

THE EXPERT'S VOICE® IN SILVERLIGHT

Pro Business Applications with Silverlight 4

*Create rich business applications customized
to your needs quickly and efficiently*

Chris Anderson

Apress®

Pro Business Applications with Silverlight 4



Chris Anderson

Apress®

Pro Business Applications with Silverlight 4

Copyright © 2010 by Chris Anderson

All rights reserved. No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the publisher.

ISBN-13 (pbk): 978-1-4302-7207-6

ISBN-13 (electronic): 978-1-4302-7206-9

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Trademarked names may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, we use the names only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

President and Publisher: Paul Manning

Lead Editor: Jon Hassell

Technical Reviewer:

Editorial Board: Clay Andres, Steve Anglin, Mark Beckner, Ewan Buckingham, Gary Cornell, Jonathan Gennick, Jonathan Hassell, Michelle Lowman, Matthew Moodie, Duncan Parkes, Jeffrey Pepper, Frank Pohlmann, Douglas Pundick, Ben Renow-Clarke, Dominic Shakeshaft, Matt Wade, Tom Welsh

Coordinating Editor: Tracy Brown

Copy Editors: Julie Hammond, Damon Larson, Heather Lang, Kim Wimpsett.

Compositor: Bytheway Publishing Services

Indexer: Toma Mulligan

Artist: April Milne

Cover Designer: Anna Ishchenko

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com.

For information on translations, please e-mail rights@apress.com, or visit www.apress.com.

Apress and friends of ED books may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Special Bulk Sales—eBook Licensing web page at www.apress.com/info/bulksales.

The information in this book is distributed on an “as is” basis, without warranty. Although every precaution has been taken in the preparation of this work, neither the author(s) nor Apress shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in this work.

The source code for this book is available to readers at www.apress.com. You will need to answer questions pertaining to this book in order to successfully download the code.

*For my grandmother Connie, a great role model for will power, optimism,
generosity, patience, and love. I miss her dearly.*

Contents at a Glance

■ Contents at a Glance	iv
■ Contents	v
■ About the Author	xix
■ Acknowledgments	xx
■ Chapter 1: Introduction	1
■ Chapter 2: Getting Started with Silverlight	13
■ Chapter 3: An Introduction to XAML	35
■ Chapter 4: The Navigation Framework	73
■ Chapter 5: Exposing Data from the Server: Using WCF RIA Services	93
■ Chapter 6: Implementing Summary Lists	141
■ Chapter 7: Building Data Entry Forms	189
■ Chapter 8: Securing Your Application	245
■ Chapter 9: Styling Your Application	269
■ Chapter 10: Advanced XAML and Data Binding	287
■ Chapter 11: Creating Custom Controls	331
■ Chapter 12: The Model-View-ViewModel (MVVM) Design Pattern	373
■ Chapter 13: Printing and Reporting	403
■ Chapter 14: Out of Browser Mode, and Interacting with the Operating System	427
■ Chapter 15: Application Deployment	483
■ Index	513

Contents

■ Contents at a Glance	iv
■ Contents	v
■ About the Author	xix
■ Acknowledgments	xx
■ Chapter 1: Introduction	1
Who This Book Is For	1
About This Book	2
About the Author	3
Silverlight Overview	4
What Is Silverlight?	4
A Short History of Silverlight	6
What Can Silverlight Bring to Your Business Application?	7
When Should You Not Use Silverlight?	8
Comparing Silverlight to Other Microsoft Platforms	9
Comparison with Adobe Flash/Flex	11
Business Applications Overview	11
Summary	12
■ Chapter 2: Getting Started with Silverlight	13
Required Tools	13
Visual Studio	13
Expression Blend 4 and SketchFlow	14

Silverlight 4 Tools	15
WCF RIA Services	15
Silverlight Toolkit.....	15
SQL Server 2008 Express Edition	15
Silverlight Spy (and .NET Reflector).....	16
Creating a Silverlight Application	16
Silverlight Application.....	17
Silverlight Navigation Application.....	18
Silverlight Business Application	18
Silverlight Class Library.....	18
WCF RIA Services Class Library.....	19
Silverlight Unit Test Application.....	19
Running and Exploring the Default Silverlight Business Application Project.....	19
Exploring the Initial Silverlight Project Structure.....	22
Project Links	24
The Web Application Project Structure.....	26
The Silverlight Application Project Structure.....	28
Recommended Project Template Modifications	31
XAP Files	32
Summary	33
■ Chapter 3: An Introduction to XAML	35
Overcoming XAML's Steep Learning Curve	35
Why Learn XAML?	36
XAML Syntax, Document Structure, and Features.....	37
Core XAML Syntax	37
Creating an Object Hierarchy.....	38
Namespaces	39
Assigning Property Values to Controls.....	41

Attached Properties	44
XAML Namespace Properties	45
Design-Time Properties	46
Markup Extensions	47
Namespaces	49
Controls	50
Base Control Classes	51
Layout Controls	53
XAML vs. Windows Forms Controls' Property Names	57
Assigning Event Handlers	58
Creating a Simple User Interface	60
Resources and Resource Dictionaries	62
Styles	64
Templates	65
Data Binding	65
Binding to an Object	66
Binding to a Collection	69
Designing User Experiences	70
The Designer/Developer Workflow	71
XAML User Interfaces: A Different Perspective	71
Summary	72
■ Chapter 4: The Navigation Framework	73
Getting Started with the Navigation Framework	73
Components of the Navigation Framework	75
The Frame Control	75
The Page Class	76
The NavigationService Object	76
The NavigationContext Object	77

Navigating Between Views	77
View URIs	77
Navigation Methods on the Frame Control	78
Navigation Methods on the NavigationService Object	78
Using a HyperlinkButton Control	79
Using the Source Property of the Frame Control	79
User-Initiated Navigation	79
Passing Data Between Views	80
Passing Data Using Query String Parameters	80
Reading Query String Parameters	81
Passing Complex Data Types Between Views	82
Deep Links	82
URI Mapping to Enable Friendly URIs	83
Integrating with the Browser History	85
Handling Navigation Events	87
Frame Events	87
View Events	88
Caching Views	89
Visual Transition Effects	90
Alternative User Interface Frameworks	91
Summary	91
■ Chapter 5: Exposing Data from the Server: Using WCF RIA Services	93
What Is WCF RIA Services?	94
How the WCF RIA Services Code Generator Works	95
How Do You Use WCF RIA Services?	96
Linking Your Silverlight and Web Projects	97
Creating Your Domain Services	97
Creating Domain Operations on Your Domain Services	97

Consuming the Domain Services.....	98
Getting Started.....	98
Creating a Data Access Layer Using the Entity Framework.....	101
Configuring the Database	101
About the Entity Framework.....	101
Creating an Entity Model	102
Domain Services.....	102
Understanding the Domain Service Life Cycle.....	103
Creating a Domain Service	104
Domain Operations	107
Query Operations.....	107
Insert/Update/Delete Operations	109
Invoke Operations.....	111
Custom Operations	111
Decorating Entities	112
Metadata Classes	113
Controlling Client Entity Generation.....	113
Validation.....	115
Presentation	118
Miscellaneous Data Annotations	118
Presentation Model Types	119
Creating the Presentation Model Class.....	120
Populating and Exposing Your Presentation Model Types.....	121
Updating Your Presentation Model Types.....	122
Sharing Code/Logic Across Tiers.....	123
Inspecting the Generated Code in the Silverlight Project	124
Encapsulating Logic in a Separate WCF RIA Services Class Library.....	125
Handling Server Errors	126

Handling Data Concurrency Violations	127
Methods of Identifying Concurrency Violations	127
Configuring the Entity Model to Check for Concurrency Violations	128
Testing Your Solution	129
Resolving Conflicts in the Domain Service	130
Transactions	131
WCF RIA Services Toolkit	132
Alternative Communication Technologies	133
WCF Services	134
WCF Data Services	137
HTTP Requests	138
Sockets	139
Summary	139
■ Chapter 6: Implementing Summary Lists	141
Exposing a Summary Collection from the Server	142
Collections and Collection Views	144
ObservableCollection<T>	144
Views	144
Consuming Data from the Server	146
Understanding the Domain Context	146
XAML-Based Approach	146
Code-Based Approach	148
Choosing the Right Approach	149
Explicitly Specifying a Domain Service Host	150
Using the BusyIndicator Control	150
Displaying the Summary List	152
Retrieving the Data from the Server	153
Configuring and Customizing a Control for Displaying Data	153

Choosing the Right Control	164
Manipulating Summary Lists	165
Querying Data Exposed by a Domain Operation	165
Filtering the Summary List	168
Sorting the Summary List	170
Grouping the Summary List	172
Paging the Summary List	175
Drilling Down on a Record	178
Opening Details in a New View	179
Opening Details in a Pop-Up Window	180
Displaying Details Using the DataGrid's Row Details	183
Implementing a Master/Details View	184
Handling Errors	184
Summary	187
■ Chapter 7: Building Data Entry Forms	189
Creating the Data Entry User Interface	189
Laying Out the Data Entry Form	189
Refining the Data Entry Form's Functionality	201
Using the DataGrid for Data Entry	212
Structuring Objects for Use by Data Entry Forms	216
Implementing the INotifyPropertyChanged Interface	216
Implementing the IEditableObject Interface	220
Adding Calculated Properties to Your Classes	222
Data Validation	223
Displaying Validation Errors	223
Types of Data Validation	226
Defining Validation Rules	227
Exposing Validation Errors to the User Interface	228

Notifying the User Interface of Object-Level Validation Errors	234
The Validator Class	235
Customizing Validation Attribute Error Messages	238
Submitting Changes to the Server	238
Change-Tracking	238
Submitting Changes via the DomainDataSource Control	239
Submitting Changes via a Domain Context	239
Handling Errors.....	240
Handling Concurrency Violations.....	242
Summary	243
■ Chapter 8: Securing Your Application.....	245
Implementing Server-Side Security Using RIA Services.....	245
The ASP.NET Membership API.....	246
Configuring Your Database for Forms Authentication	247
Authentication	249
Requiring Authentication	250
Role-Based Operation Access Restrictions	251
Returning a Subset of Data Based on the User's ID or Role	253
Permitting Update/Delete Operations According to Data.....	253
Exposing Custom User Profile Data	253
User Registration	254
Avoiding SQL Injection Attacks.....	255
Sanitizing Errors Sent to the Client.....	256
Implementing Client-Side Security	256
Authenticating and Registering Users	256
Accessing and Updating User Information	259
Implementing Client-Side Restrictions	260
Storing Data Locally.....	262
Encrypting Data Passed Between the Server and the Client	262

Cross-Domain Access Policies	263
Implementing Cross-Domain Policies for HTTP-Based Communication.....	264
Implementing Cross-Domain Policies for Socket-Based Communication	265
Implementing Cross-Scheme Access Policies	265
Restricting Access to Your Application	266
Summary	268
■ Chapter 9: Styling Your Application	269
The Designer/Developer Workflow	269
Defining Style Resources.....	271
Defining a Style Resource	272
Defining Style Resources at Various Locations	273
Naming Style Resources	275
Inheriting Style Resources.....	276
Styling Constraints	277
Control Templates.....	277
Default Control Templates	278
Templating a Control	278
Control Template Structure	279
Theming.....	282
Creating a Custom Theme	282
Silverlight Toolkit Themes	283
Silverlight Navigation/Business Application Themed Project Templates	283
Icons and Images.....	283
Animation.....	284
Pixel Shaders / Effects.....	284
Miscellaneous Styling Tips	285
Defining Constants in XAML	285
Restoring a Control's Default Style.....	286

