have to add some default value for the unused columns.

Finally, If you define a new *Table* class with a new non-nullable column, you cannot read existing data into this *Table* class, because the null values for the non-nullable column will throw an exception.

In this scenario, the simplest approach is to use two versions of your schema: on startup, you would read the existing data by using the old schema. Then, you would create a new database, with the new schema, and populate it with the old data, filling in any non-nullable columns with some default value. After that, you can delete the old database file.

Associations

It is very common for tables to have schema relationships between them. Consider Figure 18-6, which shows the first two pivots in the *CoffeeStore* application in the sample code. The application provides three pivot items (machines, beans, and cups), where each pivot corresponds to the *Category* table in the database, and lists a collection of items from the *Product* table.



FIGURE 18-6 The *CoffeeStore* application with table associations for machines and beans.

All the products reside in the one *Product* table in the database, but the UI displays each category of product in a separate pivot. The pivot lists are data-bound to collections in the viewmodel, and the viewmodel filters the single *Product* table into each of the three lists on the basis of the *Product-Category* table relationship. There are three categories, and all products map to one of these three categories. Thus, there is a one-to-many relationship between the *Category* table and the *Product* table. The relationship is established in code by using three artifacts: *EntitySet*, *EntityRef*, and *Association*, as shown in Figure 18-7.

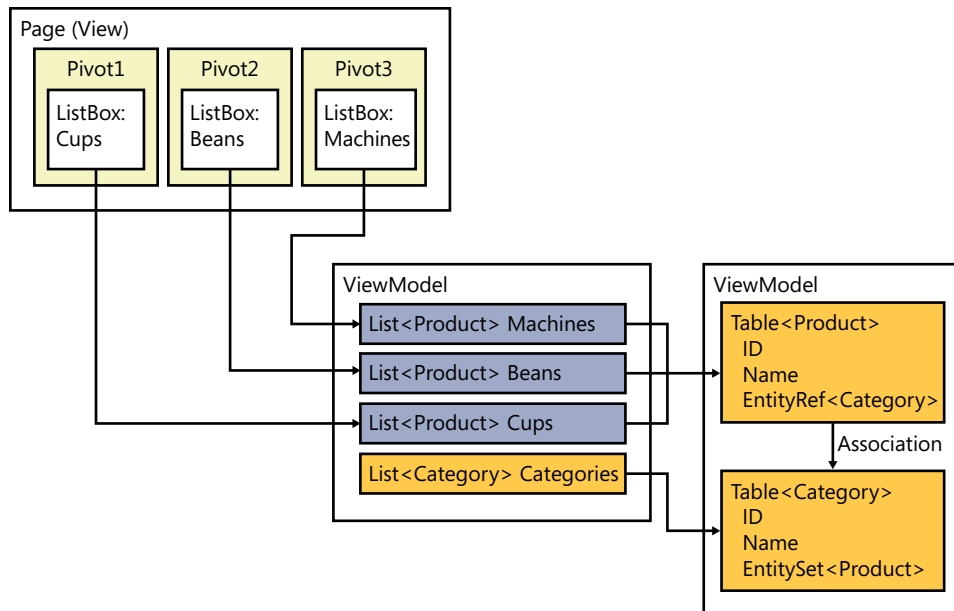


FIGURE 18-7 EntitySet, EntityRef, and Association.

The most interesting code is in the *Product* and *Category* table class definitions. The *Product* class includes a *Category* property that is defined to be an *Association*. The association specifies the corresponding storage object, which maps to an *EntityRef<T>*. The *EntityRef* is a wrapper for the *Category* object, decorating it with additional behavior. The key in this table is *categoryId*. This is a foreign key with a matching primary key in the *Category* table named *Id*. The property getter simply returns the entity from the *EntityRef*. The property setter updates the entity value, and then also updates the *Id* property on that entity; that is, the foreign key in the *Category* table.

```
[Column]
internal int categoryId;

private EntityRef<Category> category;

[Association(Storage = "category", ThisKey = "categoryId", OtherKey = "Id", IsForeignKey =
true)]
public Category Category
{
    get { return category.Entity; }
    set
    {
        category.Entity = value;
        if (value != null)
        {
            categoryId = value.Id;
        }
        NotifyPropertyChanged("Category");
    }
}
```

The *Category* table defines the other side of this association. On this side, the association is to the *EntitySet<T>*, where *T* is the *Product* type. The *EntitySet* is effectively the set of rows from the associated table (*Product*, in this example) to which this *Category* maps. Note also that the *Category* constructor initializes this *EntitySet* object.

```
private EntitySet<Product> products;

[Association(Storage = "products", OtherKey = "categoryId", ThisKey = "Id")]
public EntitySet<Product> Products
{
    get { return products; }
    set { products.Assign(value); }
}

public Category()
{
    products = new EntitySet<Product>();
}
```

The *DataContext* defines two collections: one for the total list of *Product* rows, and the other for the list of *Category* rows.

```
public class CoffeeDataContext : DataContext
{
    public CoffeeDataContext()
        : base("Data Source=isostore:/CoffeeStore.sdf")
    {
    }

    public Table<Product> Products;
    public Table<Category> Categories;
}
```

The viewmodel defines four collections. To load the data from the database into the viewmodel collections, you first load the *Category* collection, and then iterate this to populate each of the three *Product* collections, filtered on the corresponding category *Name*. In this way, you divide up the underlying *Product* items into three separate collections for use in the UI.

```
public ObservableCollection<Category> Categories { get; private set; }
public ObservableCollection<Product> Machines { get; private set; }
public ObservableCollection<Product> Beans { get; private set; }
public ObservableCollection<Product> Cups { get; private set; }

public void LoadData()
{
    if (Categories == null)
    {
        IQueryable<Category> query =
            from Category cat in coffeeDb.Categories select cat;
        Categories = new ObservableCollection<Category>(query);
    }

    foreach (Category cat in Categories)
    {
```

```

switch (cat.Name)
{
    case "machines":
        Machines = new ObservableCollection<Product>(cat.Products);
        break;
    case "beans":
        Beans = new ObservableCollection<Product>(cat.Products);
        break;
    case "cups":
        Cups = new ObservableCollection<Product>(cat.Products);
        break;
}
}
}

```



Note See the Performance Considerations section, later in the chapter, for further enhancements that you should make when using entity associations.

Isolated Storage Explorer Tool

Sometimes, it makes sense to create your SDF file in code as part of your application. On the other hand, sometimes you want to start your application with a set of preloaded data, in which case, it is not appropriate to create that data from scratch in your application. In this scenario, you should create your SDF and prepopulate it in advance.

You have two choices for creating an initial database for use in your application: either create it in code in a helper application, or create it with external tools such as SSMS or T-SQL scripts. Note, however, that the only supported approach is to create the database in code. The reason for this is that the underlying database technology should be transparent to the developer. Right now, it happens to use SQL-CE under the covers, but developers should not rely on this implementation. So long as you use only the database APIs provided in the platform—both for creating your database, and for all subsequent access in your application—then there can never be any inconsistency. As soon as you start using SSMS or T-SQL, working on a SQL-CE SDF file directly, you run the risk of introducing some behavior that is not consistent with the phone APIs.

Consider the example applications shown in the screenshots in Figure 18-8. These show the *DbCreator* application in the sample code that creates the database, and the *DbConsumer* application that consumes the database. Only the *DbConsumer* would be published. The UI is very similar in both applications, which is a deliberate testing technique.

CONTOSO			CONTOSO		
db creator			db consumer		
1	HL Road Frame - Black, 58	\$1,431.50	1	HL Road Frame - Black, 58	\$1,431.50
2	HL Road Frame - Red, 58	\$1,431.50	2	HL Road Frame - Red, 58	\$1,431.50
3	Sport-100 Helmet, Red	\$34.99	3	Sport-100 Helmet, Red	\$34.99
4	Sport-100 Helmet, Black	\$34.99	4	Sport-100 Helmet, Black	\$34.99
5	Mountain Bike Socks, M	\$9.50	5	Mountain Bike Socks, M	\$9.50
6	Mountain Bike Socks, L	\$9.50	6	Mountain Bike Socks, L	\$9.50
7	Sport-100 Helmet, Blue	\$34.99	7	Sport-100 Helmet, Blue	\$34.99
8	AWC Logo Cap	\$8.99	8	AWC Logo Cap	\$8.99
9	Long-Sleeve Logo Jersey, S	\$49.99	9	Long-Sleeve Logo Jersey, S	\$49.99
10	Long-Sleeve Logo Jersey, M	\$49.99	10	Long-Sleeve Logo Jersey, M	\$49.99
11	Long-Sleeve Logo Jersey, L	\$49.99	11	Long-Sleeve Logo Jersey, L	\$49.99
12	Long-Sleeve Logo Jersey, XL	\$49.99	12	Long-Sleeve Logo Jersey, XL	\$49.99
13	HL Road Frame - Red, 62	\$1,431.50	13	HL Road Frame - Red, 62	\$1,431.50
14	HL Road Frame - Red, 44	\$1,431.50	14	HL Road Frame - Red, 44	\$1,431.50
15	HL Road Frame - Red, 48	\$1,431.50	15	HL Road Frame - Red, 48	\$1,431.50
16	HL Road Frame - Red, 52	\$1,431.50	16	HL Road Frame - Red, 52	\$1,431.50
17	HL Road Frame - Red, 56	\$1,431.50	17	HL Road Frame - Red, 56	\$1,431.50
18	LL Road Frame - Black, 58	\$337.22	18	LL Road Frame - Black, 58	\$337.22
19	LL Road Frame - Black, 60	\$337.22	19	LL Road Frame - Black, 60	\$337.22
20	LL Road Frame - Black, 62	\$337.22	20	LL Road Frame - Black, 62	\$337.22
21	LL Road Frame - Red, 44	\$337.22	21	LL Road Frame - Red, 44	\$337.22
22	LL Road Frame - Red, 48	\$337.22	22	LL Road Frame - Red, 48	\$337.22

FIGURE 18-8 The *DbCreator* helper application (on the left), and the *DbConsumer* application (right).

DbCreator is a helper application whose sole purpose is to create the database that the final application will use. Using this approach, it is also useful (although obviously not essential) for the helper application to display the data so that you can readily confirm that the data is created correctly. As you've seen in previous examples, you would create an application with a custom *DataContext*, one or more *Table* classes, and a viewmodel. In this helper application, the database is created in one method, which can be invoked from the UI.

```

internal void CreateData()
{
    Products = new ObservableCollection<Product>();

    Products.Add(new Product()
        { Id = 680, Name = "HL Road Frame - Black, 58", Price = 1431.5 });
    Products.Add(new Product()
        { Id = 706, Name = "HL Road Frame - Red, 58", Price = 1431.5 });
    Products.Add(new Product()
        { Id = 707, Name = "Sport-100 Helmet, Red", Price = 34.99 });
    ...etc, for all rows to be inserted into the database.

    adventureWorksDb.Products.InsertAllOnSubmit(Products);
    adventureWorksDb.SubmitChanges();
}

```



Note A common approach to creating initial data is to create it in Excel and then use Excel's export feature to write the data out as an XML file. You can then use LINQ-to-XML in your application to read the data in.

Whichever route you take to create the database, you then need to transfer the generated database onto the device or emulator for testing. Later, you will also need to add it to your main application project for packaging as part of the solution. During development, you can simply copy the database file manually, using the Isolated Storage Explorer tool (ISETool), which ships with the Windows Phone 7.1 SDK, as described here:

1. Run your helper application (either on the emulator, or on a device), and then create the database.
2. Use the ISETool to take a snapshot of your database from the emulator or device on to your local desktop.
3. At any time thereafter, you can deploy the database to the isolated storage for the application (so long as the application itself is installed). This becomes very useful while testing, during which you can deploy updated or known-good versions of the database at any time.

The ISETool is a simple command-line tool that you can use to import or export SDF files to and from isolated storage, either for the emulator or for a physical phone. The command-line syntax for the tool is as follows:

```
ISETool <ts|rs> <xd|de> <appId> <path>
```


The command-line arguments are described in Table 18-1.

TABLE 18-1 Isolated Storage Explorer Tool Arguments

Parameter	Description
<i>ts</i>	Takes a snapshot of contents of the specified application's isolated storage and copies it to a desktop location.
<i>rs</i>	Restores a snapshot from a desktop location to isolated storage.
<i>xd</i>	Copies to or from the XDE Emulator.
<i>de</i>	Copies to or from an attached physical device.
<i>appld</i>	The Product ID of the project, as specified in the application's WAppManifest.xml file.
<i>path</i>	The absolute desktop path from which files are downloaded or to which they are uploaded.

When taking or restoring a snapshot, any existing files in the target folder will be deleted before the snapshot is transferred. To download the database file for the *DbCreator* application on the emulator, use the following command line (for 64-bit computers, replace %ProgramFiles% with %ProgramFiles(x86)%):

```
"%ProgramFiles%\Microsoft SDKs\Windows Phone\v7.1\Tools\IsolatedStorageExplorerTool\ISETool.exe"  
ts xd {7c6c23c5-6d25-4387-9cce-3ac4815ff40d} C:\Temp\DbCreator
```

This will create (or overwrite) a folder named *IsolatedStore* in the specified path and dump the entire contents of the application's isolated storage into that folder. This storage can include persisted application settings and other files as well as the database (SDF) file. Having taken a copy onto the desktop, you can then deploy the consuming application (which does not create its own data), and then use the ISETool again to restore the snapshot to the emulator or device. Note that you must specify the *IsolatedStore* folder in the path parameter for a restore operation, but not for taking a snapshot.

```
"%ProgramFiles%\Microsoft SDKs\Windows Phone\v7.1\Tools\IsolatedStorageExplorerTool\ISETool.exe"  
rs xd {EF2BBD34-6CBE-43A2-B6F5-A521E28C39EB} C:\Temp\DbCreator\IsolatedStore
```

As you can see from the preceding ISETool commands, the helper application that creates the database and the final application that consumes the database are different applications, with different *ProductId* values. The ISETool is very simple: it copies isolated storage for the application specified; it doesn't require that the snapshot is restored to the same application as that from which it was taken.

While you have the SDF file on the desktop, you can also use tools such as SSMS and Visual Studio's Server Explorer to query the SDF file, including both the data and the schema. To do this, in SSMS, go to the Object Explorer and click Connect. Specify SQL Server Compact as the server type, specify the path to the SDF, and then click Connect. You can then use T-SQL to work with the database, as shown in Figure 18-9. It's worth repeating at this point that creating an SDF file outside a Windows Phone application and then consuming it within a Windows Phone application is not supported. However, it might be useful to use desktop tools to examine the database after you've created it in a Windows Phone helper application.

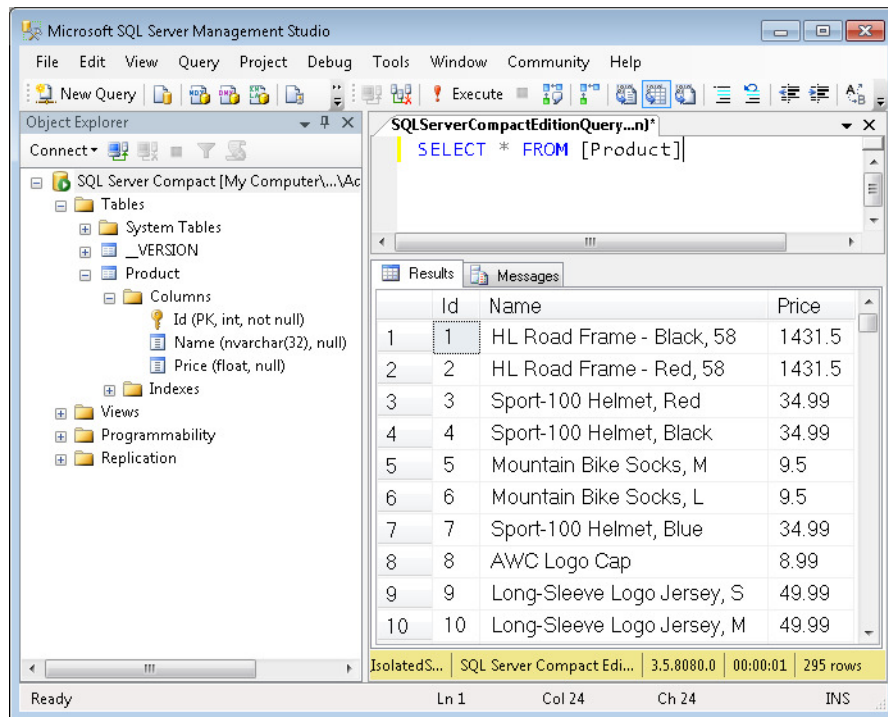


FIGURE 18-9 Working with a local database in SSMS.

To work with the SDF in Visual Studio, open the Server Explorer, and then click Add A Connection. In the Add Connection dialog box, change the data source to SQL Server Compact 3.5. Next, browse to specify the path to the SDF, and then click OK, as shown in Figure 18-10. SSMS and Visual Studio offer similar data manipulation language (DML) and querying capabilities for querying and modifying SDF databases, although SSMS has more data definition language (DDL) capabilities,

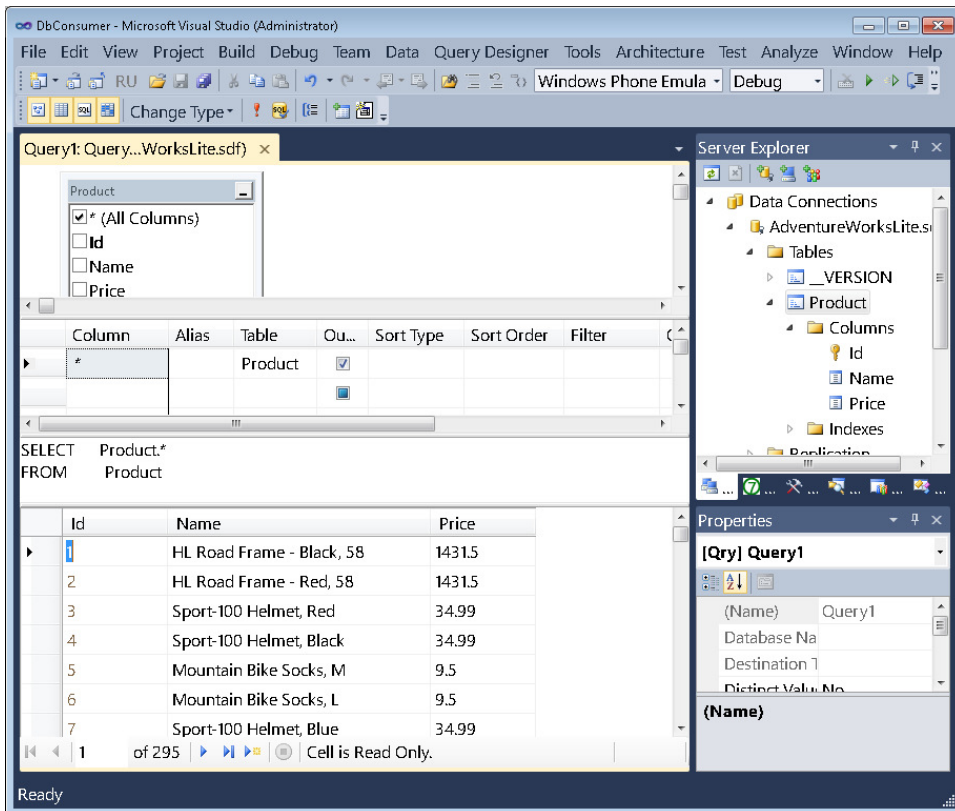


FIGURE 18-10 Working with a local database in Visual Studio Server Explorer.

There's also an open-source SQL Server Compact Toolbox tool available on codeplex at <http://sqlcetoolbox.codeplex.com/>. This is a Visual Studio add-in that provides additional features for working with SQL-CE databases, including scripting database objects, data editing, SQL Server database graphs, database diffs, and so on, as shown in Figure 18-11.

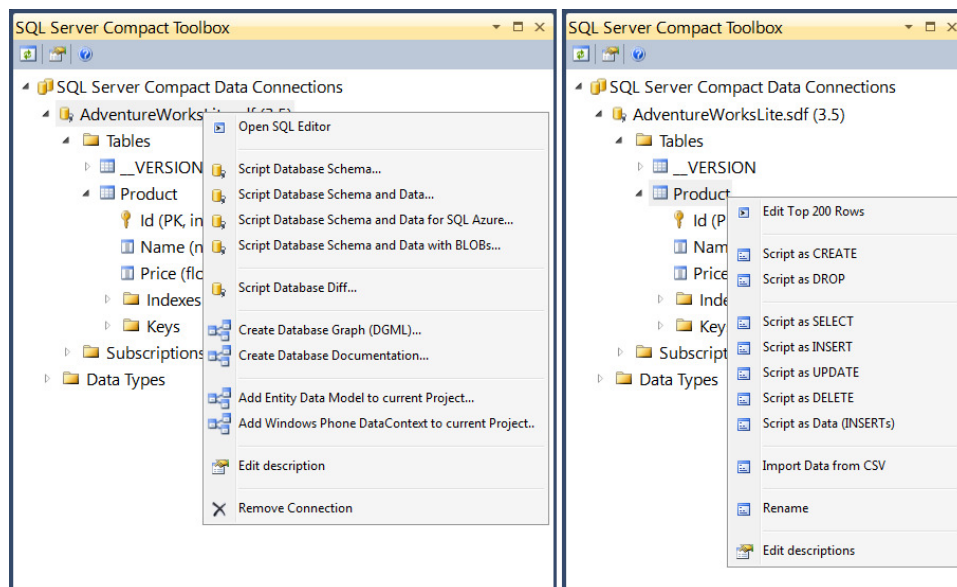


FIGURE 18-11 The Visual Studio SQL Server Compact Toolbox add-in.

The ISETool can also be useful when you're testing the upgrade scenario. You can run any version of the application, and then take a snapshot of the database with the ISETool, before making any schema changes. You can then redeploy any version of the database at will, ensuring that you can thoroughly test the upgrade scenario. Finally, the tool can be used to erase isolated storage; all you have to do is to restore a snapshot, specifying an empty *IsolatedStore* folder. As always, restoring a snapshot will overwrite the target isolated storage, and in this case, will result in empty storage.

When you eventually come to publish your application, you cannot, of course, use the ISETool to deploy the database. So, you must add the SDF file to your application solution so that it is packaged in your XAP before submitting to marketplace.

As a general principle, you should not package such data as a resource embedded in your assembly, because the user will pay the price of a bloated assembly (slower load time, more memory consumption, and possibly more battery consumption) every time he runs your application. Instead, you should set its build action to *Content* so that it is loosely packed in the XAP, not in the assembly. If the data is always read-only, that's all you need to do. You can access the file from the install location at runtime.

More likely, the data is read-write (or at least, the database will be read-write, even if you're only appending new data). In this case, you can fetch the file from the application install folder on startup, and copy it to the application's isolated storage. You only need to do this upon first startup of the application (or if you subsequently upgrade the application). The *DbConsumer_PackagedDB* solution in the sample code takes this approach. The SDF file is added to the project with the build action set to *Content*.

You want to copy this file from the install folder to isolated storage upon first startup, but choosing a good place in the code to do this is not so simple. You typically need the data to be fully available for use before the first page that uses the data is loaded. In the simple examples that you've seen thus far, this is generally the first page in the application. In a realistic application, it might well be that only a small subset of the data—if any—is likely to be needed so early, and in this case you can defer moving the data until after the critical startup phase. Recall that there are marketplace certification constraints on how fast your application loads its first page and how soon it is responsive to user interaction.

For the purposes of this simple application, you can copy the data in the *Application_Launching* event handler in the *App* class. First, you get the root isolated storage for the application and determine if the store already contains the database; you only proceed if it doesn't (this will be upon first startup). Next, use *GetResourceStream* to access the original SDF file and read it into a byte array in memory, create a new file in isolated storage (in this example, using the same name as the original, although this is not required), and then write out the contents of the byte array.

```
public static string SdfName = "AdventureWorksLite.sdf";

private void Application_Launching(object sender, LaunchingEventArgs e)
{
    using (IsolatedStorageFile isoFile =
        IsolatedStorageFile.GetUserStoreForApplication())
    {
        if (!isoFile.FileExists(SdfName))
        {
            StreamResourceInfo streamInfo = App.GetResourceStream(
                new Uri(SdfName, UriKind.Relative));
            if (streamInfo != null)
            {
                byte[] bytes = new byte[streamInfo.Stream.Length];
                streamInfo.Stream.Read(bytes, 0, bytes.Length);

                using (IsolatedStorageFileStream isoStream =
                    isoFile.OpenFile(SdfName, FileMode.OpenOrCreate))
                {
                    isoStream.Write(bytes, 0, bytes.Length);
                }
            }
        }
    }
}
```



Note Even when you defer copying the database to a less-critical time, if your database is large, you should consider defining a chunk size (maybe as large as 1 MB, or so) and then perform the operation in chunks. If necessary, you can even examine your use of the data in the initial pages loaded in the application, and then devise a suitable chunking algorithm that matches your expectations of how the user will navigate through the application. If that doesn't give you enough of a performance benefit, you can also consider compressing the database into a zip file before you add it to the XAP as *Content*. The XAP itself is compressed, but then exploded onto the device, and you want to minimize the size of the database in the application install folder, because after you copy it to isolated storage, it is unlikely to be used again in the application (barring some "application reset" scenario).

Finally, be aware that if your application needs only read access to the database, you don't have to copy it to isolated storage. Instead, add it to the project as a *Content* file, and when the application is installed, the SDF will be deployed to the application's install folder. Applications cannot write to this location, but they can read from it. The only change you need to make to the code is in the connection string in the *DataContext* class: replace the *isostore* virtual root with *appdata*.

```
public AdventureWorksDataContext()  
    //: base("Data Source=isostore:/AdventureWorksLite.sdf")  
    : base("Data Source=appdata:/AdventureWorksLite.sdf;File Mode=read only;")  
{  
    ObjectTrackingEnabled = false;  
}
```

If you do specify an *appdata* path, you must also specify read-only mode, or the application will throw an exception upon startup. By the same token, if you deploy to an *appdata* path, any attempt to write to the database (including deletes) will throw an exception. If your database is read-only, you can also set *ObjectTrackingEnabled* to *false*. Doing this will suppress the object tracking that the *DataContext* normally performs as part of change management. This will improve the performance of your application and also reduce the working set. As you would expect, if you turn off object tracking and then attempt to submit any changes to the database, this will throw an exception.

Performance Considerations

You've seen already that you can improve performance by suppressing change tracking for read-only databases. You'll also save time in this scenario by avoiding copying the SDF from the application folder to isolated storage.

Another optimization that you should always adopt is to implement both *INotifyPropertyChanged* and *INotifyPropertyChanging* interfaces. So far, the table classes that have been defined have implemented *INotifyPropertyChanged* only. This is sufficient to complete the data-binding chain between the UI and the database. However, you can save both time and memory by implementing *INotifyPropertyChanging* also. The reason for this lies in the way LINQ-to-SQL performs change tracking. To determine whether or not an entity has changed, the system will create two copies of each object. One is a faithful copy of the original object representing the data in the database. The other copy is

the one that is changed by the application. When you eventually submit changes to the database, the system compares the two copies, and submits only items that have actually changed.

The purpose of *INotifyPropertyChanged* is to notify the system that a change *has* taken place. By contrast, *INotifyPropertyChanging* notifies the system that a change *is about* to take place. The key here is that if you implement *INotifyPropertyChanging*, the system can use this as the trigger to create the second copy of the data, and avoid doing so, otherwise. The following code shows how you would update the *ShoppingItem* table class from earlier examples:

```
[Table]
public class ShoppingItem : INotifyPropertyChanged, INotifyPropertyChanging
{
    private int id;

    [Column(
        IsPrimaryKey = true, IsDbGenerated = true,
        DbType = "INT NOT NULL Identity", CanBeNull = false,
        AutoSync = AutoSync.OnInsert)]
    public int Id
    {
        get { return id; }
        set
        {
            if (id != value)
            {
                NotifyPropertyChanging("Id");
                id = value;
                NotifyPropertyChanged("Id");
            }
        }
    }

    private String name;

    [Column]
    public String Name
    {
        get { return name; }
        set
        {
            if (name != value)
            {
                NotifyPropertyChanging("Name");
                name = value;
                NotifyPropertyChanged("Name");
            }
        }
    }

    public event PropertyChangedEventHandler PropertyChanged;
    private void NotifyPropertyChanged(string propertyName)
    {
        PropertyChangedEventHandler handler = PropertyChanged;
        if (handler != null)
```

```

        {
            handler(
                this, new PropertyChangedEventArgs(propertyName));
        }
    }

    public event PropertyChangedEventHandler PropertyChanging;
    private void NotifyPropertyChanging(string propertyName)
    {
        PropertyChangedEventHandler handler = PropertyChanging;
        if (handler != null)
        {
            handler(
                this, new PropertyChangedEventArgs(propertyName));
        }
    }
}

```

Keep in mind that if you're implementing *INotifyPropertyChanging* in your entity classes, then you have a further opportunity to notify the system of impending data changes (and therefore also improve performance) when you're using entity associations. Referring back to the earlier *CoffeeStore* sample, when you construct the *EntitySet<T>*, you can use the overload that takes two *Action<T>* parameters; these are delegates that will be invoked when a *Product* entity is added to or removed from the set of mapped rows for this *Category*.

```

public Category()
{
    //products = new EntitySet<Product>();
    products = new EntitySet<Product>(
        new Action<Product>(AttachProduct),
        new Action<Product>(DetachProduct));
}

private void AttachProduct(Product product)
{
    NotifyPropertyChanging("Product");
    product.Category = this;
}

private void DetachProduct(Product product)
{
    NotifyPropertyChanging("Product");
    product.Category = null;
}

```

Another optimization is to add a version column to your table classes. This is specific to Windows Phone, and it dramatically improves bulk updates and deletes.

```

[Column(IsVersion = true)]
private Binary version;

```

The *DbCreator_WithVersion* and *DbCreator_NoVersion* solutions in the sample code illustrate this optimization by using an *UpdatePrices* method in the viewmodel that executes an arbitrary price update for all rows in the table. Simple tests (over 100 iterations) show that executing this operation

without the version column takes an average of 530 milliseconds; with the version column, it takes an average of 50 milliseconds.

```
internal void UpdatePrices()
{
    Stopwatch stopwatch = Stopwatch.StartNew();
    foreach (Product p in Products)
    {
        p.Price *= 1.1;
    }
    adventureWorksDb.SubmitChanges();
    stopwatch.Stop();
    Debug.WriteLine("elapsed time = {0}", stopwatch.ElapsedMilliseconds);
}
```

Database Encryption

If you want to increase security in your application, you can choose to encrypt your database. This can be done only at the time the database is created, so it is useful only in scenarios for which the user provides the password and the application creates the database dynamically at runtime. It is not useful if you need to create the database in advance. Figure 18-12 shows the *ShoppingList_CRUD_Encrypted* solution in the sample code, which takes this approach.

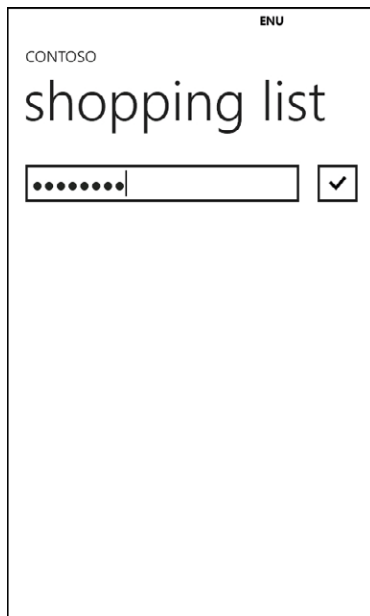


FIGURE 18-12 Encrypting a database with a password.

The UI initially presents only a *PasswordBox* and a *Button* control inside a *Grid*. The user can enter a password and tap the button to confirm. At this point, the application attempts to connect to the

database with this password. If that succeeds, it sets up the data-binding between the viewmodel and the UI, as normal, shows the rest of the UI, and hides the password grid.

```
private void confirmPassword_Click(object sender, RoutedEventArgs e)
{
    if (App.ViewModel.ConnectDatabase(password.Password))
    {
        DataContext = App.ViewModel;
        contentPanel.Visibility = Visibility.Visible;
        passwordGrid.Visibility = Visibility.Collapsed;
    }
}
```



Note In general, it is a best practice to prompt for a new password *twice*. That is, any time you ask users to create a password you should ask her to input it two times, and then compare the two copies, thus helping to ensure that the user didn't accidentally mistype the password by mistake on the first attempt.

The viewmodel exposes a *ConnectDatabase* method, callable from the UI, which takes the user's password, initializes the custom *DataContext* object with this, and then attempts to connect to the database and load the data, if there is any. The same mechanism is used to create the database the first time, and for each subsequent attempt to open it.

```
public bool IsConnected;
private String password;

internal bool ConnectDatabase(String userPassword)
{
    password = userPassword;

    try
    {
        shoppingDb = new ShoppingDataContext(password);
        if (!shoppingDb.DatabaseExists())
        {
            shoppingDb.CreateDatabase();
        }
        LoadData();
        IsConnected = true;
    }
    catch (Exception ex)
    {
        Debug.WriteLine(ex.ToString());
        IsConnected = false;
    }
    return IsConnected;
}
```

At the bottom layer of the application code, the *DataContext* uses the password to compose the database connection string, passed to the base class.

```

public ShoppingDataContext(String password)
    : base(String.Format(
        "Data Source=isostore:/ShoppingList.sdf;Password='{0}'", password))
{
}

```

If you create an encrypted database in this manner, you can still take and restore snapshots by using the ISETool, but you cannot open it with any desktop tools such as Visual Studio or SSMS.

Encrypting Data and Credentials

In Chapter 13, “Security,” you saw how to use the *AesManaged* and related classes to encrypt data locally on the phone. The one major piece of missing functionality in version 7 was the ability to store passwords and other security credentials on the phone itself. This meant that whenever you needed a password, such as whenever you encrypt or decrypt data, you would have to ask the user to supply it. Windows Phone 7.1 introduces a reduced-scope form of the Data Protection API (DPAPI), with which you can encrypt and decrypt confidential data such as passwords. Every application receives its own decryption key, which is created when the user runs the application for the first time, and which persists on the phone across application updates and phone reboots. When your code uses the DPAPI, this implicitly uses the auto-generated decryption key, such that the data is always encrypted on a per-application basis.

Figure 18-13 shows a variation of the sample discussed in Chapter 13. This version is named *SimpleEncryption_DPAPI* in the sample code.

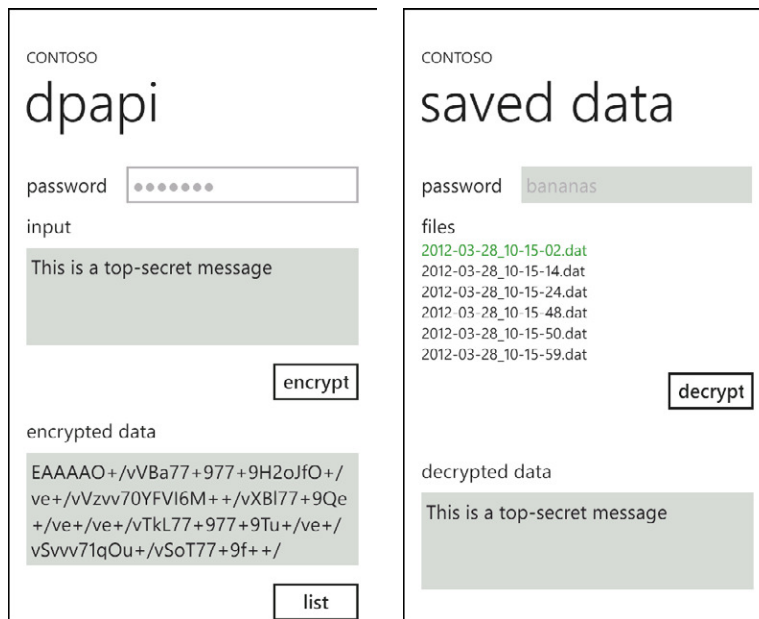


FIGURE 18-13 Encrypting and decrypting passwords and data by using DPAPI.

This version of the application modifies the UI slightly from the version in Chapter 13. Here, the user is asked to enter a password, and this single password is now used for all encryption. Once she's entered a password, the corresponding *PasswordBox* is disabled. When she goes to the second page to see a list of stored encrypted data, the application shows a read-only *TextBox* rather than a *PasswordBox* and populates this *TextBox* directly from the stored encrypted password (this is for illustration only; in a real application, you would not normally display a stored password).

In the code, you first declare a string for the name of the file in which you'll store the encrypted password. This is declared in the *App* class so that it is accessible across both pages.

```
public const String PasswordFile = "passwordFile.dat";
```

Next, the code-behind for the main page: when the user taps the Encrypt button, you fetch her password and input data, as before, and then encrypt and store the data in isolated storage, as before. The additional behavior is that you now also encrypt and store the password itself. In the *EncryptAndStorePassword* helper method, you convert the password to a byte array, and then encrypt it by using the DPAPI *Protect* method. Next, create a file in the application's isolated storage, and then write the encrypted password data out to the file.

```
private void encrypt_Click(object sender, RoutedEventArgs e)
{
    String filePath = String.Format("{0:yyyy-MM-dd_hh-mm-ss}.dat", DateTime.Now);
    using (IsolatedStorageFile store = IsolatedStorageFile.GetUserStoreForApplication())
    {
        EncryptAndStorePassword(store, passwordBox.Password);
        passwordBox.IsEnabled = false;
        EncryptData(store, filePath, passwordBox.Password);
        ShowEncryptedData(store, filePath);
    }
}

private void EncryptAndStorePassword(IsolatedStorageFile store, String password)
{
    byte[] passwordBytes = Encoding.UTF8.GetBytes(password);
    byte[] protectedBytes = ProtectedData.Protect(passwordBytes, null);

    using (IsolatedStorageFileStream fileStream =
        store.CreateFile(App.PasswordFile))
    {
        using (Stream writer = new StreamWriter(fileStream).BaseStream)
        {
            writer.Write(protectedBytes, 0, protectedBytes.Length);
        }
    }
}
```



Note You still encrypt data with the password, not the protected password, so you can recover the data even in the event that the protection keys change (for example, if your application supports roaming across different devices or platforms via the cloud, each device will have a different protection mechanism or key but the password stays the same).

On the second page, the previous functionality remains broadly as before: you provide a list of encrypted files in isolated storage (excluding the password file) and allow the user to select a file from the list. Then, you decrypt this file by using the stored password. The additional behavior here is that you no longer prompt the user to enter the password. Instead, you fetch the encrypted password from isolated storage, decrypt it by using the DPAPI *Unprotect* method, and then convert it from a byte array to a string for subsequent display in the UI.

```
private String GetPassword()
{
    using (IsolatedStorageFile store = IsolatedStorageFile.GetUserStoreForApplication())
    {
        using (IsolatedStorageFileStream fileStream =
            store.OpenFile(App.PasswordFile, FileMode.Open, FileAccess.Read))
        {
            using (Stream reader = new StreamReader(fileStream).BaseStream)
            {
                byte[] protectedBytes = new byte[reader.Length];
                reader.Read(protectedBytes, 0, protectedBytes.Length);
                byte[] decryptedBytes = ProtectedData.Unprotect(protectedBytes, null);
                String passwordText =
                    Encoding.UTF8.GetString(decryptedBytes, 0, decryptedBytes.Length);
                return passwordText;
            }
        }
    }
}
```

Notice how there's an overload of both the *Protect* and *Unprotect* methods that takes an additional parameter, which is a byte array known as the *entropy* information. This is used to increase the complexity of the encryption. If you provide entropy information in the *Protect* call, you must provide that same entropy information in the corresponding *Unprotect* call.

Keep in mind also that this technique can be used in combination with a local database. That is, if only a relatively small proportion of the data needs to be encrypted, it might make sense to use the DPAPI to encrypt just those items before submitting them to the database, instead of encrypting the entire database. There's an obvious trade-off here: encrypting the entire database will generally be slower. On the other hand, encrypting only selected items of data will increase complexity and risk. Also, this would generally only be useful if the data is specific to the phone where it is stored and will never need to be read on any other device.

Contacts and Calendar

With Windows Phone 7.1, you now have access to the user's contacts and calendar data, albeit in a read-only manner. This is another example of the ecosystem model of Windows Phone development, whereby you can build custom applications that integrate closely with standard features of the phone.



Note With great power comes great responsibility. It is very important that developers consider this feature—and its ramifications—very carefully. By their nature, contacts and calendar information consists of sensitive data, and you should always be very cautious with regard to how you work with this data, if at all. The Windows Phone platform offers a good set of built-in Launchers and Choosers, which you can take advantage of in your application to work with contacts and calendar data. If you decide that your application really needs more than that, then you should have a privacy policy in place and adhere to it. For example, you should not send any sensitive data anywhere off the device, or if you do, then you should send only hashes and use Secure Sockets Layer (SSL) for communication. You should not store the data independent of the original contacts/calendar stores, or if you do, you should store it encrypted, and you should give the user a way to erase this data. You should give the user a way to turn off the feature(s) in your application that uses contacts, calendars, and so on.

The phone platform provides an aggregated view of the user's contacts and calendar (not including the social media data, such as Facebook) and exposes these views via the *Contacts* and *Appointments* classes in the API. Figure 18-14 shows the *SimpleContacts* and *SimpleCalendar* applications in the sample code, with which the user can tap the button to fetch a list of contacts or appointments and display them in the UI.

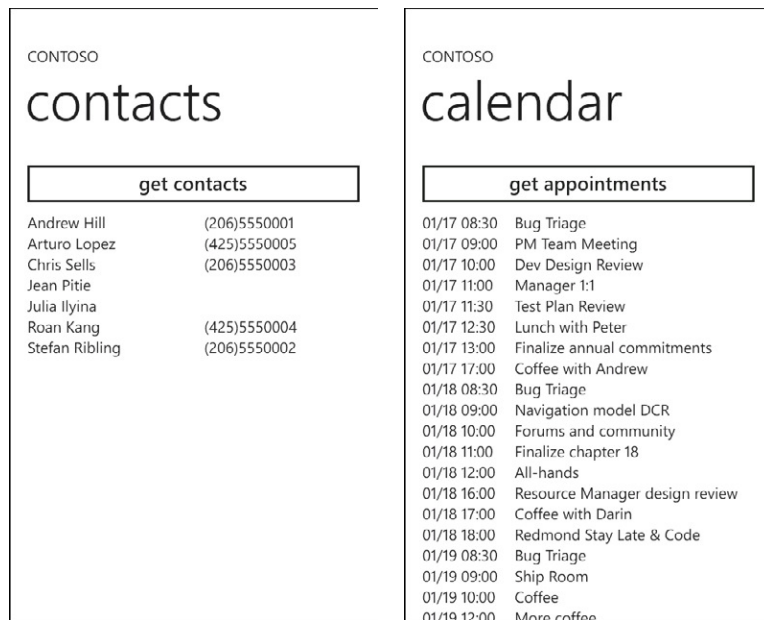


FIGURE 18-14 You can use the *Contacts* and *Appointments* APIs in your application.

You can test the *Contacts* API on either the emulator or the phone, although the emulator has only a very small list of *Contacts*. For the *Appointments* API, you must test on a physical device, because the emulator does not contain calendar data.

In both of these sample applications, the views are data-bound to simplified contact and appointment types, for which only the data of interest is defined.

```
public class ContactLite
{
    public String Name { get; set; }
    public String Phone { get; set; }
}

public class AppointmentLite
{
    public String Start { get; set; }
    public String Subject { get; set; }
}
```

The pattern for retrieving contacts and appointment data is very similar. In the contacts application, the page defines a collection of these *ContactLite* objects, which will later be populated based on the full *Contacts* data. When the user taps the button, you hook up the *SearchCompleted* event handler for the *Contacts* type, and then invoke the *SearchAsync* method. This takes three parameters: a search string, a filter to narrow the scope of the search, and an arbitrary object state that will be passed to the *SearchCompleted* handler. In this example, you pass an empty search string, and specify no filter; this will retrieve all *Contact* objects. It is more common to work with a filtered set of contacts, especially as some users might have hundreds or even thousands of them.

In the event handler, the *Contact* objects are returned in the *Results* collection, and you can iterate this to process each one. In this example, you use the *DisplayName* directly, but extract the first *ContactPhoneNumber* from the collection of *PhoneNumbers*. This collection can be empty, so you test for this.

```
public ObservableCollection<ContactLite> Contacts =
    new ObservableCollection<ContactLite>();

private void getContacts_Click(object sender, RoutedEventArgs e)
{
    Contacts contacts = new Contacts();
    contacts.SearchCompleted += contacts_SearchCompleted;
    contacts.SearchAsync(String.Empty, FilterKind.None, null);
}
```

```

private void contacts_SearchCompleted(object sender, ContactsSearchEventArgs e)
{
    if (e.Results != null)
    {
        foreach (Contact c in e.Results)
        {
            if (c.PhoneNumbers != null)
            {
                String phone = String.Empty;
                var p = c.PhoneNumbers.FirstOrDefault<ContactPhoneNumber>();
                if (p != null)
                {
                    phone = p.PhoneNumber;
                }
                Contacts.Add(new ContactLite { Name = c.DisplayName, Phone = phone });
            }
        }
        contactsList.ItemsSource = Contacts;
    }
}

```

The code patterns are closely paralleled in the calendar sample: again, you hook up the *Search Completed* event, and then invoke the *SearchAsync* method. The *Appointments* version of *SearchAsync* takes the start and end times for the search as well as the maximum number of items to return.

```

public ObservableCollection<AppointmentLite> Calendar =
    new ObservableCollection<AppointmentLite>();

private void getAppointments_Click(object sender, RoutedEventArgs e)
{
    Appointments appts = new Appointments();
    appts.SearchCompleted += appts_SearchCompleted;
    appts.SearchAsync(DateTime.Now, DateTime.Now.AddDays(7), 100);
}

private void appts_SearchCompleted(object sender, AppointmentsSearchEventArgs e)
{
    if (e.Results != null)
    {
        foreach (Appointment a in e.Results)
        {
            Calendar.Add(new AppointmentLite {
                Start = a.StartTime.ToString("MM/dd HH:mm"), Subject = a.Subject});
        }
        appointmentsList.ItemsSource = Calendar;
    }
}

```

Note that you can also data-bind directly to the *Contact* and *Appointment* data, if required, including to nested properties. For example, to bind to the first *ContactPhoneNumber* in the collection of *PhoneNumbers*, you would use the *Path* syntax in the *Binding* declaration. As always, the *Path* attribute can be omitted in many cases, so the following two declarations are equivalent:

```

<TextBlock Text="{Binding PhoneNumbers[0].PhoneNumber}"/>
<TextBlock Text="{Binding Path=PhoneNumbers[0].PhoneNumber}"/>

```


You can also use LINQ expressions on the search results. The following example extracts a subset of *Contact* items from the full set returned by the initial search.

```
private void contacts_SearchCompleted(object sender, ContactsSearchEventArgs e)
{
    if (e.Results != null)
    {
        contactsList.DataContext =
            from Contact c in e.Results
            where c.DisplayName.StartsWith("A")
            select c;
    }
}
```

If you know that you want to work with only a subset of the data, then a better approach is to provide a search string and/or a filter condition in the initial search. This improves performance and memory consumption by returning only those items that match the search criteria. So, to achieve the same results set as the preceding code, you would execute a search as follows, passing additional non-null parameters in the *SearchAsync* call:

```
private void getContacts_Click(object sender, RoutedEventArgs e)
{
    Contacts contacts = new Contacts();
    contacts.SearchCompleted += contacts_SearchCompleted;
    //contacts.SearchAsync(String.Empty, FilterKind.None, null);
    contacts.SearchAsync("A", FilterKind.DisplayName, null);
}

private void contacts_SearchCompleted(object sender, ContactsSearchEventArgs e)
{
    //if (e.Results != null)
    //{
        //contactsList.DataContext =
        //    from Contact c in e.Results
        //    where c.DisplayName.StartsWith("A")
        //    select c;
        contactsList.DataContext = e.Results;
    //}
}
```

Sync Framework

In Chapter 11, “Web and Cloud,” you saw how you could build a phone application that uses data stored in the cloud or behind some arbitrary web server. In Chapter 12, “Push Notifications,” you saw how you could push data changes (or at least, notification of data changes) to the phone to keep the data on the phone up to date. Another technology that you can use is the Microsoft Sync Framework. This framework is designed for solutions that potentially involve complex open-ended synchronization ecosystems, for which multiple devices and servers need to be kept in sync. This is particularly useful for scenarios in which multiple parties in the ecosystem can be updating the data independently, yet all need periodically to resynchronize. The framework implements change tracking so that

you don't transfer data back and forth that has not changed. This is sophisticated enough to allow for periodic changes, such that if multiple changes occur during the configured refresh period, then only the final net changes are sent rather than all the intermediate changes. Client applications that use the framework also get the benefits of a local cache, which means that they can continue to function even when offline.

You can download an early release of the Sync Framework from <http://msdn.microsoft.com/en-us/sync>. This includes an SDK for both client and server applications, plus samples and documentation. Be aware that this is an early release, so you should be careful of using this in production systems—in particular, you should be careful to make backups of all data before applying the tools.

Figure 18-15 shows an application (the *SyncClient* solution in the sample code) that pulls data from a SQL Server database sitting behind a web service (the *CoffeeWeb* application in the sample code). With this application, users in the field can enter review scores for cafés. The user can enter new values for the café score, and then tap the refresh button to upload these scores to the server. Many such users could be doing the same thing at the same time, and the sync framework is responsible for resolving update conflicts. In addition, the server computes the average score dynamically, and then this data is returned to the phone clients.



FIGURE 18-15 The sync framework client.

Building this solution involves the following major tasks:

- Set up a suitable data source. This example uses a SQL Server database.
- Generate a sync configuration (stored in an XML file) to determine which columns and tables from the database should be synchronized.

- Provision the database. This adds insert/update/delete triggers and additional tables used for change tracking.
- Generate server-side and client-side proxy code to establish the sync connection.
- Incorporate the server-side proxy code into a suitable web service.
- Incorporate the client-side proxy code into your client application.

Service Configuration

Generating the sync configuration, provisioning the database, and generating the proxy code can all be done by using two helper tools provided with the Sync Framework: SyncSvcUtil (a command-line tool) and SyncSvcUtilHelper (a GUI front-end to the command-line tool). The example application uses a Coffee database, which has two tables to track cafés and café scores. The sample code includes a SQL script to generate this database and provide some initial data. Having set up the database, the next step is to run the SyncSvcUtilHelper tool, as shown in Figure 18-16.

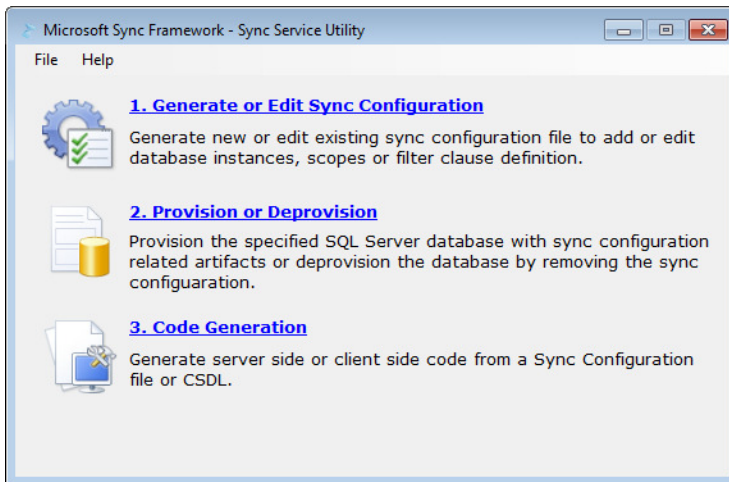


FIGURE 18-16 The SyncSvcUtilHelper tool.

Click the link to generate a sync configuration. The configuration defines the connection to the database, which columns in which tables to be synchronized, and any filters that you want to apply in order to scope down the data transfers. In the tool, you must first specify the name and location of the output configuration file (which will have a .config extension), the server and database names, and connection credentials, as shown in Figure 18-17.

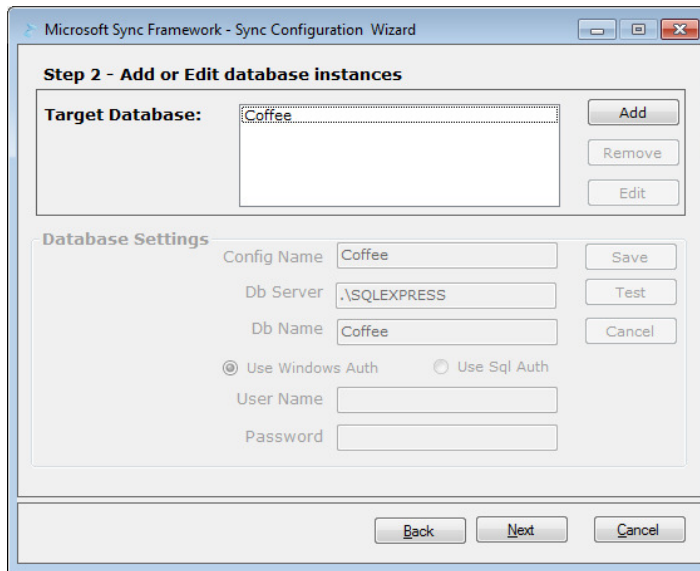


FIGURE 18-17 Configuring the server and database names.

Next, you define at least one sync scope. This is effectively the collection of data to be synchronized. In this example, there is only one scope, named *DefaultScope*, which uses the *dbo* schema name. Then, you choose the columns and tables to be included in this sync scope. This example uses all columns in the *Cafe* table, but none of the columns of the *CafeScore* table, because that table is used for computing averages on the server and is not of any interest to the client (see Figure 18-18). Note that if you want to include a table in a sync scope, it must have a primary key, and the primary key must be included in the list of columns in the sync scope.

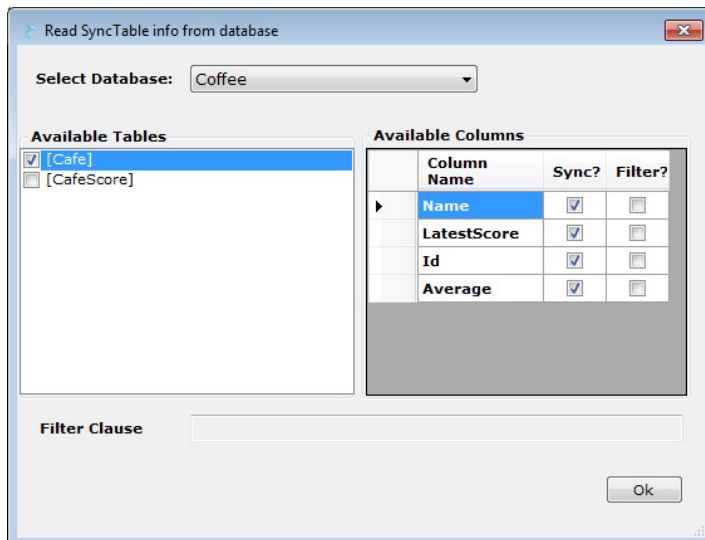


FIGURE 18-18 Configuring the tables and columns in the sync scope.

When you complete the sync scope, this generates the sync configuration file. At any time, you can rerun the tool to edit or overwrite the file, or you can edit the file manually.

Database Provisioning

The next major task is to provision the database, again using the SyncSvcUtilHelper tool. This step takes the sync configuration as input and modifies the database directly, adding tables and triggers. You should make a backup of the database before you perform this step. Figure 18-19 shows the *Coffee* database schema before and after provisioning. After provisioning, there are additional triggers in the *Cafe* table, and there is an additional *Cafe_tracking* table.

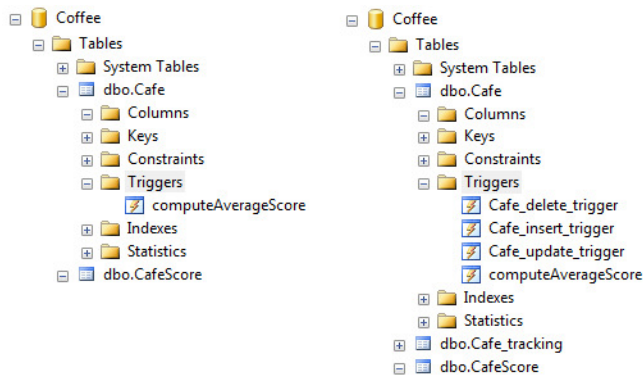


FIGURE 18-19 The Coffee database, before and after provisioning.

Code Generation

The third step is to generate the proxy code for both the server and the client, which is also done with the SyncSvcUtilHelper tool. First, you run the tool to generate code for the server, specifying the Server codegen target, the output directory for the generated files, the language (C# or Visual Basic), the namespace to use in the generated code, and the file name prefix to use, as shown in Figure 18-20. Generating code for the client follows exactly the same steps, except that you choose Isolated Storage as the codegen target (the generated proxies can be used for both Windows Phone and desktop Silverlight client applications).

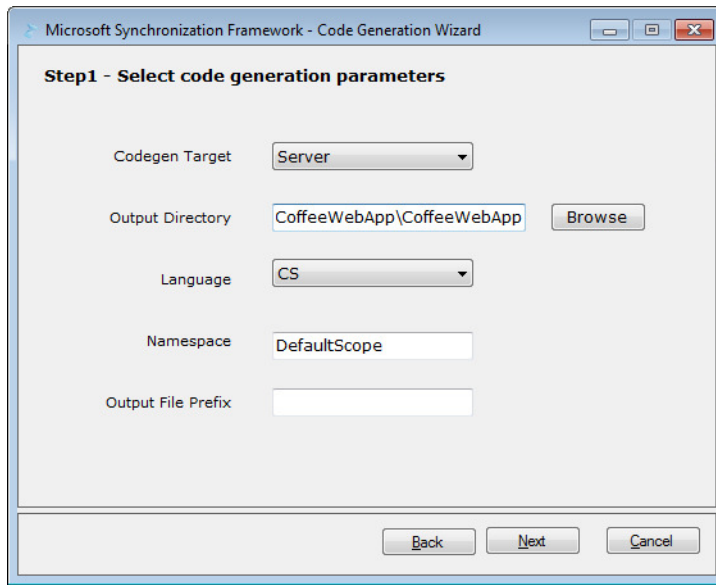


FIGURE 18-20 Generating server-side sync proxy code.

The server application in this example (the `CoffeeWebApp` solution in the sample code) is based on the Visual Studio ASP.NET Empty Web Application project type. First, add references to `Microsoft.Synchronization.Services.dll` and `System.Data.DataSetExtensions.dll`. Add the generated server proxy code files to this project. Using the aforementioned configuration, these will be named `CoffeeEntities.cs` and `CoffeeSyncService.svc` (which will also pull in the generated `CoffeeSyncService.svc.cs` file). You then need to update the service code, but first you need a database connection string. You could put this in the code, but it's conventional to put it in the `web.config`.

```
<connectionStrings>
  <add
    name="CoffeeConnectionString"
    connectionString="Data Source=.\SQLEXPRESS;Initial Catalog=Coffee;Integrated
Security=true;" />
</connectionStrings>
```

Then, in the code, uncomment or replace the placeholder code that the tool generated. You must set the server connection string, the sync scope name, and the sync object schema name. You can optionally also set the serialization format, the conflict resolution policy, the download batch size, the debugging error level, and enable the service diagnostic page.

```
public class CoffeeSyncService : SyncService<CoffeeOfflineEntities>
{
    public static void InitializeService(ISyncServiceConfiguration config) {
        config.ServerConnectionString =
            ConfigurationManager.ConnectionStrings["CoffeeConnectionString"].ToString();
        config.SetEnableScope("DefaultScope");
        config.SetSyncObjectSchema("dbo");
    }
}
```

```

        config.SetDefaultSyncSerializationFormat(SyncSerializationFormat.ODataJson);
        config.SetConflictResolutionPolicy(ConflictResolutionPolicy.ServerWins);
        //config.SetDownloadBatchSize(100);
        config.UseVerboseErrors = true;
        config.EnableDiagnosticPage = true;
    }
}

```

With the sync proxy code in place, you can build the server. In Solution Explorer, right-click the SVC file, and then select View In Browser. This uses a URL with the *\$syncscopes* parameter, and should return you a standard OData XML response. If you execute the URL again, but replace *\$syncscopes* with *\$diag*, this will run the sync diagnostics on the service and return a summary report, as shown in Figure 18-21.

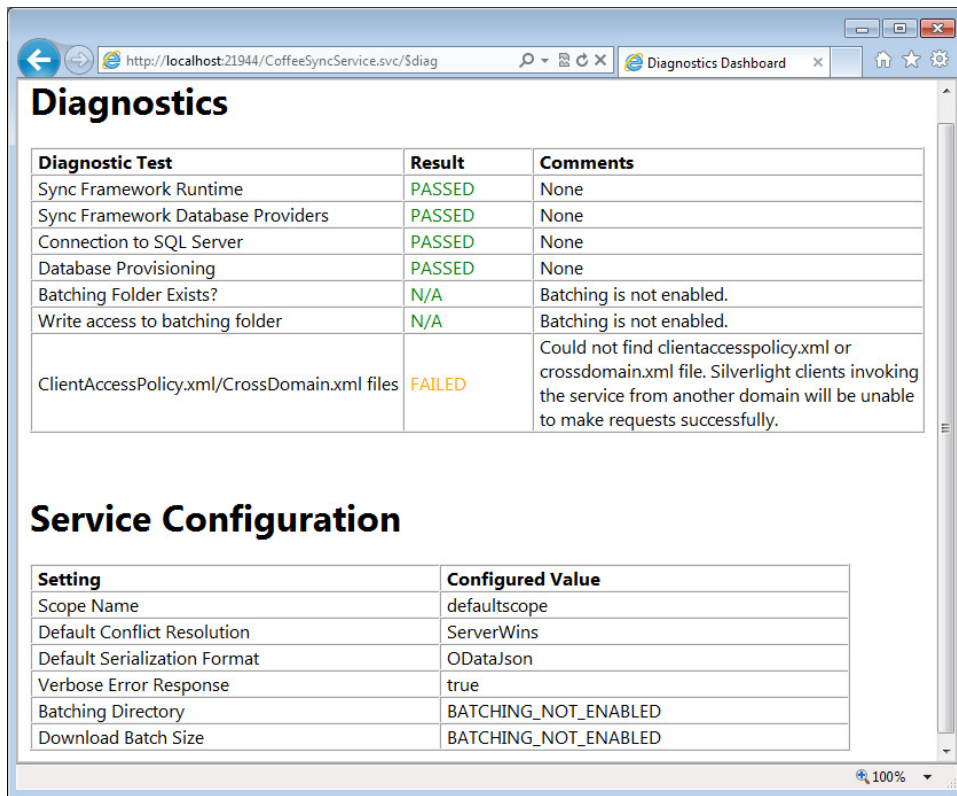


FIGURE 18-21 The sync service diagnostics report.

Assuming the diagnostics all pass (you can ignore the cross-domain warning for Windows Phone projects: these are only relevant for desktop Silverlight), you can now go on to build the client application. This is a standard Windows Phone application, with the UI set up appropriately for the café data. First, add references to `Microsoft.Synchronization.ClientServices.dll`, `mscorlib.extensions.dll` and `System.Runtime.Serialization.Json.dll`. Then, add the generated `CoffeeEntities.cs` and `CoffeeOfflineContext.cs` files. In this example, the page code simply instantiates a *CoffeeOfflineContext* object,

pointing it to the service URL, and hooks up the *LoadCompleted* event on this object. In this example, you're using localhost for the service URL; in a real application, of course, you'd use a real server path. In the event handler, you refresh the local cache, and update the UI with the collection of entities returned from the service. There's also an App Bar button with which the user can trigger a refresh whenever he wants.

```
private DefaultScope.CoffeeOfflineContext context;

public MainPage()
{
    InitializeComponent();

    context = new DefaultScope.CoffeeOfflineContext(
        "Coffee", new Uri("http://localhost:21944/CoffeeSyncService.svc"));
    context.LoadCompleted += context_LoadCompleted;
    context.LoadAsync();
}

private void context_LoadCompleted(object sender, LoadCompletedEventArgs e)
{
    Dispatcher.BeginInvoke(() =>
    {
        context.CacheController.RefreshAsync();
        cafeList.ItemsSource = context.CafeCollection;
    });
}

private void appBarSync_Click(object sender, EventArgs e)
{
    context.SaveChanges();
    context.CacheController.RefreshAsync();
}
```

In some applications, refreshing based off a UI trigger is appropriate. In others, you might want to set up a timer to invoke *RefreshAsync* more frequently or outside user control. In most scenarios, you would also want to apply one or more filters in order to restrict what data is synchronized. You can find more details on the Sync Framework at <http://msdn.microsoft.com/en-us/sync/bb821992>.

Summary

In this chapter, you saw how Windows Phone 7.1 brings extensive support for structured data in the form of local databases and LINQ-to-SQL. This includes full CRUD support, schema version management, table relationship, and phone-specific performance optimizations. By using the Isolated Storage Explorer tool, you can work with your database on the desktop during development, and you can even enlist the support of desktop tools such as SSMS and Visual Studio. Data encryption is enhanced in version 7.1, with full-database encryption, and with scoped DPAPI support for securely storing credentials on the phone. You can continue to take part in the holistic UX by integrating closely with the user's contacts and calendar information, and you can build sophisticated intermittently connected data synchronization systems by using the Sync Framework.

Framework Enhancements

In addition to a range of major new features, Windows Phone 7.1 also significantly strengthens and extends the platform. Many of these improvements have been brought in as part of the move from Microsoft Silverlight 3.0 to Silverlight 4.0, and are therefore heavily user interface (UI)-oriented. These include better navigation and backstack support, enhanced controls, more chrome (system tray and application bar) programmability, and a set of new data-binding-related features.

Navigation Enhancements

Version 7.1 includes two categories of navigation enhancement. First, the frame and page navigation APIs have been updated to provide greater flexibility and more information during navigation. Second, you now have additional APIs with which you can manipulate the backstack directly, thereby affecting the results of navigation.

Frame and Page Navigation

Windows Phone 7.1 includes enhancements to the *Frame*, *Page*, and navigation APIs, including the ability to determine whether a navigation is truly cancelable. Navigations that are initiated by the user by interacting with your application UI are generally cancelable, whereas navigations initiated by the user interacting with hardware buttons or initiated by the system are generally not cancelable. It is common to provide navigation UI within your application, including *Hyperlink* and *Button* controls. However, there are scenarios for which, even though the user has gestured that he wants to navigate, you might want to intercept the request and prompt for confirmation. For example, if the user has edited a page or entered data, but he hasn't yet confirmed the new input or changes, you would prompt him to save first when he tries to navigate away.

Since version 7, you have been able to override the *OnNavigatingFrom* method. This provides a *NavigatingCancelEventArgs*, which exposes a *Cancel* property. However, there was no way in version 7 to determine whether the event was truly cancelable or whether it was actually a non-cancelable system navigation. In your code, therefore, you might set the *Cancel* event and perform other logic based on this, when in fact the event was not cancelled at all. In version 7.1, the event is enhanced with an *IsCancelable* property to definitively establish whether an attempt to cancel will actually succeed, and if you can't cancel the navigation, you would take other steps to handle the scenario (perhaps saving the user's input to a temporary file or other mitigating actions, depending on the context). You can see this at work in the *NewNavigation* solution in the sample code.

```
protected override void OnNavigatingFrom(NavigatingCancelEventArgs e)
{
    Debug.WriteLine("OnNavigatingFrom");

    if (e.IsCancelable)
    {
        MessageBoxResult result = MessageBox.Show(
            "Navigate away?", "Confirm", MessageBoxButton.OKCancel);
        if (result == MessageBoxResult.Cancel)
        {
            e.Cancel = true;
        }
    }
    else
    {
        Debug.WriteLine("Navigation NOT cancelable");
    }
}
}
```

Windows Phone 7.1 also exposes a *NavigationMode* property on the *NavigationEventArgs* object that is passed into the *OnNavigatedTo* and *OnNavigatedFrom* method overrides. The value of *NavigationMode* will be either *New* or *Back*, which identifies the direction of navigation. The *Back* value is self-explanatory; if the value is *New*, this indicates that this is a forward navigation. The *NavigationMode* type includes the values *Forward* and *Refresh* also, but these are not used in Windows Phone. Typically, you would perform some conditional operation based on this value. The following code snippet merely prints a string to the debug window. The application has two pages: *MainPage* and *Page2*, and the user can navigate back and forth between them.

```
public partial class MainPage : PhoneApplicationPage
{
    ... irrelevant code omitted for brevity.
    protected override void OnNavigatedTo(NavigationEventArgs e)
    {
        Debug.WriteLine("MainPage.OnNavigatedTo: {0}", e.NavigationMode);
    }

    protected override void OnNavigatedFrom(NavigationEventArgs e)
    {
        Debug.WriteLine("MainPage.OnNavigatedFrom: {0}", e.NavigationMode);
    }
}

public partial class Page2 : PhoneApplicationPage
{
    protected override void OnNavigatedTo(NavigationEventArgs e)
    {
        Debug.WriteLine("Page2.OnNavigatedTo: {0}", e.NavigationMode);
    }

    protected override void OnNavigatedFrom(NavigationEventArgs e)
    {
        Debug.WriteLine("Page2.OnNavigatedFrom: {0}", e.NavigationMode);
    }
}
```

When the application starts and the *MainPage* is loaded, the following output is produced:

```
MainPage.OnNavigatedTo: New
```

As the user navigates forward from *MainPage* to *Page2*, you would expect to see the following debug output—the navigation is forward as far as both pages are concerned:

```
MainPage.OnNavigatedFrom: New  
Page2.OnNavigatedTo: New
```

Then, as the user navigates back from *Page2* to *MainPage*, you would expect to see the following debug output—again, for both pages, the navigation is backward:

```
Page2.OnNavigatedFrom: Back  
MainPage.OnNavigatedTo: Back
```

If the user is on *Page2* and then navigates forward out of the application by tapping the Start button, and then navigates back into the application again, you would see the following output (the *New* [forward] navigation out of the application, followed by the *Back* navigation, back into the application):

```
Page2.OnNavigatedFrom: New  
Page2.OnNavigatedTo: Back
```

Finally, consider another new property of both the *NavigationEventArgs* and the *NavigatingCancelEventArgs*: the *IsNavigationInitiator* property. This is a Boolean value that notifies you whether the navigation is from an external source; that is, the user navigated from outside the application into the application. In the following, you're going to modify the debug output statements to include this property value:

```
Debug.WriteLine("Page2.OnNavigatedTo: {0}, {1}", e.NavigationMode, e.IsNavigationInitiator);
```

Now, if the user starts the application (which loads *MainPage*), navigates internally to *Page2*, taps Start to navigate forward out of the application, taps the Back button to return into the application, and then finally, back from *Page2* to *MainPage*, you will see the output that follows. When the navigation is to or from an external source (including the initial launch of the application from the Start page), the value of *IsNavigationInitiator* is *false*. For internal navigation, the value is *true*.

```
MainPage.OnNavigatedTo: New, False  
MainPage.OnNavigatedFrom: New, True  
Page2.OnNavigatedTo: New, True  
Page2.OnNavigatedFrom: New, False  
Page2.OnNavigatedTo: Back, False  
Page2.OnNavigatedFrom: Back, True  
MainPage.OnNavigatedTo: Back, True
```

Backstack Management

In Chapter 7, “Navigation State and Storage,” you saw how the user’s navigation history is maintained within an application in a history list called the backstack. The backstack is managed as a last-in, first-out (LIFO) stack. That is, as the user navigates forward through the pages of an application, each page from which she departs is added to the stack. As she navigates back, the frame releases its reference to the current page (which is likely then to become available for garbage collection), and the previous page in the navigation history is popped off the stack to become the new current page.

Windows Phone 7.1 includes the following additional API support for working with the backstack:

- The *NavigationService* class now exposes a *BackStack* property, which is an *IEnumerable* collection of *JournalEntry* objects. Each page in the backstack is represented by a *JournalEntry* object. The *JournalEntry* class exposes just one significant property: the *Source* property, which is the navigation URI for that page.
- The *NavigationService* class now exposes a *RemoveBackEntry* method, which is used to remove the most recent entry from the backstack. You can call this multiple times if you want to remove multiple entries.
- The *NavigationService* class now exposes an *OnRemovedFromJournal* virtual method, which you can override. This is invoked when the page is removed from the backstack, either because the user is navigating backward, away from the page, or because the application is programmatically clearing the backstack. When the user navigates forward, the previous page remains in the backstack by default (of course).

Here’s the sequencing of the new APIs in relation to the *OnNavigatedFrom* override. When the user navigates backward away from the page, the methods/event handlers will be invoked in the following order: first the *OnNavigatedFrom* override, then the *JournalEntryRemoved* event handler, and then the *OnRemovedFromJournal* override.

Previously, your application was in control of forward navigation, in the sense that you determined where the user could navigate to within the application. You could perform forward navigation to any URL. On the other hand, you had very limited control over backward navigation. You could provide your own navigation UI (buttons or links), and implement these to invoke the *NavigationService.GoBack* method, but that always goes back to the immediately preceding page in the backstack. Similarly, when the user taps the hardware Back button, it always navigates back to the previous page on the backstack.

With the new APIs, you can modify the user’s navigation experience, such that going back doesn’t necessarily always take her back to the previous page. To be clear: you can still use only the *NavigationService* to navigate forward to a specific URL, or back one page in the backstack. The difference is that you can now remove entries from the backstack—up to and including all entries—such that navigating back no longer necessarily takes the user back to the immediately preceding page. These two new APIs effectively make the Non-Linear Navigation Service (discussed in Chapter 7) redundant.



Note As with some other features introduced with version 7.1, the ability to manipulate the backstack is a powerful one that affords you more flexibility than you previously enjoyed, but it also gives you a way to break conformance with the Metro paradigm. You should use this only after very careful consideration, and only if you're sure that you can't avoid it.

Figure 19-1 shows the *ClearBack_Thumbs* solution in the sample code, which illustrates how you can manipulate the backstack.

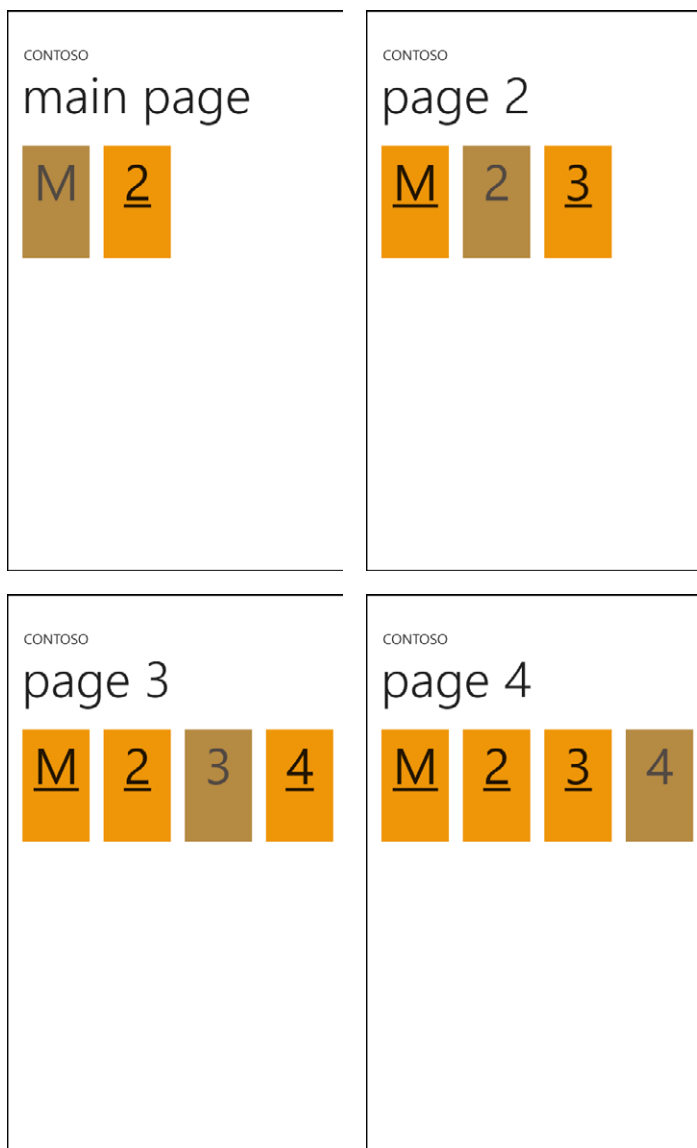


FIGURE 19-1 Maintaining linear navigation while manipulating the backstack.

The application contains four pages, and each page shows a number of page thumbnails. When the user is on the first page (*MainPage*), he can only navigate either forward to *Page2* or backward out of the application. So, on *MainPage*, you show only a thumbnail for *MainPage* (grayed-out, because it's the current page) and *Page2* (with a hyperlink, which he can tap to go to that page). When he's on *Page2*, he can navigate forward to *Page3* or backward to *MainPage*; therefore, you show the corresponding thumbnails and hyperlinks for the two navigation targets, and you gray-out the current thumbnail. Keep in mind that you're using hyperlinks here for the sake of convenience, but you should generally leave backward navigation to the hardware Back button. There is normally no good reason to add your own in-application user experience (UX) for backward navigation, except in the rare case when you're manipulating the backstack and navigating back to somewhere other than the default previous page.

So, the model you're adopting here is that the user can navigate forward only through the pre-defined sequence (that is, visiting each page, in order). When navigating forward, you can only ever navigate to the very next page: you cannot skip pages going forward. So, when the user is on the *MainPage*, you show *Page2* as the only page to which he can navigate from here.

However, when he's on *Page3* or *Page4*, you allow him to skip pages when navigating backward. Take a look at how this works in code by considering *Page4*. On all pages, when the user wants to go to *MainPage*, you treat this as a logical "home" navigation and invoke the *NavigationService* to navigate explicitly to *MainPage.xaml*.

```
public partial class Page4 : PhoneApplicationPage
{
    private void GoHome_Click(object sender, RoutedEventArgs e)
    {
        NavigationService.Navigate(new Uri("/MainPage.xaml", UriKind.Relative));
    }
}
```

For this to work correctly, you must override the *OnNavigatedTo* method on the *MainPage*, and implement it to remove all entries from the backstack. In this way, you ensure that this is always the beginning of the page stack for this application, and that regardless of how he arrived here, if the user taps the Back button from here, this will always navigate back out of the application. Doing this minimizes the confusion to the user: even though you are manipulating the backstack, you maintain a high degree of consistency with the hardware Back button behavior.

```
public partial class MainPage : PhoneApplicationPage
{
    protected override void OnNavigatedTo(NavigationEventArgs e)
    {
        while (NavigationService.CanGoBack)
        {
            NavigationService.RemoveBackEntry();
        }
    }
}
```

Looking again at *Page4*, if the user is on *Page4* and wants to go back to *Page3*, you simply invoke the *NavigationService.GoBack* method, which again implicitly maintains consistency with the Back

button. However, if the user is on *Page4* but wants to navigate back to *Page2* (skipping *Page3*) you enable this by removing *Page3* from the *BackStack* before navigating back. This also maintains consistency with the hardware Back button. This means that if the user's forward navigation sequence was *M | 2 | 3 | 4*, and he then skipped back *4 | 2*, if he subsequently taps the Back button from *Page2*, this will take him back to *MainPage*, not *Page3*.

```
public partial class Page4 : PhoneApplicationPage
{
    private void GoBack_Click(object sender, RoutedEventArgs e)
    {
        NavigationService.GoBack();
    }

    private void GotoPage2_Click(object sender, RoutedEventArgs e)
    {
        NavigationService.RemoveBackEntry();
        NavigationService.GoBack();
    }
}
```

By the same token, when on *Page3*, the user could have arrived at *Page3* either from *Page2* (navigating forward), or from *Page4* (navigating backward). Because you're maintaining consistency with the hardware Back button, you never navigate forward to a previous page. So, regardless of whether he arrived at *Page2* by navigating forward or backward, going back from *Page3* always goes back in the logical sequence; that is, to *Page2*. This simple approach ensures consistency regardless of how many pages there are in the application. Note also that the *PhoneApplicationFrame* class also exposes the exact same *BackStack* and *RemoveBackEntry* APIs, although typically you would use these from the *NavigationService*.



Note If you do find yourself manipulating the backstack in this fashion, you could also consider encapsulating this behavior in an extension method to the frame; perhaps by using something like *ReturnToPage(page)*. This method would enumerate the backstack and remove the appropriate number of pages before navigating back (or navigate forward and clear the stack if the page doesn't exist).