Summary	286
■ Chapter 10: Advanced XAML and Data Binding	287
Advanced XAML	287
Comments in XAML	287
Defining Constants	288
Using OR to Combine Enumerated Values in XAML	289
Triggers, Actions, and Behaviors	289
Blendability	299
Creating Consolidated Namespaces	301
MoXAML Power Toys	303
Advanced Data Binding.....	303
Assigning the Source of a Binding.....	304
Binding to a Resource	308
Defining Resources in the Code-Behind (For Binding To).....	311
Binding to Nested Properties	311
Binding to Indexed Properties	312
Binding to Collection Views	312
Enhancing Data Binding	317
Data Binding in Code	323
Getting and Setting Attached Property Values in Code.....	324
Additional Tips	324
Summary	329
■ Chapter 11: Creating Custom Controls	331
Adding Functionality to an Existing Control	331
Creating User Controls.....	332
Creating a Simple User Control	332
Exposing Properties.....	333
Exposing Methods	345

Exposing Events	346
Determining If in Design-Time or Runtime Mode	347
Constraining the User Control's Size	347
Creating Custom Controls	347
Creating the Custom Control	348
The Control Structure	349
Defining the Control's Default Template.....	350
Defining the Control's Behavior	359
Content Controls	365
Containing a Single Control as Content	365
Containing Multiple Controls as Content	367
Attached Properties	369
Summary	370
■ Chapter 12: The Model-View-ViewModel (MVVM) Design Pattern.....	373
The Purpose of Implementing MVVM.....	374
Why You Should Implement MVVM	375
MVVM Theory.....	376
The Layers	376
View / View Model Configurations	379
MVVM in Practice.....	380
Creating the Layers	380
Connecting the Layers	383
View and ViewModel Interactions	385
Layer Interaction Summary	392
Hotly Debated Implementation Details	393
MVVM and RIA Services.....	395
Implementing the Model.....	395
Implementing the View Model	396

Implementing the View	399
Frameworks	401
Summary	401
■ Chapter 13: Printing and Reporting	403
The Importance of Implementing Reporting	403
Potential Solutions for Implementing Reporting	404
Generating a PDF	404
Generating HTML	405
Generating Office Documents	405
Using Silverlight 4 Printing Functionality	405
Third-Party Report Viewers	405
Office Integration Using COM	407
Choosing a Reporting Strategy	407
Printing Functionality in Silverlight	407
Generating and Displaying a PDF Report	409
Generating a Report on the Server	409
Displaying the Report on the Client	418
Summary	426
■ Chapter 14: Out of Browser Mode, and Interacting with the Operating System	427
Out of Browser	427
Configuring OOB Mode	428
Installing the Application to Run Out of Browser	429
Determining Installation Status	431
Determining Whether Running Inside or Outside the Browser	432
Where/How is the Application's .xap File Stored and Run?	432
Interacting with the OOB Window	433
Checking for Updates	434

Debugging Out of Browser	436
Uninstalling the Application.....	436
Toast Notifications.....	437
Caching Data Locally	439
Caching to Isolated Storage.....	439
Caching to a Client-Side Database	449
Detecting Network Connection Availability	451
File System Open/Save Dialogs	451
The Open File Dialog.....	452
The Save File Dialog	456
Drag Drop Target	459
Clipboard	460
Full Screen.....	462
Initiating Full Screen Mode.....	462
Detecting the Switch to and from Full Screen Mode.....	463
Retaining Full Screen Mode When Unfocused.....	463
Keyboard Access	464
Elevated Trust.....	464
Enabling Elevated Trust.....	465
File System Access.....	466
COM Automation.....	471
Custom Chrome	478
Other Restrictions Lifted By Elevated Trust.....	480
Restrictions Imposed by Elevated Trust	481
Disabling Elevated Trust Applications with a Windows Group Policy	481
Summary	482
■ Chapter 15: Application Deployment	483
Deploying the Application to the Server	483

Server Requirements	484
Using Xcopy	485
Publishing	486
Creating a Web Package	487
Creating a Setup Program	488
Deploying the Application to the Client.....	488
The Default Silverlight Installation Process	489
Customizing the Silverlight Installation Experience	490
Pushing the Silverlight Runtime to Users in the Enterprise.....	493
Building a Client Desktop Installer.....	494
Improving on the Default Application Loading Screen.....	495
Creating the Application Pre-loader Files	496
Designing the Application Pre-loader	497
Updating the Application Download Progress	499
Configuring the Application Pre-loader in the HTML File.....	500
Testing the Application Pre-loader	500
Partitioning Your Application	502
Initial Steps.....	502
Assembly Caching	502
Downloading Modules on Demand	505
Digitally Signing Your Application.....	506
Summary	511
■ Index	513

About the Author

■ **Chris Anderson** has been a professional developer for over 10 years, specializing in building desktop, web, and mobile business applications using Microsoft technologies for industries as wide ranging as accounting, property valuation, mining, the fresh produce industry, logistics, field services, sales, and construction. He holds a Bachelor of Engineering in Computer Systems with a Diploma in Engineering Practice.

Chris is now a co-owner of Peer Placements (a specialist recruitment firm for software developers), in addition to working as a consultant, author, and Australian representative of the SilverlightShow.net website. Currently specializing in Silverlight Chris has spoken on this topic at conferences such as Code Camp Australia, TechEd Australia, Silverlight Code Camp Australia, REMIX Australia, and numerous Sydney Silverlight Designer and Developer Network (SDDN) meetings (for which he is a co-organizer). Chris maintains a blog at <http://chrisa.wordpress.com>, and can be found on Twitter at <http://twitter.com/christhecoder>.

Acknowledgments

I had no idea what I was getting into when taking on this book. Writing a book is a long, arduous task that consumes your entire life, but now that this book is complete I would like to thank those who were instrumental in helping it come to fruition.

I'd like to start by thanking the members of the Apress team with whom I worked closely throughout the writing process. I'm grateful for their guidance and encouragement. I'd like to thank Jonathon Hassell and Anne Collett, and in particular Tracy Brown, who was extremely patient with me even as deadlines were rapidly approaching. My great thanks also go to my technical editor Ilia Iordanov, who, with an eagle eye, spotted many issues I had missed, and made some great suggestions.

I would especially like to thank Greg Harris (www.GregMHarris.com) who gave his time to review every chapter in its draft form, and who provided me with extensive feedback. I really appreciate the amount of time and effort Greg voluntarily spent on this project. His contributions are invaluable.

I would also like to thank Miguel Madero and Pencho Popadiyn, who provided additional valuable feedback on the Model-View-ViewModel Design Pattern chapter, and Sergey Klementiev for providing a solution to a problem I had with RIA Services. My thanks also go to all the Silverlight bloggers who provided solutions to difficult problems. I've credited you in the text when you've provided information that helped me along my way.

From a personal perspective, I'd really like to thank my sister Michelle, chef extraordinaire, for providing me with home-cooked frozen meals that kept me decently fed for a month when I was working furiously on this book and barely leaving the house. It was a huge help. I'd also like to thank Mark Wallis, who covered for me with my clients whilst I had my head down writing. I hope they'll forgive my absence.

CHAPTER 1



Introduction

Silverlight is a web-based platform from Microsoft that can be used to build rich Internet applications (RIAs). Of the many new possibilities it enables, one receiving a lot of attention is its huge potential as a platform for building business applications. Despite its relative youth (having just reached its third birthday), it has achieved remarkable maturity, with its development progressing at a rapid rate—now at version 4. With just nine months separating versions 2 and 3, and a further nine months separating versions 3 and 4, its tremendous growth in that short time demonstrates the enormous amount of effort and commitment that Microsoft is pouring into Silverlight to bring it to the forefront as one of its core development platforms. Community support for the technology has been overwhelmingly positive, gaining more and more momentum as companies increasingly choose it over more mature platforms. This ensures its future as a viable platform for delivering applications that you can be sure will only grow and become ever more popular.

If you have previously only been exposed to Windows Forms and ASP.NET development, then it will no doubt be a brave new world and one with a steep learning curve, but ultimately the benefits make the initial effort worthwhile. However, with this book to help you through your journey, you should find that a lot of pain will be taken out of this learning curve, guiding you in the right direction while you build your first business application in Silverlight.

Who This Book Is For

This book covers not just Silverlight as a technology, but also the important concepts in designing and developing business applications. From structuring your application to actually implementing features, it covers most of the lifespan of developing a business application in Silverlight.

To follow along with this book, you should know your way around Visual Studio well, have a reasonable degree of familiarity with C# and its features (such as generics and LINQ, although experience with a similar language such as Visual Basic or Java may suffice), be comfortable with designing and querying databases, and have a good understanding of object-oriented design (OOD). Some prior business application development experience (or understanding of what building a business application entails) is an advantage, but isn't absolutely necessary, as I will discuss the reasoning behind many of the concepts unique to business application development as needed. However, you may need to refer to alternative sources for more information when the concepts are particularly complex and detailed. For example, a good understanding of design patterns (such as those described in the influential book *Design Patterns*, by the authors typically referred to as the Gang of Four [Addison-Wesley, 1994]) is important when designing business applications, but complete coverage of these in the context of designing Silverlight applications is really a whole book in its own right.

About This Book

This is not a book simply about Silverlight as a technology, and it isn't about covering Silverlight from end to end. Silverlight features such as streaming video, Deep Zoom, animations, sounds, and others that are not generally a core requirement in business applications will not be covered. This is a book specifically targeting the needs of designing and developing business applications, and how to do so when using Silverlight as a development platform. That said, you may be well versed in the process of business application design and merely be interested in how to apply your knowledge to developing business applications in Silverlight—this book caters to you too. You may even pick up a few things as you go!

You may have noticed that many books, web articles, and presentations take a rather simplistic view when discussing building business applications in Silverlight, and these techniques rarely hold up to scrutiny in real-world projects. Instead of just giving you all the pieces and leaving you to fit them together yourself, this book is designed to act as your guide when creating your own business application, leading you through the difficult problems you will encounter and providing one or more solutions for each problem. Business applications involve more than simply retrieving some data and displaying it on the screen—this book will allow you to peer into all the nooks and crevices that represent good business and application design practices.

This book is designed to be read from start to end rather than simply being a reference guide. Concepts will be introduced gradually to save overwhelming you with too much information at one time. This means that when some concepts are introduced, we will take shortcuts in other areas in order to focus your attention on the concept being taught, and move on to more robust implementations in subsequent chapters. Therefore, some of the Silverlight-specific lessons in early chapters are not designed to be put into immediate use, as they won't necessarily reflect all the recommended development practices in real-world projects, but are provided merely as a means of easing you into the world of Silverlight development.

■ **Note** For example, the Model-View-ViewModel (MVVM) design pattern commonly used in Silverlight/WPF business application development is not covered until the later parts of the book. You will probably want to use this pattern in your applications, but the earlier chapters forgo this pattern in their examples as it introduces a lot of complexity that will potentially confuse you when attempting to focus on and convey other important concepts. Despite its position in the later parts of the book, incorporating this design pattern into your project should not be an afterthought. Therefore, this book should not be considered a step-by-step guide, but an all-encompassing guide to all the concepts important to building business applications in Silverlight.

By the completion of this book you will have learned most of the techniques for designing robust business applications and how to implement these techniques with a Silverlight-based project. While it would be impossible to cover each and every concept that could be employed when designing and developing business applications, this book will cover those that are the most important for developing a well-structured business application.

Unfortunately, it simply isn't possible to explore the complete myriad of scenarios that you may encounter in your application requirements, nor is it possible to cover every possible solution to a problem. The primary decision that needed to be made when writing this book was to decide on a single means for communicating between the server and the client. Due to the wide array of means for a

Silverlight application to communicate with a server, it would be simply impossible to cover them all and do them justice. For this reason I chose to cover one technology in depth. For most end-to-end business applications in Silverlight (which is the primary focus of this book), RIA Services is your best option for handling communication between the Silverlight client and the server, and therefore this book will focus on using that technology as the framework for this task.

■ **Note** If you decide RIA Services is not suitable for your needs, don't despair—the functionality covered in this book is not solely dependent on RIA Services, and there should still be large amounts of information useful to your project.

Note that you don't need to use every concept taught in this book in your own applications, as at times some of the concepts may be excessive (depending on the size and type of project you are working on) or inappropriate. These concepts will be noted with the scenarios that they are most suitable for, in order to help you make an informed choice.

About the Author

My name is Chris Anderson and I will be guiding you on your journey building business applications in Silverlight. Throughout my career as a developer I've designed and built many business applications across multiple platforms and a number of industries, including construction, finance, sales, logistics, and field services, to name just a few. However, despite all my development experience, I still faced a steep learning curve when I first came face to face with Silverlight. Back then, Silverlight was still in its infancy, but I could see its future potential as a platform for applications of all types—especially in the area of business application development. It had a small runtime, could be easily deployed, had a lot of exciting potential for rich user experiences, and best yet, had the power of the .NET Framework behind it (albeit a subset, but a substantial subset).

While it wasn't a vision shared by many people at the time, there was quite an interest in the topic when I started writing about it. I published a series of articles on the topic on SilverlightShow (www.silverlightshow.net) (for Silverlight 2 beta 2, and then Silverlight 2 RTW), but it was still somewhat hard going, with a lot of hurdles to jump and issues to work around. However, it was obviously a market that Microsoft saw potential in too, and with the beta release of Silverlight 3 there was suddenly a big focus on support for developing business applications (making much of my previous work redundant in the process). The vision had suddenly become reality.

Developing business applications in Silverlight has become substantially easier since those days, but you will still face a steep learning curve that will make getting up to speed a little slower than you might like (primarily in order to effectively use XAML to design your user interface). I've specifically structured and written this book to help you through that learning curve, guide you in the right direction, and solve the problems you are likely to encounter, so you can be working and productive in minimal time.

I welcome any and all feedback (preferably constructive), and you can contact me via my blog at <http://chrisa.wordpress.com> or on Twitter at @ChrisTheCoder.

Silverlight Overview

Microsoft has a number of different platforms for building applications, with Silverlight being the newest addition to the family. Let's take a look at what Silverlight is and how it fits into the big picture of software development platforms.

What Is Silverlight?

It's important at this stage to establish exactly what Silverlight is and what it isn't. Silverlight is an RIA platform. RIAs could be considered a cross between desktop applications and web applications. They run within the user's web browser (with some exceptions), and use a browser plug-in to run the application within a sandboxed environment. RIAs are not a new concept, although not all have been a success for various reasons. Adobe Flash is a notable exception, having achieved huge popularity over the last decade, as well as a market penetration of 99 percent of Internet-enabled personal computers in mature markets and 98 percent in the enterprise (according to Adobe's own statistics available here: www.adobe.com/products/player_census/flashplayer).

You will find a number of parallels between Silverlight and Adobe Flash. Both run as browser plug-ins, providing a platform for self-contained applications running on the client (as opposed to being dependent on a server controlling the application). Of course, there are numerous differences between the two technologies, but if you take a look at the space in which Silverlight is competing, Adobe Flash (combined with Adobe Flex) is without doubt Silverlight's biggest competitor.

Market Penetration/Reach

Although Adobe Flash has an almost ubiquitous market penetration, Silverlight is rapidly catching up. Scott Guthrie (corporate vice president, .NET Developer Platform) announced at PDC (in November 2009) that the market penetration was at 45 percent of all Internet-connected devices in the world (note that this is a broader measure than that of Adobe's), and at the launch of Silverlight 4 in April 2010 he announced that it was now approaching 60 percent. For in-house business applications, the issue of Silverlight's market penetration is of little concern, as the Silverlight runtime can be incorporated into the organization's standard operating environment (SOE) and deployed automatically to all the machines throughout the organization. However, for public-facing applications, you should be aware that not all client machines will have the Silverlight runtime installed. Deploying both the Silverlight runtime and your application effectively to users is detailed in Chapter 15.

■ **Note** Another good source of statistics comparing the two platforms is www.riastats.com.

Focus

Silverlight's initial focus was toward media applications (directly targeting Adobe Flash's biggest market)—particularly in the area of video, vector graphics, and animations. However, it has more recently been focusing heavily on business applications, and is arguably the most powerful of the competing RIA platforms in this respect. With advanced support for data binding, a rich set of standard

and supporting controls, and the RIA Services framework, it has also become a viable platform for many business applications.

Compatibility

The Silverlight browser plug-in works with all major browsers and across the two most popular operating systems—Microsoft Windows and Apple Macintosh (Intel x86–based machines only). The Moonlight project (by the team from Novell working on Mono, a cross-platform open source implementation of the .NET Framework) can be used to run Silverlight applications on Linux, but unfortunately its releases lag behind the Microsoft releases.

At the time of writing, there is currently no support for mobile operating systems such as Windows Mobile (which is superseded by the Windows Phone 7 mobile operating system), Apple’s iPhone, Symbian (for Nokia devices), BlackBerry, or Google’s Android, although there are future plans to provide support for some of these devices.

Windows Phone 7

The Windows Phone 7 operating system has a Silverlight-based user interface, meaning that most applications for it will be written in Silverlight. Applications need to be written specifically to run under this operating system, as it won’t run standard Silverlight applications (i.e., you won’t be able to browse to an application in the web browser and run it). Applications written for the phone are essentially Silverlight 3 based; however, this should be enough to enable you to port much of the same code you write for your business applications targeting Silverlight 4 to a Windows Phone 7 version at some point in the future. This will open up many exciting new possibilities for sharing a large portion of your code base (and development knowledge) between both your standard and mobile applications.

The Silverlight Runtime

The Silverlight runtime is a 6MB download (small by today’s standards) including a subset of the .NET Framework and a full implementation of the Common Language Runtime (CLR) engine. This has the huge advantage of enabling you to write code in any .NET language, including (but not limited to) C# and Visual Basic—meaning that you can develop applications in exactly the same language that you are used to working with. Application code is compiled into assemblies, which (along with XAML files and other resources) are combined into a XAP file. This file can then be downloaded to the client and executed within the plug-in. Silverlight applications run within the browser are sandboxed (having extremely limited access to the host machine), meaning that Silverlight applications are inherently safe to be run and cannot cause any damage to the user’s computer (by design).

■ **Note** Silverlight applications cannot make use of assemblies that target the full .NET Framework, but assemblies targeting Silverlight can be used in applications that target the full .NET Framework.

Out-of-Browser (OOB) Experiences

While Silverlight applications are generally run from within a browser, from version 3 of Silverlight onward they can be run outside of the browser as if they were standard Windows or Macintosh applications. An icon can be placed on the desktop and/or the Start menu to open the application, and the application can automatically update itself when a new version is available on the server. The original implementation of OOB used the same sandboxing model as running the application in the browser, but in version 4 of Silverlight a new feature was added to allow the application to be installed with elevated trust permissions. This enables the application additional access to the host machine, and opens up a new range of possibilities to your Silverlight applications (when the application is being run in OOB mode). With trusted applications in Silverlight 4, you also have access to the Component Object Model (COM), which being a Windows-only feature breaks the truly cross-platform nature of the platform, but enables you to access many Windows features and integrate more tightly with other applications and the operating system.

In the .NET world, Silverlight's closest relative would be Windows Presentation Framework (WPF). Both use Extensible Application Markup Language (XAML) to define their user interfaces, and Silverlight essentially started as a web-deployed subset of WPF. If you are familiar with WPF, you will be able to write Silverlight applications reasonably easily (and vice versa), despite there being a number of features in each that are not available in the other (although they are becoming more aligned with each release of each).

A Short History of Silverlight

Silverlight has matured rapidly in its short three-year lifespan. Let's take a look at how it has grown in that time.

Before there was Silverlight, there was WPF. Introduced with version 3 of the .NET Framework, it targeted the rich client market, pioneered the use of XAML to define user interfaces, and demonstrated the new possibilities that XAML could enable. WPF applications could be deployed via a web browser (as an XBAP), but they required the full .NET Framework to be installed, and could only be run under Windows.

Silverlight 1: September 2007

Microsoft then started working on a project called WPF Everywhere (WPF/E), which was essentially designed to be a web-deployed subset of WPF that had its own runtime and ran within a browser plug-in. It used XAML to define the user interface and required you to write code in JavaScript. It had little in the way of controls, and was predominantly focused on media-type applications (video, animations, and vector graphics). This became Silverlight 1.

Silverlight 2: October 2008

With Silverlight 2, developers really started to become interested in Silverlight. It now had the CLR, a subset of the .NET Framework, and a decent set of standard controls, enabling developers to start writing rich Internet applications without needing to learn a new language.

Silverlight 3: July 2009

Silverlight 3 began to focus its use on developing business applications. It included richer data binding support, the DataForm control, validation controls, and the ability to run Silverlight applications OOB. In addition, although it was in beta, Microsoft simultaneously released a new framework called RIA Services, which provided a powerful and structured means for Silverlight applications to communicate with servers and pass data between them—making business applications easier to develop in Silverlight.

Silverlight 4: April 2010

Whereas Silverlight 3 immediately made creating business applications viable in Silverlight, Silverlight 4 smoothed many of the rough edges. Silverlight 4 introduced printing capabilities (resolving a big complaint by business application developers of a missing feature), implicit styling (making theming your application a lot easier), webcam and microphone support, elevated trust when running outside the browser, COM access, toast notifications, additional data binding capabilities, and more.

What Can Silverlight Bring to Your Business Application?

There are numerous platforms you could develop your business applications upon, so you may be wondering why you should develop business applications in Silverlight. The biggest benefits are undoubtedly the following:

- The deployment model is easy—applications are delivered over the web via a web browser.
- There's no need for the full .NET Framework—just a very small runtime.
- Applications can be written in your favorite .NET language (C#, Visual Basic, etc.).
- Applications can be run on multiple platforms (namely both Windows and Mac).
- It's faster to develop and richer than HTML-based applications.

Combined, these benefits have created a platform that many developers hope can solve the various problems that they face every day. However, there are many more benefits that can be gained using Silverlight over other technologies.

If you are used to developing HTML-based applications (e.g., using ASP.NET), then you'll gain the following benefits:

- You are no doubt used to having your applications render differently between browsers and operating systems. The big advantage of an RIA like Silverlight is that it renders in *exactly the same way* in each browser and operating system. This saves a lot of time testing and solving cross-browser issues.
- Silverlight applications are richer than HTML applications and faster to develop.
- Since Silverlight is a rich client, users don't have to deal with constant postbacks to the server, and the applications feel more responsive to the user.
- With the CLR and a subset of the .NET Framework running on the client, you don't need to write in JavaScript, and you can reuse code from other projects in your Silverlight application.

- Applications can be run offline.
- Applications can be detached and run like a standard application.

If you are used to developing rich client applications (such as with Windows Forms), then you'll gain the following benefits:

- Silverlight's use of XAML to define user interfaces enables you to create unique and flexible user interfaces that have extensive support for data binding, vector graphics, and animations.
- RIA Services makes designing applications that communicate with a server clean and easy.
- You will be able to share code with Windows Phone 7 and WPF applications.
- Users can be set up with both the Silverlight runtime and your application with ease in a matter of minutes.

As you can see, Silverlight brings a lot to the table, and this book intends to demonstrate how it can lead to brilliant business applications. However, Silverlight is not always the best choice in some scenarios—let's take a look at those now.

When Should You Not Use Silverlight?

It's important to choose the technologies you use wisely and not just choose a technology because it's "cool" or for the sake of it—you need to justify your choice and ensure that the platform will meet the needs of the business, and that the business is obtaining a reasonable return on investment (ROI). As with any technology, Silverlight is not the perfect platform for all development projects. It has strengths and weaknesses that need to be weighed, evaluated, and considered in relation to the problem domain before you commit yourself (and your business!) to that technology. You need to select the most appropriate platform to suit the requirements of the project.

Comparing Silverlight Applications to HTML-Based Applications

In comparison to HTML-based applications, a big issue is Silverlight's reach. HTML-based applications can be run almost everywhere. Desktop PCs (including Linux) and Internet-capable mobile devices all display HTML (although a little differently at times), meaning that the reach of these applications is almost 100 percent. For machines that *can* run Silverlight (Windows, Macintosh, and to an extent Linux), the user is required to install a plug-in, which isn't always possible on locked-down machines in corporate settings (without getting the IT department involved).