UI Enhancements

Windows Phone 7.1 boasts a long list of UI enhancements, mostly small incremental improvements and additions to the version 7 platform, rounding out existing features and filling some recognized gaps. For example, you can now use application icons with transparency, which allows theme colors to show through the transparent areas in the application list. Some anomalies in the behavior of *TextBox* controls when the onscreen keyboard is displayed have been fixed. An input, or "touch," thread has been added to improve performance for things such as *ListBox* scrolling. Some of these, such as the touch thread, are quite significant improvements in the platform itself, although there is no API exposed, and the list of minor fixes is quite long. In addition, some more desktop Silverlight controls

have been ported over to Windows Phone, including the *RichTextBox*, *Viewbox*, and *VideoBrush*, and the *WebBrowser* control has had a major rewrite, as is discussed in the following sections.

Enhanced Controls

Figure 19-2 shows the *Test71Controls* solution in the sample code. This application offers a *Pivot* page with four pivot items, each one demonstrating one of the new or enhanced controls.

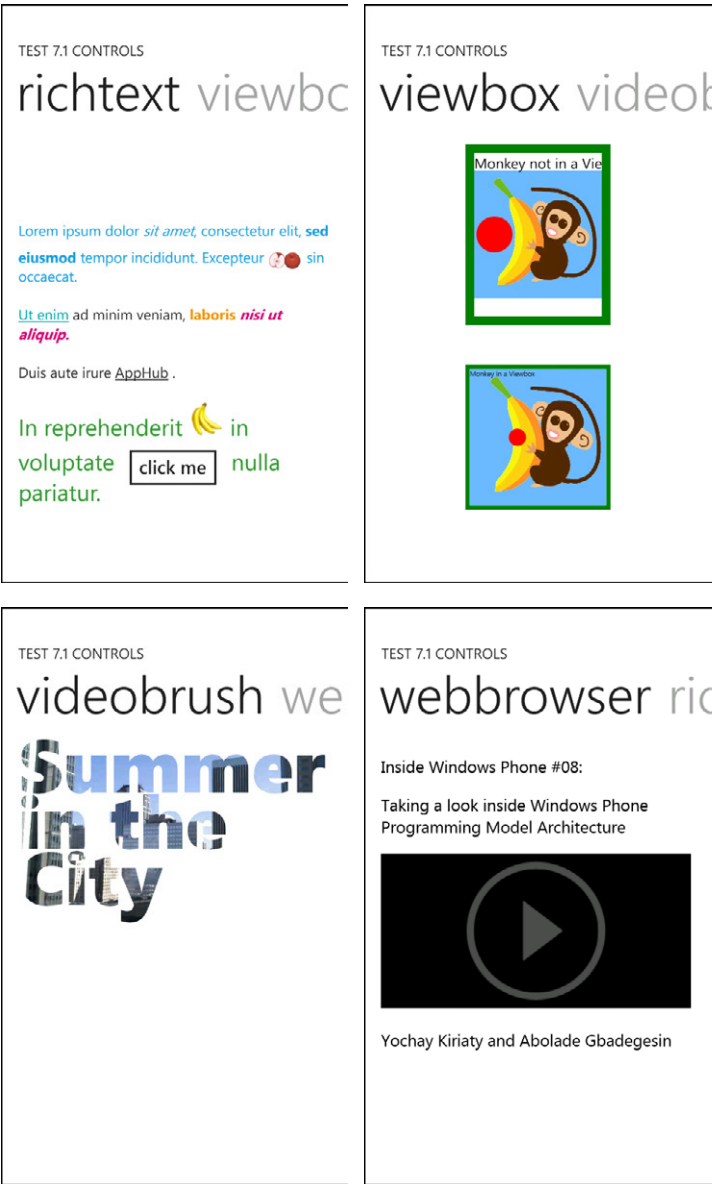


FIGURE 19-2 *RichTextBox*, *Viewbox*, *VideoBrush*, and *WebBrowser* controls

The first pivot item shows the *RichTextBox*, which provides more flexible character and paragraph formatting than either the *TextBox* or *TextBlock* controls. The second shows the *Viewbox*, which gives you another option for placing one or more items in a bounding container that constrains the sizing and scaling behavior. The third shows the *VideoBrush*, with which you can use a video file or the feed from the Silverlight webcam or phone PhotoCamera objects as the brush for any UI element. The last shows the enhanced *WebBrowser* control, which is now based on Internet Explorer 9 and supports HTML5.

The *RichTextBox* can contain zero or more *Paragraph* items, each of which can contain zero or more child items, typically *Text* items. You can control the formatting of each *Text* item in much the same way as a regular *TextBlock*, including font, size, foreground and background colors, font weight, and style. In this example, the first paragraph is set to the blue accent color, and then individual items of text within the paragraph are set to italic or bold. The second paragraph uses alternative syntax to set contained items to underline, green, bold, and mango color.

```
<RichTextBox x:Name="demoText" VerticalScrollBarVisibility="Auto" >
  <Paragraph Foreground="#FF1BA1E2">
    Lorem ipsum dolor <Italic>sit amet</Italic>, consectetur
    elit, <Bold>sed eiusmod</Bold> tempor incididunt.
  </Paragraph>
  <Paragraph/>
  <Paragraph>
    <Underline Foreground="#FF00ABA9">Ut enim</Underline> ad minim veniam,
    <Bold Foreground="#FFF09609">laboris</Bold>
    <Run
      Text="nisi ut aliquip."
      FontStyle="Italic" Foreground="#FFD80073" FontWeight="Bold"/>
    </Paragraph>
</RichTextBox>
```

As well as text, a *Paragraph* element can contain a *Hyperlink* directly, so long as you specify an external URL (that is, not a relative page URL) and the *_blank* target name. A *Paragraph* can also contain an *InlineUIContainer*, and this, in turn, can contain any *UIElement*. The following example includes two *InlineUIContainer* objects: one with an *Image* control, and the other with a *Button*:

```
<Paragraph>
  Duis aute irure
  <Hyperlink NavigateUri="http://create.msdn.com" TargetName="_blank">AppHub</Hyperlink> .
</Paragraph>
<Paragraph/>
<Paragraph>
  FontSize="{StaticResource PhoneFontSizeLarge}" Foreground="#FF339933">
  In reprehenderit
  <InlineUIContainer>
    <Image Source="bananas.png" Height="48" Width="48" />
  </InlineUIContainer>
  in voluptate
  <InlineUIContainer>
    <Button x:Name="SayHello" Content="click me" Click="SayHello_Click" />
  </InlineUIContainer>
  nulla pariat.
</Paragraph>
```

Also, of course, you can construct the contents of a *RichTextBox* dynamically in code as well as (or instead of) statically in XAML. In the following *OnNavigatedTo* override, you construct two independent *Run* objects and an *InlineUIContainer* with an *Image*. These are then added to a *Paragraph* object; in this case, the first paragraph in the existing *Blocks* collection, although you could equally well add a new paragraph to the collection.

```
protected override void OnNavigatedTo(NavigationEventArgs e)
{
    Run run1 = new Run();
    run1.Text = " Excepteur ";

    Image image = new Image();
    image.Source = new BitmapImage(new Uri("apples.png", UriKind.Relative));
    image.Height = image.Width = 48;
    image.Margin = new Thickness(0, 0, 0, -16);

    InlineUIContainer container = new InlineUIContainer();
    container.Child = image;

    Run run2 = new Run();
    run2.Text = " sin occaecat.";

    //Paragraph para = new Paragraph();
    Paragraph para1 = (Paragraph)demoText.Blocks[0];
    para1.Inlines.Add(run1);
    para1.Inlines.Add(container);
    para1.Inlines.Add(run2);
    //demoText.Blocks.Add(para);
}
```



Note Unlike desktop Silverlight, the *RichTextBox* in Windows Phone 7.1 is read-only. Also note that the *RichTextBox* is not available in the Microsoft Visual Studio toolbox by default. However, you can add it, if you want. To do this, right-click the toolbox, select *Choose Items*, and then in the list that appears, select the check box adjacent to *RichTextBox*.

To use the *ViewBox* control, you must add a reference to the Silverlight 4.0 version of System.Windows.Controls.dll. A *Viewbox* contains one child element and stretches it or scales it to fit the size of the *Viewbox*. You can control the stretch or scale levels by using the *Stretch*, *StretchDirection*, *HorizontalAlignment*, and *VerticalAlignment* properties. The *Viewbox* pivot in the example that follows shows two versions of the same *Grid*. Both contain an *Image*, a *TextBlock* and an *Ellipse*, with all the same attributes. However, the first *Grid* is simply placed in a 200x250-pixel *Border*, while the second is placed in a 200x250-pixel *Viewbox*. In the *Viewbox*, everything is scaled in proportion, with each item maintaining its aspect ratio. Contrast this with the *Border* version, in which the *Grid* expands to fill the *Border*, and the *TextBlock* is not scaled to fit.

```

<StackPanel>
    <Border Width="200" Height="250" BorderBrush="Green" BorderThickness="12">
        <Grid>
            <Image Source="monkey.png"/>
            <TextBlock Text="Monkey not in a Viewbox" Foreground="Black"/>
            <Ellipse Fill="Red" Width="50" Height="50" Margin="120,0,0,0"/>
        </Grid>
    </Border>
    <Grid Height="30"/>
    <Viewbox x:Name="monkeyBox" Width="200" Height="250">
        <Border BorderBrush="Green" BorderThickness="12">
            <Grid>
                <Image Source="monkey.png"/>
                <TextBlock Text="Monkey in a Viewbox" Foreground="Black"/>
                <Ellipse Fill="Red" Width="50" Height="50" Margin="120,0,0,0"/>
            </Grid>
        </Border>
    </Viewbox>
</StackPanel>

```

The *VideoBrush* pivot in this example defines a simple *TextBlock* whose *Foreground* is set to a *VideoBrush*. The *VideoBrush*, in turn, takes its source from a *MediaElement* object. The *MediaElement* is set to have zero opacity and to be non-hit-testable, which effectively makes it invisible. You also mute the sound because you're only interested in the visual aspects of the video here.

```

<Grid>
    <MediaElement
        x:Name="videoElement" Source="SeattleSummer.wmv"
        IsMuted="True" Opacity="0.0" IsHitTestVisible="False"/>
    <TextBlock
        Text="Summer in the City"
        LineStackingStrategy="BlockLineHeight" LineHeight="78"
        FontFamily="Segoe WP Black" FontSize="103" TextWrapping="Wrap"
        Margin="{StaticResource PhoneHorizontalMargin}">
        <TextBlock.Foreground>
            <VideoBrush SourceName="videoElement" Stretch="UniformToFill" />
        </TextBlock.Foreground>
    </TextBlock>
</Grid>

```



Note If you want the video to loop continuously, you could hook up the *MediaEnded* event to reset the position to the start, and then begin playing again, as shown here.

```

private void videoElement_MediaEnded(object sender, RoutedEventArgs e)
{
    videoElement.Position = new TimeSpan(0);
    videoElement.Play();
}

```

Chapter 16, “Enhanced Phone Services,” demonstrates further examples of the *VideoBrush*, including using it as a camera viewfinder and for an augmented reality application.

The final example in this application shows the enhanced *WebBrowser* control. This supports HTML5, and one of the interesting new tags in HTML5 is the `<video>` tag. To demonstrate the HTML5 support, you will navigate to a local page with a remote video link, and provide a mechanism for the user to play the video via on-screen controls. HTML5 video is the only standard way to embed video on web pages that is supported by multiple mobile devices. It is therefore critical in building cross-browser applications. Also note that Internet Explorer 9 (and therefore the *WebBrowser* control) on version 7.1 will play HTML5 video in H.264, which is the most commonly used video format for mobile web applications. In addition, HTML5 video on the phone is hardware accelerated, just as it is with the desktop browser. In the example, the XAML declaration for the *WebBrowser* is simple, as illustrated here:

```
<phone:WebBrowser x:Name="browser"/>
```

The work of setting up the browser is done in code by hooking up the *Loaded* event on the *WebBrowser* control. In this handler, you fetch an arbitrary HTML page from the application folder, and then navigate the browser to that page.

```
private void browser_Loaded(object sender, RoutedEventArgs e)
{
    StreamResourceInfo sri =
        App.GetResourceStream(new Uri("VideoPage.html", UriKind.Relative));
    using (StreamReader reader = new StreamReader(sri.Stream))
    {
        String html = reader.ReadToEnd();
        browser.NavigateToString(html);
    }
}
```

The HTML page itself is where the video is defined. This is a simple HTML document, added to the project with its build action set to *Content*.

```
<!doctype html>
<html>
  <head>
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
  </head>
  <body>
    <div id="main">
      Inside Windows Phone #08:
      <p />
      Taking a look inside Windows Phone Programming Model Architecture
      <p />
      <video
        src="http://ecn.channel9.msdn.com/o9/ch9/f076/8a77bfbcb-5e80-4bcb-aa44-9e1a0116f076/
          IWPS08TakingALookInToArch_ch9.wmv"
        controls/>
      <p />
      Yochay Kiriathy and Abolade Gbadegesin
    </div>
  </body>
</html>
```

The browser on the phone has access to an application's isolated storage. So, if you prefer, you can copy your local HTML pages (and any assets they require) from the application folder into isolated storage, as shown in the example that follows. This is a more convenient approach if you have multiple HTML pages, especially if they also refer to their own local assets (JavaScript files, images, and so on). Furthermore, you could perform this operation the first time the application runs (or the first time the user navigates to this page or pivot item), which means that subsequent uses of the browser would simply use the files in isolated storage directly, simplifying the code.

```
private void browser_Loaded(object sender, RoutedEventArgs e)
{
    using (IsolatedStorageFile store =
        IsolatedStorageFile.GetUserStoreForApplication())
    {
        if (!store.FileExists("VideoPage.html"))
        {
            using (IsolatedStorageFileStream stream =
                new IsolatedStorageFileStream("VideoPage.html", FileMode.Create, store))
            {
                Stream htmlPage = Application.GetResourceStream
                    (new Uri("VideoPage.html", UriKind.Relative)).Stream;
                using (StreamReader reader = new StreamReader(htmlPage))
                {
                    string data = reader.ReadToEnd();
                    using (StreamWriter writer = new StreamWriter(stream))
                    {
                        writer.Write(data);
                    }
                }
            }
        }
        browser.Source = new Uri("VideoPage.html", UriKind.Relative);
    }
}
```

Chapter 11, “Web and Cloud,” discusses the core functionality of the *WebBrowser* control in more detail. The principles of interacting with the browser in your application remain the same across versions 7 and 7.1, although the support for HTML5 in version 7.1 makes the script interoperability feature more interesting. This is particularly true when you want to build a web application that works across multiple browsers, on both desktop and mobile devices—including multiple mobile device platforms.

The *AppBar* and *SystemTray* Classes, and the *ProgressIndicator* Property

The version 7.1 *AppBar* exposes one additional property: *Mode*. This can be set to either *Default* or *Minimized*. The minimized mode is intended for use in scenarios for which you want to optimize your use of screen real estate—and most particularly on panorama pages. As discussed in Chapter 3, “Controls,” it's generally better to remove as much clutter as possible from panorama pages, including chrome. If you must use an App Bar on a panorama page, you should consider the

minimized mode, and also perhaps set *Opacity* to zero. Using this approach, the App Bar ellipsis will always be visible, but the rest of the App Bar only takes up space if the user taps the ellipsis to open it.

In version 7, the *SystemTray* class exposed only one property: *IsVisible*. This has been enhanced in version 7.1 to expose four new properties: *BackgroundColor*, *ForegroundColor*, *Opacity*, and *ProgressIndicator*. Figure 19-3 shows the *NewSystemTray* solution in the sample code, which utilizes these properties via four App Bar buttons.

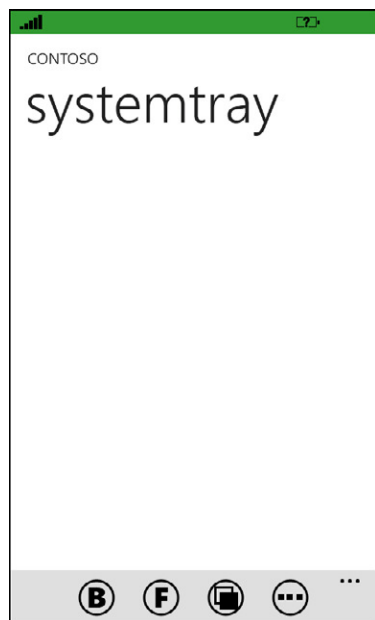


FIGURE 19-3 An application that uses the new *SystemTray* properties.

The App Bar buttons in this example perform the following operations:

- Toggle the *BackgroundColor* of the system tray between the current accent color and the default background color.
- Toggle the *ForegroundColor* of the system tray between the current accent color and the default foreground color.
- Cycle the *Opacity* between 1.0, 0.5, and 0.0.
- Attach a *ProgressIndicator*, and then toggle it between determinate and indeterminate mode.
- There's a slight quirk in the way the *SystemTray* color properties work. The *BackgroundColor* defaults to transparent—that is 00000000—and the *ForegroundColor* defaults to white (FFFFFFFF), regardless of the current theme (dark or light). In other words, the exposed *BackgroundColor* and *ForegroundColor* properties of the *SystemTray* object are not used unless they are explicitly set. Also note that the transparent value is not the same as *Colors.Transparent* (00FFFFFF or “transparent white”). To allow for this, you first determine which theme is in use, and then cache suitable black or white colors for later use.

```

private Color ForegroundColor;
private Color BackgroundColor;
private Color AccentColor;
private Color TransparentBlack = new Color() { A = 0, R = 0, G = 0, B = 0 };

public MainPage()
{
    InitializeComponent();

    Visibility v = (Visibility)Resources["PhoneDarkThemeVisibility"];
    if (v == Visibility.Visible)
    {
        BackgroundColor = TransparentBlack;
        ForegroundColor = Colors.White;
    }
    else
    {
        BackgroundColor = Colors.White;
        ForegroundColor = TransparentBlack;
    }

    AccentColor = (Color)Resources["PhoneAccentColor"];
}

```

To toggle the system tray background and foreground colors between two colors, you have to allow for the initial default property (that is, a third color—either true black or white).

```

private void setBackColor_Click(object sender, EventArgs e)
{
    if (SystemTray.BackgroundColor == TransparentBlack ||
        SystemTray.BackgroundColor == BackgroundColor)
    {
        SystemTray.BackgroundColor = AccentColor;
    }
    else
    {
        SystemTray.BackgroundColor = BackgroundColor;
    }
}

```

Although *Opacity* is typed as a double, as it relates to the system tray, it affects page layout more like a Boolean in the sense that you get a 32-pixel high space at the top of the page if the system tray *Opacity* is 1.0. If the system tray *Opacity* is <1.0, you don't get the 32-pixel space (and the acceptable range is 0.0 to 1.0).

```

private void setOpacity_Click(object sender, EventArgs e)
{
    if (SystemTray.Opacity == 1.0)
    {
        SystemTray.Opacity = 0.0;
    }
    else if (SystemTray.Opacity == 0.0)
    {
        SystemTray.Opacity = 0.5;
    }
}

```

```

    else
    {
        SystemTray.Opacity = 1.0;
    }
}

```

If you enable the *ProgressIndicator* on the *SystemTray* object, this occupies space right at the extreme outer edge of the system tray (at the top of the screen, when the device is in portrait orientation). An indeterminate progress bar displays a repeating pattern of dots to indicate that progress is ongoing, but that the current percentage completion is undefined. A determinate progress bar, on the other hand, shows a colored indicator bar within the overall control, whose length is proportional to the total length of the control, and represents the percentage of progress completed so far. You set the length of this colored indicator by setting the *Value* property. If you set *IsIndeterminate* to *true*, any *Value* property you assign will be ignored. In this example, you toggle between indeterminate and determinate mode, and set the *Value* to an arbitrary number between 0.0 and 1.0. In the code snippet that follows, the commented-out line that invokes the *SetProgressIndicator* method would have exactly the same effect as the subsequent line that sets the *ProgressIndicator* property.

```

private ProgressIndicator progress;

private void setProgress_Click(object sender, EventArgs e)
{
    if (progress == null)
    {
        progress = new ProgressIndicator();
        progress.IsVisible = true;
        progress.IsIndeterminate = true;
        progress.Text = "working...";
        //SystemTray.SetProgressIndicator(this, progress);
        SystemTray.ProgressIndicator = progress;
    }
    else
    {
        progress.IsIndeterminate = !progress.IsIndeterminate;
        progress.Value = 0.5;
    }
}

```



Note You can optionally set some text to display just below the progress bar. If you do, and the user then taps the system tray to drop down the regular system icons, your text will be hidden (regardless of the opacity setting). Also note that the progress bar on the system tray uses the current accent color; thus, if you set the system tray *BackgroundColor* to the accent color, then any progress bar would be invisible.

The Clipboard API

Windows Phone 7.1 introduces programmatic support for the clipboard, albeit in a constrained manner. You can set text into the system-wide clipboard, but you cannot extract text from it programmatically. This constraint is for security and privacy reasons, and it ensures that the user is always in control of where the clipboard contents might be sent. Figure 19-4 shows the *TestClipboard* application in the sample code.

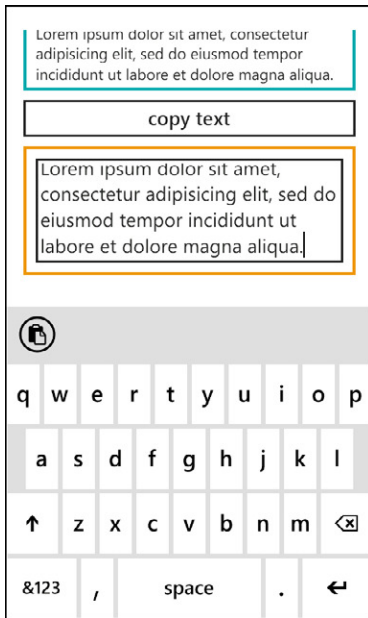


FIGURE 19-4 A simple clipboard application.

This example offers a *RichTextBox* at the top with a *Button* below it, and a regular *TextBox* at the bottom. The *RichTextBox* is populated with some dummy text. The reason for this choice of controls is that you want to get your text from a control that supports selection of its contents, which the *Rich TextBox* does. You also need to make an editable text control available into which the user can paste; hence, the *TextBox*. When the user taps the button, you arbitrarily select some or all of the text in the *RichTextBox*, and then set this text into the clipboard by using the static *Clipboard.SetText* method.

```
private void copyText_Click(object sender, RoutedEventArgs e)
{
    textSource.SelectAll();
    Clipboard.SetText(textSource.Selection.Text);
}
```

After this, if and when the user chooses to tap the regular *TextBox*, the standard phone UI will present a paste icon, indicating that there is some text in the clipboard. If the user taps this icon, the clipboard contents are pasted into the *TextBox*. This last operation is outside of your control, and is handled entirely by the phone platform. Note that the clipboard is cleared currently whenever the phone lock engages.

32 Bits per Pixel

Windows Phone 7 devices supported image rendering of 16 bits per pixel (bpp). In version 7.1, this was increased to 32 bpp (although the actual hardware screen might still be 16 bpp or some other value less than 32). Even in version 7.1 projects, the default is still 16 bpp, but you can specify that you want to use 32 bpp for your application by adding an attribute to your WMAPPManifest file. The *16bpp* and *32bpp* solutions in the sample code and Figure 19-5 highlight the difference.

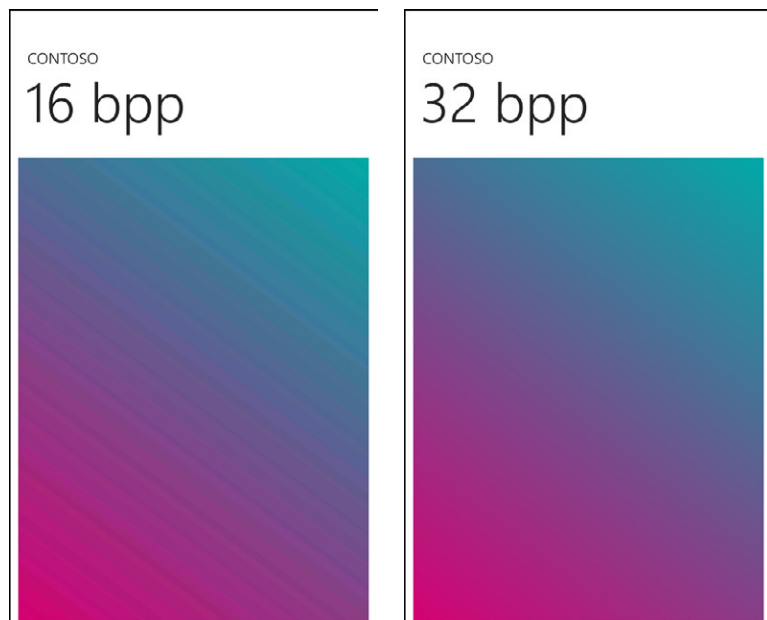


FIGURE 19-5 A linear gradient rendered at 16 bpp (on the left) and 32 bpp.

Rendering at 16 bpp results in obvious banding in image gradients. This is significantly reduced or eliminated at 32 bpp. Similar behavior is evident with photos, too. This application uses a simple *LinearGradientBrush*, running from magenta at one end to teal at the other. To select 32 bpp, you add the *BitsPerPixel* attribute to the *App* element in your manifest, and then set it to 32. This is an application-wide setting; you cannot set different values on a per-page basis.

```
<App xmlns="" ProductID="{1000b48b-c81f-4dff-bfa2-fc822521dcea}" Title="Foo"
  RuntimeType="Silverlight" Version="1.0.0.0" Genre="apps.normal" Author="Foo author"
  Description="Sample description" Publisher="Foo"
  BitsPerPixel="32">
```

Be aware that while this can dramatically improve the presentation of your images, there is a small price to pay in terms of performance. You should test your application to see if the slightly slower rendering is actually noticeable to the user, because the improvement in presentation almost always will be.

Background Image Decoding

In Windows Phone 7, image decoding happens on the UI thread. As discussed in Chapter 14, “Go to Market,” you should minimize the work that you do on the UI thread wherever possible to avoid impacting the UX. Image decoding in version 7 could sometimes result in UI stuttering or a noticeable lag in responsiveness. Windows Phone 7.1 supports background image decoding. To use this feature, you set the *CreateOptions* attribute on each image for which you want it to apply. You can do this in XAML, as shown in the following:

```
<Image>
  <Image.Source>
    <BitmapImage
      UriSource="SunsetDowntown.jpg" CreateOptions="BackgroundCreation"/>
  </Image.Source>
</Image>
```

Alternatively, you can also do this in code:

```
BitmapImage bmp = new BitmapImage();
bmp.CreateOptions = BitmapCreateOptions.BackgroundCreation;
bmp.UriSource = new Uri("SunsetDowntown.jpg", UriKind.Relative);
theImage.Source = bmp;
```

Touch Thread

The scrolling behavior of the *ListBox* and *ScrollViewer* in Windows Phone has been improved in version 7.1. To achieve this, the platform functionality that listens to the touch gestures has been moved to a separate thread. Because this is a platform change (rather than an API change), it means that even version 7 applications automatically benefit from this improvement on phones that are running version 7.1 OS.

One side-effect of this change is that *ScrollViewer* properties are not updated immediately; instead, they are deferred until the user completes the touch gesture. Also, the *ManipulationDelta* events are no longer raised on the UI thread when the user is dragging inside a *ScrollViewer*. These changes improve performance, but if you have an application that relies on the specifics of the previous behavior, you can choose to opt out of the improvements by setting the *ScrollViewer.ManipulationMode* property on the *ListBox* in question to *Control*, as indicated in the following snippet (the default value is “System”):

```
<ListBox
  x:Name="MainListBox" Margin="0,0,-12,0" ItemsSource="{Binding Items}"
  SelectionChanged="MainListBox_SelectionChanged"
  ScrollViewer.ManipulationMode="Control">
```

Silverlight 4.0

Windows Phone 7 was based on a slightly modified form of desktop Silverlight 3.0. Windows Phone 7.1 is based on a modified form of desktop Silverlight 4.0. In moving from Silverlight 3.0 to 4.0, more of the desktop Silverlight features have been brought forward and ported to the Windows Phone version of the runtime. In addition to a raft of minor enhancements, the major features introduced are implicit styles, command binding, and a set of data-binding improvements.

Implicit Styles

Chapter 2, “UI Core,” shows how you can define styles in your application for use in multiple UI elements in XAML. A traditional Silverlight 3.0 style has a key value, and you specify that key value in the element to which you want to apply the style. The code that follows defines a named style that can be applied to *TextBlock* elements. This would typically be defined within a resources section in your XAML, and most often in the App.xaml so that it can be used across multiple pages.

```
<Style x:Key="GradientTextStyle" TargetType="TextBlock">
  <Setter Property="FontSize" Value="{StaticResource PhoneFontSizeLarge}"/>
  <Setter Property="Margin" Value="{StaticResource PhoneHorizontalMargin}"/>
  <Setter Property="TextWrapping" Value="Wrap"/>
  <Setter Property="Foreground">
    <Setter.Value>
      <LinearGradientBrush StartPoint="0,1" EndPoint="1,0">
        <GradientStop Offset="0" Color="#FFD80073"/>
        <GradientStop Offset="1" Color="#FF00ABA9"/>
      </LinearGradientBrush>
    </Setter.Value>
  </Setter>
</Style>
```

Having defined the style, you could then go ahead and apply it to any element of the specified target type, by specifying the key name in the *Style* attribute for that element.

```
<TextBlock Text="Lorem ipsum dolor sit amet consectetur elit." Style="{StaticResource
GradientTextStyle}"/>
```

Such styles are known as “explicit” or “named” styles. Silverlight 4.0 supports these as well as implicit styles. These are defined in almost the same way as named styles; the difference is that you do not define a key value for the style. Without a key value, the style will be applied implicitly to all elements of the specified target type, without an explicit *Style* reference. Simply remove the key from the style definition

```
<Style TargetType="TextBlock">
  ... unchanged definition omitted for brevity.
</Style>
```

and then apply the style implicitly, like so:

```
<TextBlock Text="Lorem ipsum dolor sit amet consectetur elit."/>
```

At any time, for any element of the target type, you can override the implicit style if you need to. To do this, you can set the *Style* to null (using the *{x:Null}* syntax), or you can simply apply an explicit style, instead. You can also retain the implicit style, but override one or more of the style's properties.

```
<TextBlock
    Text="Sed do eiusmod tempor incididunt ut labore."
    Style="{x:Null}"/>
<TextBlock
    Text="Ut enim ad minim veniam, quis nostrud mant."
    Style="{StaticResource PhoneTextExtraLargeStyle}"/>
<TextBlock
    Text="Ellaamco laboris nisi ut aliquip ex eapants."
    FontSize="60" />
```

Finally, you can define style hierarchies. To do this, you use the *BasedOn* attribute. Of course, this attribute requires a key name, so if you define a style hierarchy, only the last style in the tree can be implicit; all the others must have key values.

```
<Style x:Key="GradientTextStyle" TargetType="TextBlock">
    <Setter Property="Foreground">
        <Setter.Value>
            <LinearGradientBrush StartPoint="0,1" EndPoint="1,0">
                <GradientStop Offset="0" Color="#FFD80073"/>
                <GradientStop Offset="1" Color="#FF00ABA9"/>
            </LinearGradientBrush>
        </Setter.Value>
    </Setter>
</Style>

<Style TargetType="TextBlock" BasedOn="{StaticResource GradientTextStyle}">
    <Setter Property="FontSize" Value="{StaticResource PhoneFontSizeLarge}"/>
    <Setter Property="Margin" Value="{StaticResource PhoneHorizontalMargin}"/>
    <Setter Property="TextWrapping" Value="Wrap"/>
</Style>
```

Figure 19-6 shows the results of applying this style (this is the *TestImplicitStyles* solution in the sample code). The first *TextBlock* uses the style implicitly, as defined. The second sets a null style, the third replaces it with a different style, and the fourth uses the implicit style but overrides the *FontSize* property.

```
<TextBlock Text="Lorem ipsum dolor sit amet consectetur elit."/>
<TextBlock Text="Sed do eiusmod tempor incididunt ut labore." Style="{x:Null}"/>
<TextBlock Text="Ut enim ad minim veniam, quis nostrud mant." Style="{StaticResource
PhoneTextExtraLargeStyle}"/>
<TextBlock Text="Ellaamco laboris nisi ut aliquip ex eapants." FontSize="60" />
```

CONTOSO

implicit styles

Lorem ipsum dolor sit amet
consecturi elit.

Sed do eiusmod tempor incididunt ut labore.

Ut enim ad minim venia

Ellamco laboris
nisi ut aliquip ex
eapants.

FIGURE 19-6 Using and overriding implicit styles.

Command Binding

Silverlight 4.0 adds support for *ICommand* on the *ButtonBase* and *Hyperlink* classes (and therefore, also their derived classes). The point of the *ICommand* interface is to allow your viewmodel to expose commands that can be data-bound as properties to view controls. Using this approach means that you don't need to supply additional code behind your view in order for it to interoperate with your viewmodel because the binding can be done entirely in XAML. This increases the decoupling between view and viewmodel.

Consider Figure 19-7, which shows the *MvvmDataBinding_ ICommand* solution in the sample code. This is a variation of the *MvvmDataBinding* solution described in Chapter 4, "Data Binding."

The application offers three buttons: one to load data into the viewmodel by using the traditional approach with a custom button *Click* handler; a second button to load data by using the *ICommand* approach; and a third button to clear the data (also using *ICommand*). Below that is a *ListBox* that is data-bound in the normal way to a collection of employee data held in the viewmodel. The application model uses a view (the page) and a viewmodel (*EmployeesViewModel*), backed by a model (*EmployeeModel*). In the traditional approach, the first button is defined with a *Click* handler, as shown in the following:

```
<Button x:Name="loadData" Content="load data (old school)" Click="loadData_Click"/>
```

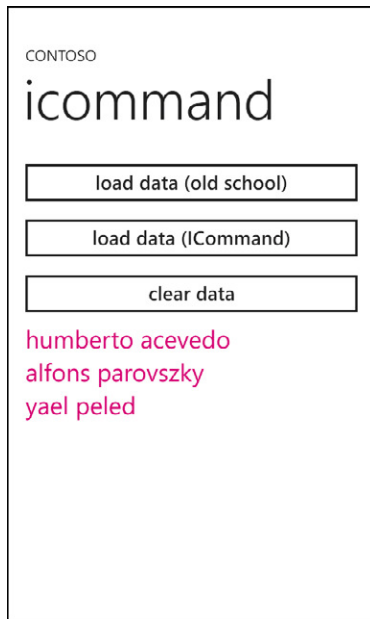


FIGURE 19-7 Data binding and command binding.

This requires a corresponding handler method in the view code-behind, as demonstrated here:

```
private void loadData_Click(object sender, RoutedEventArgs e)
{
    App.ViewModel.LoadDataOldSchool();
}
```

This in turn calls into a method on the viewmodel to load the data (in this case, simulating a more realistic data source). The viewmodel is set up with an *ObservableCollection<T>*, initialized in the constructor. The traditional method to load data simply creates some dummy employees.

```
private ObservableCollection<EmployeeModel> data;
public ObservableCollection<EmployeeModel> Data
{
    get { return data; }
    private set { }
}

public EmployeesViewModel()
{
    data = new ObservableCollection<EmployeeModel>();
}

public void LoadDataOldSchool()
{
    data.Add(new EmployeeModel { Name = "humberto acevedo" });
    data.Add(new EmployeeModel { Name = "alfons parovszky" });
    data.Add(new EmployeeModel { Name = "yael peled" });
}
```

All this works just fine, but it does mean that the view has to have a method to handle the *Click* event on the button, and this in turn must invoke some functionality on the viewmodel. The alternative approach, using *ICommand*, is slightly more decoupled, which is generally a good thing for maintainability and testability. To set this up, you need a class that implements the *ICommand* interface. *ICommand* defines two methods—*Execute* and *CanExecute*—and one event—*CanExecuteChanged*. This class is a façade between the consumer (the page) and the implementation (the viewmodel).

The *Execute* delegate can be set to a method on the viewmodel that will perform the desired operation when the user taps the button. The *CanExecute* delegate can be set to another method on the viewmodel that determines whether or not the button can be executed, and if not, the system disables the button automatically. The *CanExecuteChanged* event is raised when the value of the executability of the command changes.

```
public class Command : ICommand
{
    private Func<object, bool> canExecuteDelegate;
    private Action<object> executeDelegate;
    public event EventHandler CanExecuteChanged;

    public Command(Action<object> executeDelegate)
    {
        this.canExecuteDelegate = (e) => true;
        this.executeDelegate = executeDelegate;
    }

    public Command(
        Func<object, bool> canExecuteDelegate, Action<object> executeDelegate)
    {
        this.canExecuteDelegate = canExecuteDelegate;
        this.executeDelegate = executeDelegate;
    }

    public bool CanExecute(object parameter)
    {
        return canExecuteDelegate(parameter);
    }

    public void Execute(object parameter)
    {
        if (executeDelegate != null)
        {
            executeDelegate(parameter);
        }
    }
}
```

This is how you use the *Command* class in the viewmodel. First, the constructor is enhanced to initialize two *ICommand* objects: one for loading the data, and the other for clearing the data. The first initialization passes both a *CanExecute* delegate and an *Execute* delegate. If the *CanExecute* delegate returns true, the *Execute* delegate will be invoked; otherwise, it will not. The *Execute* delegate, named

LoadDataDelegate here, simply parallels the old-school behavior of creating some dummy employees. The *CanExecute* delegate in this example always returns true. Realistically, it would be doing something meaningful like checking the availability of network connectivity so that the *Execute* delegate can download data from the web, and so on.

The second *ICommand* object is used to clear the collection of data. You initialize this one by using the constructor overload that takes only an *Execute* delegate. This *ICommand* object will therefore have a null *CanExecute* delegate. The *Command* class is implemented always to return *true* in this case.

```
public EmployeesViewModel()
{
    data = new ObservableCollection<EmployeeModel>();
    loadDataCommand = new Command(CanLoadData, LoadDataDelegate);
    clearDataCommand = new Command(ClearDataDelegate);
}

private void LoadDataDelegate(object parameter)
{
    data.Add(new EmployeeModel { Name = "humberto acevedo" });
    data.Add(new EmployeeModel { Name = "alfons parovszky" });
    data.Add(new EmployeeModel { Name = "yael peled" });
}

private ICommand loadDataCommand;
public ICommand LoadDataCommand
{
    get
    {
        return loadDataCommand;
    }
}

private bool CanLoadData(object parameter)
{
    return true;
}

private void ClearDataDelegate(object parameter)
{
    Data.Clear();
}

private ICommand clearDataCommand;
public ICommand ClearDataCommand
{
    get
    {
        return clearDataCommand;
    }
}
```

The final piece of this puzzle is to command-bind the UI. The second and third buttons are bound by using the *Command={Binding}* syntax, binding the first to the *LoadDataCommand* object in the viewmodel, and the second to the *ClearDataCommand* object, both of which are *ICommand* objects.

```
<Button x:Name="loadData" Content="load data (old school)" Click="loadData_Click"/>
<Button Content="load data (ICommand)" Command="{Binding LoadDataCommand}"/>
<Button Content="clear data" Command="{Binding ClearDataCommand}"/>
<ListBox ItemsSource="{Binding Data}" >
    <ListBox.ItemTemplate>
        <DataTemplate>
            <TextBlock
                Text="{Binding Name}"
                FontSize="{StaticResource PhoneFontSizeLarge}"
                Margin="{StaticResource PhoneHorizontalMargin}"
                Foreground="#FFD80073"/>
            </DataTemplate>
        </ListBox.ItemTemplate>
    </ListBox>
```

Data-Binding Enhancements

The data-binding enhancements introduced with Silverlight 4.0 include the following:

- Improved formatting capabilities, with the *StringFormat*, *TargetNullValue*, and *FallBackValue* attributes.
- The ability to sort or group a data-bound collection by using the *CollectionViewSource* class and the *SortDescriptions* and *GroupDescriptions* properties.
- The introduction of the *DataServiceCollection<T>* class, which simplifies binding for data returned by Windows Communications Foundations (WCF) Data Services.
- The use of *INotifyDataErrorInfo*, which brings greater flexibility for data validation.

The *StringFormat* attribute in a data-binding definition has been brought forward from desktop Silverlight to the phone. This means that you have all the rich formatting capabilities available to you when you display data-bound values. Figure 19-8 is a screenshot of the *TestStringFormat* solution in the sample code, which exercises a very small number of the possible formatting options. The full list is documented on MSDN at [http://msdn.microsoft.com/en-us/library/26etazsy\(VS.95\).aspx](http://msdn.microsoft.com/en-us/library/26etazsy(VS.95).aspx).