Comparing Silverlight Applications to Rich Desktop Applications

In comparison to rich desktop applications, Silverlight applications have restricted functionality and features. Only a subset of the .NET Framework is available to Silverlight applications (although it is a fairly reasonable subset), and the capabilities and features of the controls are somewhat limited too, so you will find that some things tend to be more difficult (or impossible) to do in Silverlight than in the desktop counterpart. Silverlight is also not designed for integrating with other applications, the underlying operating system, or hardware devices. Silverlight applications are designed to be

sandboxed, and although the elevated trust and COM features in Silverlight 4 enable you to get around many of these limitations, you do need to run the applications outside of the browser (with elevated trust) in order for those workarounds to be possible.

These are the primary considerations you will have to take into account before choosing Silverlight as a platform. Determine the current and potential future requirements of your application, identify any features that conflict with Silverlight's limitations, and then decide whether Silverlight is really the best-suited platform for your needs.

Comparing Silverlight to Other Microsoft Platforms

Let's compare Silverlight with the other core platforms Microsoft provides and identify the primary pros and cons of each.

WPF

Shared features:

- Both use XAML for defining user interfaces (with Silverlight essentially a subset of WPF).
- Both can be deployed via a browser as sandboxed applications.

Pros:

- WPF has minimal restrictions, and it requires no sandbox (unless deployed via a browser as an XBAP without full trust turned on).
- WPF has the power of the full .NET Framework behind it.

Cons:

- WPF requires the full .NET Framework to be installed on the machine.
- WPF is limited to running on Windows only.
- WPF does not have the hype and community/Microsoft support that Silverlight has.

Windows Forms

Shared features:

- Both can be downloaded via a browser (although Windows Forms applications cannot be run within the browser like Silverlight applications) and update themselves automatically when a new version is available (using click-once with Windows Forms applications).

Pros:

- Windows Forms has minimal restrictions and no sandbox.

- It has the power of the full .NET Framework behind it.
- It is very easy to develop (no need to understand XAML).
- It is mature, having been around since the inception of the .NET Framework.
- It requires no graphical skills to design user interfaces with the standard style of Windows applications.

Cons:

- It requires the full .NET Framework to be installed on the machine.
- It is limited to running on Windows only.
- It is an aging platform with waning support.
- Data binding capabilities aren't as rich.
- User interfaces aren't as flexible and don't have the same styling capabilities.

ASP.NET

Shared features:

- Both ASP.NET and Silverlight are available via a web browser (for Silverlight applications, the Silverlight plug-in must be installed first).

Pros:

- ASP.NET runs on almost all Internet-connected devices.
- It does not require a plug-in to be installed.
- It has the power of the *full* .NET Framework behind it (server side only).
- It is mature, having been around since the inception of the .NET Framework.

Cons:

- It requires the full .NET Framework to be installed on a Windows server running IIS (although the Mono project provides a potential alternative).
- Client-side code must be written in JavaScript, requiring learning a different language.
- It can render differently between browsers, requiring a lot of testing.
- Data binding capabilities aren't as rich.
- It isn't as rich or responsive (although Ajax and jQuery can help).
- It cannot run with elevated trust to integrate with other software and hardware.

Comparison with Adobe Flash/Flex

With Adobe Flash/Flex being Silverlight's biggest competitor, it's worth comparing the two (at a high level).

Similar features:

- Both are rich Internet application platforms deployed via a web browser.
- Both can run applications outside of the browser (Silverlight with no additional requirements, Flash with Adobe Air).
- Both use XML-based markup languages to define their user interfaces.

Pros:

- The Flash runtime is almost ubiquitous, installed on almost all Internet-capable PCs.
- The core Flash runtime is a smaller download (1.8MB vs. 6MB), although running outside the browser requires the Adobe Air runtime (which is 15MB).

Cons:

- Code must be written using ActionScript, a JavaScript derivative, which is not as capable or popular as managed code languages (e.g., there's no equivalent to LINQ and generics), and is specific to Flash/Flex (i.e., code can't be reused with other applications).
- It requires an additional download to run applications outside the browser (Adobe Air), and applications must be packaged separately to run in this runtime (whereas the same Silverlight application will run both within and outside the browser).

Business Applications Overview

Business applications—sometimes also referred to as line-of-business (LOB) applications—are core applications critical to the running of a business. Businesses the world over rely on software in order to function, much of which has been custom developed to capture and support the unique processes and needs of the business—without this software, they would not be able to function effectively, or even at all.

Business applications tend to have the following features in common:

- The ability to access and maintain data in a central data store
- Server/client communication
- A workflow providing a structured means of entering/modifying that data
- A user interface providing an efficient means of entering/modifying data
- Business rules

- Data validation rules
- The ability to obtain information from the system (via reports)
- Security

Each of these facets of business applications will be investigated in this book, with information on how to implement each in Silverlight to produce a complete end-to-end business application.

Summary

Now that we've discussed both Silverlight and business applications separately, it's time to put the two together and start learning how to build business applications in Silverlight.

CHAPTER 2



Getting Started with Silverlight

To start developing with Silverlight, you need to set up your development environment with a number of tools that enable you to develop for Silverlight, and install all the related programs and tools that are typically required for a standard business application project. This chapter will walk you through getting your development environment set up so you can start developing for Silverlight. Then we'll take a look at the project templates that are added to your development environment, and finally we'll explore the structure of a typical Silverlight application.

Required Tools

The best place to find all the tools you need to get started with Silverlight is the Microsoft Web Platform Installer 2.0, which you can download from here: www.microsoft.com/web/downloads/platform.aspx. You will find the tools that you require for Silverlight development in the Tools category on the Web Platform tab.

Alternatively, you can visit the Get Started page on the official Silverlight web site, at <http://silverlight.net/GetStarted>, which has links to the required downloads. The following subsections give a breakdown of the various tools required (or recommended) for Silverlight development, and what each does.

■ **Note** The following sections give you a starting point for what you should have installed in order to begin developing Silverlight applications; however, these are not all the tools you'll eventually need—additional tools will be introduced as required throughout the book.

Visual Studio

Theoretically, you don't need an integrated development environment (IDE) to develop Silverlight applications, but that scenario is only for the most hardcore of developers. Any sane developer will use an IDE, and Visual Studio is arguably the best IDE available. Potential IDEs that support Silverlight development range from free upward, and their functionality and ease of use is generally proportional to the amount they cost (i.e., if you have the latest professional edition of Visual Studio, you will have the most functionality available to you and be far more productive). The Professional edition (or higher) of Visual Studio 2010 is your best option by far as an IDE for Silverlight development, as it has no limitations to constrain you, but there are some free alternatives if you wish to take that path, with the best option currently supporting Silverlight 4 development being Visual Web Developer 2010 Express.

However, for the sake of simplicity, any instructions throughout this book will refer to the action to be taken in the professional edition of Visual Studio 2010.

■ **Note** Note that Visual Studio 2008 does not support Silverlight 4 development—you will need to upgrade to Visual Studio 2010 if you haven't done so already.

Expression Blend 4 and SketchFlow

Expression Blend could be described as a designer for XAML user interfaces and animations, although its capabilities have been extended from version 3 onward to include the ability to edit code. You could technically build an entire Silverlight application using Expression Blend only, but it's not designed to be used as such, and Visual Studio is much better suited for this purpose (Expression Blend is designed to be used primarily to target the user interface design).

Expression Blend is targeted toward designers (i.e., user interface, user experience, and graphics designers) as its users (rather than developers), with a focus on helping support and enhance the developer/designer workflow. This book is primarily focused on a developer audience; therefore, we'll spend most of our time in Visual Studio when designing user interfaces. Designing user interfaces is much easier in Visual Studio 2010 than it was in Visual Studio 2008, as Visual Studio 2010 now has a much better user interface designer than was previously available (which was essentially no designer at all). However, it is not as feature rich or powerful as what you'll find in Expression Blend (particularly when it comes to creating animations). If you're serious about creating advanced user interfaces, Expression Blend will be a very useful tool in your arsenal.

SketchFlow is a prototyping tool that you can use in designing your application (mapping out the flow of your screens and their layouts), from which a Silverlight (or WPF) project can automatically be generated. SketchFlow is a part of Expression Blend and targets user experience designers.

Expression Blend is not a free tool—it must be purchased as a standalone product or as a part of Expression Studio. There is a 60-day trial available from the Microsoft web site if you want to trial the tools before committing your hard-earned money. SketchFlow is bundled as a feature of Expression Blend. Expression Design (also a part of Expression Studio) can also be a useful tool to have (as a vector drawing program) as well, but neither tool is essential for Silverlight development.

■ **Note** Both Visual Studio and Expression Studio are included in the MSDN Premium and the Expression Professional subscriptions (the Expression Professional subscription is a new type of MSDN subscription specifically targeting designers and web developers), which you may find to be the most cost-effective means of obtaining these tools. These subscriptions also include licenses for Office, Windows, and various other Microsoft software products, and are great value for your money. Startup software businesses less than three years old can take advantage of the Microsoft BizSpark program that provides you an MSDN subscription (with all these tools) and numerous other benefits for \$100 when you *leave* the program—a fantastic way to get the tools for no initial investment. Details and eligibility requirements can be found at www.microsoft.com/bizspark.

Silverlight 4 Tools

Visual Studio 2010 comes with support for Silverlight 3 out of the box, but you will need to install Silverlight 4 Tools in order to develop Silverlight 4 applications. The Silverlight 4 Tools download includes the Silverlight 4 SDK, the Silverlight 4 developer runtime, WCF RIA Services, and support for Silverlight 4 in Visual Studio. Once these tools are installed, you will find new project templates available in Visual Studio for creating Silverlight applications.

WCF RIA Services

WCF RIA Services provides a prescriptive pattern and framework for creating end-to-end Silverlight applications. The WCF RIA Services code generator “projects” your business object model on the server onto your Silverlight application along with shared business logic, and it provides the infrastructure that enables methods on the server to be called by your Silverlight application. We’ll take a thorough look at WCF RIA Services in Chapter 5. This is now included as a part of the Silverlight 4 Tools installer, but there is an associated toolkit that you can install with additional features (most of which are still under development) if you wish.

Silverlight Toolkit

The Silverlight Toolkit is a free and open source project hosted on CodePlex, developed and maintained by Microsoft. This is an ongoing project with regular releases that adds greatly to the available controls that you can use in your Silverlight project. The Silverlight 4 SDK only contains basic user interface controls (TextBox, Button, Canvas, Grid, etc.), so the Silverlight Toolkit provides a raft of more advanced controls, such as TreeView, Calendar, and Charting, among many others.

Blacklight is a similar open source project hosted on CodePlex, but unlike the Silverlight Toolkit it is a community-run project and does not attempt to overlap with the controls available in the Silverlight Toolkit. It’s not listed on the Silverlight.net Get Started page, but you can get it from here: www.codeplex.com/blacklight.

SQL Server 2008 Express Edition

Business applications (as described in Chapter 1) are data-centric applications—hence they (almost always) interact with some sort of database. You have unlimited options of databases to choose from—the database you choose has no bearing on your Silverlight application, as Silverlight applications can’t read or write to a remote database directly—you need to provide a set of services to expose the data from the server to client applications. As long as you can communicate with the database in a standard .NET application, you can provide a way to allow your Silverlight application to communicate with it. For the purposes of this book, we’ll be using SQL Server 2008 Express Edition as our database of choice, since it’s free and reasonably full featured. In most business settings you would probably want to use the full SQL Server 2008 product, with which the code provided will work without any changes. Earlier versions of SQL Server would be acceptable too. You can download SQL Server 2008 Express Edition from www.microsoft.com/express/sql/download. Ensure (at a minimum) that you choose the Runtime with Management Tools option.