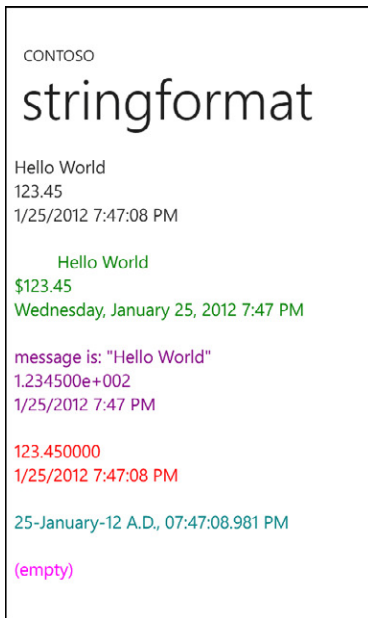


FIGURE 19-8 A demonstration of the *StringFormat* data-binding attribute.

Compare the screenshot with the XAML where these *StringFormat* values are declared. Observe the use of the backslash to escape the meaning of special characters in the formatting string, such as the open "{" and close "}" brackets and the comma (,). Also note that you cannot use the backslash to escape double quotation marks; instead, you must use the XML *"* delimiter.

```
<TextBlock Text="{Binding StringValue}" />
<TextBlock Text="{Binding DoubleValue}" />
<TextBlock Text="{Binding DateTimeValue}" />
<Grid Height="30"/>
<TextBlock Text="{Binding StringValue, StringFormat=\{0\,20\}}" Foreground="Green"/>
<TextBlock Text="{Binding DoubleValue, StringFormat=c}" Foreground="Green"/>
<TextBlock Text="{Binding DateTimeValue, StringFormat=f}" Foreground="Green"/>
<Grid Height="30"/>
<TextBlock Text="{Binding StringValue, StringFormat='message is: &quot;{0}&quot;'}"
    Foreground="Purple"/>
<TextBlock Text="{Binding DoubleValue, StringFormat=e}" Foreground="Purple"/>
<TextBlock Text="{Binding DateTimeValue, StringFormat=g}" Foreground="Purple"/>
<Grid Height="30"/>
<TextBlock Text="{Binding DoubleValue, StringFormat=\{0:n6\}}" Foreground="Red"/>
<TextBlock Text="{Binding DateTimeValue, StringFormat=G}" Foreground="Red"/>
<Grid Height="30"/>
<TextBlock Text="{Binding DateTimeValue, StringFormat='dd-MMMM-yy g, hh:mm:ss.fff tt'}"
    Foreground="Teal"/>
```

You can also use the *TargetNullValue* and/or *FallBackValue* attributes. At the bottom of the page, you declare three more *TextBlock* controls, each bound to the same string property in another model instance; in this case, the string property is set to null in code. The first variation does not specify what to do in the case of a null value, so nothing is displayed. The second specifies that the string "(empty)" should be used, via the *TargetNullValue* attribute. The third variation uses the *FallBackValue* attribute to specify that the string "unknown" should be displayed if something goes wrong with the data-binding. In the screenshot in Figure 19-8, only the second one results in displayed text, in this instance.

```
<TextBlock Text="{Binding StringValue}" Foreground="Magenta"/>
<TextBlock Text="{Binding StringValue, TargetNullValue=(empty)}" Foreground="Magenta"/>
<TextBlock Text="{Binding StringValue, FallBackValue=unknown}" Foreground="Magenta"/>
```

Another feature brought over from desktop Silverlight is the ability to sort or group a collection as part of data-binding. This uses the *CollectionViewSource* class and the *SortDescriptions* and *Group Descriptions* collection properties. Figure 19-9 shows two different versions of the *TestGrouping* solution in the sample code. When the user selects a store item from the list of stores in the first column, the view updates the second column with those products that are associated with the selected store. The key here is that this is all done via data-binding—there is no *SelectionChanged* event handler in the code, for instance.



FIGURE 19-9 Data-binding with *CollectionViewSource* objects, and sorting and grouping collections.

In both cases, the model consists of a *StoreModel* class to represent a store as well as a string property for the store name and a collection property for the store products. The store *Product* model, in turn, consists of a string for the name and a double for the price.

```

public class StoreModel
{
    public String Name { get; set; }
    public ObservableCollection<Product> Products { get; set; }

    public StoreModel(String name)
    {
        Name = name;
        Products = new ObservableCollection<Product>();
    }
}

public class Product
{
    public String Name { get; set; }
    public double Price { get; set; }

    public Product(String name, double price)
    {
        Name = name;
        Price = price;
    }

    public override string ToString()
    {
        return String.Format("{0} - {1:C2}", Name, Price);
    }
}

```

The viewmodel is a collection of *StoreModel* objects, and the constructor creates some demonstration data (this is an arbitrary collection of stores and products, in no particular order).

```

public class StoreViewModel : ObservableCollection<StoreModel>
{
    public StoreViewModel()
    {
        StoreModel grocery = new StoreModel("grocery");
        grocery.Products.Add(new Product("peas", 2.50));
        grocery.Products.Add(new Product("sausages", 3.00));
        grocery.Products.Add(new Product("coffee", 10.00));
        grocery.Products.Add(new Product("cereal", 3.00));
        grocery.Products.Add(new Product("milk", 2.50));
        this.Add(grocery);

        StoreModel pharmacy = new StoreModel("pharmacy");
        pharmacy.Products.Add(new Product("toothpaste", 3.99));
        pharmacy.Products.Add(new Product("aspirin", 5.25));
        this.Add(pharmacy);

        StoreModel bakery = new StoreModel("bakery");
        bakery.Products.Add(new Product("croissants", 5.00));
        bakery.Products.Add(new Product("bread", 4.00));
        bakery.Products.Add(new Product("vanille kipferl", 6.50));
        bakery.Products.Add(new Product("amandines", 5.00));
        this.Add(bakery);
    }
}

```

In the XAML, you define two *CollectionViewSource* objects as resources. The first is bound to the *StoreViewModel*; that is to say, all stores. The second *CollectionViewSource* is bound to the first *CollectionViewSource*, specifying the *Products* within that collection as the path. This effectively provides a pivot mechanism on the stores.

```
<phone:PhoneApplicationPage.Resources>
    <local:StoreViewModel x:Key="shoppingItems" />
    <CollectionViewSource x:Key="cvs1" Source="{Binding Source={StaticResource
        shoppingItems}}"/>
    <CollectionViewSource x:Key="cvs2" Source="{Binding Source={StaticResource cvs1},
        Path=Products}"/>
</phone:PhoneApplicationPage.Resources>
```

You then define two *ListBox* controls. For the first one, set its *ItemsSource* to the first *CollectionViewSource*, and then data-bind the *TextBlock* in the item template to the store name property. For the second *ListBox*, set its *ItemsSource* to the second *CollectionViewSource*; data-bind the *TextBlock* in the item template implicitly to the whole *Product* item. Recall that the *Product* item overrides *ToString* to render both the product name and price.

```
<ListBox ItemsSource="{Binding Source={StaticResource cvs1}}" Grid.Row="1">
    <ListBox.ItemTemplate>
        <DataTemplate>
            <TextBlock Text="{Binding Name}" Margin="{StaticResource PhoneHorizontalMargin}"/>
        </DataTemplate>
    </ListBox.ItemTemplate>
</ListBox>
<ListBox ItemsSource="{Binding Source={StaticResource cvs2}}" Grid.Column="1" Grid.Row="1">
    <ListBox.ItemTemplate>
        <DataTemplate>
            <TextBlock Text="{Binding}" Margin="{StaticResource PhoneHorizontalMargin}"/>
        </DataTemplate>
    </ListBox.ItemTemplate>
</ListBox>
```

This results in the experience shown in the first screenshot; the first column lists all stores, and the second column lists only those products for the currently selected store. The second screenshot has two further enhancements: sorting and grouping. The stores are sorted alphabetically, and the products within each store are grouped according to price. This is achieved very simply in XAML by specifying a *SortDescription* for the first *CollectionViewSource*, and a *GroupDescription* for the second.

```
<CollectionViewSource x:Key="cvs1" Source="{Binding Source={StaticResource shoppingItems}}">
    <CollectionViewSource.SortDescriptions>
        <scm:SortDescription PropertyName="Name"/>
    </CollectionViewSource.SortDescriptions>
</CollectionViewSource>

<CollectionViewSource x:Key="cvs2" Source="{Binding Source={StaticResource cvs1},
    Path=Products}">
    <CollectionViewSource.GroupDescriptions>
        <PropertyGroupDescription PropertyName="Price" />
    </CollectionViewSource.GroupDescriptions>
</CollectionViewSource>
```

Both *SortDescriptions* and *GroupDescriptions* are collection properties, which means that you can specify multiple sorting and grouping definitions for each *CollectionViewSource*, if required. Also note that the use of *SortDescriptions* requires an XML namespace reference to the *System.ComponentModel* namespace in *System.Windows.dll*.

Another significant data-binding enhancement is the introduction of the *DataServiceCollection<T>* class. This provides simplified binding for data returned by WCF Data Services. The key to this class is that it derives from *ObservableCollection<T>*, which implements *INotifyCollectionChanged* and *INotifyPropertyChanged*, allowing it to update bound data automatically. *DataServiceCollection<T>* is discussed in Chapter 17, "Enhanced Connectivity."

The final data-binding enhancement to consider is *INotifyDataErrorInfo*. You would implement this interface on your viewmodel or model class to signify whether there are currently any validation errors on the object. Silverlight 4.0 also brings with it the *IDataErrorInfo* interface, which is almost identical. You are encouraged to use *INotifyDataErrorInfo* in preference to *IDataErrorInfo*. The difference is that *INotifyDataErrorInfo* exposes an event that can be raised when there is a validation error. This removes the need for validation to be immediate; instead, you could perform validation asynchronously (perhaps querying a web service), and then raise the event when you eventually determine the result.

By the same token, you can use this to perform validation across multiple properties, for which you cannot fully determine whether an individual property is valid until you have examined other properties. This applies especially in circumstances when you need to perform not just cross-property validation, but whole-entity validation. It might be that no single property is invalid but that the combination of several (or all) of the property values is invalid.

Figure 19-10 presents the *MoreValidation* solution in the sample code is a minor variation on the *BindingValidation* sample from Chapter 4.

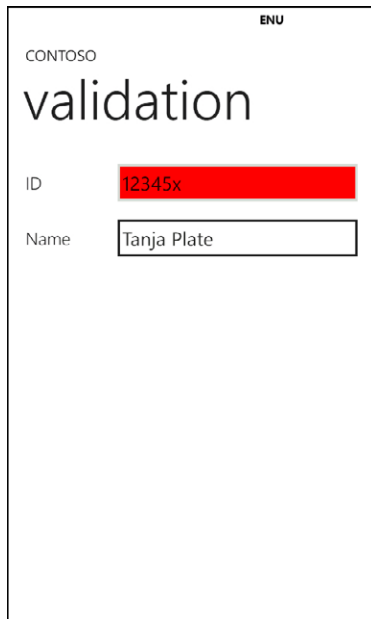


FIGURE 19-10 Data-binding validation via *INotifyDataErrorInfo*.

The XAML is unchanged and specifies that the *idText* *TextBlock* has data-binding validation associated with it, and that the validator error handler is at the scope of the containing grid.

```
<Grid
  x:Name="ContentPanel" BindingValidationError="ContentPanel_BindingValidationError">
  ... irrelevant code omitted for brevity.
  <TextBox x:Name="idText">
    <TextBox.Text>
      <Binding
        Mode="TwoWay" Path="ID"
        NotifyOnValidationError="true"
        ValidatesOnExceptions="true"/>
    </TextBox.Text>
  </TextBox>
</Grid>
```

As before, the binding validation event handler repaints the background of the *TextBox* either red or the default background color, depending on whether an error has been added to or removed from the collection. This code is also unchanged from the version in Chapter 4.

```
private void ContentPanel_BindingValidationError(
  object sender, ValidationErrorEventArgs e)
{
  Debug.WriteLine("ContentPanel_BindingValidationError");

  TextBox t = (TextBox)e.OriginalSource;
```



```

        if (e.Action == ValidationErrorEventAction.Added)
        {
            t.Background = new SolidColorBrush(Colors.Red);
        }
        else if (e.Action == ValidationErrorEventAction.Removed)
        {
            t.ClearValue(TextBox.BackgroundProperty);
        }

        e.Handled = true;
    }
}

```

The new code is in the model class itself. This now implements *INotifyDataErrorInfo*, which defines the *ErrorsChanged* event (the *GetErrors* method) and the *HasErrors* property. To support these, you define a *Dictionary<T>* to hold the collection of errors. When you validate one of the model properties (*ID*, in this example), if there is a validation error, you add an entry to the dictionary, and then raise the *ErrorsChanged* event.

```

public class Employee : INotifyDataErrorInfo
{
    public String Name { get; set; }
    private String id;
    public String ID
    {
        get { return id; }
        set
        {
            errors.Remove("ID");
            int tmp;
            if (Int32.TryParse(value, out tmp))
            {
                id = value;
            }
            else
            {
                //throw new ArgumentOutOfRangeException("value must be an integer");
                errors.Add("ID", "value must be an integer");
            }

            EventHandler<DataErrorsChangedEventArgs> handler = ErrorsChanged;
            if (handler != null)
            {
                handler(this, new DataErrorsChangedEventArgs("ID"));
            }
        }
    }

    private Dictionary<String, String> errors = new Dictionary<String, String>();
    public event EventHandler<DataErrorsChangedEventArgs> ErrorsChanged;

    public IEnumerable GetErrors(string propertyName)
    {

```

```

        if (!errors.ContainsKey(propertyName))
        {
            return String.Empty;
        }
        else
        {
            var tmp = errors[propertyName];
            return new String[] { tmp };
        }
    }

    public bool HasErrors
    {
        get { return errors.Any(); }
    }
}

```

Note that if you anticipate having to support more than one error per property, then the simple *Dictionary* shown in the preceding code would not be sufficient. In that case, you'd need something like a *Dictionary<String, List<String>>*, instead.

Summary

In this chapter, you saw how the move from Silverlight 3.0 to Silverlight 4.0 has brought with it a raft of improvements, both major and minor. Many of the perceived gaps in the version 7 support for page navigation, backstack management, and data-binding have been filled. Other improvements are more related to the quality of the UX, such as the move to 32 bpp for rendering images, background image decoding for optimizing the UI responsiveness on the phone, and the introduction of a touch thread on the platform.

Tooling Enhancements

In Chapter 19, “Framework Enhancements,” you saw how the Windows Phone 7.1 SDK introduced a range of improvements in the application platform, the programming model, and the Microsoft Silverlight runtime. In this chapter, you’ll see how the developer tooling support has also improved. The emulator has been significantly enhanced, as has the debugger. There’s a new marketplace test kit designed to help you finalize your application prior to submitting it for publication. Probably the most significant new tooling feature is the Windows Phone Performance Analysis tool, also known as the Profiler, which you can use to build a profile of your application’s CPU and memory consumption. As you can imagine, this can be of tremendous assistance when you’re trying to identify problem areas in your application, or for optimizing its performance.

Emulator Improvements

The emulator in Windows Phone 7.1 has a number of improvements, mostly specific to either the accelerometer or location. Improvements to support sensor development (including accelerometer) are discussed in Chapter 16, “Enhanced Phone Services.” The location support follows a similar vein, in that it allows you to test location-related code to simulate changes in location over time, without the use of a physical device. The Location tab in the emulator offers a Bing map and a number of controls for manipulating location, as shown in Figure 20-1. By default, the Live feature is enabled, which uses real geolocation capabilities to track your current position. If your code is using a *GeoCoordinate Watcher*, and this is turned on, then your application running in the emulator will receive the position data.

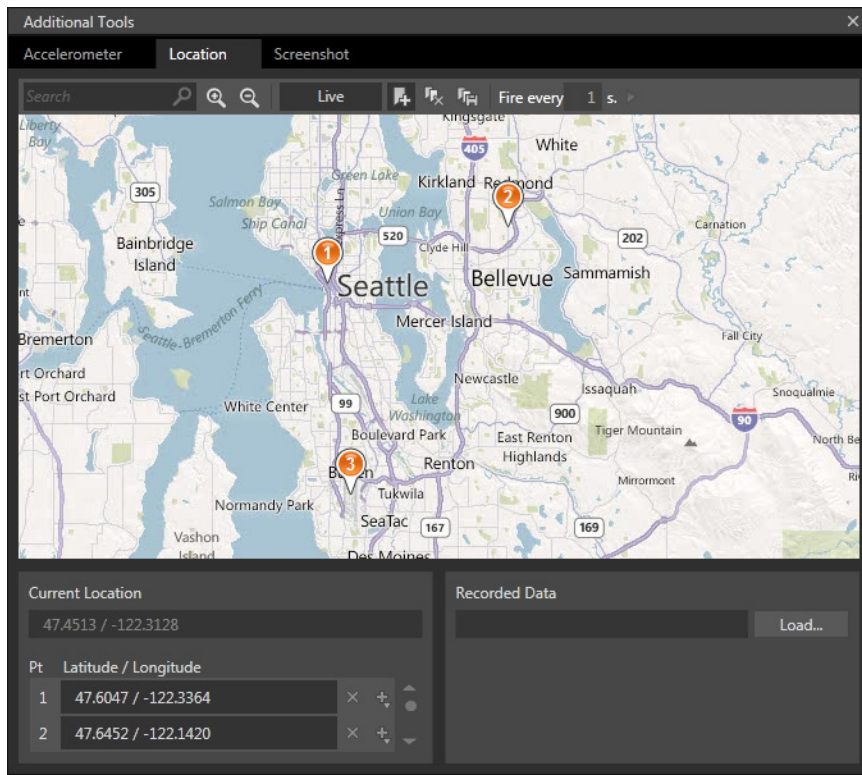


FIGURE 20-1 You can use the emulator to simulate location changes.

You can also click a spot on the map in the emulator to simulate changes in location; each change will raise a *GeoPositionChangedEvent* in your application. This actually makes it more useful than testing on a device, because you can easily simulate many location changes—and large ones, at that—in a way that would be difficult (or simply impossible) using a physical device. As an alternative to Live location behavior, you can tap the map to add push-pins at any position. Each push-pin is tracked in the Latitude/Longitude list at the bottom of the emulator. Having set up a series of location changes, you can then play back the series, and again, each change will raise an event in your application. You can also specify the time interval in seconds between each position change. Finally, you can save a series of locations to an XML file, and then reload it in a subsequent emulator session, or create the file manually for use in the emulator. The file format is very simple: each position is represented by a *GpsData* element, with *latitude* and *longitude* attributes, as shown here:

```
<?xml version="1.0" encoding="utf-8"?>
<WindowsPhoneEmulator xmlns="http://schemas.microsoft.com/WindowsPhoneEmulator/2009/08/
SensorData">
  <SensorData>
    <Header version="1" />
    <GpsData latitude="47.6047023940839" longitude="-122.336365090179" />
    <GpsData latitude="47.6451686929145" longitude="-122.142042257115" />
    <GpsData latitude="47.4513249698352" longitude="-122.312802411839" />
  </SensorData>
</WindowsPhoneEmulator>
```

The third tab in the version 7.1 emulator is the Screenshot tab. You can use this to take a screenshot of your running application and save it as a PNG file to disk. You can take a screenshot whenever you like, as often as you like, and the full emulator screen is captured, including any chrome that might be displayed (system tray or application bar). If you're using this feature to create the images required for marketplace submission, you should set the emulator size to 100 percent so that you get a full 480x800-pixel image.

Debugger Experience

In Chapter 15, “Multi-Tasking and Fast App Switching,” you saw how the version 7.1 application lifecycle behavior has been enhanced to improve responsiveness when the user is switching between applications. The Microsoft Visual Studio debugger has also been updated to accommodate the modified runtime behavior. There's a new option in the Debug tab of the project settings with which you can set the behavior of your application at the point of deactivation.

In Windows Phone 7, in the normal case, an application was tombstoned upon deactivation. In version 7.1, the normal case is that the application is made dormant but is not tombstoned. The application might eventually be tombstoned; for example, if it falls off the backstack or if memory resources fall below a critical threshold. So, you need to test the behavior in both the tombstoning and the non-tombstoning cases. You can toggle the debug option to change the behavior for each test run, as shown in Figure 20-2.

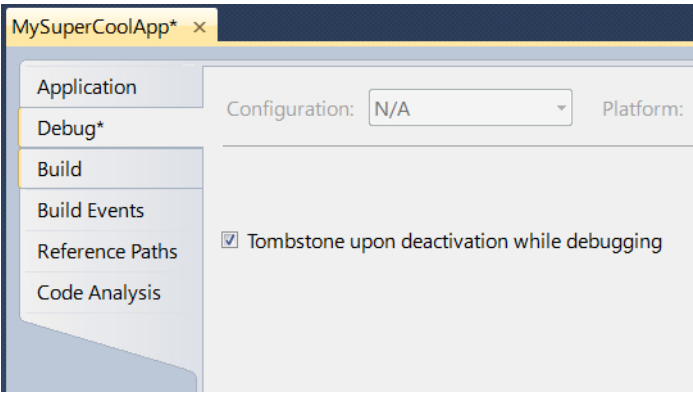


FIGURE 20-2 You can set your application to tombstone upon deactivation while debugging.

The behavior of the debugger under different conditions is summarized in Table 20-1.

TABLE 20-1 Debugger Behavior with Different Application Conditions

Application Type	User Action	Version 7.1 Debugger Behavior
Standard application running in the foreground	Taps the Back button	The debug session ends.
	Taps the Start button	Tombstone or fast application switching (FAS), depending on the debugger setting. In either case, the debug session remains active.
	Activates after tombstone or FAS	Attaches to the activated application. In the FAS case, attach to the same process. In the tombstone case, attaches to the new process.
	Launches a new instance after tombstone or FAS	The debug session ends, and the new instance is launched in non-debug mode.
	Taps the Stop Debugging button in Visual Studio	The debug session ends, the application is terminated, and the backstack entry for the application is removed.
Application with a Background agent (Audio, periodic, or resource intensive)	Taps the Back button	The debug session is kept alive.
	Taps the Start button	Tombstone or FAS, depending on the debugger setting. In either case, the debug session remains active.
	Activates after tombstone or FAS	Attaches to the activated application. In the FAS case, attach to the same process. In the tombstone case, attaches to the new process.
	Launches a new instance after tombstone or FAS	The debug session remains alive, the application is launched in debug mode, and the debugger attaches to the foreground application.
	Background agent kicks in	The debug session remains alive, and the debugger attaches to the background agent.
	Taps the Stop Debugging button in Visual Studio	The debug session ends, the application and background agent are terminated, and the backstack entry for this application is removed.
Application with a background service (alarm, push notification, or background transfer service)	Taps the Back button	The debug session ends.
	Taps the Start button	Tombstone or FAS, depending on the debugger setting. In either case, the debug session remains active.
	Activates after tombstone or FAS	Attaches to the activated application. In the FAS case, attach to the same process. In the tombstone case, attaches to the new process.
	Launches a new instance after tombstone or FAS	The debug session ends, and the new instance is launched in non-debug mode.
	Taps the Stop Debugging button in Visual Studio	The debug session ends, the application is terminated, and the backstack entry for the application is removed.

Marketplace Test Kit

The marketplace ingestion process checks your application’s capabilities and overwrites the entries in the WMAppManifest.xml. When the application is later installed on a user’s phone, these capabilities are used to determine the security sandbox in which the application will run, constraining the application to only those features that it is known to use. Chapter 1, “Vision and Architecture,” described the core tools available for use in Windows Phone development, including the Capability Detection tool, which you can use to confirm the specific capabilities used in your application, prior to submitting it for publication. This tool is not shipped with the version 7.1 SDK. Instead, it introduces the marketplace test kit, which covers the same functionality, plus additional functionality for testing other marketplace requirements. The test kit is available in Visual Studio under the Project menu, and it is only for use on a version 7.1 project; you cannot use the test kit for a version 7 project.

The menu is dynamically constructed, and you need to select the project itself (or any part of the project) in Solution Explorer before the test kit option becomes available. Also note that the test kit uses rules and test cases that are continually synchronized with the rules and test cases used in the full marketplace ingestion process. Upon startup, the test kit will make a web service call to the marketplace to check for updated rules or test cases. If needed, it will prompt you to install any updates. The test kit offers four tabs:

- **Application Details** This is where you specify marketplace images.
- **Automated Tests** Use this for validating the static marketplace requirements.
- **Monitored Tests** This is where the test kit monitors your application’s behavior during execution.
- **Manual Tests** You can use this tab to track your manual testing results against a set of test cases.

On the Application Details tab, you specify the application tile and screenshot images used in marketplace. These must all be in PNG format, with no transparency defined, and using the sizes as noted in Table 20-2. All of these are required, including the screenshot, although you can also specify up to seven additional (optional) screenshots. These requirements can be updated from time to time, and the details will be indicated on each placeholder image in the test kit itself, as shown in Figure 20-3.

TABLE 20-2 Marketplace Image Requirements

Image	Required Size in Pixels
Large application tile	173x173
Small application tile	99x99
Marketplace application tile	200x200
Application screenshot	480x800

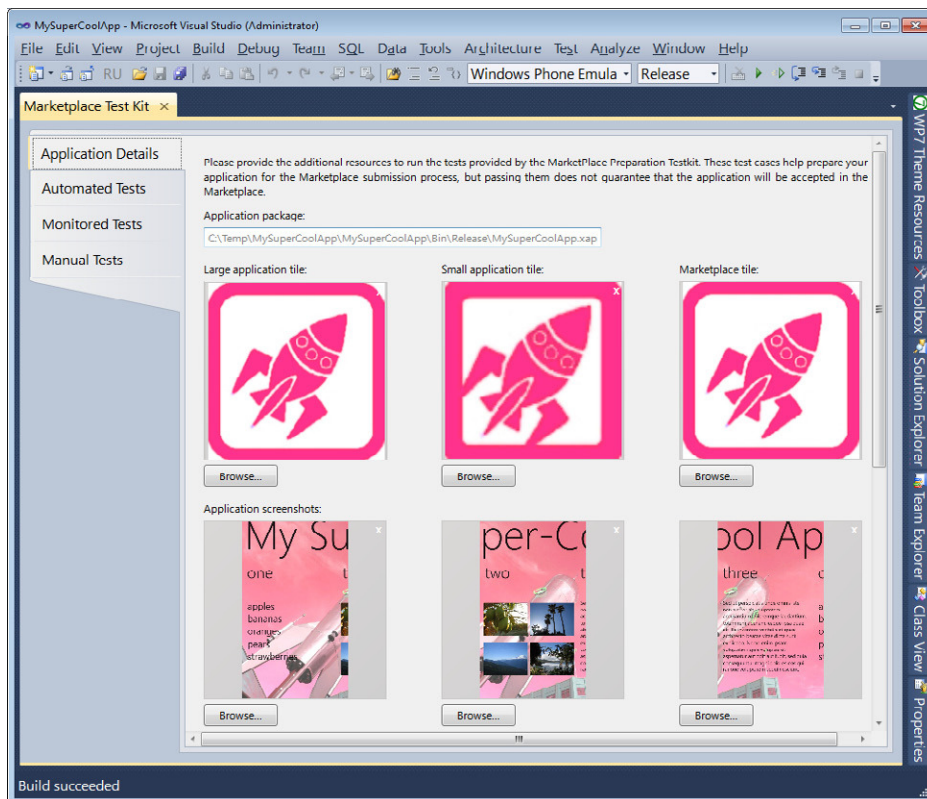


FIGURE 20-3 Specify your marketplace images in the Application Details tab of the marketplace test kit.

The Automated Tests tab validates the static marketplace requirements; that is, it validates the size of your XAP, the capabilities required by your application, the application icon and tile images, and your marketplace screenshots, as shown in Figure 20-4. Before you can run these tests, you must build a release version of your XAP.



Note The test kit sometimes incorrectly reports that the ID_CAP_NETWORKING capability is required by your application. To double-check the test kit's report, you can also run the Capability Detection tool from the version 7 SDK. Then, you should update your WMApManifest.xml to specify only those capabilities that this tool reports.

The Capability Detection tool scans your intermediate language (IL) code to identify which classes are used, and then determines the required capabilities from that scan. This works even if you use obfuscation on your code. However, it cannot detect APIs that you use via reflection. This means that it is possible to build an application that uses unsupported APIs; thus, this application can pass the marketplace test kit check as well as the production marketplace ingestion process. The problem here is that this effectively bypasses a set of reliability tests, and your application could then simply fail at runtime. The bottom line is that if you use reflection in your application, you must be very careful to avoid any unsupported APIs.

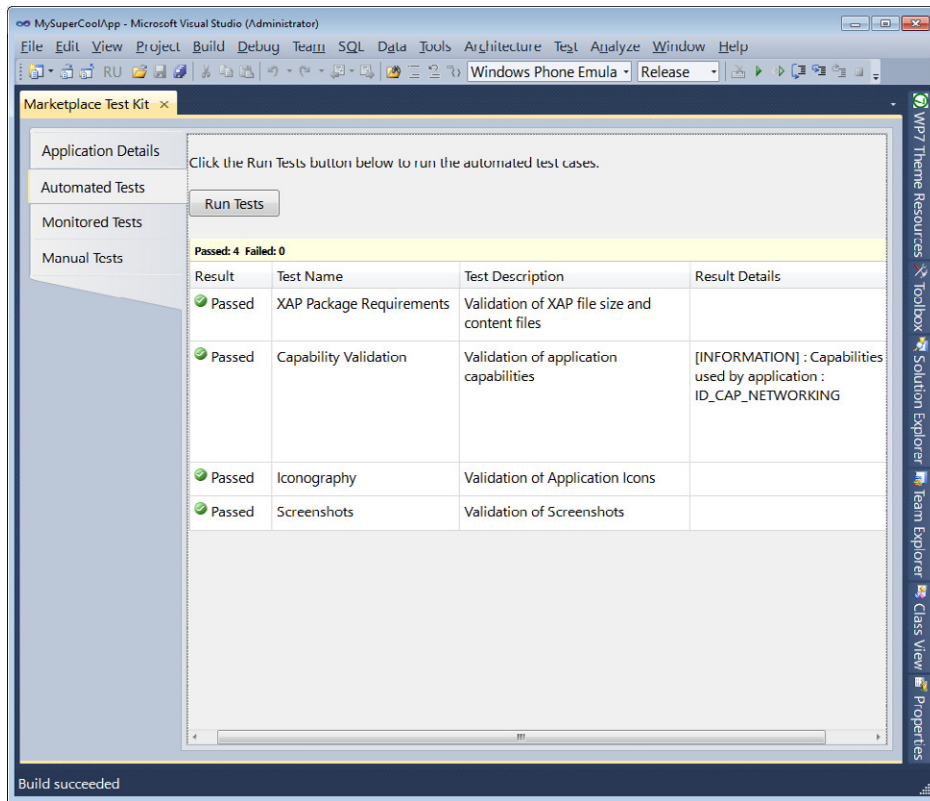


FIGURE 20-4 Perform static validation by using the Automated Tests tab.

Apart from reflection, the tool also cannot detect APIs that you only use in XAML and not in code. There are two places where this is especially important: the *WebBrowser* control and the *Media Element* control. For both of these, you must give them an *x:Name* property value (even if you don't need to access them in the code-behind) so that the IL scanner will see them.



Note If the tools report that you're not using any specific capabilities, and you therefore remove *all* the capability entries from your manifest, the next time you open the project, you'll get a spurious error message related to this. Specifically, the message will alert you that, "you are using a project created by previous version of Windows Phone Developer Tools CTP." You can safely ignore this message.

The Monitored Tests tab (Figure 20-5) examines your application as you run it and exercise its functionality. The tests look for certain specific behaviors, which include your application starting up within the published maximum required time (5 seconds to first screen, 20 seconds to user interface (UI) responsiveness), the peak memory is within published limits (90 MB), all exceptions are handled so that they do not propagate out of your application, and you do not interfere with the Back button behavior in an unexpected way.

You must run these tests on a connected device—performance on the emulator is not a good representation of physical device performance, and the test kit will only allow you to run these tests on a device. This is an opportunity to test your application thoroughly. You should exercise all code paths, navigate to all pages, and perform all operations that the user might perform. This especially includes scenarios at the edges of your application’s control, such as the use of Launchers and Choosers, the behavior when there’s an incoming phone call or SMS, fast application switching and tombstoning. Testing at this stage should be destructive; your aim is to try to cause failures in the application, so that you can catch them before submitting to the marketplace. Note that monitored test results are not persisted anywhere, so you should note the results for each test run on-screen before closing the test kit.

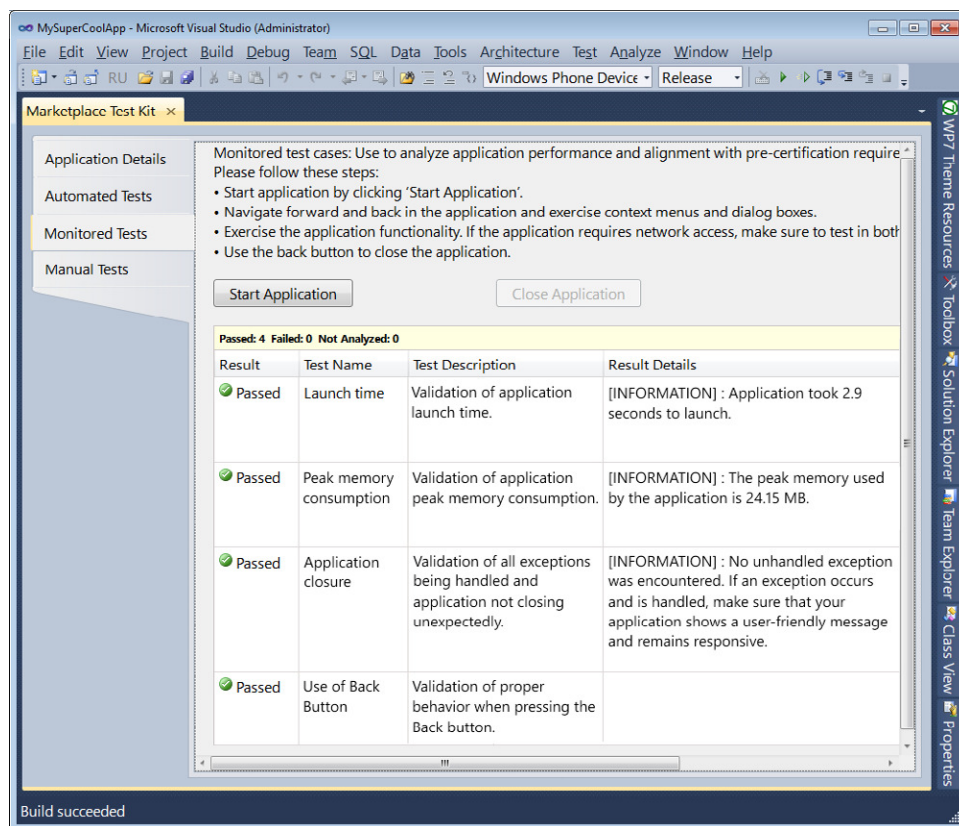


FIGURE 20-5 The Monitored Tests tab oversees behavior as you exercise your application.

The Manual Tests tab provides about 50 specific test cases, which you are asked to work through manually. Each test case has a link to the specific certification requirement on MSDN. The test kit does not monitor these tests; rather, they are for your benefit to help guide you through a comprehensive set of scenarios for which you should be testing. These tests are designed to match the tests performed during marketplace ingestion. Note that some of the tests might not apply to your application

(for example, your application might not make use of game-specific or media-specific features, background audio, background transfers, and so on). For tests that don't apply, simply leave them as pending. The aim here is to ensure that you test all cases that do apply, and that your application passes these tests. It is obviously in your own interests to flag tests as failed until you fix the cause of the failure. Thus you should use the test kit here as a bug tracking tool, as shown in Figure 20-6.

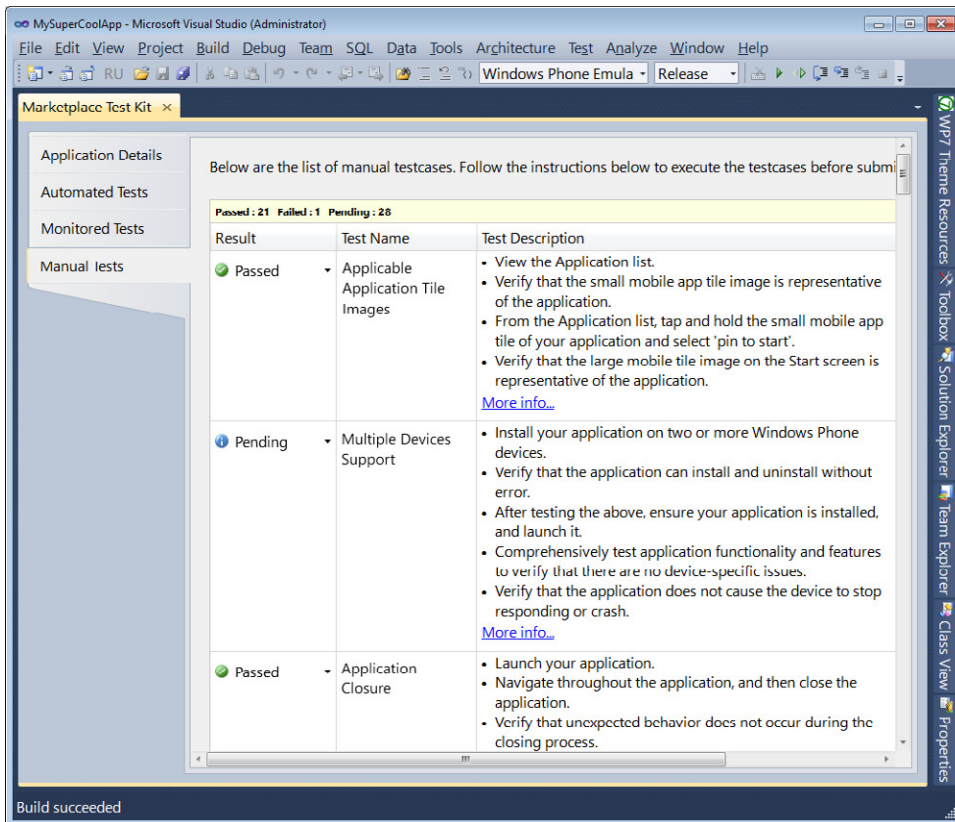


FIGURE 20-6 You can use the Manual Tests tab as a bug tracking tool.

When you run the test kit, it creates a folder named SubmissionInfo in your project, and then copies any marketplace image files you specified into this folder. It also creates two XML files: Settings.xml (a simple list of the marketplace image files), and ManualTestResults.xml (logs the results of all manual tests run for the application). Note that once you have added an image to the list in the Application Details tab, you can't remove it via the UI, although you can replace it with another image (so long as the new image has a different name). Alternatively, you can close the test kit and edit the Settings.xml directly, or you can simply delete it altogether so that the test kit can recreate it from scratch next time you run it. When you've completed all tests, you can also use the Submission Info folder as your source for images when you finally submit your application to marketplace.

The Profiler

The version 7.1 SDK ships with the Windows Phone Performance Analysis tool, or Profiler, as an integrated component in Visual Studio. You can think of this as a much more sophisticated and powerful version of the debug performance counters discussed in Chapter 8, “Diagnostics and Debugging.” The idea is that you start the profiling tool, which launches your application, and you then exercise the application’s functionality. The Profiler will track metrics on your application’s use of resources, and then write these out to a log file. After you stop the application, you can examine the log in the Profiler. This presents a graphical display as well as text descriptions of what was going on during any selected time period. There are multiple options for configuring what kind of profiling information you want to track, and suggestions for how to improve your application’s performance in cases where it is approaching or exceeding acceptable limits.

The following sections use the *ProductCatalog* application in the sample code as a reference. The initial version of this application has no performance optimizations, many of which can be easily identified through use of the Profiler. The application is shown in Figure 20-7.



FIGURE 20-7 The *ProductCatalog* sample is used to demonstrate profiling.

The application has three pages:

- **MainPage** This page has a background image and a short list of menu items, corresponding to the product categories. The user can tap on a category item in the list to navigate to the *CategoryPage* for that item.
- **CategoryPage** This has a *ListBox* of product items, each with a small *Image* control and a *TextBlock*. From here, the user can tap an individual item to select it. This navigates to the *DetailsPage* for that item.
- **DetailsPage** This page has a larger *Image* control and a *TextBlock*, both of which are inside a *ScrollView*. Both the *CategoryPage* *ListBox* items and the *DetailsPage* use the same image file, which is scaled according to the requirements of each page.

To start the Profiler, you should first build a release version of your application, and then deploy it to an attached device (profiling on the emulator or with a debug build is normally only useful as a reference point). Then, select the Start Windows Phone Performance Analysis option on the Debug menu. If you're using Visual Studio Ultimate, don't be distracted by the similarly named Start Performance Analysis option: that option will not work with Windows Phone projects. Also note that even if you did want to profile a debug build, you cannot run the Profiler during a debug session, and you cannot debug during a Profiler session. The Profiler first asks you if you want to focus on execution or memory. An execution pass pays more attention to visual rendering, frame rates, and function call sequences. A memory pass pays more attention to object allocation, deallocation, and garbage collection. You should do multiple profile passes, sometimes focusing on execution, sometimes memory. Within the two major options, you also have more detailed choices. These are discussed later on in this section.

After you have configured the profiling pass, click the link to launch the application. This injects instrumentation into the application and deploys it to the device, and profile recording starts just as the application is launched. You then run through the behavior on the application that you want to examine, and finally exit out of the application by pressing the Back key. Then, stop the Profiler. At this point, the profile recorder writes out the log. The key log file is an SAP file in your project folder, and this is also added to your project. The file is named by using a combination of the project name and a timestamp. The file itself is an XML that contains references to the raw binary logs.

The Profiler creates these logs in a subfolder of a folder named PerformanceLogs, which it creates in your project folder. Each profile pass creates a fresh subfolder. This way, you have a useful history of performance that you can track as you make optimizations to your code. After the log is written out, the profile analyzer kicks in. This reads the log back in, parses it, and generates a graphical display of the behavior over time. Figure 20-8 shows a sample run of about 45 seconds. You can reload an SAP file from an earlier run at any time by just double-clicking it in Solution Explorer. You can also open multiple SAP files at the same time so that you can do side-by-side comparisons.

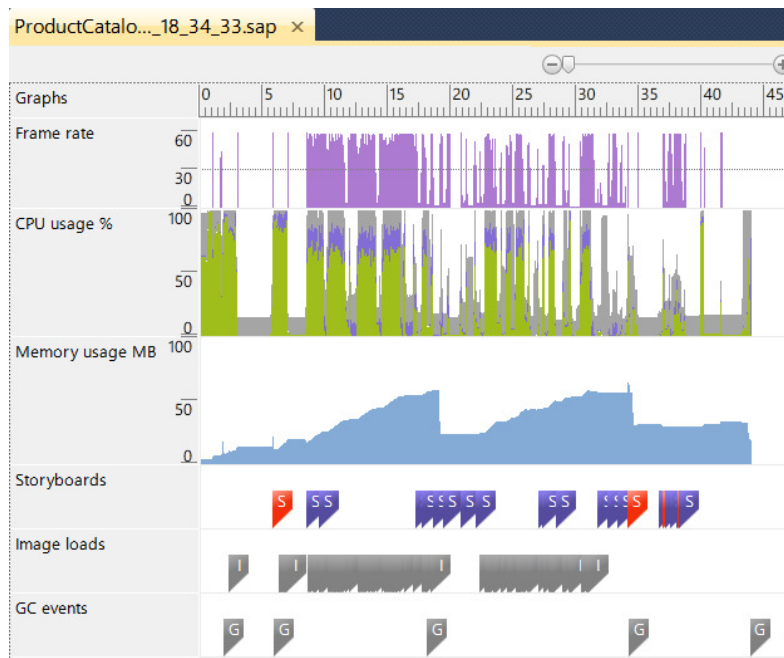


FIGURE 20-8 It's useful to perform a quick high-level pass first.

You should perform a quick pass first, running through all the major features of the application, but not drilling down into too much detail or spending more than a couple of minutes on the pass. This will give you a big-picture view of your application's performance. After that, you should perform more targeted passes, focusing on individual areas such as (but not limited to):

- All the functionality of a single page at a time
- All the page navigation transitions
- All the areas where you make web connectivity calls
- All the areas where you perform local database operations
- All the areas that are heavy in user input
- Application behavior across FAS at different points

It is also worth performing a long-running pass, in which you might concentrate on performance degradation over time that are perhaps caused by memory leaks, and so on. Note, however, that the log files can become extremely large very quickly. Even a 60-second pass typically results in a 10 MB log file. Also note that you should keep an independent track of time, so that you can match up the interesting events in the profile analysis with the time you performed given operations in the application. If you open the system clock from the desktop toolbar, it shows an enhanced clock that tracks seconds, which is often all you need.

You can delete an SAP file from your project when you no longer need it, but this doesn't delete any of the raw binary log files; you should then delete those manually. Also note that the Profiler generates an AssemblyCache folder, and the contents of this can be deleted after you stop the Profiler. This is because they are not used in the subsequent parse/analyze phase, and you do not need to keep them on disk, as they will be regenerated with each fresh pass.

After you have carried out the quick high-level pass, you can zoom the display to get a more detailed view. Figure 20-9 zooms in on the first 10 seconds of the high-level pass.

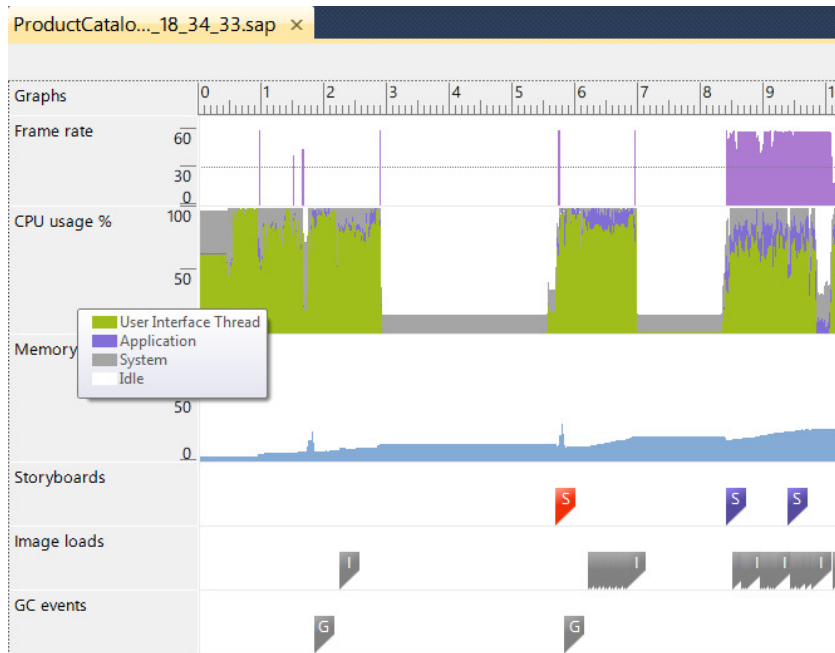


FIGURE 20-9 Zoom in to see a detailed view of the profile analysis.

The graph is divided into six bands, as follows:

- **Frame Rate** This is the rate (in frames per second [FPS]) at which the system was redrawing the screen. Recall from Chapter 8 that the best frame rate you can achieve is ~60 FPS, and that you should not be happy with a rate below 30 FPS. The scale of the Y-axis on the graph is from 0 to 60. The graph is uni-colored for the entire range and has a demarcation line at 30 FPS to indicate if any frames are dropping below the acceptable range.
- **CPU Usage %** Like the desktop Task Manager, this indicates how busy the CPU was at any given time. If you hover over the CPU usage band, the Profiler displays a legend that shows the color scheme used for CPU usage, which is as follows:
 - Green for the UI thread. You should expect the UI thread to be doing a lot of work, but you don't want it at its maximum utilization all the time.
 - Purple for non–UI threads in your application (including the render/compositor thread and any background threads your application creates, directly or indirectly).

- Grey for system threads. These are not directly within your control, but it's useful to know if there happens to be heavy system activity at a given time, because this might skew the results for your application. Also, of course, the system might actually be doing work on behalf of your application, in addition to unrelated work, so you can sometimes tune system CPU usage indirectly.
- White for idle percentage. A high idle percentage means that the CPU is not busy, so this should translate into a higher degree of responsiveness in your application.
- **Memory Usage MB** This is the amount of memory being used by your application. It is the application's working set, and does not account for memory used by the system on behalf of your application, nor does it account for GPU memory.
- **Storyboards** This band displays an "S" flag on the timeline whenever a storyboard event occurs. This typically corresponds to the start of an animation, including page transition animations. There are two different storyboard flags:
 - A red flag indicates that the storyboard is CPU-bound; in other words, it is running on the CPU, which you should generally try to avoid, if possible.
 - A purple flag indicates a storyboard that is not CPU-bound; in other words, it is running on the GPU, using the render thread. As much as possible, you want your storyboards to not be CPU-bound.
- **Image Loads** This displays an "I" flag on the timeline whenever your application loads image assets into memory.
- **GC Events** A "G" flag appears on the timeline whenever the system's memory garbage collector (GC) performs a garbage collection. A flag is raised whenever you explicitly call `GC.Collect` in your application and whenever the system performs a collection in response to memory pressure.

From the display in Figure 20-9, you can see that during this sample pass, CPU was at 100 percent for the first 3 seconds. This is when the application was first launched, so this is expected. This period also included an image load (for the background image on the main page) and a garbage collection. The GC is often invoked just after launch, because launching an application takes a lot of work and uses a lot of types that will no longer be required thereafter.

Between 3 seconds and 5.5 seconds on the timeline, nothing much happened. It typically takes the user a second or two to realize that the application has fully loaded, before she starts interacting with it. At 5.6 seconds, there's a CPU-bound storyboard, another GC collection, a spike in CPU usage, memory consumption and frame rate, and then a whole series of image loads. This corresponds with the user choosing an item from the menu on the main page, which triggers a navigation to the *CategoryPage* for that category. This in turn initializes the *ListBox* on the page and loads the list items, including their image files. The sequence continues in Figure 20-10.

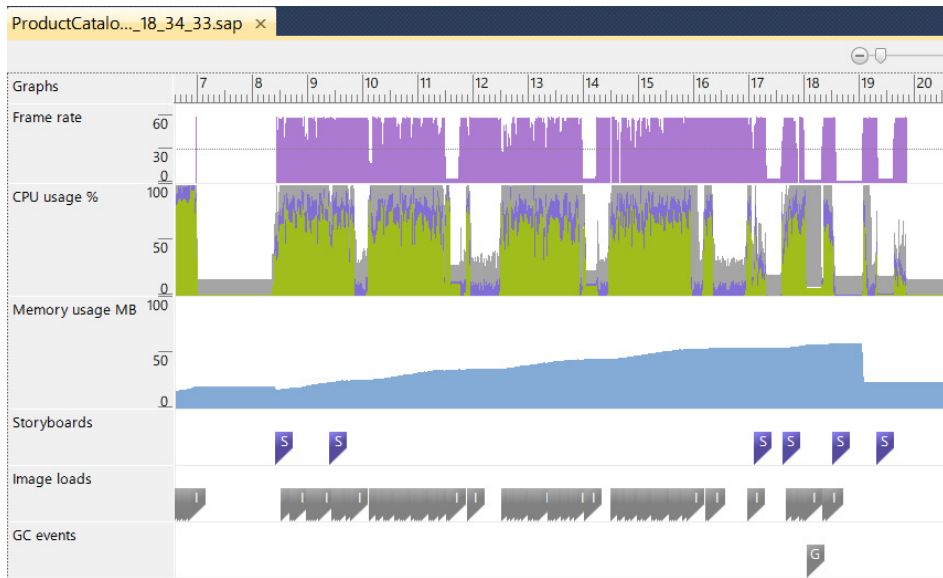


FIGURE 20-10 This profile shows high frame rate choppiness and increasing memory consumption.

Several interesting things are happening from the 8.5-second mark onward. The frame rate shows a lot of choppiness, oscillating between very low rates around 5 FPS and the target 60 FPS. This period corresponds with high CPU usage, and steadily increasing memory consumption. There are also many image load events during this time. All of this can be explained by the user scrolling through the *ListBox* on the *CategoryPage*. There are many items in the list, each with an image that needs to be loaded, and the more items are loaded, the more memory is consumed. The frame rate is unacceptably low in several places, so if your application exhibits behavior like this, you would definitely want to fix it before publication. Figure 20-11 shows that this choppiness becomes worse.

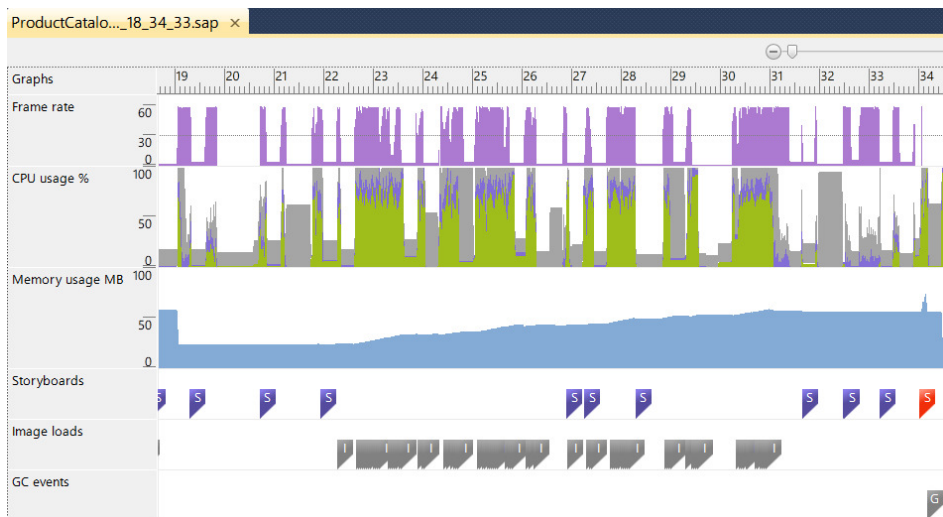


FIGURE 20-11 The choppy frame rate worsens over time.

It's interesting that the memory consumption drops suddenly at about the 19-second mark, along with the CPU usage. Also at this time, the frame rate band shows that nothing is being drawn. This probably corresponds to a point at which the user stopped scrolling for a second or two. Then, she started scrolling again, and the previous pattern of increasing frame rate choppiness and increasing memory consumption is repeated. The repeated non-CPU-bound storyboards indicate points at which the user changed scrolling direction. The CPU-bound storyboard at the 34-second mark indicates the start of another page transition, which is when the user selected an item from the list, and navigated to the *DetailsPage*.

Figure 20-12 shows another profile pass, after making some optimization changes to the code.

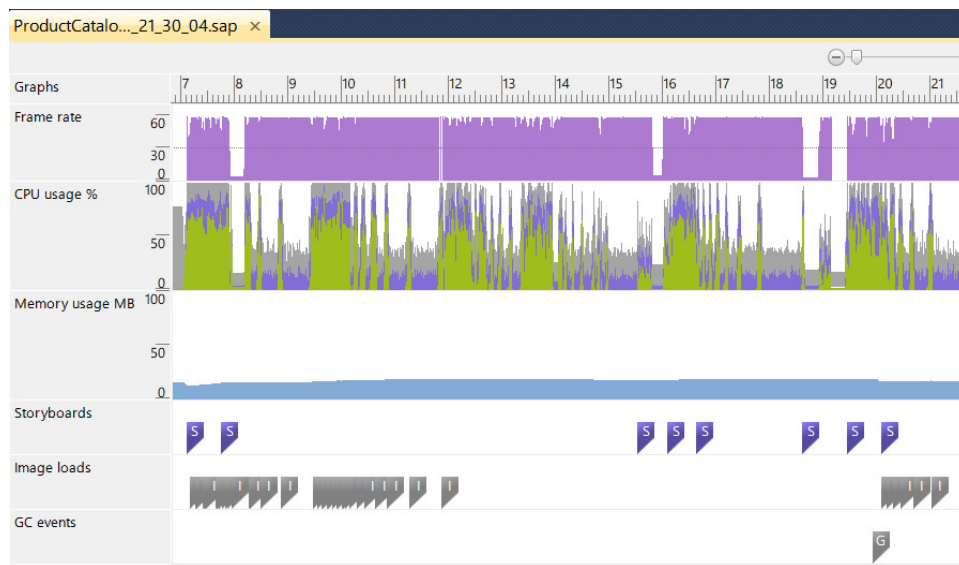


FIGURE 20-12 Profile analysis showing improved performance.

In this pass, the *CategoryPage* is loaded at the 7-second mark, and then the user starts scrolling. Scrolling in this version is significantly less choppy, and the frame rate is much more consistently high, even when the user changes scrolling direction. Also note that memory consumption remains much lower than before.

The changes in the code are as follows:

- Set the build action for each image file to *Content* rather than *Resource*. This speeds up startup time.
- Set *CacheMode* on each *Image* control to *BitmapCache*, which allows the system to cache the texture and use the render thread/GPU rather than the UI thread/CPU.
- The first version loads all of the data for all products for all categories when the viewmodel is constructed at startup, even though the user might never actually visit some (or any) of the categories. The optimized version defers loading of any category data until the user selects a category to which to navigate, and then only loads the data for that category. This improves

startup time, reducing it from 3 seconds to 2 seconds. This could make all the difference between passing and failing certification: recall that you're only allowed 5 seconds to startup. The downside is that if and when the user does navigate to a category, she pays a performance penalty at that point; however, this is a less critical point than startup, so this technique is normally a good trade-off.

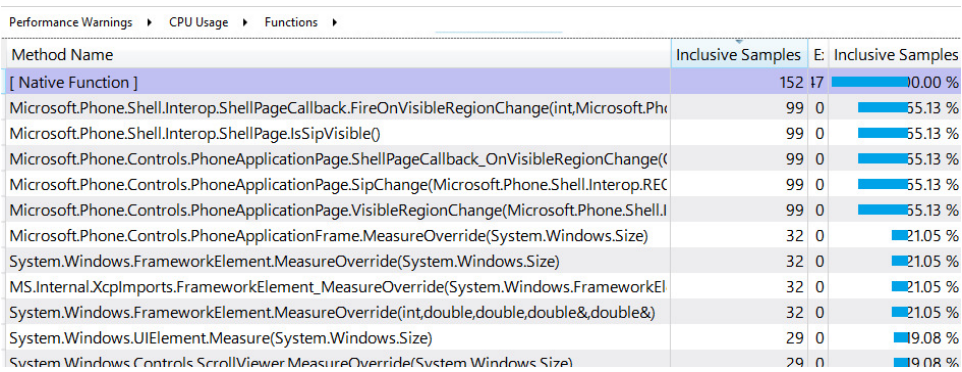
- Provide two sizes of the image: a thumbnail size, used in the *CategoryPage* list, and a full size, used in the *DetailsPage*. Also size these exactly to the size of the *Image* control in each page. This solves two issues. First, there is no longer any scaling required, because the image assets are the same size as the *Image* controls. Second, in the *CategoryPage*, where many images are loaded, only the small thumbnails are used. This saves considerable memory, and the larger images are only loaded if and when the user actually navigates to a *DetailsPage*—and then only that one large image is loaded, not all of them.

In addition to zooming in to the graph, you can also select any arbitrary portion of the timeline to see a detailed analysis section. This section aims to identify any recognized patterns of resource usage (frame rate, CPU, and memory consumption) and provides indications as to how to investigate further. An example is shown in Figure 20-13, in which the time 5.5 seconds to 8 seconds is selected.



FIGURE 20-13 You can select a portion of the graph to investigate more closely.

In this example, the Profiler has identified that the UI thread is doing an above-normal amount of work. The tool indicates that you can use the Performance Warnings menu at the bottom of the graph to inspect the threads more closely to try to determine the reason for this behavior, as shown in Figure 20-14.



Performance Warnings > CPU Usage > Functions			
Method Name	Inclusive Samples	E: Inclusive Samples	
[Native Function]	152	17	10.00 %
Microsoft.Phone.Shell.Interop.ShellPageCallback.FireOnVisibleRegionChange(int,Microsoft.Pho	99	0	55.13 %
Microsoft.Phone.Shell.Interop.ShellPage.IsSipVisible()	99	0	55.13 %
Microsoft.Phone.Controls.PhoneApplicationPage.ShellPageCallback_OnVisibleRegionChange(C	99	0	55.13 %
Microsoft.Phone.Controls.PhoneApplicationPage.SipChange(Microsoft.Phone.Shell.Interop.REC	99	0	55.13 %
Microsoft.Phone.Controls.PhoneApplicationPage.VisibleRegionChange(Microsoft.Phone.Shell.I	99	0	55.13 %
Microsoft.Phone.Controls.PhoneApplicationFrame.MeasureOverride(System.Windows.Size)	32	0	21.05 %
System.Windows.FrameworkElement.MeasureOverride(System.Windows.Size)	32	0	21.05 %
MS.Internal.XcpImports.FrameworkElement.MeasureOverride(System.Windows.FrameworkEl	32	0	21.05 %
System.Windows.FrameworkElement.MeasureOverride(int,double,double,double&,double&)	32	0	21.05 %
System.Windows.UIElement.Measure(System.Windows.Size)	29	0	19.08 %
System.Windows.Controls.ScrollViewer.MeasureOverride(System.Windows.Size)	29	0	19.08 %

FIGURE 20-14 You can drill down into the Performance Warnings details to isolate problems.

Drilling down into the Functions list under the CPU Usage option, this example shows a series of calls to various UI element objects, notably *VisibleRegionChange* events, and *MeasureOverride* calls. This ties up nicely with what you’d expect to see at this point, because this is actually the point at which the user tapped one of the *MainPage* menu options to navigate to the *CategoryPage*, triggering page transitions and size calculations for the incoming page’s visual tree. In other words, in this particular case, there is no issue, and performance is within expected parameters. The function report has columns for the total number and percentage of both inclusive and exclusive samples. Inclusive samples are the samples collected during the execution of the indicated method, including any child methods called from this method. Exclusive samples are scoped strictly to the current method; they exclude child methods.

To drill down on the frame-rate choppiness that you observed earlier, you would go to the Performance Warnings menu, and then select the Frames view. Next, sort the frames by clicking the CPU Time column header, and then select the frame(s) with the highest CPU time. From the Frames view, select the Visual Tree option to identify the specific UI elements that are consuming the most CPU time. This should indicate whether you have a particularly expensive element, in terms of template expansion or visual tree changes.

The Performance Warnings menu offers many ways of looking at the profile data: you can focus on CPU usage, memory usage, or frame rate. You can drill down into threads or functions. You can examine the function call tree at any point, and wherever the Profiler encounters one of the methods in your code, it provides a link in the report so that you can quickly jump to the code if you need to, as shown in Figure 20-15.

Performance Warnings ▶ CPU Usage ▶ Functions Call Tree

Method Name	Module Name
MS.Internal.FrameworkCallbacks.CreateKnownObject(int,UInt32,String,int&,UI	System.Windows.dll
MS.Internal.TypeProxy.CreateInstance(UInt32)	System.Windows.dll
<>c__DisplayClass30.<GetCreateObjectDelegate>b__2a()	System.Windows.dll
System.Reflection.ConstructorInfo.Invoke(Object[])	mscorlib.dll
System.Reflection.RuntimeConstructorInfo.InternalInvoke(Object,System	mscorlib.dll
[Native Function]	[Native Modules]
ProductCatalog.MainViewModel.get_LoremIpsums()	ProductCatalog.dll
ProductCatalog.MainViewModel.LoadInPace()	ProductCatalog.dll
System.Uri..ctor(String,System.UriKind)	System.dll
[Native Function]	[Native Modules]
System.Windows.Application.LoadComponent(Object System.Uri)	System.Windows.dll

FIGURE 20-15 You can jump from the function call tree to your own code by clicking the provided link.

It's also instructive to perform a memory analysis pass; the Profiler will often suggest that you do this when it finds a suspicious memory usage pattern. This is obviously useful if you see a pattern of memory consumption that steadily increases and never seems to drop, which is often an indication of a memory leak. A memory pass is very intrusive, so you should expect very poor performance and responsiveness while this is active. The option to perform a memory pass is in the initial Profiler dialog. The simplest memory analysis drill-down report gives you a table of allocations and GC collections for the selected period, as shown in Figure 20-16.

Performance Warnings ▶ Heap Summary ▶

Category	Instances	Total Size (KB)
Retained Allocations at Start	9794	433.422
New Allocations	19570	10595.297
Collected Allocations	14015	10358.016
Retained Allocations at End	15349	670.703
Retained Silverlight Visuals at Start	128	1010.316
Retained Silverlight Visuals at End	295	23548.598

FIGURE 20-16 Perform a memory analysis pass to look for usage patterns and leaks.

The memory analysis offers fewer options for examining the data, but one aspect you should generally look at is a table showing the types allocated, as shown in Figure 20-17. To get this report, go to the Performance Warnings menu, select Heap Summary, and then click Types. The most important column here is for total allocated size; this is the product of the type size and the number of instances of that type that were allocated. It is common, as in this example, to find that strings take up the highest volume of allocations per type.

Performance Warnings ▸ Heap Summary ▸ Types ▸						
Type Name	Instances	Total Size (Bytes)	Max Size (Bytes)	Avg Size (Bytes)	Total Allocated Size %	Allocating Module
System.String	502	50868	12788	101	3.05 %	mscorlib.dll
System.String	625	39548	4244	63	7.92 %	N/A
System.Char[]	133	26708	8208	200	12.10 %	mscorlib.dll
System.RuntimeType	393	9432	24	24	4.27 %	System.Windows.dll
System.Byte[]	19	7444	4108	391	3.37 %	mscorlib.dll
System.Collections.Generic.Dictionary	1	6908	6908	6908	3.13 %	mscorlib.dll
System.Int32[]	66	6528	1736	98	2.96 %	mscorlib.dll
System.Collections.Generic.Dictionary	6	5672	3164	945	2.57 %	mscorlib.dll
System.Collections.Hashtable+bucket[]	19	4944	1296	260	2.24 %	mscorlib.dll
System.String[]	124	2852	96	23	1.29 %	mscorlib.dll
System.Windows.CoreDependencyPrc	113	2712	24	24	1.23 %	System.Windows.dll
System.Reflection.MethodInfo[]	28	2176	284	77	0.99 %	mscorlib.dll

FIGURE 20-17 Examine the memory analysis for a breakdown by type.

You can execute a profiler pass across FAS, but not across tombstoning. If your application falls off the backstack during a Profiler pass, the pass is terminated. In the FAS case, during the time when your application is in a dormant state, you should expect to see zero CPU usage assigned to any of your application's threads, including the UI and render/compositor thread. There should also be zero frames rendered, zero images loaded, and zero storyboard events. However, you would expect to see system CPU usage, because the system is of course doing other things while your application is dormant. You would also expect to see that your application's memory consumption remains flat throughout the dormant period; the system doesn't take away your memory while you're dormant, but it doesn't allow you to allocate any more until reactivation.

UserVoice Forums

UserVoice is a company based in San Francisco that provides hosted feedback forums. Microsoft offers two main UserVoice feedback forums for Windows Phone: one for users, and one for developers. The developer forum is at <http://wpdev.uservoice.com/forums/110705-app-platform>. This allows application developers to provide feedback that goes straight to the teams that design and build the various pieces of the phone platform. Using this forum, anyone can make suggestions for improvements to the platform, file bugs, and read and comment on other people's feedback.

There's also a Windows Phone application that provides a phone client to the UserVoice forum, which is available at <http://www.windowsphone.com/en-US/apps/b5466109-2b8d-46f4-9461-c959e433ae4a>. This has a pivot-based UI, as shown in Figure 20-18, in which you can choose the specific forum that you want to use, and then the category of feedback. You can add your own feedback comments or vote on other people's comments, either anonymously or by logging in with a valid email address.



FIGURE 20-18 You can use the WPDev application to send feedback to Microsoft.

You are strongly encouraged to use the feedback mechanisms to make your voice heard. The Windows Phone development teams at Microsoft are very keen to accept feedback on where to improve the product, what additional features developers need, issues with the APIs, the tools or the documentation, and so on. Please read the terms of service available on the site.

Portable Library Tools

The version of Silverlight used in the Windows Phone platform is slightly different from the desktop version of Silverlight, and both of these runtimes are significantly different from the full Microsoft .NET common language runtime (CLR) and that of the runtime on Xbox consoles. There is an increasing trend for developers to build applications that run across multiple platforms. This is challenging because of the differences in the runtimes. This is where the Portable Class Library project comes in. This is an add-in to Visual Studio that offers a new class library project type. This project type allows you to specify multiple platforms to target. It also uses a special set of runtime assemblies that contain a subset of features known to be common across the selected platforms.

You can download the Portable Class Library project from the Visual Studio gallery, either through the Extension Manager in Visual Studio, or directly from this MSDN link at <http://visualstudiogallery.msdn.microsoft.com/b0e0b5e9-e138-410b-ad10-00cb3caf4981/>. Note that this works only with Visual Studio Professional and above. When installed, this adds the new project type to the root nodes for Visual C# and Visual Basic in the New Project dialog, as shown in Figure 20-19.

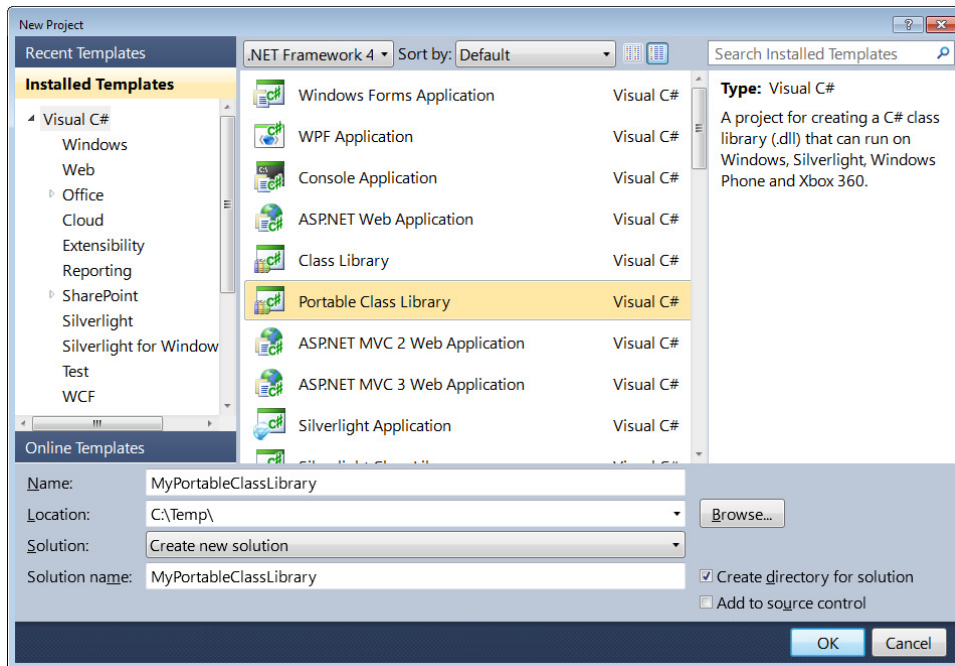


FIGURE 20-19 The Portable Class Library project type is available for both C# and Visual Basic.

The subset assemblies are installed by default to a standard location for reference assemblies such as %ProgramFiles%\Reference Assemblies\Microsoft\Framework\NETPortable\v4.0\. Within this folder, there are subfolders for each of the supported permutations of target runtimes. The default permutation is .NET Framework 4, Silverlight 4, and Windows Phone 7. You can change this on the Library tab of the project properties dialog, as shown in Figure 20-20. You can use a portable library in Windows Phone 7.0, 7.1, or 7.1.1.

If you change the target runtimes, the corresponding runtime assembly references are updated, and the project is closed and re-opened. If you had added other assembly references, some of these might no longer be valid in the new permutation; these will be indicated with an exclamation point icon in the references list in Solution Explorer.

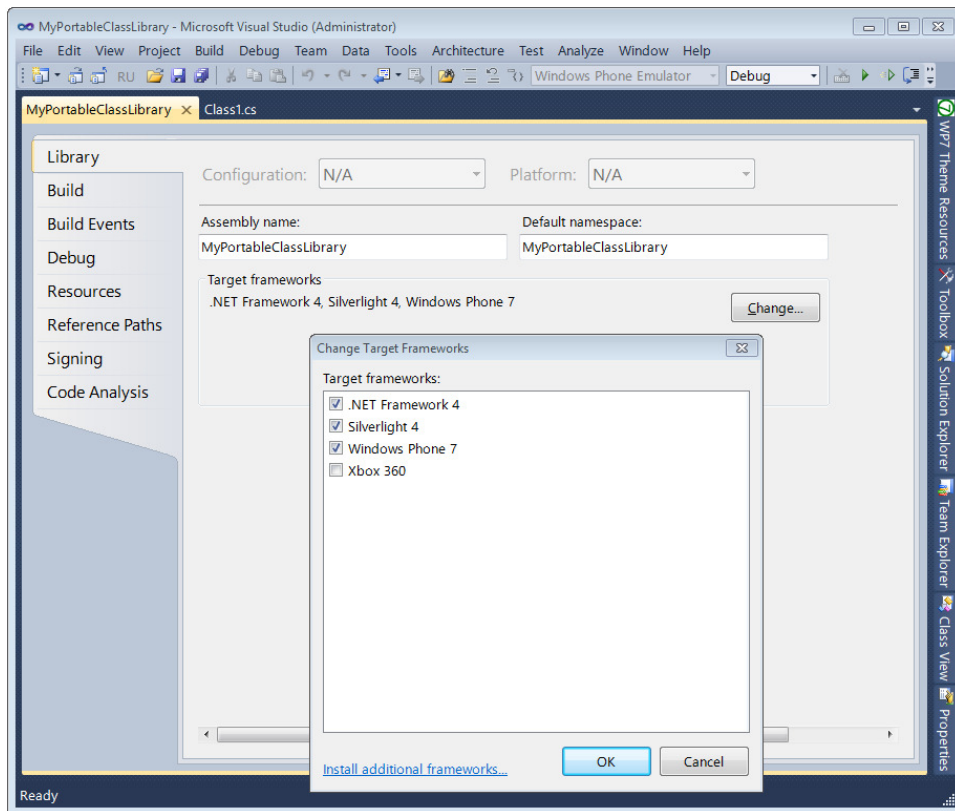


FIGURE 20-20 You can change the target runtimes in the project properties dialog.

Figure 20-21 shows the *SilverlightProductCatalog* application in the sample code. This is a regular desktop/browser Silverlight application, which is designed to mirror the behavior of the *Product Catalog* Windows Phone application.



FIGURE 20-21 The desktop Silverlight version of the *ProductCatalog* application.

Both the Silverlight version and the Windows Phone application share a common library. There's obviously a lot of commonality between desktop Silverlight and Silverlight for Windows Phone, so if you only need to share libraries across these two platforms, you can often do this by just using a simple Windows Phone Class Library project. In this example, this approach gives you the freedom to share not only code, but also data. The first screenshot in Figure 20-22 shows this approach, where the *LibraryViewModel* shared library contains all the image data files as well as the model and view-model classes.

Contrast this with the second screenshot, in which the *PortableViewModel* shared library contains only shared code, and each application project has its own independent copy of the *Images* data. The second version of the solution uses the Portable Class Library project, and it is a limitation of this project type that you cannot use resources with the build action set to *Resource*, only resources set to *Embedded Resource*. This makes it more challenging to load the data resources into the viewmodel for data binding. However, this is a minor limitation, and it is less important if you also need to target the full .NET CLR and/or Xbox platforms. The Portable Class Library project gives you a good platform-agnostic starting point for a shareable library. If you build a lot of applications, it makes sense to invest the time in building a core set of platform-agnostic libraries, on top of which you can layer a set of platform-specific but application-agnostic libraries.

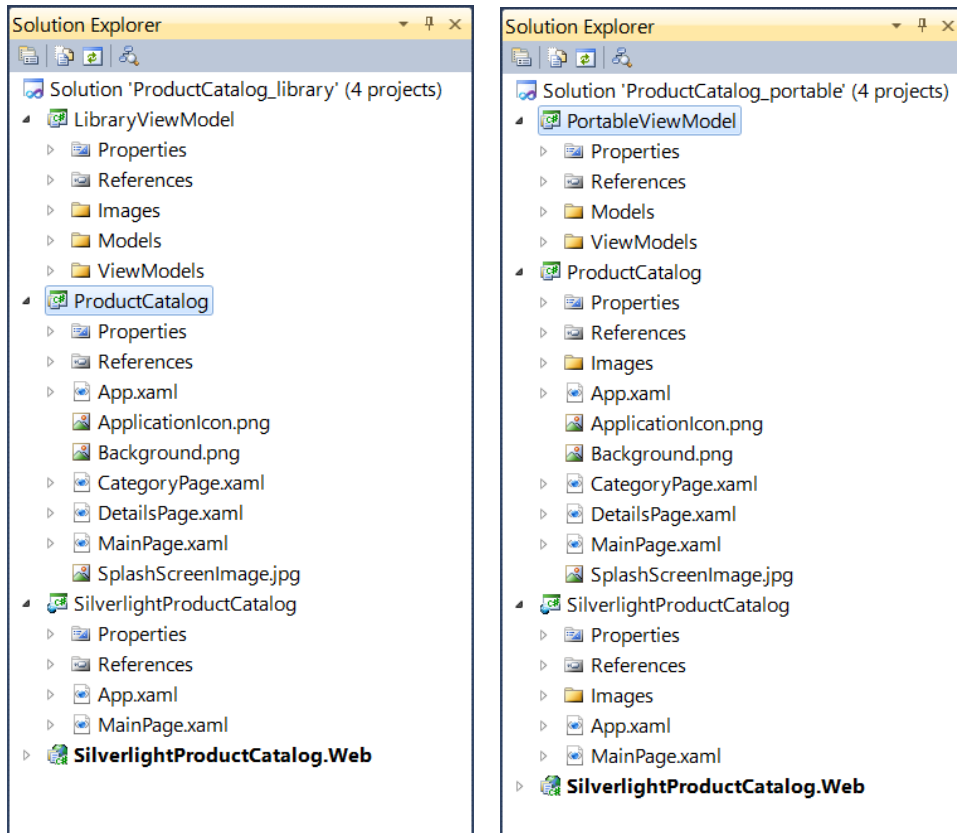


FIGURE 20-22 You can share libraries across Silverlight and Windows Phone projects.

Async Framework

Chapter 14, “Go to Market,” discussed your choices for executing multiple threads and the reasons why you should offload as much work as possible from the UI thread, in particular. Any operation that might take a noticeable amount of time should be offloaded so that you do not make the UI unresponsive. In Chapter 11, “Web and Cloud,” you saw how all web APIs—such as *WebClient* and *HttpWebRequest*—are asynchronous. The Windows Phone platform enforces asynchronous web operations precisely because such operations do take a noticeable amount of time, and must not be allowed to run synchronously.

In fact, these APIs use what is known as the Event-based Asynchronous Pattern (EAP). In this pattern, you invoke a method, named *<OperationName>Async* which returns void, and which initiates an asynchronous operation. Prior to invoking this method, you hook up a handler for the event that will be raised when the operation completes. In some cases, you have the option to hook up other events such as for ongoing progress reports or error conditions.

To make it easier to work with asynchronous operations, Microsoft has released an early version of the Visual Studio Async Framework. This introduces the Task-based Asynchronous Pattern (TAP), which is a lot cleaner than EAP. It provides additional language (C# and Visual Basic) compiler support with which you can write asynchronous operations in a more elegant way, with less code. This is available for download from <http://www.microsoft.com/download/en/details.aspx?displaylang=en&id=9983>.

Consider the *AvatarWebClient_EAP* solution in the sample code. This is a variation of the sample in Chapter 11: it uses the traditional EAP approach to fetch avatar images from Xbox Live. The sequence of operations is as follows: instantiate a *WebClient* object, hook up the *OpenReadCompleted* event, invoke the *OpenReadAsync* method, eventually retrieve the data in the *OpenReadCompletedEventArgs* passed into the event handler method, and finally, process the returned data.

```
private void getAvatarButton_Click(object sender, RoutedEventArgs e)
{
    GetAvatarImage(
        new Uri(String.Format(
            "http://avatar.xboxlive.com/avatar/{0}/avatar-body.png",
            gamerTagText.Text)));
}

private void GetAvatarImage(Uri avatarUri)
{
    WebClient webClient = new WebClient();
    webClient.OpenReadCompleted +=
        new OpenReadCompletedEventHandler(webClient_OpenReadCompleted);
    webClient.OpenReadAsync(avatarUri);
}

private void webClient_OpenReadCompleted(object sender, OpenReadCompletedEventArgs e)
{
    if (!e.Cancelled && e.Error == null)
    {
        BitmapImage bmp = new BitmapImage();
        bmp.SetSource(e.Result);
        avatarImages.Add(bmp);
    }
}
```

This all works just fine, and once you're used to EAP coding, it's not especially complicated, nor especially troublesome to write. However, the TAP model is certainly a lot cleaner; you don't have to hook up any events, and you don't have to write any event handlers. This version of the solution (*AvatarWebClient_TAP* in the sample code) requires a reference to the *AsyncCtpLibrary_Phone.dll*, which is part of the Async Framework. The application is shown in Figure 20-23.