Silverlight Spy (and .NET Reflector)

This is not a mandatory tool to obtain, but it's certainly recommended, as it is extremely handy when debugging your Silverlight application. Silverlight Spy, created by Koen Zwikstra of First Floor Software, allows you to inspect the internals of a Silverlight application (including its file structure, XAML visual tree, and styles), monitor its memory and processor usage, and monitor its network connector (among many other features). It also integrates with .NET Reflector (a free tool from Red Gate Software) to decompile the assemblies within the XAP file and view their code. From a debugging perspective it's an invaluable tool. Early versions of Silverlight Spy were free, but with its version 3 release it became a commercial product with a price attached. You can download Silverlight Spy from <http://silverlightspy.com/silverlightspy> and .NET Reflector from www.red-gate.com/products/reflector.

Creating a Silverlight Application

Once you have your development environment configured, you can begin creating your first Silverlight project. Open the New Project window in Visual Studio and take a look at the new Silverlight project templates by clicking the Silverlight folder (shown in Figure 2-1).

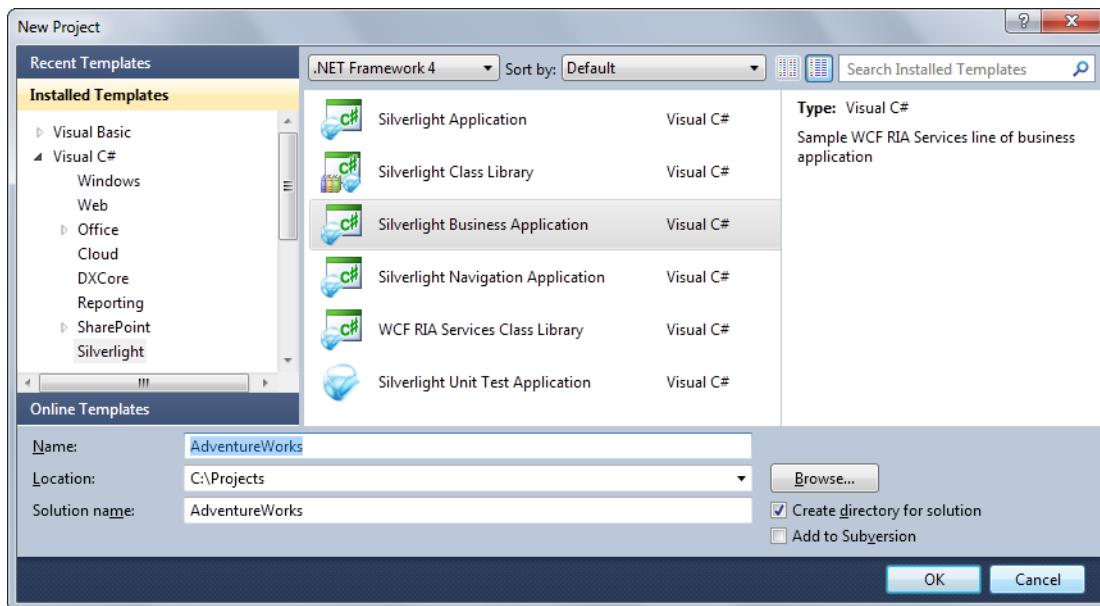


Figure 2-1. The New Project window

You will find six project templates in the Silverlight folder. Let's have a look at when and how you would use each of them.

Silverlight Application

This is a basic project template for Silverlight projects, which will simply include a single page to start with. Back in Silverlight 2, this was the only project template available to create a Silverlight application, and it didn't provide much to get started with (it was somewhat a blank slate). This might be useful for creating widgets, games, banners, advertisements, and video players, for example—when initial project infrastructure isn't particularly required. For business applications, on the other hand, you will find the Silverlight Navigation Application or Silverlight Business Application project templates a much better starting point.

When you create a project using this template, it will display a window allowing you to create a corresponding web application or web site to host your Silverlight project, and select a version of Silverlight for you application to target (as shown in Figure 2-2).

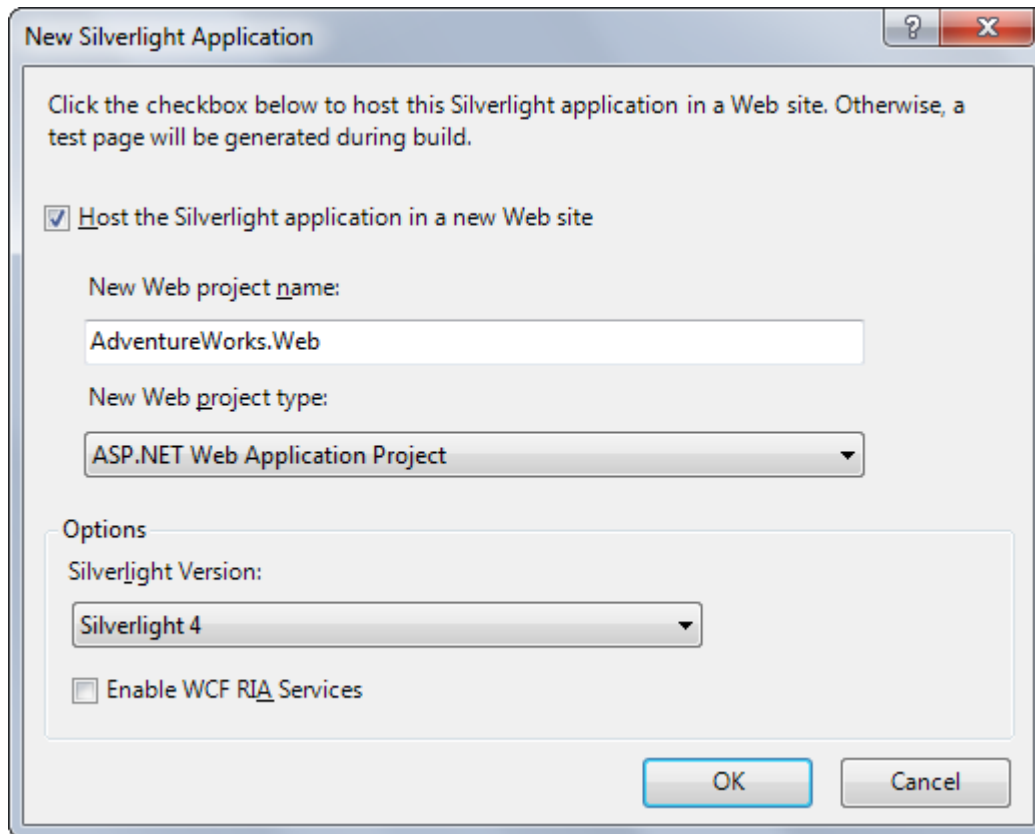


Figure 2-2. The Silverlight project wizard

Technically, you can create a Silverlight application as a standalone project (when you run it, an HTML page will be automatically generated to host it, and it will run in the browser with a local file URL); however, there is little reason to do this, and you will have limited capabilities if you choose to do so.

Whenever you create a Silverlight project, you will want to host your Silverlight application in a web application/site, and the web project will be set as the default project in your solution.

Silverlight Navigation Application

Silverlight 3 introduced a navigation framework that essentially provides an application framework for Silverlight applications. This project template implements the navigation framework and provides a good starting point for using it. We'll take a look at the features of the navigation framework in Chapter 4; however, this project template is ideal when you are creating an application (as opposed to a widget).

Silverlight Business Application

The Silverlight Business Application template is similar to the Silverlight Navigation Application template, but with a number of additional features. It automatically assumes you want a web application project, and links the Silverlight project and the web application project with WCF RIA Services. It includes login/registration screens and some built-in authentication functionality. In addition, all strings are stored in resource files so that the application can be easily localized and support different languages. This will be the template we will be basing our application upon.

■ **Note** If you don't see the Silverlight Business Application project template in the New Project dialog, ensure that you have selected version 4 of the .NET Framework in the framework selection drop-down.

Silverlight Class Library

A Silverlight Class Library project is essentially the same as a standard class library project, but is marked for use in Silverlight projects (since Silverlight targets its own version of the .NET Framework). You would use this project template if you were creating a custom control library, reusable code library, or business logic library, for example. Because libraries cannot be run on their own, you won't be asked whether to create a web project to host it.

■ **Note** It is worth noting that Silverlight projects *cannot* reference assemblies created targeting the full .NET Framework. However, projects targeting the full .NET Framework *can* reference Silverlight assemblies. Therefore, if you need to share a class library assembly between both types of projects, you are best off creating a Silverlight class library. Alternatively, you could create two projects (one for each project type) and link the files between them—a technique that may be useful if you have additional functionality that you want to add to the class library targeting the full .NET Framework not supported by Silverlight.

WCF RIA Services Class Library

The default scenario for WCF RIA Services is that you create your business objects to share between the server and the client within your web project, and these will be replicated within your Silverlight project. However, this scenario doesn't create an ideal separate "middle tier" where your business objects are contained within a separate assembly and are reusable between applications. This is where the WCF RIA Services Class Library project template comes in. When you use this template, it creates not one but two separate projects (one for the server and the other for the client) that are linked. Also, the automatic business object replication is done between these two projects rather than your Silverlight and web projects, providing much more flexibility and enabling a much better structure for your solution. This will be discussed further in Chapter 5.

■ **Note** As with the Silverlight Business Application template, the WCF RIA Services Class Library template will not appear in your project templates if you do not have version 4 of the .NET Framework selected in the framework selection drop-down.

Silverlight Unit Test Application

The Silverlight Unit Test Application project template is installed with the Silverlight Toolkit. This creates a project that you can use to unit test your Silverlight application with the Silverlight Unit Test Framework.

Running and Exploring the Default Silverlight Business Application Project

Create a new project using the Silverlight Business Application template, and compile/run it as soon as it's generated without making any changes to it. As you can see from Figure 2-3, you get a reasonable starting point for your application.



Figure 2-3. User interface created by the Silverlight Business Application project template

The project template has created a basic application with a Home page, an About page, a Login window, and a Register window. There's also an Error window that's displayed if something goes wrong. It is a very basic application, but a fantastic point to get started building your own application, as it provides some basic functionality that you can easily modify and work from.

When you click the About button, the Home page is unloaded from the content area of the application, and the About page is displayed in its place. Clicking the Back button will navigate back through the pages you've just viewed. When you navigate between pages, the title of the page (displayed in the browser's title bar) will change, and the URL in the address bar will change accordingly. If you try copying the URL from the address bar, navigating to a different page, and then pasting the other URL in the address bar and pressing Enter, you'll see that the application navigates back to that previous page. This is a concept called *deep linking*, and is part of the navigation framework. We'll take a thorough look through how the navigation framework works in Chapter 4.

When you click the Login button, a modal window will pop up for you to provide your login credentials (as shown in Figure 2-4). This window can be moved around the screen, and a dark semitransparent overlay will be added to your application to direct the user's focus toward this window (and stop mouse clicks from getting to the controls on the underlying page).

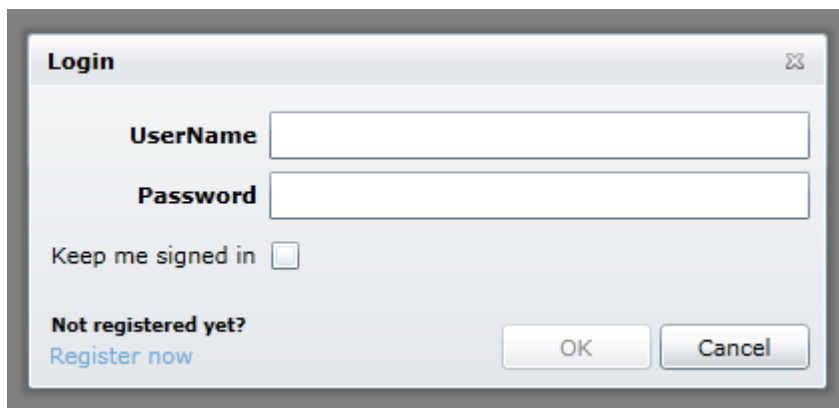


Figure 2-4. The Login window

Clicking the Register Now button in this window will display a window allowing you to register as a new user (shown in Figure 2-5). This is an example of the DataForm control in action, which is very useful for creating forms for data entry in Silverlight that include data validation. If you put your mouse cursor over the little information symbol next to the password box, a tooltip will appear providing information about the type of validation that is applied to that field. Try entering a password that doesn't conform to the rules or entering an invalid e-mail address. See how the field validates the data entered and notifies you if the validation fails. A nonconforming password entered in the Password field will turn the label for that field red, while an invalid e-mail address entered into the Email field will turn its label red and also put a red border (with a red triangle at the top right) around the text box. If you put your mouse over the red triangle or move the focus to the field, a tooltip will appear indicating what validation rule was broken.