FIGURE 20-23 Fetching avatar images by using the Async Framework.

With the TAP model, you use the *await* keyword when you invoke an asynchronous method, and this method returns a *Task* or *Task<T>* object. You can only use the *await* keyword inside a method or lambda which is declared with the *async* keyword. It is recommended practice (although not required) to name such a method with “Async” appended. So, in the async version, the sequence of operations is as follows: instantiate a *WebClient* object, invoke and await the *OpenReadTaskAsync* method, and then process the returned data. In this example, this reduces two methods and five lines of code down to just one line of code.

```
private void getAvatarButton_Click(object sender, RoutedEventArgs e)
{
    GetAvatarImageAsync(
        new Uri(String.Format(
            "http://avatar.xboxlive.com/avatar/{0}/avatar-body.png",
            gamerTagText.Text)));
}

private async void GetAvatarImageAsync(Uri avatarUri)
{
    WebClient webClient = new WebClient();
    var data = await webClient.OpenReadTaskAsync(avatarUri);

    BitmapImage bmp = new BitmapImage();
    bmp.SetSource(data);
    avatarImages.Add(bmp);
}
```

Under the covers, the *OpenReadTaskAsync* method is an extension method on the *WebClient* class. Internally, this method is implemented by using a more or less traditional EAP model, hooking up an event handler for the *OpenReadCompleted* event on the *WebClient* object. This is really the key to the Async Framework: it doesn't do anything you couldn't do yourself manually, it's just that it does a lot of the work for you, enlisting the compiler's support to make your coding simpler. If you think about it, this is the same model as the language extensions in C# 3.0—especially lambda expressions and LINQ. These didn't provide anything extra that you couldn't do before, they just provided new keywords and language constructs that made things a lot easier, cleaner, less error-prone, and more maintainable. Note that the Async Framework is in unsupported early release form; it's very instructive to experiment with it, but all the usual caveats apply with regard to using unsupported tools for production code.

Summary

This chapter explored the improvements in developer tooling in Windows Phone 7.1. The emulator provides a set of new features, including support for sensor and location-based applications that would otherwise be difficult to test. The debugger improvements are less obvious but now cover all the new version 7.1 features such as fast application switching and background agents. The new marketplace test kit helps you finalize your application prior to submitting it for publication, in terms of gathering the required image files, and also in terms of formalizing the final testing phases and aligning your tests with marketplace certification requirements. The most powerful new feature is undoubtedly the Profiler, with which you can track your application's use of CPU and memory resources, including breakdowns of thread activity, resource and animation events, memory allocations, and so on. This gives you considerable scope for identifying anomalous behavior, especially performance issues, and comparing the results after applying fixes. Finally, the Portable Class Library project simplifies the process of building cross-platform applications, and the Async Framework gives you a taste of the direction things are going in the next version of Visual Studio and the language compilers.

Index

Symbols

<wp:Param/> element, 640

A

- absolute layout type, 46
- AccelerationHelper class (Level Starter Kit), 311
 - Averaging, 311
 - Low-Pass Filtering, 311
 - Optimal Filtering, 311
 - usage, 311–313
- Accelerometer class, 305
 - accelerometer, 592–594
 - AccelerometerReading event, 305
 - CurrentValueChanged event vs. ReadingChanged event, 593
 - MangoAccelerometer solution, 592
 - Microsoft.Devices.Sensors namespace, 305
 - ReadingChanged event, 306
 - TimeBetweenUpdates property, 593
- accelerometer (physical), 305–317
 - AccelerationHelper class (Level Starter Kit), 311
 - Accelerometer class, 305, 592–594
 - accuracy, issues affecting, 305
 - and screen lock, 321
 - enhanced in Phone 7.1, 592–594
 - FilteredAccelerometer application, 308
 - Level Starter Kit, 311–315
 - MangoAccelerometer solution, 592
 - OrientationHelper class (Level Starter Kit), 311
 - power usage by, 321
 - Reactive Extensions for .NET (Rx .NET) and, 308–310
 - shake, 315–317
 - TestAccelerometer application, 306
 - TestShake solution, 316
 - update interval for, 593
- AccelerometerReading event, 305
- accent color
 - PhoneAccentBrush resource, 27
 - PhoneAccentColor resource, 27
 - Silverlight/.NET/HTML colors vs., 27
 - ThemeAccent sample, 27
- Activated event and tombstoned applications, 187
- ActivatedEventArgs, 556
- ActiveX controls, 358
- AdControl
 - advertisements, adding to application with, 540
 - background behavior of, 541
 - behavior of, UX, 542
 - Capabilities Detection tool, capability testing with, 544
 - capability requirements for, 544
 - DataBoundAppWithAds solution, 543
 - data service requirements with, 542
 - Marketplace TestKit, capability testing with, 544
 - memory requirements for, 542
 - Microsoft pubCenter, registering with, 540
- AdControls
 - Visibility property, setting dependent on Trial mode status, 545
- Add Service Reference Wizard, 650
- AdventureWorksLT2008R2Model, 650
- advertisements
 - Microsoft pubCenter, registering with, 540
 - UI layout considerations for, 542
- advertising
 - AdControl, 540
 - DataBoundAppWithAds solution, 543
 - DataBoundAppWithAds_TrialMode solution, 544
 - filtering, 541
 - incorporating in application, procedure for, 542
 - reporting for, 542
- advertising in applications, 540–544

AesManaged object (data encryption)

- AesManaged object (data encryption), 460–461
- Alarm object
 - IntervalTraining application, 560
 - ScheduledActionService in, 560
- alarms
 - accuracy of, 562
 - Alarms application, 560
 - anatomy of, 560
 - constraints on, 560
 - IntervalTraining application, 560
 - re-boot survival of, 566
- Alarms application, 560
- alerts
 - alarms, 560–562
 - reminders, 563–566
- AllowReadStreamBuffering property (HttpWebRequest class), 329
- analytics engines
 - Comscore, 546
 - Google, 546
 - Webtrends, 546
- Android phones, CPNS push notifications for, 445
- animation(s)
 - Background Image, 79
 - as code resources, 57
 - Content, 79
 - PanoramaItem Header, 79
 - Panorama Title, 79
 - Pivot control, 72
 - transition animations, 12
- Anonymous User ID (ANID), 276
- AppBarAnimator solution, 176
- App class, 131
 - Application_Launching event handler, 691
- App Connect, 656, 657
- App.Exit
 - and user experience, 192
 - and XNA applications, 192
 - lack of, 192
- AppHub
 - and publishing applications, 20
 - registering as developer on, 20
 - Shake Gesture Library, 315
 - source for, 18
- AppHub portal
 - adding information for testers to submission in, 528
 - canceling a submission with, 529
 - constraints on artwork for, 527
 - location of, 523
 - pricing/availability, setting, 528
 - Reports tab, location of, 533
 - testing/validation after submission with, 529
 - uploading artwork to, 527
 - using, 525–534
- App Instant Answer, 656, 657
- Apple Push Notification Service (APNS), 443
- Application Bar
 - AppBarAnimator solution, 176
 - AppBar class, 175
 - AppBarIconButton class, 175
 - AppBarMenuItem class, 175
 - buttons, limit on number of, 176
 - icons provided by Microsoft, 176
 - icons provided by third parties, 176
 - menu item support in, 178
 - opacity settings, 176
 - visual tree and, 180
- AppBar class, 38, 74, 175
 - constraints on modification, 38
 - enhancements to in Phone 7.1, 725–728
 - modifying in application, 38
 - Panorama, 79
 - properties of, 38
- AppBarIconButton, 637
- AppBarIconButton class, 175
- AppBarMenuItem class, 175
- AppBarMode enumeration
 - Mode property of, 536–537
- AppBar property (PhoneApplicationPage class), 176
- Application Bar (Touch UI), 175–180
- Application class, 40
 - ResourceDictionary, 59
 - RootVisual property, 40
- ApplicationCurrentMemoryUsage property (DeviceExtendedProperties class), 280
- application icons
 - ApplicationIcon.png application image, 30
 - current theme and, 32
- ApplicationIdCredentialsProvider, data binding with, 377
- ApplicationIdleDetectionMode setting
 - battery consumption and, 195
 - screen locking and, 194
- application image(s)
 - ApplicationIcon.png, 30
 - Background.png, 30
 - Metro guidelines and, 31
 - setting, 30

- SplashScreenImage.jpg, 30
- standard, 30–32
- WAppManifest.xml, defined in, 30
- Application_Launching event handler (App class), 691
- Application Model, 181–204
 - App.Exit, lack of, 192
 - closing applications, 186–187
 - closing vs. deactivating applications, 182
 - deactivation (non-tombstone case), 189–190
 - deactivation (tombstone case), 187–189
 - exceptions, unhandled, 191
 - launchers/choosers, 195
 - lifetime events, 181–196
 - obscuring/unobscuring applications, 192–195
 - resource constraints and, 181
 - user expectations, 195–196
- ApplicationPeakMemoryUsage property (DeviceExtendedProperties class), 280
- application(s)
 - adding audio files during development, 585
 - constraints on modifying Application Bar, 38
 - entry point, Panorama, 77
 - Exit method, lack of, 192
 - foreground vs. non-foreground state, 181
 - generic background agents (GBAs), relationship with, 572
 - object tree, 58
 - standard application UI model, 38
 - SupportedOrientations, 170–180
 - third-party access to data from, 495
 - username/password availability to third party, 496
 - using the Application Bar, 38
- applications
 - checking for updates on the marketplace, 530–532
 - MarketplaceDetailTask Launcher, updates downloaded with, 532
 - trial mode, 544–546
 - Zune marketplace atom feed, 531
- Appointments class
 - emulator, inability to test on, 701
 - retrieving data from, 701
 - SimpleCalendar application, 700
- Apps
 - anatomy of, 21–32
 - constraints on, in Metro, 14
 - inter-process communication between, 15
 - marketplace manifest, extracting information from, 276
 - native code and, 15
 - required behaviors of, 17
 - WAppManifest.xml file, 22
- architecture and design, 11–36
 - application anatomy, 21–32
 - design goals, 11–12
 - developer guidelines, 15
 - developer tools, 18–20
 - Metro, 12–15
 - Microsoft User Experience Design Guidelines, 11
 - project types, 23–25
 - Silverlight vs. XNA, 17–18
 - standard images, 30–32
 - themes/accent colors, 25–29
 - vision for, 11–15
 - of Windows Phone 7, 16
 - Windows Phone XAP, 22–23
- ARDisplay control, 616
- ArrangeOverride method, 51
- ASP.NET, 481
- assemblies, navigating between, 216–217
- AssemblyCatalog (MEF), 547
- AssemblyInfo.cs, 679
- AssemblyVersion attribute (AssemblyInfo.cs)
 - schema version numbers related to, 679
- associations
 - Association (artifact), 681
 - EntityRef (artifact), 681, 682
 - EntitySet (artifact), 681, 683
 - schema, between, 681
- AsyncCallback, 644
- Asynch framework, 771–774
 - AvatarWebClient_EAP solution, 772
 - AvatarWebClient_TAP solution, 772
 - Event-based Asynchronous Pattern (EAP) and, 771–772
 - lambda expressions and, 774
 - LINQ and, 774
 - Task-based Asynchronous Pattern (TAP) and, 772–774
 - Visual Studio Async Framework, 772
- asynchronous data, 141
- attachable properties model, 67
- attached properties, 64–67
- audio
 - DynamicSoundEffectInstance class, 339–343
 - FM tuner, 349–351
 - input/manipulation of, 328–343

audio, *continued*

 SoundEffectInstance class, 329–331

 XNA Microphone class, 335

 XNA SoundEffect class, 329–331

AudioPlayerAgent vs. BackgroundAudioPlayer, 582

audio/video APIs, 324

audio/video hardware, 323–324

augmented reality (AR) application, 612–617

 DirectionalViewfinder solution, 612

 Geo Augmented Reality Toolkit (GART), 616–617

 TestGart solution, 616

authentication types (web), 472

 basic, 472

 forms, 472

 Windows, 472

Authenticode certificate

(marketplace requirement for), 451

AutoResetEvent, 645

AvatarWebClient_EAP solution, 772

AvatarWebClient solution, 355

AvatarWebClient_TAP solution, 772

Azure. *See* Windows Azure

Azure Tables and Queues proxies, 394

AzureTablesHandler proxy, 397

B

Back button (hardware), 37, 89

 backstack navigation with, 182

 OnBackKeyPress, 91

BackgroundAgentDemo solution, 572

background agents

 background audio, for, 587–589.

See also background audio agents

 constraints on, 560

 debugger behavior for applications with, 750

 generic. *See* generic background agents (GBAs)

background audio agents, 580–589

 background agent implementation, 587–589

 BackgroundAudioPlayer class, 580

 BapApp solution, 584

 constraints on memory/CPU time, 581

 defining playlist for, 587

 GBAs vs., 580

 GetNextTrack and GetPreviousTrack methods,

 modifications for, 588

 main application implementation, 585–587

 OnPlayStateChanged method,

 changes required for, 589

 OnUserAction method, modifications for, 588

 Universal Volume Control (UVC) and, 581

BackgroundAudioPlayer class, 580

 AudioPlayerAgent vs, 582

 BapApp solution, 584

 OnPlayStateChanged override, 582

 OnUserAction override, 582

 Zune media queue (ZMQ), as proxy to, 580

Background Image, 79

 CreateOptions attribute of, 731

 decoding, 731

Background.png application image, 30

BackgroundProperty, 119

backgrounds

 composite images, 80

 embedded logos, 80

 embedded text, 80

 low-contrast images, 80

 Panorama, image size in, 81

 photographic, 80

background service, debugger behavior with, 750

BackgroundTransferDemo application, 566

BackgroundTransferRequest

 Background Transfer Service API, 566

 usage of, 567–568

Background Transfer Service API, 566–570

 BackgroundTransferDemo application, 566

 BackgroundTransferRequest, 566

 BackgroundTransferService queue, 566

 constraints on, 566

 downloading files using, 566

 OnNavigatedTo event, overriding, 569

 TransferProgressChanged event, 568

 Xbox Live marketplace, similarity to, 566

 Zune marketplace, similarity to, 566

BackgroundWorker class, 501

 ProgressChanged event, 502

 ReportProgress method, 502

 ThreadPool, usage of, 504

BackgroundWorker (threading API), 500–501

 event handlers for, 502

BackKeyPress, 96

backstack

 active applications and, 182

 dead-end pages and, 199

 management in multi-page applications, 201

 number of applications in, 189

backstack management

 ClearBack_Thumbs solution, 717–719

 enhancements in Phone 7.1, 716–719

 sequence of APIs on backward navigation, 716

- BackStack property (NavigationService class), 716
- BallManipulation solution, 148
- BapApp solution, 584
- BaseIntropectionRule base class, 469
- Basic authentication
 - binding configurations for, 484
 - implementing with SSL, how to, 489–494
 - vulnerability of, 487
- battery consumption and PowerLevelChanged event, 432
- BeginAccept method, 644, 645
- behaviors, custom, 163–167
- Behavior<T> base class
 - AssociatedObject property, 164
 - use in building custom behaviors, 164
- beta testing
 - marketplace support for, 534
- BindingConverters_Color solution, 114
- BindingConverters_FontWeight solution, 114
- BindingOperations.SetBinding, 105
- BindingValidationError event, 118
- BindingValidation solution, 118
- BindToShellToast method, 433
- Bing. See also search extensibility, enhanced in Phone 7.1
- Bing maps
 - static vs. dynamic, for performance optimization, 516
- Bing Maps, 376–382
 - GeocodeService, 379
 - Geo-location and, 378
 - ImageryService, 379
 - key creation for, 376–377
 - map control, using, 376–378
 - OverheadMap control and, 617
 - RouteService, 379
 - SearchService, 379
 - SimpleBingMaps solution, 377
 - SOAP services, use of, 379–382
 - TestBingMaps solution, 378
 - TestGeocodeService application, 380
 - web services, 379
- BingMapsDirectionsTask launcher, 621
- BingMapsTask launcher, 621
- bitly, 398–400
 - API, 398
 - API return string, 399
 - hardcoding username/account key information and, 399
 - TestBitly sample code, 398

- BitmapCache mode (UI)
 - rendering phase of element, skipping with, 510
- Blend UI, 167
- BouncingBall solution, 161, 507
- BouncingStoryboard solution, 510
- branding, 80
- brushes, 57
- BufferDuration property (XNA Microphone class), 335
- BufferingChanged event (MediaElement class), 329
- BufferNeeded event (DynamicSoundEffectInstance class), 339–343
 - TestDynamicSounds_Controls solution, 345
 - TestDynamicSounds solution, 340
- BufferReady event (XNA Microphone class), 336, 332
- bug tracking with Manual Tests tab, 755
- build action
 - set to Content, 56
 - set to Resource, 56
- Button
 - TrackAction behavior, adding to, 548
- ButtonBase class, 734
- Button control, 41, 42
 - Border, 42
 - click behavior, 54
 - ContentControl, 42
 - Control class, 42
 - elements of, 42
 - Grid, 42
- Button family, 70
- button(s)
 - Back, 37
 - hardware, 37
 - internal Mouse event control, 158
 - Search, 37
 - Start, 37

C

- Cache visualization (performance), 277
- Calibrate event (Compass class), 598
- CameraCaptureTask, 300
- camera pipeline, 608–612
 - CameraCaptureTask chooser, 608
 - PhotoCamera class, 608
 - SimpleCamera solution, 608
- cancelable navigations, 713
- CanExecuteChanged event (ICommand class), 736–738
- CanExecute method (ICommand class), 736–738

- Canvas, 46, 49
 - Canvas.Left property, 49
 - Canvas.Top property, 49
- Capabilities Detection tool
 - AdControl capabilities testing and, 544
- Capability Detection tool
 - detection limits on, 753
 - marketplace test kit vs., 751
 - obfuscated code and, 752
 - reflected APIs and, 752
- CaptureImageAvailable event (PhotoCamera class), 609, 610
- certificate authorities and SSL, 488
- Certificate Export Wizard
 - exporting security certificates with, 490
 - output format of, 490
 - Personal Information Exchange PKCS #12 (PFX) file, output saves as, 490
- certificates, security
 - Certificate Export Wizard, 490
 - Common Name (CN) of, 489
 - exporting to phone from developer computer, 490–491
 - MMC Certificate Snap-in, 490
 - WP7CertInstaller project, used to export to phone, 493–494
- certification requirements, memory, 280
- chambers, heirarchy of, 455
- changeButton_Click handler, 112
- ChannelUriUpdated event, 424
- ChannelUriUpdate event, 425
- chassis requirements/specifications, 37
 - buttons, 37
 - and developers, 38
 - graphics processing unit (GPU), 37
 - keyboard, hardware (optional), 167
 - light sensor, 37
 - multi-touch input system, 144
 - multi-touch, support for, 37
 - proximity sensors, 37
 - screen configuration, 37
 - screen, physical size of, 37
- chat application, remote peer-to-peer, 642
- CheckBox (internal Mouse event control), 158
- ChildWindow, 89, 94, 95
- Choosers, 195
- ClearBack_Thumbs solution, 717–719
- Click events vs. Mouse/Manipulation events, 158
- Click handler (Pin button), 635
- Clipboard API, 729
 - Clipboard.SetText method, 729
 - TestClipboard application, 729
- CloseSocket method, 647
- Closing events, 186–187
- Cloud 2 Device Messaging Framework (C2DM)
 - Common Push Notification Service (CPNS) and, 443–447
 - Microsoft Push Network Services vs., 443
- CloudChuck solution, 389
- cloud services
 - Project Hawaii, 398
 - Windows Azure, 387–398
- CLR Security model, 454
 - Critical code attribute, 454
 - SafeCritical code attribute, 454
 - transparency layers in, 454
 - Transparent code attribute, 454
- CodeAnalysisPath (MSBuild property), 467
- code, generated
 - diagnostics, running on, 709
 - modifying, 707–710
- code resources, 57
 - animation(s), 57
 - brushes, 57
 - colors, 57
 - styles, 57
 - templates, 57
- CoffeeStore application, 681
- CoffeeWeb application, 704
- CoffeeWebApp solution, 708
- CollectionDataBinding_DTDData solution, 124
- CollectionDataBinding solution, 106
- CollectionDataBinding_xaml solution, 121
- collections
 - collection types, 87
 - CollectionViewSource class, 740–743
 - data template, 109
 - dynamic data-bound, 111
 - filter queries, 138
 - GroupDescriptions property, 740–743
 - grouping queries, 138
 - performance, effect on, 107
 - SortDiscriptions property, 740–743
 - sort queries, 138
 - TestGrouping solution, 740
- CollectionViewSource class, 138, 738
 - GroupDescriptions property, 738, 740–743
 - SortDescriptions property, 738
 - SortDiscriptions property, 740–743

- TestGrouping solution, 740
- colors as code resources, 57
- Common Language Runtime (CLR), 62
- Common Name (CN) of certificates, 489
- Common Push Notification Service (CPNS), 443–447
 - and Apple Push Notification Service (APNS), 443–447
 - and Cloud 2 Device Messaging Framework (C2DM), 443–447
 - usage, 446–447
- Compass class, 595–599
 - Calibrate event, 598
 - CurrentValueChanged event, 598
 - declination, computing, 595
 - IsDataValid property, 597
 - required references for, 596
 - SimpleCompass solution, 595
- compass (sensor API), 595–599
 - calibrating, 598
 - calibration requirements for, 592
 - testing for device support of, 596
- Compose method (MEF), 547
- composite images, 80
- CompositeTransform, 152
- CompositionHost (MEF), 547
- Comscore, 546
- configurable diagnostics, 261–265
- ConnectAsync, 648
- ConnectionSettingsTask launcher, 621
- Connection type sockets, 643
- connectivity features, enhanced in Phone 7.1, 627–666
 - OData Client, 650–656
 - push, tile, and toast notifications, 627–642
 - search extensibility, 657–665
 - sockets, 642–649
 - TCP sockets, 643–649
- ConsoleDiagnostics solution, 268
- contact info chooser, enhancements to, 621
- Contacts class
 - retrieving data from, 701
 - SimpleContacts application, 700
 - testing on emulator, 701
- Content
 - build action set to, 56
 - resources vs., 55, 56
 - Resource vs., for performance optimization, 514
- Continuation object, 655
- Control.ClearValue method, 119

- controls, 69–98
 - Button, 158
 - CheckBox, 158
 - RadioButton, 158
 - standard, 69–89
- CPU usage, as shown by Profiler, 759
- CreateDatabase method (DataContext class), 671
- cross-site restrictions, 358
- Cross-Site Scripting (XSS) protection, 500
- CryptoStream object (data encryption), 460–461
- CurrentValueChanged event (Compass class), 598
- CurrentValueChanged event (Gyroscope class), 602
- CurrentValueChanged event (Motion class), 606
 - and augmented reality applications, 615
 - Attitude property, 606
 - center-point calculations using, 607
 - RenderTransform property, 606
- CurrentValueChanged vs. ReadingChanged event (Accelerometer class), 592–593
- Custom Control(s), 49–52
 - ArrangeOverride, 51
 - MeasureOverride, 51
 - UserControl vs., 49
- CustomerDataService.svc.cs, 653
- CustomerWebApp, 650
- customization of Silverlight controls, 70
- CustomPanel solution, 50

D

- database performance
 - DbCreator_NoVersion solution, 694
 - DbCreator_WithVersion solution, 694
 - INotifyPropertyChanged interface and, 692–694
 - INotifyPropertyChanging interface, 692–694
- DatabaseSchemaUpdater class
 - DatabaseSchemaVersion property, 680
 - Microsoft.Phone.Data.Linq namespace, 680
- DatabaseSchemaVersion property (DatabaseSchemaUpdater class), 680
 - DbCreator_NoVersion solution, 694
- databases, local
 - access to, from application, 691–692
 - associations, 681–684
 - building classes for, 669–670
 - CoffeeStore application, 681
 - Content, packaging as, for performance optimization, 690
 - create/read operations on, 669–675

databases, local, *continued*

- DataContext class, 668
 - DbCreator application, 685
 - DbCreator_WithVersion solution, 694
 - development model of, 668
 - downloading large databases, 692
 - encryption, 695–697
 - error-throwing events, 692
 - Isolated Storage Explorer tool, 684–692
 - LINQ-to-SQL and, 667–697
 - Model-View ViewModel (MVVM) use with, 669–710
 - performance considerations with, 692–695
 - performance optimizations for, 673–674
 - querying files in SMSS/Visual Studio, 688–690
 - read-only access, 692
 - restricted access to, in code, 668
 - schema updates, 677–681
 - ShoppingList_CR solution, 669
 - ShoppingList_CRUD_Encrypted solution, 695
 - ShoppingList_CRUD solution, 675
 - SQL-CE, use of, 684
 - SQL Server Management Studio (SSMS), 667
 - System.Data.Linq.Mapping namespace, 669
 - update/delete operations, 675–677
- data binding
- ApplicationIdCredentialsProvider and, 377
 - BindingOperations.SetBinding, 105
 - {Binding} specifiers, 105
 - {Binding} syntax, 103
 - collections, 106–113, 740–743
 - CollectionViewSource class, 738, 740–743
 - DataServiceCollection<T> class, 738
 - data template, 109
 - dependency properties, 62
 - dynamic collections, 111
 - element binding, 116–117
 - FallBackValue attributes, 738, 740
 - FrameworkElement, 103
 - GroupDescriptions property (CollectionViewSource class), 738
 - INotifyDataErrorInfo interface, 738, 743
 - INotifyPropertyChanged, 127, 377
 - MoreValidation solution, 743–744
 - MvvmDataBinding ICommand solution, 734
 - Nodatabinding solution, 99–102
 - one-way binding, 103
 - performance, effect on, 107
 - simple, 102–106
 - SortDescriptions property (CollectionViewSource class), 738
 - StringFormat attribute, 738
 - TargetNullValue attribute, 738, 740
 - TestGrouping solution, 740
 - TestStringFormat solution, 738
 - two-way binding, 103
 - validation and, 118–121
- Databound Application project type, 632
- DataBoundApp_modified solution, 140
- DataBoundApp solution, 132
- DataBoundAppWithAds solution, 543
- DataBoundAppWithAds_TrialMode solution, 544
- DataContext, 107, 134
- DataContext class, 668, 669, 671, 672, 679
- CreateDatabase method, 671
- DataContract attribute, 235
- DataContractSerializer and, 236
 - XmlSerializer and, 236
- DataContractJsonSerializer, 235
- referencing requirements for, 237
 - XmlSerializer vs., 237
- DataContractSerializer, 235, 651
- DataContract attribute and, 236
 - referencing requirements of, 237
 - requirements/constraints on, 236–237
 - Windows Communication Foundation and, 236
 - XmlSerializer vs., 236
- data encryption
- AesManaged object, 460–461
 - CryptoStream object, 460–461
 - Rfc2898DeriveBytes class, 460–464
 - RNGCryptoServiceProvider class vs. Random class, 462
 - SimpleEncryption solution, 460
 - StrongerEncryption solution, 462
 - supported cryptographic functions, 459–464
- DataMemberAttribute, 651
- DataMember attribute type, 235
- Data Protection API (DPAPI), 697–699
- Protect method, 698
 - SimpleEncryption_DPAPI, 697
 - Unprotect method (DPAPI), 699
- data serialization options, 235–242
- DataContract attribute type, 235
 - DataContractJsonSerializer, 235
 - DataContractSerializer, 235
 - DataMember attribute type, 235
 - file size, comparisons between, 239
 - SerializeOptions_Perf solution, 241

- SerializeOptions solution, 237
- XmlSerializer, 235
- data service
 - yncSvcUtilHelper tool, 707
- DataServiceClient phone application, 650
- DataServiceClient_State, 654
- DataServiceCollections, 655
- DataServiceCollection<T> class, 738, 743
- DataServiceCollection<T> prooperty, 652
- DataServiceContext, 651
- DataServiceQuery<T> property, 651
- data services
 - CoffeeWeb application, 704
 - CoffeeWebApp solution, 708
 - SimpleCalendar application, 700
 - SimpleContacts application, 700
 - SyncClient solution, 704
 - SyncSvcUtilHelper tool, 707
- DataServiceState class, 654
- DataServiceState object, 655
- DataServiceState.Serialize method, 655
- data support, 667–710
 - calendar, 699–703
 - contacts, 699–703
 - encryption, 697–699
 - local databases and LINQ-to-SQL, 667–697
 - Synch Framework, 703–710
- DataSvcUtil tool, 650
- DataTemplate, 132
- data templates, 108–110, 113–141
- data tracking
 - privacy concerns with, 549
- data validation
 - INotifyDataErrorInfo interface, 743
 - MoreValidation solution, 743–744
- DbCreator application, 685
- DbCreator_NoVersion solution, 694
- DbCreator_WithVersion solution, 694
- Deactivated events, 187–189
 - and backstack, 189
 - fast application switching and, 557
 - fast reactivation (non-tombstone case), 183, 189–190
 - time limit to complete, 213
 - tombstone case, 183
- deactivation/reactivation
 - emulating in debugger, 749
 - fast application switching and, 555–558
- debugger behavior with different application conditions, 750

- debugging, 249–294
 - device and user information, 273
 - DeviceInfo solution, 273
 - Diagnostics_Email solution, 257–258
 - Diagnostics_Persist solution, 259
 - Diagnostics_SettingsExpando solution, 264–265
 - Diagnostics_Settings solution, 261
 - Fiddler, 291–292
 - FloatingDiagnostics_AppScope solution, 255
 - FloatingDiagnostics_Behavior solution, 253
 - and idle detection disabling, 270
 - improved tools for in SDK 7.1, 749–750
 - in Visual Studio, 249–250
 - lock-screen, 269–270
 - Manual Tests tab as bug tracking tool, 755
 - MediaPlayer, 270
 - MethodBase class, 252
 - Microsoft Network Monitor, 289–291
 - Panorama, use in, 255–257
 - pinning tiles, 638
 - Pivot, use in, 255–257
 - post-release, 257–259
 - Reflector, 301
 - release builds, 267
 - ScreenCapture application, 265
 - Silverlight Spy, 293
 - SimpleDiagnostics solution, 255
 - SimpleDiagnostics_UEH solution, 253
 - size constraint when e-mailing log files, 259
 - StackTrace class, 252
 - System.Diagnostics.Debug.WriteLine API, 250
 - TestMemory solution, 280–281
 - tombstoning, 269–270
 - UnhandledException event, 253
- DecibelMeter application, 336
- Deep Zoom applications, 382–387
 - converting code from desktop to phone applications, 384–387, 386
 - creating, 383–384
 - creating image file for, 384
 - Deep Zoom Composer tool, 383–387
 - MultiScaleImage control, 384
 - TestDZ solution, 382
 - zoom settings, default, 386
- Deep Zoom Composer tool, 383–387
- Delete button (LiveTiles application), 628
- deleting
 - tiles, 631
- Dependency Injection (DI), 125

DependencyObject

- DependencyObject, 62
 - DependencyObject.GetValue/SetValue method, 62
 - ResourceDictionary, 58
- DependencyObject.GetValue/SetValue method, 62
- dependency properties, 62–64
 - animation, 62
 - data binding, 62
 - visual inheritance, 62
- DependencyProperty, 62, 66
- Deserialize method (DataServiceState class), 654
- design-time data, 123–124
- DetailsPage, 641
- developer tools, 18–20, 747–774
 - debugger, improvements on, 749–750
 - development cycle, 19–20
 - emulator, 747–749
 - GeoCoordinateWatcher, improved testing for, 747
 - marketplace test kit, 751–755
 - portable library tools, 767–771
 - Profiler, 756–766
 - Screenshot tab, 749
 - Silverlight for Phone Toolkit, 19
 - Silverlight Media Framework, 19
 - UserVoice, 766–774
 - Windows Azure Toolkit for WP7, 19
 - Windows Phone 7.5 Training Kit, 19
 - Windows Phone 7 Training Kit for Developers, 19
 - Windows Phone Developer Guide, 19
 - Windows Phone Developer Tool, 18
 - Windows Phone Developer Tools January 2011 Update, 18
- development process, data storage during, 230
- Device Emulator (XDE), 267–269, 284–289
 - Automation, 286
 - features of, vs. device, 285
 - forced logging on release builds, 267
 - GeoCoordinateWatcher, testing on, 747–748
 - GeoPositionChangedEvent, raising in, 748
 - hardware requirements for, 285
 - improvements to, in the 7.1 SDK, 747–749
 - Phone 7.1.1, low memory support in, 628
 - picture library, lack of, 303
 - Screenshot tab, 749
 - testing accelerometer in, 594
- DeviceExtendedProperties class, 273, 280
 - ApplicationCurrentMemoryUsage property, 280, 625
 - ApplicationPeakMemoryUsage property, 280
- DeviceStatus API as replacement to, 273
- DeviceStatus class vs., 623
- DeviceTotalMemory property, 280, 625
- DeviceUniqueld property, 273
- ID_CAP_IDENTITY_DEVICE property, 277–278
- TryGetValue function, 273
- device information
 - Anonymous User ID (ANID), 276
 - DeviceExtendedProperties class, use of, 273
 - DeviceInfo solution, 273
 - Microsoft.Devices.Environment class, 274
 - Microsoft.Devices.Environment class, use of, 273
 - Microsoft .NET System.Environment class, 274
 - NetworkInterface class, 274
 - NetworkInterface class, use of, 273
 - PhoneApplicationService class, 275
 - PhoneApplicationService class, use of, 273
 - System.Environment class, use of, 273
 - use of, 273–277
- DeviceInfo solution, 273
- Device.InstallApplication, 286
- DeviceNetworkInformation class, 623
 - enhanced in Phone 7.1, 623–625
 - NetworkAvailabilityChanged event, 623
 - NewDeviceInfo solution, 623
- device orientation (Touch UI), 170–175
- device(s)
 - low physical memory, performance
 - enhancements for, 556
 - XDE vs., 285
- device security
 - PC tethering and (version 7.1), 449
 - SD cards and, 449
 - Windows Phone Live and lost devices, 449
- DeviceStatus API, 273
- DeviceStatus class
 - ApplicationCurrentMemoryUsage property, 626
 - ApplicationMemoryUsageLimit property, 625
 - DeviceExtendedProperties class, as replacement for, 623
 - enhanced in Phone 7.1, 623–625
 - KeyboardDeployedChanged event, 623
 - NewDeviceInfo solution, 623
 - PowerSourceChanged event, 623
- DeviceTotalMemory property (DeviceExtendedProperties class), 280
- DeviceUniqueld property (DeviceExtendedProperties class), 273
- diagnostics, 249–294
 - configurable, 261–265

- ConsoleDiagnostics solution, 268
- device and user information, 273
- DeviceInfo solution, 273
- Diagnostics_Email solution, 257–258
- Diagnostics_Persist solution, 259
- Diagnostics_SettingsExpando solution, 264–265
- Diagnostics_Settings solution, 261
- emulator console output, 267–269
- fixed controls for, 255–257
- FloatingDiagnostics_AppScope solution, 255
- FloatingDiagnostics_Behavior solution, 253
- FloatingDiagnostics sample code, 250–251
- memory, 279–284
- MethodBase class, 252
- Microsoft Network Monitor, 289–291
- Panorama, use in, 255–257
- performance counters, 277–284
- persisting logs and, 259–261
- Pivot, use in, 255–257
- post-release, 257–259
- ScreenCapture application, 265
- screen captures, 265–267
- sending screen captures via e-mail, 267
- setting up pop-up windows for, 250–255
- simple, 250–272
- SimpleDiagnostics solution, 255
- SimpleDiagnostics_UEH solution, 253
- size constraint when e-mailing log files, 259
- StackTrace class, 252
- System.Diagnostics.Debug.WriteLine API, 250
- TestMemory solution, 280–281
- UnhandledException event, 253
- Diagnostics_Email solution, 257–258
- Diagnostics_Persist solution, 259
- Diagnostics_SettingsExpando solution, 264–265
- Diagnostics_Settings solution, 261
- dialog boxes
 - ChildWindow, 89
 - Popup, 89, 91
 - System.Windows.Visibility, 89
- Direct3D 10 Level 9 support, 37
- DirectDraw, 37
- DirectionalViewfinder solution, 612
- Dispatcher object, 653
- Dispatcher property (UI elements)
 - SynchronizedContext object vs., 503
 - threads, using with, 501
- Dns.GetHostName, 644
- Dotfuscator Windows Phone Edition, 537–540
 - cleaning up unused code and, 539

- Control Flow, 537
- Linking, 537
- output of, 539
- PreMark, 538
- Removal, 538
- runtime errors with, possibility of, 538
- String Encryption, 538
- DoubleTap (logical touch gesture), 143
- Double-Tap (Pivot control behavior), 73
- DownloadChanged event (MediaElement class), 329
- downloading files with Background Transfer Service API, 566
- DownloadStringAsync method, 353–355
- Drag (logical touch gesture). See Pan/Drag (logical touch gesture)
- DynamicCollectionBinding solution, 111
- dynamic layout, 46
- dynamic layout and StackPanel, 46, 48
- DynamicSoundEffectInstance class, 339–343
 - BufferNeeded event, 339–343
 - SoundEffectInstance class vs., 343
 - TestDynamicSounds_Controls solution, 345
 - TestDynamicSounds solution, 340
 - usage, 340–343

E

- EAP. See Event-based Asynchronous Pattern (EAP)
- elements, limits on movement of, 145
- Elevated-Rights Chamber (ERC), 455
- e-mail
 - constraint on size of log file in, 259
 - use in post-release debugging/diagnostics, 258–259
- EmailComposeTask, 621
- emulator
 - Appointments class not available on, 701
 - Contacts class and, 701
 - LicenseInformation.IsTrial method, 545
 - testing trial mode in, 545
- EnableCacheVisualization flag (performance), 277
- EnableCacheVisualization flag (UI), 509
 - GPU and, 509
- EnableRedrawRegions flag (performance), 277
- EnableRedrawRegions flag (UI), 508
- encryption
 - databases, 695–710
 - Data Protection API (DPAPI), 697–699
 - Protect method (DPAPI), 698–699

encryption

- encryption, *continued*
 - ShoppingList_CRUD_Encrypted solution, 695
 - SimpleEncryption_DPAPI, 697
 - Unprotect method (DPAPI), 699
- EndAccept method, 645
- EnqueueCallback method (SLUTF), 522
- EnqueueTestComplete method (SLUTF), 522
- enterprise users and security, 458–459
- environment design with Expression Blend, 124
- ErrorCode property, 432
- ErrorOccurred event
 - ErrorCode property, 432
 - ErrorType property, 432
 - PowerLevelChanged event, 432
 - Push_Better\PnClient solution, 432
 - SubscribeToNotifications method and, 432
- ErrorType property, 432
 - ChannelOpenFailed, 432
 - PayloadFormatError, 432
- EventArgs object, 54
- Event-based Asynchronous Pattern (EAP). See also Async framework
 - AvatarWebClient_EAP solution, 772
 - Task-based Asynchronous Pattern (TAP) vs., 772
- event handler(s), 54, 118
 - asynchronous data and, 141
 - connecting methods, 55
 - EventHandler<T>, building custom with, 166
 - infinite loop, 130
 - Microsoft .NET standard, 54
 - multiple, 120
 - order of, 54
 - PositionChanged, 378
 - SessionChanged, 410
- exceptions, unhandled, 191
 - debugging and, 191
 - FatalError solution, 191
 - inappropriate usage of to exit applications, 191
 - lifecycle events and, 191
 - marketplace certification and, 191
 - UnhandledException handlers and, 191
- Exchange ActiveSync (EAS), 458
- Execute method (ICommand class), 736–738
- Expression Blend
 - behaviors, 163–167
 - Silverlight vs., 124
 - Visual Studio vs., 56
- Extended Backus-Naur Form (EBNF) grammar features of, 521

- extensions
 - with App Connect, 657
- ExtraFile attribute, 659
- Extras.xml, 657, 659

F

- Facebook, 497
 - data not available to applications, 700
- Facebook applications, 400–404
 - FacebookClient GetAsync method, 403
 - FacebookOAuthClient GetLoginUrl method, 402
 - FacebookOAuthResult object, 403
 - IsScriptEnabled property and, 401
 - permissions requests, 402
- Facebook C# SDK
 - supported services of, 400
 - TestFacebook solution, 401
- FallBackValue attributes, 738, 740
- fast application switching, 555–558
 - Deactivated events and, 557
 - internal behavior of, 557
 - IsApplicationInstancePreserved property and, 556
 - OnNavigatedFrom events and, 557
 - reattaching resources after dormancy, 558
- fast reactivation, 189–190
 - defined, 183
 - purpose of, 190
- FatalError solution, 191
- feedback forums, 766–767
- Fiddler
 - debugging/diagnostics using, 291–292
 - use in XDE, 291–292
 - use on physical device, limits on, 292
- FilteredAccelerometer application, 308
- filtering large datasets, 71
- filter queries, 138
- flick
 - horizontal, 78
 - vertical, 78
- Flick and Tap (GestureService), 161–162
- FlickGestureEventArgs, 161
 - HorizontalVelocity property, 161
 - VerticalVelocity property, 161
- Flick (logical touch gesture), 143, 145
- flick (manipulation events), 148–150
- Flickr, 495
- Flick Right/Left, 73

- Flick Up/Down, 73
- FloatingDiagnostics_AppScope solution, 255
- FloatingDiagnostics_Behavior solution, 253
- FloatingDiagnostics sample code, 250–251
- FMRadio class, 349–351
 - supported regions, 346
 - testing in emulator, 351
 - TestRadio solution, 349
- FM tuner, 349–351
- form authentication
 - Windows Communications Foundation (WCF) Service, requirement for, 474–479
- FormsAuthClient application, 472
- forms authentication, 472–483
 - ASP.NET, configuring for, 476–477
 - client side, 479–483
 - client side, implementation of, 480
 - configuring IIS Manager to test, 473
 - GetDataAsync method, 480
 - GetDataCompleted event, 480
 - LoginAsync method, 480
 - LoginCompleted event, 480
 - roles, use in restricting access, 481
 - server side, 473–479
 - service binding settings for, 479
 - Visual Studio and, 475
 - vulnerability of, 487
- FormsAuthServer application, 472
- FPS counter thread, 278
- Fragment
 - navigation and, 217–220
 - NavigationQueryString solution, 217
 - usage, 218
- Frame, 38
- frame rate counter (performance), 277–278
 - Intermediate Surface Counter, 278
 - Profiler, 759
 - Render Thread FPS, 278
 - Screen Fill Rate, 278
 - Surface Counter, 278
 - TestPerfCounters solution, 279
 - Texture Memory Usage, 278
 - User Interface Thread FPS, 278
- FrameReported events (Touch class), 155–156
- FrameworkElement, 103
 - DataContext, 103
 - ResourceDictionary, 59
- framework(s), 713–719
 - backstack management, 716–719
 - Clipboard API, 729

- frame/page navigation, 713–715
- navigation, 713–719
- Silverlight 4.0, 732–746
- touch events, dedicated thread for, 731
- UI enhancements, 719–731
- FxCop, 466–471
 - \$(CodeAnalysisPath) MSBuild property, modifying for, 467
 - BaseIntrospectionRule base class and, 469
 - creating rule sets, custom, 470
 - metadata requirements for, 468
 - MyCodeAnalysisRule solution, 467
 - Visual Studio, use in, 467

G

- GamelInviteTask chooser, 621
- GC (garbage collector) events as shown in Profiler, 760
- GDI, 37
- generic background agents (GBAs), 570–580
 - and XAP file inclusions, 571
 - BackgroundAgentDemo solution, 572
 - default expiry of, 571
 - ExpirationTime, setting, 576
 - GeoCoordinateWatcher behavior when used in, 578
 - main application and, 572
 - memory limitations on, 571
 - NotifyComplete method, 578
 - PeriodicTask class, 570
 - permitted/prohibited operations in, 571
 - Phone 7.1.1, lack of support in, 628
 - polling to update UI from, 573
 - ResourceIntensiveTask class (GBA), 570–571
 - starter code for, 579
 - WMApManifest, adding to, 579
- Geo Augmented Reality Toolkit (GART)
 - ARDisplay control, 616
 - augmented reality (AR) applications and, 616–617
 - IARView interface, creating custom views with, 617
 - required references for, 616
 - SLARToolkit vs., 617
 - source for, 616
 - TestGart solution, 616
- GeoCoordinateWatcher class, 318–322
 - background agents, behavior when used in, 578

- GeoCoordinateWatcher class, *continued*
 - Bing Maps and, 378–379
 - DesiredAccuracy property of, 319
 - Dispatcher.BeginInvoke and, 320
 - GeoCoordinate type, 320
 - GeoPosition objects, 319
 - MovementThreshold property, 321
 - PositionChanged event, 319
 - Position property, 320
 - raw data exposed by, 573
 - Reactive Extensions and, 322
 - saving test locations to file, 748
 - SimpleGeoWatcher application, 318
 - sources of data for, 318
 - TestBingMaps solution, 378
 - TestGeoCoordinates application, 320
 - testing on the emulator, 747–748
 - Geolocation
 - accessing on phone, 378
 - and Bing maps, 378
 - PositionChanged event handler, 378
 - TestBingMaps solution, 378
 - GeoPositionChangedEvent, raising in the XDE, 748
 - GestureEventArgs type, 162
 - GestureListener event, 158, 162
 - gestures
 - Flick Right/Left, 73
 - flicks, 72
 - Flick Up/Down, 73
 - horizontal flick, 78
 - horizontal pan, 78
 - modeling with mouse events, 154
 - Pan Right/Left, 73
 - Pan Up/Down, 73
 - Pinch-and-Stretch, 73
 - Tap, Double-Tap, or Press-and-Hold, 73
 - use and usability guidelines, 144
 - vertical pan or flick, 78
 - GestureService (Silverlight Toolkit), 158–163
 - GestureListener, 158
 - memory usage of, 163
 - problems with, 162–163
 - XNA TouchPanel, performance issues caused by, 162
 - GetChild method (VisualTreeHelper class), 42
 - GetChildrenCount method (VisualTreeHelper class), 42
 - GetDataAsync method, 480
 - GetDataCompleted event, 480
 - GetParent method (VisualTreeHelper class), 43
 - GetResourceStream method (Application class), 333
 - GlobalElementChange solution, 43
 - GoBack method (NavigationService class), 215, 716
 - Google, 497, 546
 - Google Analytics, 548
 - location of, 548
 - Google.WebAnalytics, 547
 - graphics processing unit (GPU), 37
 - Direct3D 10 Level 9 support, 37
 - DirectDraw, 37
 - GDI, 37
 - Graphics Processing Unit (GPU), 506–507
 - Grid, 40, 41, 75, 118
 - RowDefinitions, 48
 - ShowGridLines property, 46
 - GroupDescriptions collection property (CollectionViewSource class), 738
 - grouping queries, 138
 - Gyroscope class, 600–603
 - gyroscope (sensor API), 600–603
 - CurrentValueChanged event, 602
 - Micro-Electromechanical Systems (MEMS) device, 600
 - pitch, roll, and yaw, 600
 - SimpleGyro solution, 600
- ## H
- Habitant item, 641
 - handler(s)
 - catch-all, 120
 - changeButton_Click handler, 112
 - Hardin, David, 493
 - hardware
 - accelerometer, 305–317
 - applications processor requirements, 296
 - audio/video requirements, 323–324
 - digital camera requirements, 296
 - FM tuner, 349–351
 - graphics processor requirements, 296
 - memory requirements, 296
 - microphone, 335–343
 - minimum requirements for Windows Phone 7, 295–296
 - power requirements, 296
 - screen requirements, 296
 - sensor requirements, 296
 - universal volume control (UVC), 343
 - wireless requirements, 296
 - hardware sensors, 603

Hold (logical touch gestures), 143
 Home button
 and linking tiles, 637
 “home” feature, 636
 horizontal flick, 78
 horizontal pan, 78
 HttpNotification event, 424
 HttpRequest
 vs. WebClient, using with threads, 505–506
 WebClient vs., performance and, 515
 HttpRequest class, 357
 SimpleHttpRequest solution, 357
 vs WebClient class, 357
 HttpResponse class
 extension methods for, 441
 m_Headers member, 430
 push notifications and, 430
 HttpResponseExtensions class, 441
 HyperlinkButton class, 215
 HyperlinkButton (LiveTiles application), 628
 Hyperlink class, 734

I

IAApplicationService (Silverlight extension), 335
 IARView interface and custom GART views, 617
 ICommand class
 CanExecuteChanged event, 736–738
 CanExecute method, 736–738
 Execute method, 736–738
 IDataErrorInfo vs. INotifyDataErrorInfo interface, 743
 ID_CAP_IDENTITY_DEVICE property, 277–278
 ID_CAP_NETWORKING capability, 752
 IDictionary, 58
 ID_REQ_MEMORY_90 (markeplace manifest element), 626
 IEnumerable, 112
 IEnumerable<T>.OfT method, 44
 ILoopingSelectorDataSource, 87
 image rendering
 enhancement of in Phone 7.1, 730
 performance hit with higher bpp, 730
 images
 background, decoding of, 731
 dynamically creating, 629
 Page2 link Click handlers, 629
 as resources, 55
 image scaling
 PictureDecoder API, using, 514
 implicit styles
 defining in Silverlight 4.0, 732–734
 hierarchies of, 733
 TestImplicitStyles solution, 733
 in-application page backstack, 196
 ingestion. See marketplace ingestion
 Initial fly in, 72
 Initialized event (PhotoCamera class), 609
 InitializeService method, 653
 INotifyCollectionChanged, 111, 126, 138
 INotifyDataErrorInfo interface
 ErrorsChanged event, 745
 GetErrors method, 745
 HasErrors property, 745
 IDataErrorInfo interface vs., 743
 MoreValidation solution, 743–744
 INotifyPropertyChanged, 104, 111, 126, 133, 138, 377, 651
 INotifyPropertyChanged interface, 692
 INotifyPropertyChanging interface, 692
 Input Method Editor (IME). See also Software Input Panel (SIP)
 auto-completion, 169
 predictive word-completion, 169
 Input Method Editor (IME), 70
 InputScope, 167
 Chat input scope, 168
 TelephoneNumber input scope, 168
 intellectual property
 protecting, 537–540
 inter-application backstack. See backstack
 Intermediate Surface Counter (performance), 278
 Internet Explorer vs. WebBrowser control
 security, 500–501
 Internet Information Services (IIS) Manager
 configuring for Basic authentication, 483
 configuring to test forms authentication, 473
 creating SSL certificates with, 489
 Trusted Root Certification Authorities (CA) store, 489
 inter-process communication, 15
 IntervalTraining application, 560
 Inversion of Control (IoC), 125
 iPhone, CPNS push notifications for, 445
 IPEndPoint object, 644
 IPv4 sockets, 644
 IsCancelable property (OnNavigatingFrom method), 713
 IsValid property (Compass class), 597

IsInertial property (ManipulationCompletedEventArgs)

IsInertial property
(ManipulationCompletedEventArgs), 150
IsNavigationInitiator property
(NavigatingCancelEventArgs/NavigationEventArgs
objects), 715
IsoDataBound solution, 231
isolated storage, 226–246

- accessing, performance optimization for, 517–518
- access to, in WebBrowser control, 358
- cleaning up unused internal data, 231
- data serialization options, 235–242
- deleting data in, by user, 231
- during development, 230
- helpers, 243–246
- IsoDataBound solution, 231
- IsolatedStorageFile API, 226
- IsolatedStorageSettings dictionary class, 226
- IsolatedStorageSettings, saving, 231
- memory limits on, 229
- path to, 226
- persistence during application upgrade, 230
- simple persistence and, 227–231
- SimplePersistence_directory solution, 229
- SimplePersistence solution, 227
- subdirectories in, 229
- submitting updates to AppHub/marketplace, 530
- TestIsoStorage application, 243
- ViewModel, persisting, 231–235

Isolated Storage Explorer Tool, 233
Isolated Storage Explorer tool (ISETool), 684–692

- arguments for, 687
- command line syntax for, 686
- DbCreator application, 685

IsolatedStorageSettings object, 227
IsScriptEnabled property, 500
IsToastOk property, 440
ItemsControl, 87

- ItemsSource, 106
- ListBox, 87
- ListBox control, 106
- ListPicker, 87

ItemsSource, 87, 106, 107
ItemTemplate, 110
IValueConverter, 86, 114

J

Javascript

- ScriptNotify event, 361
- Silverlight, interaction with, 359–365

JavaScript Object Notation (JSON) format data

- filtering, 374
- vs. XML formatted data, 373
- WCF Data Services\DataServiceClient(JSON-filterable) solution, 374
- WCF Data Services\DataServiceClient(JSON) solution, 373

JPG images (Page2 link Click handlers), 629

K

KeyboardDeployedChanged event (DeviceStatus class), 623
keyboard (hardware)

- and orientation, 173
- emulating during design process, 169

keyboard input (Touch UI), 167–170
Key name, 57

L

Landscape modes

- LandscapeLeft, 173
- LandscapeRight, 173
- Panorama, 79

Language-Integrated Query (LINQ), 667
Language-Integrated Query (LINQ) operations, 138, 308
large datasets, filtering, 71
launchers/choosers, 195, 297–301

- backstack and, 297
- MediaPlayerLauncher class, 321
- navigation with, 297
- SimpleTasks application, 298
- using in applications, 298

Launchers/Choosers, enhancements to in Phone 7.1, 621–622

- AddressChooserTask chooser, 621
- BingMapsDirectionsTask launcher, 621
- BingMapsTask launcher, 621
- ConnectionSettingsTask launcher, 621
- contact information chooser, 621
- EmailComposeTask, 621
- GameInviteTask chooser, 621

- MediaPlayerLauncher, 621
- NewBingMaps solution, 622
- PhotoResult object return value, 621
- SaveContactTask chooser, 621
- SaveRingtoneTask chooser, 621
- ShareLinkTask launcher, 621
- ShareStatusTask launcher, 621
- WebBrowserTask, 621
- Launching events, 186
- layer decoupling, 121–131
- layout engine, runtime behavior of, 41
- LayoutRoot control, 44
- Least-Privileged Chamber (LPC), 455
- Level Starter Kit, 311–315
 - AccelerationHelper class, 311
 - OrientationHelper class, 311
- LicenseInformation.IsTrial method, 545
- lifecycle events, 181–196
 - Activated event, 187
 - and user expectations, 195–196
 - asymmetry of, 189
 - Closing case, 183
 - Deactivated case, 183
 - Deactivated events, 187–189, 213
 - fast reactivation, 183
 - fast reactivation (non-tombstone case), 189
 - Launching event, 186
 - TestActivation sample code, 184
 - tombstoning, 183
 - unhandled exception case, 183
- light-up features, 535–537
 - TestLightUp solution, 536
- line of business (LOB) users and security, 458–459
- LINQ-to-SQL
 - CoffeeStore application, 681
 - DataContext class, 668
 - local databases and, 667–697
 - ShoppingList_CR solution, 669
 - ShoppingList_CRUD solution, 675
 - SQL Server Management Studio (SSMS), 667
 - System.Data.Linq.Mapping namespace, 669
- ListBox, 41, 75
 - DataTemplate, 132
 - filtering, 71
 - IEnumerable, 112
 - ItemTemplate, 110
 - scrolling, limits on, 145
- ListBox control, 106
- ListBox elements
 - best practices for using, 512–513

- ListBoxItem template, 43
- ListBox.SelectionChanged handler, 634
- ListPicker, 87
- List<T> (collection type), 87
- Live/SkyDrive REST API, 409
- LiveTiles application, 628
- LoadAsync/LoadCompleted model, 652
- LoadCompleted event, 653
- LoadCompleted handler, 654, 655
- LoadContext method, 435–436
- LoadData method, 75
- LocalOffsetZ property (TextBlock), 613
- local tiles, 628–632
- lock-screen
 - accelerometer and, 321
 - ApplicationIdleDetectionMode setting and, 194
 - debugging, 269–270
 - disabling using UserIdleDetectionMode setting, 194
 - Obscured event and, 194
- logical touch gestures, 143–146
 - DoubleTap, 143
 - Flick, 143, 145
 - Hold, 143
 - Pan/Drag, 143, 145
 - Pinch, 143
 - Stretch, 143
 - Tap, 143
- LoginAsync method, 480
- LoginCompleted event, 480
- LongListSelector, 86
- LoopingSelector control, 86, 87

M

- Maecenas item, 641, 642
- magnetometer. See compass (sensor API)
- MainPage class, 119
 - DataContext, 134
 - PushViewModel class, changes when using, 436
- MainPage.xaml.cs, 654
- MainViewModel, 634
- managed code and marketplace applications, 453
- Managed Extensibility Framework (MEF)
 - AnalyticsApplicationService, 547
 - ApplicationLifetimeObjects, 548
 - AssemblyCatalog, 547
 - Compose method, 547
 - CompositionHost, 547

Managed Extensibility Framework (MEF)

- Managed Extensibility Framework (MEF), *continued*
 - MSAF, as basis for, 546–549
 - WebPropertyId, 548
- ManagedMediaHelpers class, 327–329
 - MediaParsers.Phone library project and, 324
 - Mp3MediaStreamSource.Phone library project and, 324
 - required library projects for, 324
 - source for, 327
 - usage, 327–328
- MangoAccelerometer solution, 592
- ManipulationCompletedEventArgs, 150
- Manipulation events
 - Click events vs., 158
 - combining Mouse events and, 156–157
 - ManipulationDelta, 151
 - ManipulationStarted event, 147
 - ManipulationXXX events, 146
 - multi-touch, 150–152
 - OnManipulationStarted method, 147
 - single touch (flick), 148–150
 - single touch (tap), 146–148, 147
- Manual Tests tab (marketplace test kit), 754, 755
- Map control, 376–378
 - key creation, 376–377
 - Panorama, 79
 - SimpleBingMaps solution, 377
- marketplace, 523–534
 - AppHub portal, setting prices/availability with, 528
 - beta testing, 534
 - beta testing, support for, 534
 - certification/publication, 523–530
 - checking for updates to application, 530
 - information required for submission, 523
 - MarketplaceData solution, 530
 - MarketplaceDetailTask Launcher, 532
 - Marketplace Test Kit and, 524
 - obfuscation of intellectual property, 537–540
 - portal for submissions, location of, 523
 - reports, 533
 - updates, 530–532
 - updates, restrictions on, 534–535
 - Zune marketplace atom feed, 531
- marketplace applications
 - information access disclosures to user, 456
 - instances, limit on number of, 183
- marketplace certification requirements
 - Authenticode certificate, 451
 - Automated Tests tab (marketplace test kit), testing with, 752
 - BindToShellToast method, 433
 - developer registration, 451
 - exception handling, 191
 - load time, 57
 - managed code and, 453
 - marketplace test kit, testing for compliance, 751–755
 - memory consumption, 191
 - navigation patterns, 201
 - Push_Better\PnClient solution, 433
 - responsiveness, 187
 - security, 451–453
 - unhandled exceptions and, 191
- MarketplaceData solution, 530
- MarketplaceDetailTask Launcher, 532
- marketplace ingestion
 - Manual Tests tab's mimicry of, 754
 - security constraints on, 451
 - WebBrowser control requirements for, 500
- marketplace injection
 - requirements, evolving, 532
- Marketplace Test Kit, 524
 - accessing, 751
 - Application Details tab, 751
 - Automated Tests tab, 752
 - Capability Detection tool, 752
 - developer tools, 751–755
 - Manual Tests tab, 754
 - Monitored Tests tab, 753–755
 - Windows Phone Capability Detection Tool vs., 272
- Marketplace TestKit
 - and AdControl capability testing, 544
- MasterVolume property (SoundEffect class), 340
- MeasureOverride method, 51, 65
- MediaElement class, 325–327
 - AutoPlay property of, 326
 - BufferingChanged event, 329
 - DownloadChanged event, 329
 - IsMuted property of, 326
 - NaturalDuration property of, 329
 - Pause method, 332
 - Play method, 332
 - Position property of, 329
 - protocols not supported by, 327
 - Stretch property of, 326
 - TestMediaElement solution, 325
 - TestVideo solution, 326

- Volume property of, 326
- vs. XNA SoundEffect class, 329
- MediaElement control, 329–332
- media files, 55, 57
- MediaLibrary type, reference requirements for, 302
- MediaParsers.Phone library project, 324
- media playback, 324–332
 - ManagedMediaHelpers class, 327–329
 - MediaElement class, 325–327
 - MediaElement control, 329–332
 - MediaPlayerLauncher class, 321
 - MediaStreamSource class, 327–329
 - TestMediaPlayer application, 325
- MediaPlayer
 - debugging, 270
 - Windows Phone Connect Tool, 270
- MediaPlayerLauncher class, 321
 - Orientation property, new, 621
 - TestMediaPlayer application, 325
- media services, 323–352
 - audio input/manipulation, 328–343
 - audio/video APIs, 324
 - audio/video hardware, 323–324
 - DecibelMeter application, 336
 - DynamicSoundEffectInstance class, 339–343
 - FM tuner, 349–351
 - ManagedMediaHelpers class, 327
 - MediaElement class, 325–327
 - MediaElement control, 329–332
 - MediaPlayerLauncher class, 321
 - MediaStreamSource class, 327
 - Microphone class (XNA), 335
 - Music and Videos Hub, 343–345
 - playback, 324–332
 - sample media, sources for testing, 327
 - slider control, implementing, 330–332
 - SoundEffect class (XNA), 329–331
 - SoundEffectInstance class, 329–331
 - SoundFx solution, 336
 - TestMediaHub solution, 343
 - TestRadio solution, 349
 - TestSoundEffect solution, 333
 - TestVideo solution, 326
 - video, encode/decode requirements for, 319–320
- MediaStreamSource class, 327
- memory
 - backstack, management of, 196
 - cleaning up unused internal data, 231
 - diagnostics, 279–284
 - paging, in Phone 7.1.1, 626–627
 - TestMemory solution, 280–281
 - usage, as shown by Profiler, 760
- MergedDictionaries property
 - ResourceDictionary, 60
 - search order of, 60
- MessageBox, 89, 90
- MessageSendPriority class, 441
- MessageSendResult class, 441
- MethodBase class, 252
- Metro, 633, 636
- Metro design guidelines, 69, 84
- Metro guidelines, 12–15, 39
 - and attempts to move content past boundary, 145
 - application design and, 13
 - application images and, 31
 - constraints imposed by, 14
 - transition animations, 12
 - typography and, 13–14
 - user interface (UI) and, 12–15
- Metro user experience (UX), 70
- m_HttpResponseHeaders member, 430
- Micro-Electromechanical Systems (MEMS) device, 600
- Microsoft Advertising SDK, 541–544
- Microsoft ASP.NET, 362
- Microsoft Cross-Platform Audio Creation Tool (XACT), 324
- Microsoft.Devices.Environment class, 273, 274
- Microsoft.Devices.Sensors namespace, 305
- Microsoft Interoperability Strategy Group, 444
- Microsoft .NET standard and event handlers, 54
- Microsoft .NET System.Environment class, 274
- Microsoft Network Monitor (NetMon)
 - TCP Analyzer expert, 290
 - using, 289–291
- Microsoft.Phone.Data.Linq namespace
 - DatabaseSchemaUpdater class, 680
- Microsoft pubCenter, 540–544
 - application, registering with, 541–542
 - categories of ads in, 541
 - location of, 540
 - regional limits on, 541
- Microsoft Push Notification Service (MPNS), 413–416
 - Apple Push Notification Service (APNS) vs., 443
 - Cloud 2 Device Messaging Framework (C2DM) vs., 443
- Microsoft Silverlight Analytics Framework (MSAF)
 - event tracking with, 546–549

- Google.WebAnalytics, 547
 - MEF, based on, 546
 - required references for, 547
 - TestAnalytics solution, 547
 - Webtrends.WebAnalytics.WP7, 547
- Microsoft Silverlight security model, 454
- Microsoft User Experience Design Guidelines, 11
- Microsoft.WebAnalytics assembly
 - as bridge to third-party analytics services, 547
- Microsoft.Xna.Framework.Games class, 18
- Microsoft.Xna.Framework.Graphics class, 18
- MMC Certificate Snap-in, 490
- MobileTestPage
 - NavigateBack method, 519
 - Silverlight Unit Testing Framework (SLUTF) and, 519
- Model-View ViewModel (MVVM)
 - and database operations, 669–710
 - benefits/drawbacks, 141
 - Model, 125
 - pattern, 125–131
 - Push Notification Server-Side Helper Library and, 441
 - View, 125
 - ViewModel, 125
- Monitored Tests tab, 753–754
- MoreValidation solution, 743
- Motion4 solution, 607
- motion APIs, 603–608
 - hardware, susceptibility to errors, 603
 - Motion4 solution, 607
- Motion class, 604–608
 - CurrentValueChanged event, 606
 - increased sensor accuracy using, 604
 - Motion4 solution, 607
 - SimpleMotion solution, 604
 - testing for device support of, 605
- MouseAndManipulation solution, 156
- MouseButtonEventArgs class, 54
- Mouse events, 152–155
 - Click events vs., 158
 - combining Manipulation events and, 156–157
 - modeling gestures with, 154
 - MouseEnter, 152
 - MouseEventArgs, 153
 - MouseLeave, 152
 - MouseLeftButtonDown, 152
 - MouseLeftButtonUp, 152
 - MouseMove, 152
 - MouseWheel, 152
 - MouseWheel event, unused, 155
- MovementThreshold property, 321
- MPNS. See Microsoft Push Notification Service (MPNS)
- multi-Pivot pages, 74
- multiple platforms, targeting, 766–767
- MultiScaleImage. See Deep Zoom applications
- MultiScaleImage control, 384
- multi-tasking, 559–590
 - Alarms and Reminders, 559
 - background agents, generic, 570–580
 - background audio, 580–589
 - Background Transfer Service API, 559
 - Generic Background Agents, 559–560
- multi-touch (manipulation event), 150–152
- Music and Videos Hub, 343–345
 - testing on emulator/physical device, 348
 - TestMediaHub solution, 343
- MvvmDataBinding ICommand solution, 734
- MvvmDataBinding_Model solution, 129
- MvvmDataBinding solution, 126
- MyCodeAnalysisRule solution, 467
- MyPhotoExtra solution, 302
- MyPictureExtension solution, 618
- MySpace, 497
- MyTargetPage, 658

N

- Name, 57
- NAudio library
 - source for, 343
 - WAV file headers and, 343
- Navigate method, 215
- NavigateUri property (HyperlinkButton class), 215–216
- NavigatingCancelEventArgs (OnNavigatingFrom method)
 - Cancel property of, 713
 - IsNavigationInitiator property, 715
- navigation
 - application/page state and, 206–213
 - between applications, 197
 - between multiple assemblies, 216–217
 - canceling, 221–223
 - Deactivated events and, 213
 - disk I/O, performance issues with, 229
 - Fragment/QueryString, 217–220
 - implementation using HyperlinkButton, 216

- isolated storage and, 226–246
- marketplace certification and, 201
- NavigateUri, 215–216
- NavigationMode property, 220–221
- NavigationQueryString solution, 217
- NavigationService, use of, 206–207
- Non-Linear Navigation Service (NLNS), 223–226
- OnNavigatedFrom, 205
- OnNavigatedTo property, 205
- options within Windows Phone platform, 215–226
- re-routing navigation/URI mappers, 221–223
- ReRouting solution, 216
- separate assembly of pages, 216–217
- TestNavigation solution, 208
- TestNlns application, 225
- TestUriMapping solution, 222
- user experience and, 213
- NavigationContext, 637, 639
- NavigationContext property (Page class), 219
- navigation controls
 - Back button (hardware), 196
 - Button control, 196
 - HyperlinkButton control, 196
 - NavigationService, 196
- navigation enhancements in Phone 7.1
 - backstack management, 716–719
 - ClearBack_Thumbs solution, 717–719
 - frame/page, 713
 - NavigationEventArgs object, 714
 - NavigationMode property (NavigationEventArgs object), 714
- NavigationEventArgs object
 - Content property of, 220
 - IsNavigationInitiator property, 715
 - NavigationMode property of, 714
 - overrides of OnNavigatedTo/OnNavigatedFrom and, 220
 - Uri property of, 220
- navigation mappers
 - re-routing, 221–223
 - ReRouting solution, 216
- navigation model, 205–248
 - options for navigation, 215
 - resurrection, detecting, 213–215
 - state, 205–215
- NavigationMode property (NavigationEventArgs object), 220–221, 714
 - emulation of in Windows Phone 7, 220
 - in Windows Phone 7.1 vs Windows Phone 7, 220

- NavigationQueryString solution, 217
- NavigationService class
 - BackStack property, 716
 - GoBack method, 215, 216, 716
 - Navigate method, 215
 - OnRemovedFromJournal virtual method, 716
 - RemoveBackEntry method, 716
- NavigationUri, 630
- NetMon. See Microsoft Network Monitor (NetMon)
- NetworkAvailabilityChanged event (DeviceNetworkInformation class), 623
- network calls
 - performance optimization and, 516
- networking calls
 - requirements for, 499
- NetworkInterface class, 273, 274
- network monitoring, 289–291
- NeutralResourcesLanguage attribute (assembly)
 - requirement for, by marketplace, 532
- NewBingMaps solution, 622
- NewDeviceInfo solution, 623
- NewSystemTray solution, 726
- NoDataBinding solution, 99, 100
- Nonlinear Navigation Service (NLNS), 223–226
 - implementation of, 218–220
 - made redundant in Phone 7.1, 716
 - source of, 223
 - TestNlns application, 225
- notification batching interval values, 427
- NotifyCollectionChangedEvent, 111
- NotifyOnValidationError, 118
- NotifyPropertyChangedEvent, 111

O

- OAuth 1.0, 495–497
- OAuth 2.0, 497
- obfuscation of intellectual property, 537–540
 - Dotfuscator Windows Phone Edition, 537–540
- object tree, 58
- Obscured event(s), 193–195
 - NavigatedFrom event vs., 193
 - TestObscured application, 193
 - Unobscured event and, 194
- ObservableCollection<T> (collection type), 111, 133, 743
- OData Client
 - enhanced features in Phone 7.1, 650–656

OData protocol

- OData protocol
 - vs.SOAP protocol, for performance
 - optimization, 516
- OnBackPressed, 96
- OnNavigatedFrom events
 - behavior, 205
 - fast application switching and, 557
 - overriding handlers, 211
 - overriding to persist client push notification settings, 431
- OnNavigatedFrom override, 655
- OnNavigatedTo events
 - behavior, 205
 - overriding handlers, 211
 - overriding, in WCF Data Service clients, 371
- OnNavigatedTo method, 634
 - overriding, 631
- OnNavigatedTo override, 635, 641, 654, 655
- OnNavigatedTo, overriding, 637
- OnNavigatingFrom method
 - IsCancelable property, 713
 - NavigatingCancelEventArgs, 713
- OnRemovedFromJournal virtual method (NavigationService class), 716
- On-Screen Keyboard (OSK). See Software Input Panel (SIP)
- Opacity property
 - Visibility property vs., for performance
 - optimization, 514
- Open Data (OData) client, 366
- Open Data (OData) vs SOAP format, 365
- OpenReadAsync method, 355–356
- OpenReadTaskAsync method, 774
- orientation, 173
- OrientationChanged event, 172, 175
- OrientationHelper class (Level Starter Kit), 311, 313–322
- orientation (Touch UI), 170–175
- OverheadMap control, 617

P

- Page2 link Click handlers, 629
- PageCreationOrder sample code, 201
- Page elements, 38
 - Page Sizes, 38
 - PhoneApplicationPage type, 38
- page model, 196–203
 - forward navigation controls, 196

- page creation order, 201–203
 - PageCreationOrder sample code, 201
- Page object, 219
- page size, 38
- Paley, Mark, 615
- pan
 - horizontal, 78
 - vertical, 78
- Pan/Drag (logical touch gesture), 143, 145
- Panel class, 46
 - Canvas, 46
 - Grid, 46
 - StackPanel, 46
- Panorama, 136
 - Background Image animation, 79
 - background images, choosing, 80
 - Content animation, 79
 - debugging/diagnostics, use in, 255–257
 - DefaultItem property, 84
 - entry point with, 77
 - Orientation property, 82
 - Pivot compared with, 80
 - Pivot vs., performance and, 515–516
 - Title animation, 79
- Panoramaltem, 77
- Panoramaltem Header, 79
- Panorama Title, 79
- Pan Right/Left, 73
- Pan Up/Down, 73
- ParameterizedThreadStart delegate (Thread class), 501
- PC tethering, 449
- peer-to-peer chat application, remote, 642
- People Hub, 636
- performance
 - ProductCatalog application, 756
 - testing for using Profiler, 756–766
- performance counters
 - Cache visualization (performance), 277
 - diagnostics and, 277–284
 - EnableCacheVisualization flag, 277
 - EnableRedrawRegions flag, 277
 - Frame rate counter, 277
 - Intermediate Surface Counter, 278
 - Redraw regions, 277
 - Render Thread FPS, 278
 - Screen Fill Rate, 278
 - Surface Counter, 278
 - TestPerfCounters solution, 279
 - Texture Memory Usage, 278

- User Interface Thread FPS, 278
- performance optimization, 505–517
 - assemblies, factoring pages out to separate, to improve, 516
 - Bing maps, static vs. dynamic, 516
 - BitmapCache, using, 510
 - BouncingBall solution, 507
 - BouncingStoryboard solution, 510
 - constructors, minimizing for, 516
 - EnableCacheVisualization flag and, 509
 - inline XAML UI element declarations and, 514
 - isolated storage, accessing, 517–518
 - JPG vs. PNG images and, 513
 - ListBoxes and, 512–513
 - network calls and, 516
 - non-UI tips for, 515
 - OData protocol vs. SOAP protocol, for, 516
 - Panorama vs. Pivot controls, 515–516
 - redrawing regions, checking for, and, 508
 - Resource vs. Content, image embedding as, 514
 - re-using desktop code for Phone and, 514
 - scaling images and, 513
 - static vs. dynamic images and, 513
 - UI tips for, 513–515
 - UI vs. render thread and BitmapCache, 505–512
 - Visibility property vs. Opacity property and, 514
 - WebClient vs. HttpWebRequest, 515
- PerformanceProgressBar (Toolkit)
 - TestProgressBars solution, 515
 - use for performance optimization, 515
- Performance Warnings menu (Profiler), 764
- PeriodicTask class (GBA), 570
- persisting logs, 259–261
 - Diagnostics_Persist solution, 259
 - providing option to clear, 261
- PersonViewModel, 75
- Phone 7
 - running code for 7.1 on, 535–537
- Phone 7.1 applications
 - Phone 7 applications, updating when published with, 535
- PhoneAccentBrush resource, 27
- PhoneAccentColor resource, 27
- PhoneApplicationFrame, 38, 40
- PhoneApplicationPage class, 40, 176
- PhoneApplicationPage declaration, 115
- PhoneApplicationPage.Resources, 58
- PhoneApplicationPage.State property, 206, 212
- PhoneApplicationPage type, 38
- PhoneApplicationService class, 273, 275
- PhoneApplicationService.State property, 206, 212
- PhoneCallTask, 298
- PhoneDarkThemeVisibility resource, 27
- PhoneHorizontalMargin resource, 74
- PhoneLightThemeVisibility resource, 27
- phone services, 295–322
 - accelerometer, 305–317
 - geo-location, 318–322
 - hardware, 295–296
 - launchers/choosers, 297–301
 - photo extras, 301–304
 - Reactive Extensions for .NET (Rx .NET) and, 308–310
 - tasks, 297–301
- phone services, enhanced in Phone 7.1, 591–628
 - accelerometer, 592–594
 - augmented reality (AR) application, 612–617
 - camera pipeline, 608–612
 - compass, 595–599
 - DeviceStatus/DeviceNetworkInformation classes, 623–625
 - DirectionalViewfinder solution, 612
 - Geo Augmented Reality Toolkit (GART), 616–617
 - gyroscope, 600–603
 - Launchers/Choosers, 621
 - MangoAccelerometer solution, 592
 - motion APIs, 603–608
 - PhotoCamera class, 608
 - photo extensibility, 618–621
 - sensor APIs, 591–608
- PhoneTextNormalStyle, 39
- Phone version 7.1 SDK
 - Isolated Storage Explorer Tool, 667
- PhotoCamera class, 608
 - CaptureImageAvailable event, 609, 610
 - default orientation of, 608
 - Initialized event, 609
 - rear-facing vs. forward-facing camera support, 608
 - required references for, 608
 - usage, 609
- PhotoChooserTask chooser, 621
- photo extensibility
 - Apps menu vs. Extras mechanism, 618
 - enhancements in Phone 7.1, 618–621
 - Extensions section requirement for, in WMAAppManifest.xml, 618
 - jumping off points to, 618
 - MyPictureExtension solution, 618
 - Photos_Extra_Hub, 618

photo extras applications

- Photos_Extra_Share, 618
- Photos_Extra_Viewer, 618
- testing, constraints on, 619
- WPConnect tool, testing with, 619
- photo extras applications, 301–304
 - accessing from Start menu vs. photo gallery, 303
 - creating, 302
 - debugging, 304
 - marketplace certification requirements and, 303
 - MyPhotoExtra solution, 302
 - testing requirements for, 303
- PictureDecoder API
 - performance optimization and, 514
- Pinch and Drag (GestureService), 160–161
- Pinch-and-Stretch, 73
- PinchAndStretch solution, 150
- Pinch (logical touch gesture), 143, 150, 151
- pinning
 - tiles, 632–638
- PinTiles application, 632
- p/invoke, 469
- Pivot
 - Panorama vs., performance and, 515–516
- Pivot control, 136
 - content pane, 71
 - fundamental elements, 72
 - header, 71
 - PivotItems, 71
 - title, 71
 - use in debugging/diagnostics, 255–257
- PivotFilter_CollectionViewSource solution, 138
- PivotFilter solution, 136
- pivoting on data, 137
- PivotItem, 73
- playback services
 - Music and Videos Hub, 343–345
 - slider control, implementing, 330–332
- PnClient_Mango, 639
- PNG images (Page2 link Click handlers), 629
- PnServer_Mango, 639
- polling
 - updating UI with background agent information using, 573
- Popup, 89, 91
 - IsOpen, 91
 - OnBackKeyPress, 91
 - use in debugging/diagnostics, 250–255
- Portable Library Class projects, 767–771
 - constraints on resource sharing with, 770
 - identifying target platform, 768
 - SilverlightProductCatalog application, 769
 - source for, 767
- Portrait modes
 - PortraitDown, 173
 - PortraitUp, 173
- PositionChanged event handler, 378
- post-release debugging/diagnostics, 257–259
 - Diagnostics_Persist solution, 259
 - Diagnostics_SettingsExpando solution, 264–265
 - e-mail, use in, 258–259
 - ScreenCapture application, 265
 - sending screen captures via e-mail, 267
 - size constraint when e-mailing log files, 259
- PowerLevelChanged event, 432
- PowerSourceChanged event (DeviceStatus class), 623
- Press-and-Hold, 73
- PrintVisualTree method, 42
- ProductCatalog application, 756
- Profiler, 756–766
 - Frames view, using to troubleshoot choppiness, 764
 - function report, 764
 - Functions list, location of, 764
 - function report, inclusive/exclusive samples in, 764
 - graph, explanation of, 759–760
 - log files, removing from project, 759
 - logs created by, location of, 757
 - memory analysis pass with, 765
 - Performance Warnings menu, 764
 - ProductCatalog application, 756
 - selecting arbitrary portions of timeline in, 763
 - tombstoning/FAS analysis in, 766
 - types allocated, checking, 765
 - usage, 757
 - using to identify performance problems, 761
- ProgressBar
 - TestProgressBars solution, 515
- ProgressChanged event (BackgroundWorker class), 502
- ProgressIndicator property (SystemTray)
 - determinate/indeterminate progress bar, 728
 - enhancements to, 725
 - enhancements to in Phone 7.1, 725–728
 - visibility concerns with, 728
- Project Hawaii, 398
- properties
 - updating tile, 631
- PropertyChangedEventHandler, 104, 105