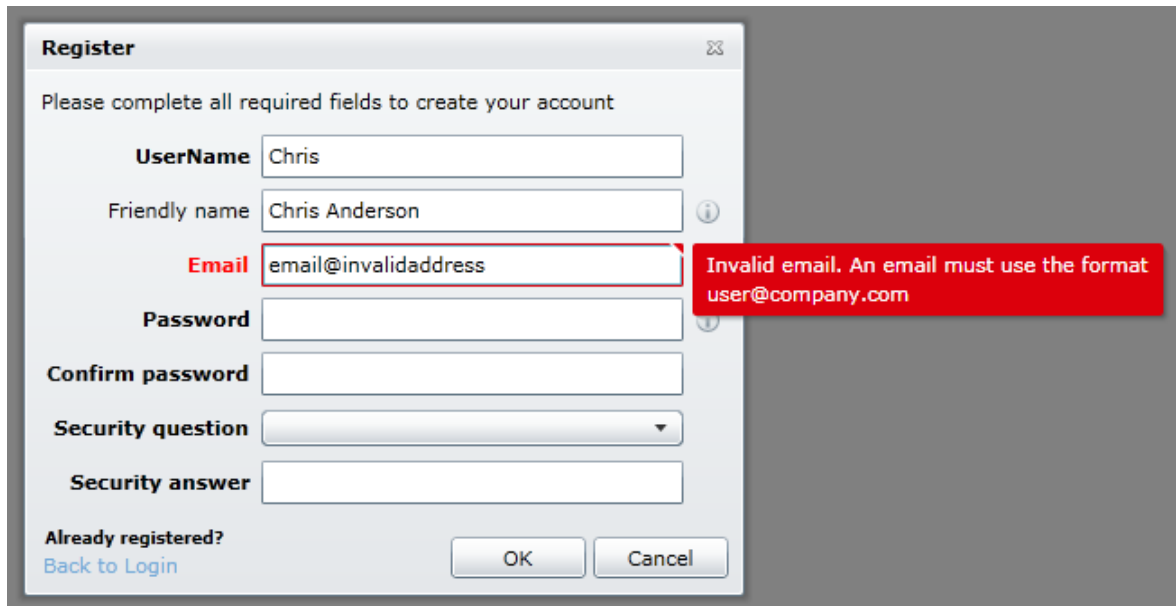


Figure 2-5. The Register window with validation errors

Register yourself as a user. When you do so, the application will create a SQL Server database with the ASP.NET membership tables, views, and so on, and attach to it as a user instance. You'll find the database file in the App Data folder in the web application project (you may need to click Show All Files to see it, as it may not have been added to the project). In Chapter 8, we'll set it up to use the AdventureWorks database to store data in. While it's creating the user on the server, a little window will pop up (as shown in Figure 2-6) with an animation to show that the application is doing something (a type of wait indicator). This is an example of the `BusyIndicator` control, which lets you know that something is happening and you need to wait for it to complete.

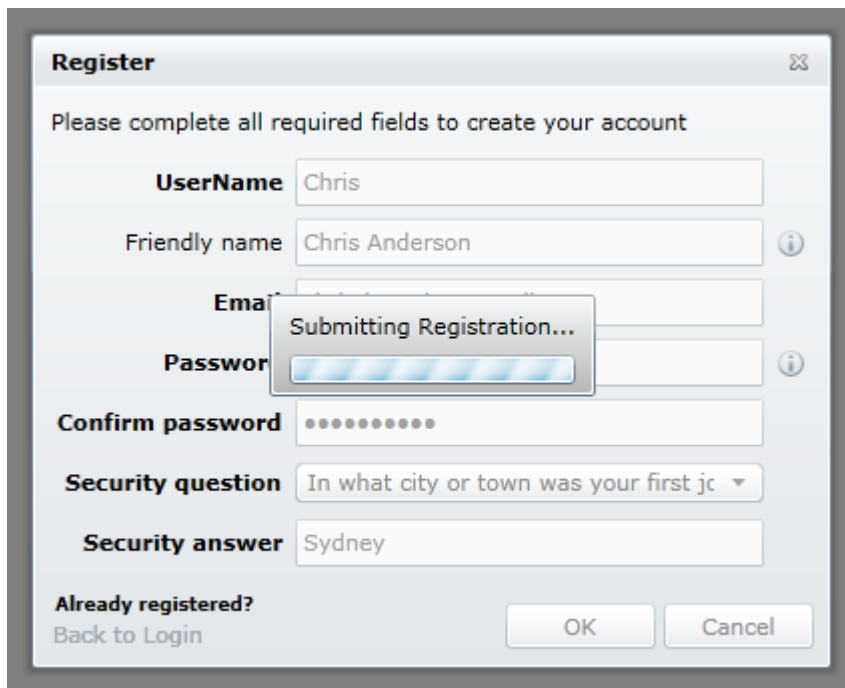


Figure 2-6. The *BusyIndicator* control

Once the new user has been created, the user interface of the application changes to hide the Login button and show a Logout button instead, along with a welcome message displaying the *friendly name* of the logged-in user (e.g., “Welcome Chris Anderson”).

So in summary, the default project template provides us with a business application–like user interface framework (with styling), some sample pages, and some security functionality. As you can see, this is pretty solid base on which to build your business application in Silverlight.

■ **Note** By default, the process of debugging the web project will use the Visual Studio Development Server (code named Cassini) to act as a host for the web site, but you can change this to run under IIS if you so desire by opening the Web tab in the project properties and changing the settings as required.

Exploring the Initial Silverlight Project Structure

Going back to our solution structure in Visual Studio, let’s now see what it took to get this amount of application functionality and learn our way around a Silverlight project. Figure 2-7 shows the structure

The image displays two side-by-side screenshots of the Visual Studio Solution Explorer, showing the project structure for a solution named 'AdventureWorks' (2 projects).

Left Screenshot (Solution Explorer):

- Solution 'AdventureWorks' (2 projects)**
 - AdventureWorks**
 - Properties
 - References
 - Assets
 - Resources
 - ApplicationStrings.resx
 - SecurityQuestions.resx
 - Styles.xaml
 - Controls
 - CustomDataForm.cs
 - Helpers
 - Libs
 - Models
 - LoginInfo.cs
 - RegistrationDataExtensions.cs
 - UserExtensions.cs
 - Views
 - Login
 - LoginForm.xaml
 - LoginRegistrationWindow.xaml
 - LoginStatus.xaml
 - RegistrationForm.xaml
 - About.xaml
 - ErrorWindow.xaml
 - Home.xaml
 - Web
 - Resources
 - ErrorResources.resx
 - RegistrationDataResources.resx
 - App.xaml
 - GlobalSuppressions.cs
 - MainPage.xaml
 - AdventureWorks.Web**

Right Screenshot (Solution Explorer):

- Solution 'AdventureWorks' (2 projects)**
 - AdventureWorks**
 - AdventureWorks.Web**
 - Properties
 - References
 - App_Data
 - ClientBin
 - BusinessApplication4.xap
 - Models
 - Shared
 - User.shared.cs
 - RegistrationData.cs
 - User.cs
 - Resources
 - ClientCultureBasedResources.cs
 - ErrorResources.resx
 - RegistrationDataResources.resx
 - Services
 - AuthenticationService.cs
 - UserRegistrationService.cs
 - BusinessApplication4TestPage.aspx
 - BusinessApplication4TestPage.html
 - Default.aspx
 - GlobalSuppressions.cs
 - Silverlight.js
 - Web.config

23

As you can see, the project template has created a sizeable solution structure with a Silverlight project and a Web Application project. Note that the web project is the startup project in the solution—*not* the Silverlight application (this was touched on earlier). The web page hosting our application is downloaded to the client, which then downloads and runs the Silverlight application itself. The Silverlight project is linked to the web project in the solution to support this process. Because the client needs to download the application from the web site, the compiled application must be available for download from an accessible location within it. By default this location is the ClientBin folder. If you expand this folder (after the solution has been compiled for the first time), you'll find the Silverlight application (a XAP file). XAP files are discussed later in this chapter.

Project Links

A link is required between Silverlight and the web projects in order for the compiled Silverlight project to be copied to the ClientBin folder in the web application project. This link is already set up when the solution is created from any of the Silverlight Application project templates, but if you open the properties for the web application project and look at the Silverlight tab (shown in Figure 2-8), you can see how this has been configured.

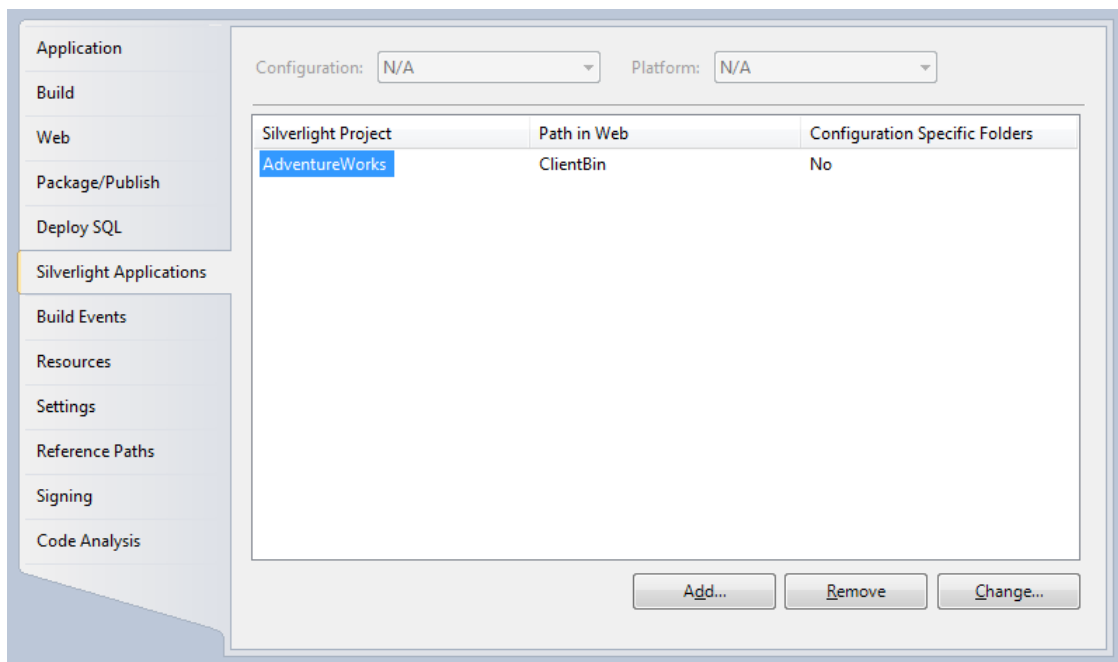


Figure 2-8. Silverlight project properties for the web application project

As you can see, the web project can be linked to multiple Silverlight projects if required, and this link will mean that whenever *either* the Silverlight project or the web project is compiled, the XAP file will be copied from the active configuration of the Silverlight project (i.e., Debug or Release) into the ClientBin folder. You can change the folder that each linked Silverlight project is copied to here if so desired, and

you can also copy the XAP file to configuration-specific folders under this folder (e.g., ClientBin\Debug and ClientBin\Release).

RIA Services generates code in the Silverlight project in order to enable you to communicate with the services defined in the web application project. If you click the Show All Files button for the Silverlight project in Solution Explorer, you will see a hidden folder called Generated Code that will contain all the code that WCF RIA Services automatically generates to communicate with the services defined in your web project (as demonstrated in Figure 2-9).