- Protect method (DPAPI), 698
 - overloading, 699
- proximity sensors, 37
- Push_Additional\PnClient solution, 431
- Push_Additional\PnServer solution, 427
- Push_Better\PnClient solution, 432, 433
- Push_Better\PnServer solution, 429
- Push Client service
 - behavior of, 414
 - HttpNotificationChannel.Open and, 414
 - Push_Simple\PnClient solution, 422
- Push enhancements, 627, 638–643
- push notification client
 - ChannelUriUpdated event, 424
 - ChannelUriUpdate event, 425
 - HttpNotification event, 424
 - mainpage.xaml.cs, required fields in, 423
 - Push_Additional\PnClient solution, 431
 - ShellToastNotification event, 424
- PushNotificationMessage class, 441
- push notifications, 413–448
 - architecture for, 413–416
 - batching intervals, 427–428
 - BindToShellToast method, 433
 - client, 422–426
 - client features, 431–441
 - Common Push Notification Service, 443–447
 - compatibility with APNS/C2DM, 443
 - ErrorOccurred events, 432–433
 - Microsoft Push Notification Service, 413–416
 - payload size limit of, 415
 - persistant client settings, 431–432
 - PowerLevelChanged event, 432
 - Push_Additional\PnClient solution, 431
 - Push_Additional\PnServer solution, 427
 - Push_Better\PnClient solution, 432, 433
 - Push_Better\PnServer solution, 429
 - Push_Simple\PnClient solution, 422
 - Push_Simple\PnServer solution, 416
 - PushViewModel class, 435–441
 - push ViewModel, implementing, 435–441
 - Push_ViewModel\PnClient solution, 435
 - Raw type of, 415
 - required functionality for, 417
 - response information for, 430
 - security in, 494–495
 - server, 416–422
 - server features, 427–430
 - Server-Side Helper Library, 441–443
 - settings page to set preferences for, 435
 - SSL and, 494–495
 - Tile type of, 415
 - Toast type of, 415
 - user opt-in/opt-out, 433–435
 - WebResponse class and, 416
 - XML payloads, 428–430
- Push Notification Server-Side Helper Library, 441–443
 - Ask to Pin Application Tile notification pattern, 443
 - Create Custom Server-Side Image notification pattern, 443
 - HttpWebResponse class, extension methods for, 441
 - HttpWebResponseExtensions class, 441
 - MessageSendPriority class, 441
 - MessageSendResult class, 441
 - notification examples in, 443
 - Push Counter Resets upon Logon notification pattern, 443
 - PushNotificationMessage class, 441
 - RawPushNotificationMessage class, 441
 - Scheduled Tile Updates notification pattern, 443
 - TilePushNotificationMessage class, 441
 - ToastPushNotificationMessage class, 441
- push notifications, types of
 - Create Custom Server-Side Image notification pattern, 443
 - One-time Push pattern, 443
 - Push Counter Resets upon Logon pattern, 443
 - Scheduled Tile Updates notification pattern, 443
 - ToastPushNotificationMessage class, 443
- Push_Simple\PnClient solution, 422
- Push_Simple\PnServer solution, 416
- push, tile, and toast, enhancements in Phone 7.1
 - local tiles, 628–632
 - pinning tiles, 632–638
 - push notifications, 638–642
- PushViewModel class, 435–441
 - changes to MainPage class when in use, 436
 - Dispatcher field in, 438
 - IsToastOk property, 440
 - LoadContext method, 435–436
 - SaveContext method, 435–436
- Push_ViewModel\PnClient solution, 435

Q

QueryString

- navigation and, 217–220
- NavigationQueryString solution, 217
- TryGetValue method and, 219
- usage, 218, 219

Quick Card, 657

R

- RadioButton (internal Mouse event control), 158
- rameworkElement.DataContext property
 - System.Data.Linq.DataContext vs., local database acces with, 668
- Raw (push notification), 415
- RawPush
- NotificationMessage class, 441
- Reactive Extensions for .NET (Rx .NET), 308–310
 - FilteredAccelerometer application, 308
 - GeoCoordinateWatcher class and, 322
 - usage, 309
- ReadingChanged event (Accelerometer class), 306
 - CurrentValueChanged event vs., 592–593
 - return values, 307
- ReceiveAsync, 648
- Redraw regions (performance), 277
- Reflector (debugging tool), 301
- RegisterAttached method, 66
- registry, editing, 267
- relative layout type, 46
- reliability
 - of sockets, 643
- reminders
 - anatomy of, 560
 - calendar reminders, 563
 - constraints on, 560
 - DatePicker, use with (Silverlight toolkit), 565
 - re-boot survival of, 566
 - TrailReminders application, 563
- remote peer-to-peer chat application, 642
- RemoveBackEntry method (NavigationService class), 716
- Render thread
 - BouncingStoryboard solution, 510
 - Graphics Processing Unit (GPU) and, 506–507
 - performance optimizations using, 506
 - usage of, 506–507
- ReportProgress method (BackgroundWorker class), 502

reports

- AppHub, available through, 533

ReRouting solution, 216

Resource

- Content vs., for performance optimization, 514

ResourceDictionary

- Application class, 59
- DependencyObject, 58
- FrameworkElement, 59
- IDictionary, 58
- Key, 59
- MergedDictionaries property, 60
- merging, 60
- object types supported, 58
- PhoneApplicationPage.Resources type, 58

ResourceIntensiveTask class (GBA), 570

resource(s), 55–61

- build action set to, 56
- content vs., 56–57
- dictionaries, resource, 57–61
- external, 61
- images, 55, 57
- Key name, 57
- media files, 55, 57
- Name, 57
- referencing other resources, 61
- {StaticResource} syntax, 58
- text files, 55, 57

response times and user expectations, 195

resurrection, detecting, 213–215

reusable XAML. See code resources

Rfc2898DeriveBytes class, 460–464

RichTextBox control, 721–722

RNGCryptoServiceProvider class, 462

RootFrame object, 220

RootVisual property, 40

- PhoneApplicationPage, 40

- RootFrame and, 220

RoutedEventArgs class, 54

routed event(s), 52–55

- BindingValidationError, 118
- visual tree, 55

RoutedEvents class, 52

RowDefinitions

- *, 48
- arbitrary, 48
- Auto, 48
- Grid, 48

rule sets, custom, 470

S

- sandboxing processes, 455
- SaveContactTask chooser, 621
- SaveContext method, 435–436
- SaveRingtoneTask chooser, 621
- ScaleFactor property, 65
- ScaleTransform, 151
- ScheduledActionService
 - Alarm object and, 560
 - ScheduledActionService.GetActions method, 561
- schema, databases
 - associations between, 681
 - supported changes to, 680
 - version numbering, 679
 - version updates to, 677–681
- ScreenCapture application, 265
- screen captures, 265–267
 - WriteableBitmap.Render method, 266
 - Zune and, 267
- screen configuration, 37
 - light sensor, 37
 - multi-touch, support for, 37
 - orientations, support for, 37
 - physical size, 37
- Screen Fill Rate (performance), 278
- screen layout, 46–49
 - Application Bar, 38
 - Page elements, 38
 - real estate available for app, 38
 - System Tray, 38
 - Visual Studio template, 39
- Screenshot tab, 749
- scripts
 - in WebBrowser control, 358
 - ScriptNotify event, 361
- ScrollViewer.ManipulationMode property (ListBox object), 731
- SD cards and device security, 449
- Search button (hardware), 37
- search extensibility, enhanced in Phone 7.1, 657–665
 - App Connect, 657–663
 - App Instant Answer, 664–665
- secondary local tiles, 638
- security, 449–502
 - application deployment and, 451–453
 - application safeguards, 450–459
 - chambers/capabilities, 455–457
 - data encryption, 459–464
 - device security, 449–450
 - Execution Manager, use for, 456–457
 - FormsAuthClient application, 472
 - FormsAuthServer application, 472
 - FxCop, 466–471
 - hard-coding security credentials, 499
 - installing certificates, 488
 - managed code constraints and, 453–455
 - missing features of, 458
 - MyCodeAnalysisRule solution, 467
 - OAuth 1.0, 495–497
 - OAuth 2.0, 497–498
 - push notifications and, 494–495
 - sandboxing processes for, 455
 - SDL tools, 464–471
 - SimpleEncryption solution, 460
 - sockets, 649
 - SSL, 488–494
 - StrongerEncryption solution, 462
 - threat modeling, 465–466
 - WebBrowser Control, implementing for, 500
 - web service IDs, securing, 498–499
 - web service security, 471–494
- Security Development Lifecycle (SDL) tools
 - FxCop, 464, 466–471
 - SDL Guidelines, 464
 - threat modeling, 465–466
 - Threat Modeling Tool, 464
- SendAsync, 648
- SendNotification method, 640
- sensor APIs
 - optional sensors available in Phone 7.1, 591
 - SensorBase<T> base class and, 591
 - SimpleGyro solution, 600
- sensor APIs, enhanced in Phone 7.1, 591–608
 - accelerometer, 592–594
 - compass, 595–599
 - gyroscope, 600–603
 - motion APIs, 603–608
 - SimpleMotion solution, 604
- SensorBase<T> base class
 - as basis for all sensor APIs, 591, 597, 601
 - IsValid property, inherited from, 597
- sensor(s)
 - accelerometer, 305–317
 - light, 37
 - power usage by, 321
 - proximity, 37
 - TestAccelerometer application, 306
- Separation of Concerns (SoC), 102, 104, 121–131
- Serialize method (DataServiceState class), 654

- SerializeOptions_Perf solution, 241
- SerializeOptions solution, 237
- service bindings
 - enableHttpCookieContainer attribute, 480
- SessionChanged event handler, 410
- ShakeGesture event
 - ShakeGestureEventArgs property, 317
 - ShakeMagnitudeWithoutGravitationThreshold property, 317
 - ShakeType value, 317
- Shake Gesture Library
 - AccelerometerHelper and, 315
 - required references for, 316
 - ShakeGesturerHelper class in, 315
 - source for, 315
 - TestShake solution, 316
- SharedAccessSignature service, 394, 397
- ShareLinkTask launcher, 621
- ShareStatusTask launcher, 621
- ShellTile.Create method, 629, 630
- ShellTile object, 631
- ShellToastNotification event, 424
- ShoppingList_CR solution, 669
- ShoppingList_CRUD_Encrypted solution, 695
- ShoppingList_CRUD solution, 675
- ShowGridLines property, 46
- ShowStatus method, 647
- side-loading applications, 452
- Silverlight
 - rendering engine optimizations, 505
- Silverlight 4.0, 69, 125, 732–746
 - CLR Security model, 454
 - CollectionViewSource class, 738
 - command binding support, 734–738
 - Cross-Site Scripting (XSS) protection, 500
 - data binding enhancements in, 738–746
 - databound application template, 23–25
 - DataServiceCollection<T> class, 738
 - Dialog class, lack of, 89
 - FallBackValue attributes, 738, 740
 - GroupDescriptions property (CollectionViewSource class), 738
 - IApplicationService extensibility mechanism, 335
 - implicit styles, defining in, 732–734
 - INotifyDataErrorInfo, 738
 - INotifyDataErrorInfo interface, 743
 - Javascript, interaction with, 359–361
 - Microsoft.Xna.Framework.Games class and, 18
 - Microsoft.Xna.Framework.Graphics class and, 18
 - Model-View ViewModel, 125
 - MvvmDataBinding ICommand solution, 734
 - panorama application template, 23–25
 - phone application template, 23–25
 - phone class library template, 23–25
 - pivot application template, 23–25
 - project template types, 23–25
 - rotation angle direction, vs. Motion, 615
 - RoutedEvents class, 52
 - runtime behavior, 48, 59
 - scrolling, 70
 - security model of, 454
 - Silverlight for Phone Toolkit, 19
 - Silverlight Media Framework, 19
 - Silverlight Spy, 293
 - SortDescriptions property (CollectionViewSource class), 738
 - StringFormat attribute, 738
 - TargetNullValue attribute, 738, 740
 - TestImplicitStyles solution, 733
 - TestStringFormat solution, 738
 - Touch/gesture enabling, 70
 - using XNA classes with, 18
- Silverlight EasingFunctions
 - usage, 510
- Silverlight layout types, 46
 - absolute, 46
 - custom layouts, 46
 - dynamic, 46
 - relative, 46
- Silverlight Media Framework, 19
- SilverlightProductCatalog application, 769
- Silverlight Spy, 293
- Silverlight Toolkit, 69
 - ChildWindow, 94
 - DatePicker, 565
 - GestureService, 158–163
 - Silverlight Unit Testing Framework (SLUTF) and, 517
 - using to build settings page, 439
 - VisualTreeExtensions, 46
 - WrapPanel, 174
- Silverlight Toolkit GestureService
 - Flick and Tap, 161–162
 - gesture event sequences, 158
 - Pinch and Drag, 160–161
- Silverlight Unit Testing Framework (SLUTF), 517–523
 - EnqueueCallback method, 522
 - ExpectedException attribute, 520
 - MobileTestPage, 519
 - required references for, 519

- Silverlight Toolkit and, 517
- Tag attribute, 521
- TestClass attribute, 519
- TestCleanup, 522
- test-driven development (TDD) and, 517
- TestInitialize attribute, 522
- TestMethod attribute, 519
- Silverlight XAML schema, 57
- Silverlight XAP, 293
- SimpleAppInstantAnswer, 664
- simple arrays (collection type), 87
- SimpleBingMaps solution, 377
- SimpleCalendar application, 700
- SimpleCamera solution, 608
- SimpleCompass solution, 595
- SimpleContacts application, 700
- SimpleDataBinding solution, 103
- SimpleDiagnostics solution, 255
- SimpleDiagnostics_UEH solution, 253
- SimpleEncryption_DPAPI, 697
- SimpleEncryption solution, 460
- SimpleEvents solution, 53
- SimpleGeoWatcher application, 318
- SimpleGyro solution, 600
- SimpleHttpRequest solution, 357
- SimpleLayout solution, 46
- SimpleMotion solution, 604
- Simple Object Access Protocol (SOAP) services, 379
- simple persistence
 - isolated storage and, 227–231
 - SimplePersistence_directory solution, 229
 - SimplePersistence solution, 227
- SimplePersistence_directory solution, 229
- SimplePersistence solution, 227
- SimpleResources solution, 61
- SimplestAppConnect, 658
- SimpleTasks application, 298
- SimpleVisualTree solution, 40
- single touch (flick), 148
- single touch (tap), 146–148
- SkyDrive, 409–411
 - Live/SkyDrive REST API, 409
 - TestLive_Photos solution, 409
- SLARToolkit, 617
- Sleep method (Thread class), 501
- Slider control (Panorama), 79
- Smart Device Connectivity component (CoreCon), 286
- SmugMug, 495
- SOAP protocol
 - OData protocol vs., for performance optimization, 516
- SocketAsyncEventArgs, 648
- SocketAsyncEventArgs class, 648
- SocketClient, 643
- sockets, 642–649
 - TCP sockets, 643–649
- SocketServer, 643
- Software Input Panel (SIP), 70, 167
 - features of, 167
 - InputScope of elements, 167
- SortDescriptions collection property (CollectionViewSource class), 738, 743
- sort queries, 138
- SoundEffectInstance class, 329–331, 343
- SoundFx_Persist solution, 342
- SoundFx solution, 336
- SoundLab download, 333
- SplashScreenImage.jpg application image, 30
- SQL-CE
 - SQL Server Compact Toolbox tool, for working with, 690
- SQL Server Compact Toolbox tool
 - SQL-CE databases, working with, 690
- SQL Server Management Studio (SSMS), 667
- SSL, 488–494
 - certificate authorities, 488
 - creating certificates for, 489
 - credential web services and, 499
 - push notifications and, 494–495
 - testing without, 487
 - Windows Phone Certificate Installer helper library, 488
- StackPanel, 41, 75, 93
- StackTrace class, 252
- standard controls
 - Panorama control, 77–85
 - Pivot control, 71–77
 - SDK, 69–70
 - toolkit controls, 86–89
- Standard-Rights Chamber (SRC), 455
- StandardTileData class, 631
- StandardTileData parameter (ShellTile.Create method), 629
- Start button (hardware), 37
- StartListening worker method, 644
- Start page, 630
- Start Windows Phone Performance Analysis vs. Start Performance Analysis, 757

- state
 - application, 206–213
 - memory limits on storing, 212
 - page, 206–213
 - persistant, 206
 - PhoneApplicationPage.State property, 206
 - PhoneApplicationService.State property, 206
 - State.ContainsKey, testing, 212
 - transient application, 206
- state dictionaries
 - and object serialization, 213
 - memory limits on, 212
 - PhoneApplicationPage.State, 206–213
 - PhoneApplicationService.State, 206–213
- StateObject parameter, 645
- State.TryGetValue, 655
- {StaticResource} syntax, 58
- Status Bar. *See* SystemTray class
- storyboard events in Profiler, 760
- Stretch (logical touch gesture), 143, 150
- StringFormat attribute, 738
- StrongerEncryption solution, 462
- styles
 - as code resources, 57
 - hierarchies of, 733
 - implicit, using, 732–734
 - TestImplicitStyles solution, 733
- SubmissionInfo folder (marketplace test kit), 755
- SubscribeToNotifications method, 432
- SupportedOrientations
 - handling orientation changes using, 170–175
 - OrientationChanged event, 172
 - PortraitOrLandscape, 170
- Surface Counter (performance), 278
- SyncClient solution, 704
- Synch Framework, 703–710
 - code generation, 707–710
 - CoffeeWeb application, 704
 - CoffeeWebApp solution, 708
 - configuration of service, 705–707
 - database provisioning, 707
 - required references for, 709
 - scope definition, 706
 - source for, 704
 - SyncClient solution, 704
 - Sync Framework: SyncSvcUtil, 705
 - SyncSvcUtilHelper tool, 705, 707
 - yncSvcUtilHelper tool, 707
- SynchronizationContext object
 - TestThreading_SyncContext solution, 503
 - threads, accessing UI with, 503–504
- SynchronizedContext object
 - Dispatcher property vs., 503
- System.Data.Linq.DataContext
 - rameworkElement.DataContext property vs., for database access, 668
- System.Data.Linq.Mapping namespace, 669
- System.Data.Services.Client.dll, 650
- System.Diagnostics.Debug.WriteLine API, 250
- System.Environment class, 273
- system memory available to app, 280
- system notifications vs. Toast notifications, 433
- SystemTray class, 38, 74, 79
 - enhancements to in Phone 7.1, 725–728
 - FPS counter thread requirements for, 278
 - new properties exposed in, 726
 - NewSystemTray solution, 726
 - Opacity property, 79
 - properties of, 38
 - quirky color properties of, 726
 - in Windows Phone 7.1, 38
- System.Windows.Media.Imaging namespace, 629
- System.Windows.Visibility, 89
- System.Windows.xaml, 41

T

- Tag attribute (SLUTF), 521
- Tag attributes (SLUTF)
 - Extended Backus-Naur Form (EBNF) grammar, usage of, 521
- TangoTest solution, 627
- Tap, 73
- TAP. *See* Task-based Asynchronous Pattern (TAP)
- Tap (logical touch gesture), 143
- tap (manipulation event), 146–148
- TargetNullValue attribute, 738, 740
- Task-based Asynchronous Pattern (TAP), 773
 - AvatarWebClient_TAP solution, 772
 - Event-based Asynchronous Pattern (EAP) vs., 772
 - Visual Studio Async Framework and, 772
- tasks, 297–301
 - CameraCaptureTask, 300
 - PhoneCallTask object, 298
 - SearchTask, 298
 - WebBrowserTask, 300
- TCP Analyzer expert, 290
- TCP sockets, 643–657
- templates
 - as code resources, 57

- screen layout, 39
- TitlePanel, 38, 39
- Visual Studio, 38
- Test71Controls solution, 720
- TestAccelerometer application, 306
- TestActivation sample code, 184
- TestAnalytics solution, 547
- TestBehaviors solution, 163
- TestBehaviors_Standard solution, 166
- TestBingMaps solution, 378
- TestBitly sample code, 398
- TestCleanup (SLUTF), 522
- TestClipboard application, 729
- TestDependencyProps solution, 64
- test-driven development (TDD)
 - Silverlight Unit Testing Framework (SLUTF) and, 517
- TestDynamicSounds_Controls solution, 345
- TestDynamicSounds solution, 340
- TestDZ solution, 382
- testers
 - using AppHub to pass information to, 528
- TestFacebook solution, 401
- TestFrameReported solution, 155
- TestGart solution, 616
- TestGeocodeService application, 380
- TestGeoCoordinates application, 320
- TestGestureService solution, 160
- TestGrouping solution, 740
- TestImplicitStyles solution, 733
- testing
 - beta testing, support for in marketplace, 534
 - dummy data for, 674–676
 - Silverlight Unit Testing Framework (SLUTF), using, 517–523
 - test-only viewmodels, 674
 - upgrade scenario with ISETool, 690
- testing (application)
 - marketplace test kit, 751–755
 - Profiler. See Profiler
- TestInitialize attribute (SLUTF), 522
- TestIsoStorage application, 243
- TestLightUp solution, 536
- TestLive_Photos solution, 409
- TestLive solution, 404
- TestLoopingSelector solution, 86
- TestManipulation solution, 146
- TestMediaElement solution, 325
- TestMediaHub solution, 343
- TestMediaPlayer application, 325
- TestMemory solution, 280–281
- TestMouse solution, 153
- TestNavigation solution, 208
- TestNlns application, 225
- TestObscured application, 193
- TestOrientation solution, 170
- TestPanorama_Template solution, 85
- TestPerfCounters solution, 279
- TestPivot solution, 73
- TestPopup solution, 89
- TestProgressBars solution, 515
- TestRadio solution, 349
- TestShake solution, 316
- TestSip solution, 168
- TestSoundEffect solution, 333
- TestStringFormat solution, 738
- TestThreading solution, 500
- TestThreading_SyncContext solution, 503
- TestUriMapping solution, 222
- TestVideo solution, 326
- TestWbc solution, 358
- TextBlock (LiveTiles application), 628
- TextBox, 118
 - data binding, 103
 - NotifyOnValidationError, 118
 - ValidatesOnExceptions, 118
- text files, 55, 57
- Text Text Revolution!, 170
- Texture Memory Usage (performance), 278
- theme, 25–29
 - application icons and, 32
 - documentation source for, 27
 - PhoneAccentBrush resource, 27
 - PhoneAccentColor resource, 27
 - PhoneDarkThemeVisibility resource, 27
 - PhoneLightThemeVisibility resource, 27
 - ThemeAccent sample, 27
- ThemeAccent sample, 27
- Thread class
 - location of, 501
 - Name property, 501
 - ParameterizedThreadStart delegate, 501
 - Sleep method, 501
 - ThreadStart delegate, 501
- threading, 499–505
 - BackgroundWorker class, 501–502
 - Render thread, 505
 - TestThreading solution, 500
 - TestThreading_SyncContext solution, 503
 - UI controls and, 501

threading

- threading, *continued*
 - UI thread, 505
 - web service client proxies with, 504
- ThreadPool
 - BackgroundWorker class usage of, 504
- ThreadPool (threading API), 500
 - when to use, 503
- ThreadStart delegate (Thread class), 501
- Thread (threading API), 500–501
 - when to use, 501
- Threat Modeling tool (SDL), 465–466
- Tile (push notification), 415, 420
- TilePushNotificationMessage class, 441
- tiles
 - local, 628–632
 - pinning, 632–638
- Tiles
 - sides of, 627
 - updating, 627
- tiles pinned to, 629
- TitlePanel, 38
- Toast enhancements, 627
- toast notification, 639, 640
- toast notifications, 638
- Toast (push notification), 415
 - payload requirements for, 420
 - requesting permission for use of, 433
- ToastPushNotificationMessage class, 441
- ToggleSwitch control (Panorama), 79
- Tokens element, 665
- tombstone case, 187–189
 - defined, 183
 - Launchers/Choosers and, 195
- tombstoning
 - Activated events vs. Launching events, 187
 - debugging, 269–270
 - emulating in debugger, 749
 - fast application switching and, 555–558
- Touch class, 155
- touches
 - finger, 144
 - palm, 144
- touch events
 - dedicated thread for, 731
 - ScrollViewer.ManipulationMode property, 731
- TouchPoint, 156
 - Position property, 156
 - Size property, 156
- TouchPointCollection, 156
- touch target
 - design and size, 144
 - dynamic sizing of, 169
 - size of, vs. touch element, 144
- Touch UI, 143–180
 - Application Bar, 175–180
 - BallManipulation solution, 148
 - behaviors, 163–167
 - BouncingBall solution, 161
 - Click events, 158
 - combining Manipulation/Mouse events, 156–157
 - DoubleTap, 143
 - drag, 145
 - finger touch size, 144
 - Flick, 143, 145
 - Flick and Tap, 161–162
 - FrameReported events, 155–156
 - Hold, 143
 - keyboard input, 167–170
 - logical touch gestures, 143–146
 - MouseAndManipulation solution, 156
 - Mouse events, 152–155
 - multi-touch, 150–152
 - orientation, 170–175
 - Pan/Drag, 143, 145
 - Pinch, 143
 - Pinch and Drag, 160–161
 - PinchAndStretch solution, 150
 - Silverlight Toolkit GestureService, 158–163
 - single touch (flick), 148–150
 - single touch (tap), 146–148
 - Stretch, 143
 - Tap, 143
 - target design and size, 144
 - TestBehaviors solution, 163
 - TestBehaviors_Standard solution, 166
 - TestFrameReported solution, 155
 - TestGestureService solution, 160
 - Test Manipulation solution, 146
 - TestMouse solution, 153
 - TestOrientation solution, 170
 - TestSip solution, 168
 - touch events and overrides, 146
 - TouchPointCollection, 156
 - TouchPoints, 156
 - touch target vs. touch element, 144
 - use and usability guidelines, 144
 - WrapOrientation solution, 174
- tracking data, 546–549
 - consent requirements for, 546

- TrailReminders application, 563
- TransferProgressChanged event, 568
- Transient Panel(s), 89
 - ChildWindow, 94
 - Popup, 89
 - System.Windows.Visibility, 89
 - Visibility, 92
- transitions
 - animations, 12
 - Pivot control, 72
- TranslateTransform, 151
- Transmission Control Protocol (TCP)
 - sockets, 642
- Transmission type sockets, 643
- Transport Layer Security (TLS) certificates. *See* SSL
- trial mode, application, 544–546
 - DataBoundAppWithAds_TrialMode solution, 544
 - LicenseInformation.IsTrial method, 545
 - switching to paid version, how to, 545
- Trusted Computing Base (TCB) chamber, 455
- TryGetValue method, 219, 273
- Twitter, 495
- TwoPivots solution, 74
- type/value converters, 114–116

U

- UI, 37–68
 - and Metro, 12–15
 - AppBar class, enhancements to in Phone 7.1, 725–728
 - attached properties and, 64–68
 - background image, decoding, 731
 - Clipboard API, 729
 - content vs. resource in, 56–57
 - control enhancements, 720–725
 - dependency properties and, 62–64
 - EnableCacheVisualization flag, 509
 - EnableRedrawRegions flag, 508
 - enhancements in Windows Phone 7.1, 719–731
 - Frame, 38
 - NewSystemTray solution, 726
 - ProgressIndicator property, enhancements to in Phone 7.1, 725–728
 - rendering behavior, 507–508
 - resources, 55–61
 - screen layout, 46–49
 - standard elements, 37–40
 - SystemTray class, enhancements to in Phone 7.1, 725–728
 - TestClipboard application, 729
 - themes/accent colors and, 25–29
 - threading and the, 501
 - touch events, dedicated thread for, 731
 - user vs. custom controls, 49–52
 - visual tree, 40–46
 - WebBrowser control as, 358
- UIElement class, 70
- UI enhancements in Phone 7.1
 - RichTextBox control, 721–722
 - Test71Controls solution, 720
 - VideoBrush control, 723
 - ViewBox control, 722
 - WebBrowser control, 724–725
- UI model, 69
- UI thread
 - BouncingBall solution, 507
 - usage of, 505–506
- UnhandledException event, 253
- unhandled exceptions. *See* exceptions, unhandled
- Universal Volume Control (UVC), 343, 581
- “unpin” feature, 633
- Unprotect method (DPAPI), 699
 - overloading, 699
- update
 - marketplace injection, restrictions on, 535
- Update button (LiveTiles application), 628
- Update method, 631
- updates
 - name change and, 535
 - restrictions on, in marketplace, 534–535
 - submitting while app is in hidden state, 535
- Updating
 - tiles, 627
- upgrading applications and isolated storage, 230
- UriMapper, 657
- URI mapper, custom, 660
- URI mappers
 - re-routing, 221–223
 - TestUriMapping solution, 222
- UriMapper type
 - MappedUri property, 223
 - re-routing navigation, 223
- UriMapping type, 223
- UserControls, 49–52
- User Datagram Protocol (UDP)
 - sockets, 642
- user expectations and lifecycle events, 195–196
- UserExtendedProperties class, 273, 276
- UserIdleDetectionMode setting, 194

user information

- user information
 - Anonymous User ID (ANID), 276
 - application safeguards on, 450–459
 - ID_CAP_IDENTITY_DEVICE property, 277
 - use of, 273–277
 - UserExtendedProperties class, use of, 273
- User Interface Thread FPS (performance), 278
- user interface (UI), 629
- UserVoice, 766–774
 - location of Windows Phone forums at, 766–767
 - phone application providing access to, 766
- UTF8 and XML payloads, 429

V

- ValidatesOnExceptions, 118
- validation, 118–123
 - BindingValidationError, 118
 - error handler, 120
 - NotifyOnValidationError, 118
 - ValidatesOnExceptions, 118
 - Validation.Errors collection, 118
- versions
 - light-up features, 535–550
 - marketplace support for, 534–537
 - Phone 7/7.1, updating applications for, 535
 - TestLightUp solution, 536
 - updating database schemas, 677–681
 - user data, saving after updates, 678
- versions, application, 534–537
- vertical flick, 78
- vertical pan, 78
- VideoBrush control, 723
- video, encode/decode requirements for, 319–320
- ViewBox control, 722
- ViewModel
 - IsoDataBound solution, 231
 - persisting in isolated storage, 231–235
- Vimeo, 495
- Visibility, 92, 93
- Visibility property, 637
 - Opacity property vs., for performance optimization, 514
- visual hierarchy. *See* visual tree
- Visual Studio
 - and overriding virtual methods, 215
 - Databound Application template, 132
 - debugging in, 249–250
 - debugging on device with, 249
 - debugging tombstoning/lock-screen in, 270
 - Expression Blend vs., 56
 - form authentication projects and, 475
 - FxCop, use in, 467
 - Panorama template-generated project, 136
 - Pivot template-generated project, 136
 - Portable Library Class add in, 767
 - Smart Device Connectivity component, 286–289
 - Solution Explorer, 56
 - starter code, 38
 - stub methods, creating in, 676
 - System.Diagnostics.Debug.WriteLine API, 250
 - Windows Phone 7 Cloud Application solution template, 394
 - Windows Phone Scheduled Task Agent project, 577
- Visual Studio Async Framework
 - Asynch framework and, 772
 - Task-based Asynchronous Pattern (TAP) introduced by, 772
- Visual Studio Databound Application template, 132, 139
- Visual Studio Panorama Application, 81
- visual tree, 40–46, 41
 - and Application Bar, 180
 - App Bar and, 180
 - Button, 41
 - Grid, 41
 - StackPanel, 41
 - VisualTreeHelper class, 42
- VisualTreeExtensions, 46
- VisualTreeHelper class, 42, 43, 44
 - GetParent method, 43
 - PrintVisualTree method, 42
 - VisualTreeHelper.GetChild method, 42
 - VisualTreeHelper.GetChildrenCount method, 42
 - wrapping methods of, 45

W

- WaitOne, 645
- WbcScript sample code, 359
- WCF Data Services, 362, 365–375
 - JSON-formatted data, 373–375
 - Open Data (OData) client, 365–373
 - Visual Studio Windows Phone application templates and, 370
 - WCF, 362
 - WCF Data Services\CustomerWebApp solution, 367

- WCF Data Services\DataServiceClient(JSON-filterable) solution, 374
- WCF Data Services\DataServiceClient(JSON) solution, 373
- WCF Data Services\DataServiceClient(Simple) solution, 370
- WCF Simple\ChuckClient solution, 362
- WCF Simple\ChuckService solution, 362
- WCF Data Services\CustomerWebApp solution, 367
- WCF Data Services\DataServiceClient(JSON-filterable) solution, 374
- WCF Data Services\DataServiceClient(JSON) solution, 373
- WCF Data Services\DataServiceClient(Simple) solution, 370
- WCF Simple\ChuckClient solution, 362
- WCF Simple\ChuckService solution, 362
- WebBrowser control, 357–361
 - ActiveX controls, disallowance of, 358
 - cross-site restrictions on, 358
 - Cross-Site Scripting (XSS) protection, 500
 - enhancement to in Phone 7.1, 724–725
 - Internet Explorer security vs., 500–501
 - isolated storage and, 358
 - IsScriptEnabled property, 500
 - marketplace ingestion requirements for, 500
 - mobileoptimized tag, 361
 - NotifyEventArgs parameter, 359
 - Panorama, 79
 - required capabilities, 359
 - script invocation and, 358, 359
 - ScriptNotify event, 361
 - Silverlight/JavaScript interaction and, 359–361
 - specifying name for, 359
 - TestWbc solution, 358
 - user interface, as, 358
 - vs. desktop version of same, 358
 - WbcScript sample code, 359
 - WebBrowser.InvokeScript method, 359
 - WebBrowser.IsScriptEnabled property, 359
 - WebBrowser.NavigateToString method, 361
 - WebBrowser.ScriptNotify event, 359
- WebBrowserTask
 - new properties for, 621
 - using, 300
- WebClient
 - HttpRequest vs., performance issues with, 515
 - UI thread and, 505
- WebClient class, 353–356
 - AvatarWebClient solution, 355
 - DownloadStringAsync method, 353–355
 - HttpRequest class vs., 357
 - OpenReadAsync method, 355–356
 - OpenReadTaskAsync method, 773
 - performance issues and, 355
 - using, 353
- WebRequest class, 329
- WebResponse class, 416
 - X-DeviceConnectionStatus header, 416
 - X-NotificationStatus header, 416
 - X-SubscriptionStatus header, 416
- web role(s), 387
- web service
 - threading for background services in, 504
- web services, 353–412
 - Bing maps, 376–382
 - bitly, 398–400
 - consumption of, 362–365
 - credentials and SSL, 499
 - Deep Zoom, 382–387
 - Facebook, 400–404
 - HttpRequest class, 357
 - Microsoft ASP.NET, consuming, 362
 - Open Data (OData) client, 365–373
 - third-party access to, using OAuth 2.0, 497
 - WCF, 362
 - WCF Data Services, 365–375
 - WCF Simple\ChuckService solution, 362
 - WebBrowser control, 357–361
 - WebClient class, 353–356
 - Windows Azure, 387–398
 - Windows Live, 404–411
- web service security, 471–494
 - authentication, 472
 - basic authentication, 483–487
 - form authentication, 472
 - FormsAuthClient application, 472
 - FormsAuthServer application, 472
 - SSL, 488–494
- Webtrends, 546
- Webtrends.WebAnalytics.WP7, 547
- Wilcox, Jeff, 517
- Windows API, 453
- Windows Azure
 - API, 388
 - AppFabric SDK, 388
 - as cloud service, 387–398
 - client/server applications, building, 387–389
 - client/server applications, flows between, 396
 - CloudChuck solution, 389

- Windows Azure, *continued*
 - implementation of application on, 388
 - SDK, 388
 - server-side configuration for application, 389–393
 - SQL Azure, 388
 - StorageClient.dll and, 397
 - Toolkit for Windows Phone, 394–398
 - web role(s), 387
 - web services, consuming by Phone apps, 389–393
 - Windows Azure Storage, 388
 - worker role(s), 387
- Windows Azure Toolkit, 394–398, 394–412
 - Azure SDK, 388
 - Azure Tables and Queues proxies, 394
 - included tools/templates/libraries, 394
 - Shared Access Signature service, 394
 - Windows Phone 7 Cloud Application solution template, 394
- Windows Azure Toolkit for WP7, 19
- Windows CE, 16
- Windows Communications Foundations (WCF) Data Service
 - DataContractSerializer and, 236
 - DataServiceCollection<T> class, 738
- Windows Communications Foundation (WCF) Service
 - AuthenticationService endpoint, 475
 - form authentication and, 474–479
 - serviceHostingEnvironment element, 475
 - system.ServiceModel element, 475
- Windows Live
 - connecting to, from Phone app, 404–411
 - Live SDK source, 406
 - Live/SkyDrive REST API, 409
 - provisioning application on, 405
 - REST API, 408
 - SessionChanged event handler, 410
 - sign-in implementation, 408
 - SkyDrive, 409–411
 - TestLive_Photos solution, 409
 - TestLive solution, 404
 - wl.basic scope, 407
 - wl.offline_access scope, 407
 - wl.signin scope, 407
- Windows Phone 7
 - API source, 18
 - documentation source, 19
 - documentation source for media support, 334
 - Exchange ActiveSync (EAS), support for, 458
 - security challenges, vs. other Windows clients, 486
 - side-loading applications, lack of support for, 452
 - version 7.1 vs., 32–35
 - vs iOS and Android, 15
 - Windows CE and, 16
- Windows Phone 7.1
 - additional features of, vs. 7, 33–36
 - compatibility with Windows Phone 7, 32
- Windows Phone 7.1 SDK
 - Isolated Storage Explorer tool (ISETool), 686
 - Microsoft Advertising SDK, 541–544
 - support for GBAs in, 572
 - System.Windows.xaml, 41
 - version 7 application testing and, 19
 - Windows Phone Capability Detection Tool, 271–272
 - Windows Phone Connect Tool, 270
- Windows Phone 7.5 Training Kit, 19
- Windows Phone 7 Cloud Application solution template
 - Cloud project, 394
 - Cloud Services project, 394
 - Phone client application, 394
- Windows Phone 7 Training Kit for Developers, 19
- Windows Phone Capability Detection Tool, 271–272
- Windows Phone Certificate Installer helper library, 488
- Windows Phone Connect Tool, 270
- Windows Phone Developer Guide, 19
- Windows Phone Developer Tools, 18
- Windows Phone Developer Tools January 2011 Update, 18
- Windows Phone devices, hardware requirements for, 295–296
- Windows Phone “Mango”, 555
- Windows Phone Performance Analysis tool.
- See Profiler
- Windows Phone SDK, 69
- Windows Phone SDK 7.1.1 Update, 19
- Windows Phone shell
 - Application Bar, 38
 - System Tray, 38
- Windows Phone Version 7.1.1
 - ApplicationWorkingSetLimit property (DeviceExtendedProperties class), 626
 - generic background agents, lack of support for, 628

- ID_REQ_MEMORY_90 (markeplace manifest element), 626
- low-memory device support in, 625–628
- TangoTest solution, 627
- Windows Presentation Foundation (WPF), 102, 643
- WMAAppManifest, 638
- WMAAppManifest.xml, 657, 659, 664, 665
 - application image entry in, 30
 - generic background agents (GBAs), adding to, 579
 - in XAP file, 22
- worker role(s), 387
- WP7CertInstaller project, 493–494
- WPConnect tool, 619
- WrapOrientation solution, 174
- WrapPanel, 174, 175
- WriteableBitmap class, 629
- WriteableBitmapEx third-party library, 629
- WriteableBitmap.Render method, 266
- WriteEndElement method, 428
- WriteStartElement method, 428

X

- XAML
 - and pinning tiles, 634
- XAP files
 - contents of, 22–23
 - GBAs and, 571
- Xbox Live, 772
- XDE. See Device Emulator (XDE)
- X-DeviceConnectionStatus header, 416
- XDocument
 - push payloads, use in building, 428
 - XmlWwrite vs., 429
- XML formatted data, 373
- XML payload push notifications, 428–430, 429
- XmlSerializer, 235
 - DataContract attribute and, 236
 - DataContractJsonSerializer vs., 237
 - DataContractSerializer vs., 236
 - requirements/constraints on, 236
- XmlWriter
 - push payloads, use in building, 428
 - WriteEndElement method, 428
 - WriteStartElement method, 428
 - XDocument vs, 429
- XNA Microphone class, 335
 - behavior during phone calls, 336

- BufferDuration property, 335
- BufferReady event, 332
- DecibelMeter application, 336
- GetSampleSizeInBytes, 335
- SoundFx_Persist solution, 342
- SoundFx solution, 336
- WAV format and, 342
- XNA SoundEffect class, 329–331
 - MasterVolume property, 340
 - MediaElement class vs., 329
 - SoundLab. source for, 333
 - TestSoundEffect solution, 333
- XNA FrameworkDispatcher, emulating, 330–343
- XNA TouchPanel, 162
- X-NotificationClass, 427
- X-NotificationStatus header, 416
- X-SubscriptionStatus header, 416
- X-WindowsPhone-Target header, 420
- XXXAsync method, 648

Y

- Yahoo!, 495

Z

- Zune
 - and screen captures, 267
 - MediaPlayer debugging and, 270
- Zune marketplace atom feed, 531
- Zune media queue (ZMQ) vs. BackgroundAudioPlayer class, 580

About the Author



ANDREW WHITECHAPEL has worked in the software industry for 30 years, including more than 20 years as a developer, working mainly in C/C++ and C#. During his 10 years at Microsoft, he has covered stints in Consulting Services, Visual Studio, and Xbox. Andrew is currently a Program Manager in the Windows Phone team, where he is responsible for core features of the application platform.

Survey Page goes here.