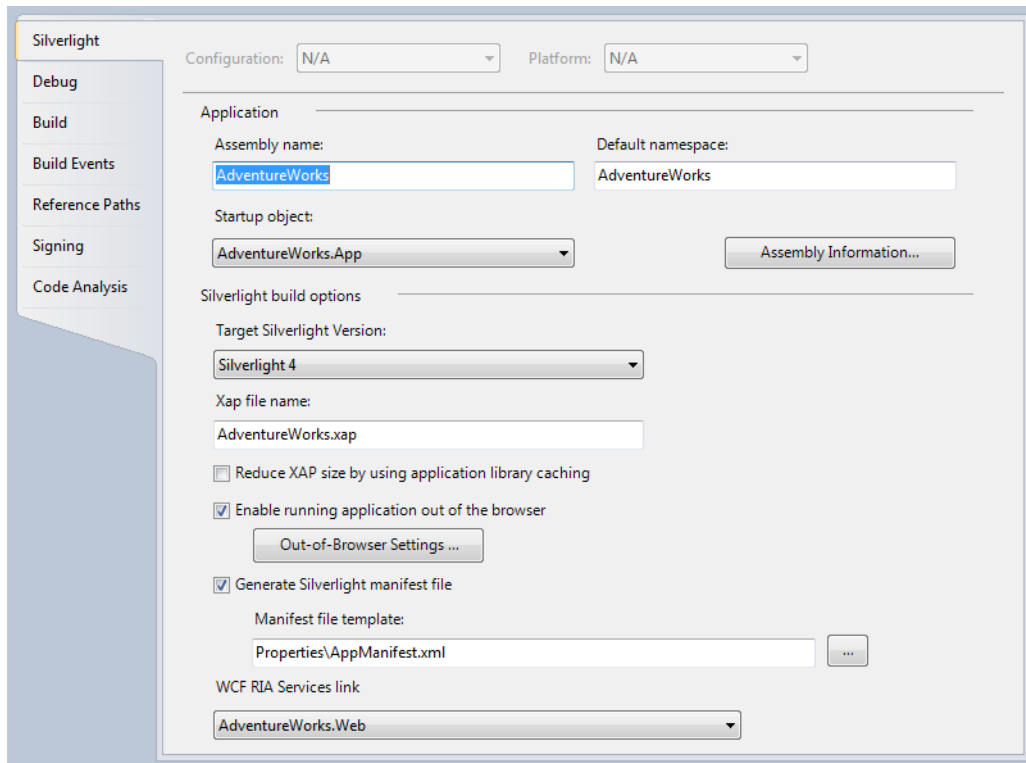


Figure 2-9. The *Generated_Code* hidden folder

In order for the code generator to know which project to generate the code into, another link is required between the Silverlight project and the web application project (but in the reverse direction). Open up the properties for your Silverlight project and take a look at the Silverlight tab (shown in Figure 2-10).

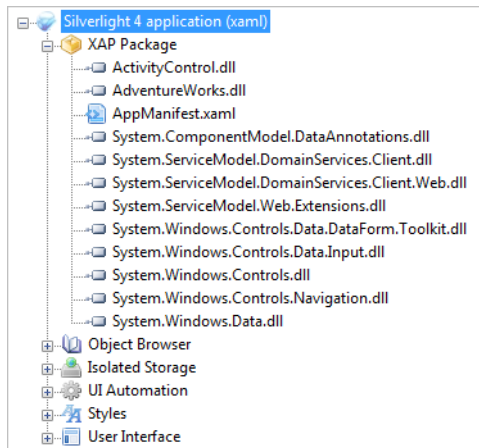


Figure 2-10. Silverlight project properties for the Silverlight project

At the bottom is the WCF RIA Services link field and a drop-down box to select the project to generate corresponding WCF RIA Services client-side code from; this defaults to the web project in the solution. When services are defined in the linked project, corresponding code will be generated in the Silverlight project to interface with them. Again, this will all be described further in Chapter 5.

■ **Note** Never modify any of the generated classes, as these changes will be overwritten the next time the project is compiled.

There are also some linked resource files between the web application project and the Silverlight project. This enables the resource files (which are used to store localized strings such as validation error messages) to be maintained in one location and shared between the two projects.

The Web Application Project Structure

The web project structure includes the following files (substitute *ApplicationName* with the name of your application):

- *ApplicationName*TestPage.aspx
- *ApplicationName*TestPage.html

These two pages can be used to host your Silverlight application (the user navigates to the page that downloads and hosts the Silverlight content), but both serve the same purpose—there's little difference (in terms of results) between the two. For now, we'll just use the HTML page and delete or ignore the ASPX page (see the "Recommended Project Template Modifications" section later in this chapter).

The Models folder (shown in Figure 2-11) contains a couple of files containing data transfer object (DTO) classes (User and RegistrationData) that will be used to pass data between the server and the client. The Shared subfolder in the Models folder contains a file (User.shared.cs) containing code to be shared between the server and the client (for which WCF RIA Services will generate corresponding code in the Silverlight project). Sharing code in this manner will be discussed in Chapter 5.

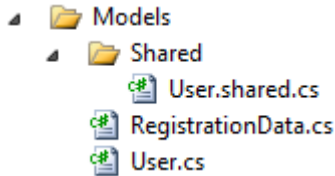


Figure 2-11. The Models folder

The Resources folder (shown in Figure 2-12) contains resource files (RegistrationDataResources.resx and ValidationErrorResources.resx) that are used to define strings used for localizing the application. As described earlier, these files are defined in this project and shared with the Silverlight project as linked files.

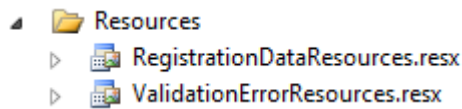


Figure 2-12. The Resources folder

The Services folder (shown in Figure 2-13) contains files for two domain service classes (AuthenticationService and UserRegistrationService) that are used to implement the server-side security functions for the application. These classes expose data and operations to the client using the WCF RIA Services framework. Again, this will be discussed further in Chapter 5.

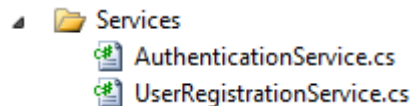


Figure 2-13. The Services folder

The GlobalSuppressions.cs file is a project suppression file used to define the messages that should be suppressed when using the Code Analysis feature of Visual Studio and should not appear in the violation list. The easy way to add violations to this file is to right-click the message in the Error List window, select the Suppress Message(s) menu item, and select the In Project Suppression File item from its submenu. Note that the Silverlight project also contains a GlobalSuppressions.cs file.

The Silverlight.js file contains various JavaScript functions to help install the Silverlight runtime (if it isn't installed), and help instantiate a Silverlight object in an HTML or ASPX page. Because this file has been "minified" to reduce its size when downloading, it is more or less unreadable, but in any case

you should have no reason to modify it and should leave it as is. If you do want to read and/or modify it, you can format it to be more readable using an online tool such as the JavaScript Beautifier (<http://jsbeautifier.org>).

The Silverlight Application Project Structure

Let's move on to the structure of the Silverlight application. You'll notice that most of the files are actually XAML files—many of which have a corresponding code-behind file (having a `.xaml.cs` extension). XAML files are XML-based files that are defined as “markup files for declarative application programming.” In simpler terms, one of XAML's primary functions is to define the structure and appearance of your application's user interface. I'll cover XAML in depth in Chapter 3. In the root of the application there are two XAML files: `App.xaml` and `MainPage.xaml`. These are two very core files in a Silverlight project, and each has a vastly different purpose.

`App.xaml` can contain the resources (such as styles) available globally in your application (in the XAML file), and handles application-wide events (in the `App.xaml.cs` code-behind file). The `App` class inherits from `Application`, and is set as the startup object for the application—you could say this class effectively represents the application itself. There is a `Startup` event (and an `Application Startup` event handler already defined) on the `App` class that is raised when the application starts; this class allows you to initialize things such as global variables if necessary.

The `Application Startup` event handler calls the `InitializeRootVisual` method, which configures the root node of the Silverlight application's visual tree (which will be discussed in Chapter 3, but in summary represents the hierarchy of objects forming the user interface). When you inspect this method, you will find that it assigns the `BusyIndicator` control to be the root node (resulting in its always remaining in the visual tree), and sets the control to disable its content while it's active. However, its content is assigned a new instance of the `MainPage` object, which really forms the basis of our user interface. The `MainPage` object is defined in `MainPage.xaml`, and could be considered the “master page” for the application into which “content pages” (aka views) are loaded.

Looking at the content of the `MainPage.xaml` file, you will find a `Frame` control that will host the views and the surroundings to this content area (such as the application header and menu). Because this page has been designed to be highly configurable through theme files, its XAML may appear to be a bit disorganized, but disregard that for now. The views to be displayed within the `Frame` control are located within the `Views` folder (shown in Figure 2-14)—with the sample views `Home.xaml` and `About.xaml` already defined for you.

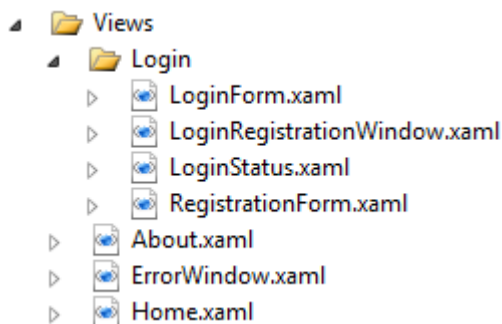


Figure 2-14. The Views folder

In the Views folder you will also find the `ErrorWindow.xaml` file, which inherits from the `ChildWindow` class (to provide modal pop-up window behavior) and can be used to display any errors that occur to the user. For an example of using this class, take a look at the `Application.UnhandledExceptions` event handler in the `App` class, which uses it to display any error messages in the application that have not been caught by a `try...catch` block. The `Login` subfolder of the Views folder contains a number of controls relating to the user login and registration functionality already implemented in the application by the project template.

The Assets folder (shown in Figure 2-15) contains a number of files containing project assets (i.e., resources and dictionaries). The `Styles.xaml` file in the Assets folder is a resource dictionary containing the styles for the application theme (i.e., defining the visual appearance of the application). This resource dictionary is merged into `App.xaml`, making the styles available globally in the application. You could consider this file to be similar in concept to a Cascading Style Sheets (CSS) file in ASP.NET projects (or any HTML web site). A number of alternative application themes are available for download that you can drop into your project (overwriting the current `Styles.xaml` file) to instantly transform the visual appearance of the application. Customizing the application styles is detailed in Chapter 9. The Assets folder also has a `Resources` subfolder containing resource files used to define various strings used throughout the application (for localization purposes).

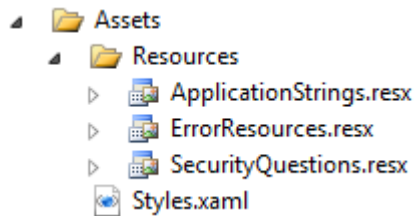


Figure 2-15. The Assets folder

The Controls folder (shown in Figure 2-16) contains a class named `CustomDataForm`. This class, which inherits from the `DataForm` control (which will be discussed in Chapter 7), changes at runtime the `TextBox` control to a `PasswordBox` control for password fields, and exposes a collection publicly from the control of the fields that have been generated. It also includes the source for the `BusyIndicator` custom control (the `BusyIndicator.xaml`, `BusyIndicator.cs`, and `VisualStates.cs` files), which is used to indicate to the user that something is happening behind the scenes.

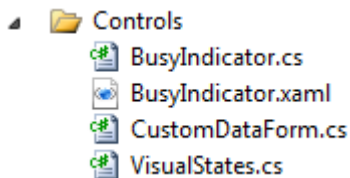


Figure 2-16. The Controls folder

The Helpers folder (shown in Figure 2-17) contains a number of Extension classes and Value Converter classes that are used throughout the default project and may be useful when you start adding your own functionality.

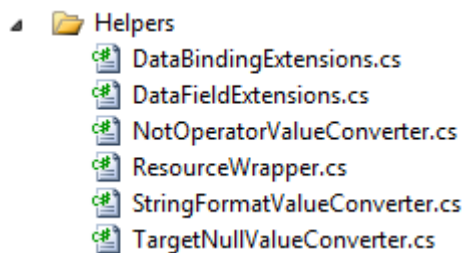


Figure 2-17. The Helpers folder

The Libs folder (shown in Figure 2-18) contains a DLL that is referenced by the project, but is not a part of the official Silverlight assemblies. `System.Windows.Controls.Data.DataForm.Toolkit.dll` is an assembly from the Silverlight Toolkit that provides the DataForm control (as was demonstrated in Figure 2-5).

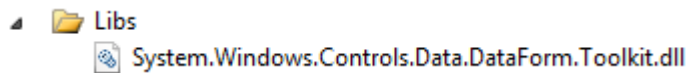


Figure 2-18. The Libs folder

The Models folder (shown in Figure 2-19) contains three files: `LoginInfo.cs`, `RegistrationDataExtensions.cs`, and `UserExtensions.cs`. The latter two classes extend the partial classes generated by the WCF RIA Services code generator (adding additional client-side functionality and properties), while the `LoginInfo.cs` file contains the `LoginInfo` class that the DataForm control in the login window can bind to (particularly to add metadata to the properties, dictating how the corresponding data entry fields will be displayed in the form).

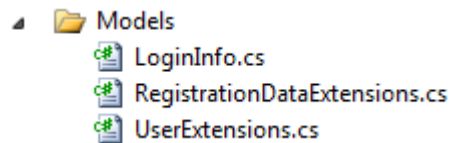


Figure 2-19. The Models folder

The Web folder contains no files, but does have a subfolder called Resources (shown in Figure 2-20), which links to the two resource files defined in the web application project, enabling the localization strings contained within them to be maintained in one location but used by both projects.

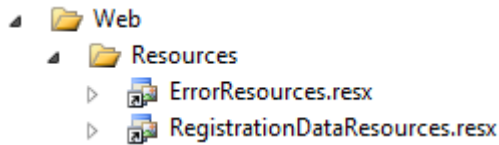


Figure 2-20. The Web folder

Recommended Project Template Modifications

As you've seen, you get quite a reasonable starting point for a business application in Silverlight, but there are a few additional changes you are likely to want to make to the default template before you get started developing.

As described previously, the following files can be found in the web application project in your solution:

- *ApplicationNameTestPage.aspx*
- *ApplicationNameTestPage.html*

These two pages do much the same thing—there's little difference (in terms of results) between the two, so you only really need one of them. ASPX pages need to be processed on the server by ASP.NET, so unless you want to do some processing in the code-behind of the page (such as checking security credentials), you can simply use the HTML page instead. You can then delete the one you don't need. On the remaining page, right-click its entry in Solution Explorer and select *Set as Start Page* from the context menu to ensure it is automatically loaded in the browser when the project is run.

When you run the Silverlight project, you might expect that the Silverlight application would have the focus, but actually it's the web page that has the focus. Therefore, if the starting page in your application has a text box with the focus explicitly set to it when the page is loaded, you will see some strange behavior. The text box may not have focus automatically, or it might have focus but not show the blinking caret. We can resolve this issue by adding a little bit of JavaScript code to the web page.

Open the HTML page to inspect its contents (in Source view). There are some styles defined for the page, a linked JavaScript file containing support functions for installing Silverlight and creating the Silverlight object, a JavaScript function that is called when an unhandled error occurs in the Silverlight application, an object tag to host the Silverlight application, and an IFrame that is used as a part of the navigation framework to handle the browser history functionality. Add the following code directly below the `onSilverlightError` JavaScript function at the top of the page:

```
function appload(sender, args)
{
    var xamlObject = document.getElementById('silverlightControl');

    if (xamlObject != null)
        xamlObject.focus();
}
```

Add this property to the Silverlight object tag:

```
id="silverlightControl"
```

And add the following line to the properties of the Silverlight object:

```
<param name="onLoad" value="appLoad" />
```

Run the project again and you will find that the text box now correctly has the focus. You can see the caret blinking, and typing (without clicking the application first) will populate the text box with the typed characters.

■ **Note** If the change doesn't seem to have worked, try refreshing the page using Ctrl+F5, as the page may have been cached and not updated with the changes made.

At this stage, you might want to also add the following property to the Silverlight object:

```
<param name="Windowless" value="True" />
```

The purpose of this will be discussed in Chapter 14, where it is necessary in order to overlay the application with an IFrame to display reports within. There are performance impacts to setting this property to True, but as most business applications aren't graphically intensive, this should not cause an issue. You are now ready to start working on your Silverlight project.

In the Silverlight project, open up the `ApplicationStrings.resx` file (under the `Assets\Resources` folder). Find the `ApplicationName` resource and change its value to the name of your application. This name will appear in the header of the application when you run it.

XAP Files

When you compile a Silverlight project, it is all combined into a single file with a `.xap` extension. This file contains the assemblies, XAML files, and any other resource or content files associated with the project, which are bundled and compressed using the ZIP file format. This XAP file can then be hosted on a web server to be downloaded by client computers, and the application can then be run. You can prove this by simply changing the file extension from `.xap` to `.zip` and unzipping the file to inspect its contents.

■ **Note** Some compression utilities (such as 7-Zip) add items to the context menu in Windows Explorer, which you can use to open the archive without needing to change the XAP file's extension. Alternatively, you can associate the `.xap` file extension with a compression utility such that you can simply double-click it to inspect its contents.

The easiest (and most feature-rich) way to inspect a XAP file, however, is to simply load it into Silverlight Spy (as shown in Figure 2-21) and inspect it that way.

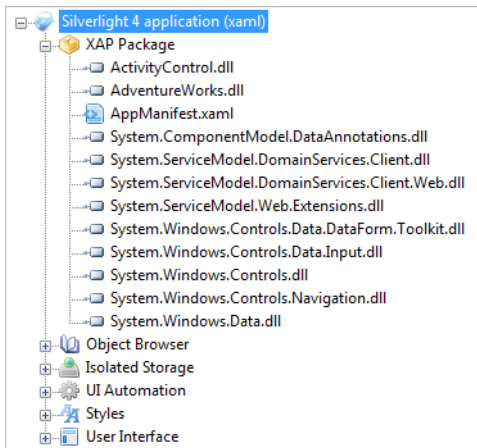


Figure 2-21. Silverlight Spy project view explorer

Summary

Now you're set up and ready to start developing Silverlight applications. You should now understand how a Silverlight solution is structured, what features have already been implemented in the solution to get you started, and how the Silverlight and web projects are connected. In the next chapter, we'll take a look at the features and syntax of XAML, and how it helps you design and build a user interface for your application.

CHAPTER 3



An Introduction to XAML

Extensible Application Markup Language (XAML; pronounced *zammel*) is an XML-based declarative markup language used to represent a hierarchy of objects.

In Silverlight, XAML is primarily used to define the layout and contents of user interfaces (although it is also used to define resource dictionaries, animations, and the application root). XAML is not a Silverlight-specific concept, however. It actually has its origins in Windows Presentation Foundation (WPF); Silverlight emerged essentially as a subset of WPF designed for use on the Web (hence its early name of WPF/E, or WPF Everywhere, before being christened with the catchier “Silverlight”).

Despite being used to define user interfaces in Silverlight and WPF, this is not XAML’s sole purpose. XAML was designed to simply represent a hierarchy of objects, giving it a rather broad scope in which it can be used. In fact, XAML has gone on to be used in multiple Microsoft technologies, including Windows Workflow Foundation (WF; used to define process workflows) and the XML Paper Specification (XPS; used to define an electronic paper format). In this chapter, however, I will be focusing specifically on Silverlight’s use of XAML, and the XAML extensions that Silverlight implements.

HyperText Markup Language (HTML) is a form of markup language that you are almost certainly familiar with, and have possibly used to define a user interface for web-based applications. HTML was designed as a markup language for structuring documents for display within a web browser, but was soon being coerced into doing things it was never designed to do (including formatting documents and positioning document elements to form application user interfaces). Scalable Vector Graphics (SVG) is another markup language that you may be familiar with; it’s used to define vector graphics.

XML was designed to provide a standardized yet extensible framework for designing new markup languages that could be used for an almost infinite number of purposes (SVG being one of those languages). XAML builds on this XML framework, and takes the capabilities of a markup language much further than ever before—primarily by combining the features found in both the HTML and SVG formats (and much more) into a single powerful markup language definition. As a result, XAML is a powerful, flexible, and extensible means of defining a hierarchy of objects that can serve an array of purposes, and hence these benefits can be harnessed by Silverlight.

Overcoming XAML’s Steep Learning Curve

Unfortunately, despite all XAML’s benefits, learning, understanding, and becoming productive in XAML is likely to be your greatest hurdle when attempting to develop your first Silverlight applications. Things will seem harder than they should be, and will invariably lead to initial frustration with the technology. The problem is that (as a general rule) the more flexible something is, the more complex it is to understand and use. XAML unfortunately suffers from this same pitfall. As a result, you will be faced with a steep learning curve. XAML introduces new concepts to the way to define and design user interfaces, and requires you to invest in a new way of thinking in how you do so.

Some examples of the types of issues you are likely to face when getting up to speed with XAML include the following:

- Things may seem harder than you initially think should be. For example, there is no property on the Button control that lets you display an image on the button. This is because XAML has enabled a more flexible method to display custom content in buttons (an example of this can be found in the “Content Element Syntax” section of this chapter).
- XAML introduces many new concepts that may not have been part of previous technologies you’ve worked with. For example, concepts such as attached properties, markup extensions, styles, and templates might be foreign to you.
- Properties you are used to finding on controls in previous technologies you’ve worked with may not exist in Silverlight controls, or will have different names from what you might expect. For example, the Button control has no Text property in Silverlight. Neither does it (or any other control) have properties that can be used to position it within the user interface. Some of these differences will be raised throughout this chapter.

Despite the challenges of learning XAML, it is worth the effort to work through them, and the reason XAML is structured the way it is will make sense in the end (as will hopefully become clear throughout this chapter), enabling you to start harnessing its power.

Why Learn XAML?

You may be wondering why you need to understand and write XAML, as Expression Blend has always been available to act as a XAML designer for building user interfaces that can take away (most of) the need to write XAML itself. In addition, Visual Studio 2010 now contains a drag-and-drop XAML designer too (although it’s not quite as fully featured as Expression Blend’s).

It’s important to remember that Silverlight as a technology revolves around its XAML core. Understanding the fundamental concepts of XAML is akin to learning how to count—you need to learn how to count before you can get a grasp on algebra. Therefore, your primary concern when learning Silverlight is to at least understand the underlying concepts of XAML, even if you don’t initially completely learn its syntax.

In the early days, there wasn’t a XAML designer in Visual Studio for Silverlight developers, which made it necessary to either write the XAML by hand or use Expression Blend. However, now that there is a XAML designer for Silverlight developers in Visual Studio 2010, you could probably get away with not learning the XAML syntax for a while, and solely use that (or the Expression Blend designer) to design your user interfaces. You will, however, need to learn and understand the underlying concepts for structuring user interfaces in XAML in order to develop them effectively.

If you’ve ever worked on an ASP.NET project, you will probably have had to resort to modifying the HTML directly instead of using the HTML designer, and you will unfortunately face this same issue with XAML at times. Therefore, it’s best to avoid having a complete reliance on the drag-and-drop designers. XAML may look confusing at first, but with a good understanding of the concepts, you will find that the syntax (along with the help of IntelliSense) becomes very easy to read and write.

The majority of this chapter is designed to help you understand the concepts, structure, and syntax of XAML, but it is also interspersed with practical examples to demonstrate the concepts and put them into perspective.

XAML Syntax, Document Structure, and Features

XAML files are easily spotted by their `.xaml` extension, and often (but not always) have a corresponding code-behind file (`.xaml.cs`). In this section we'll take a look at the structure of these XAML files, the XAML syntax, and the most important features of XAML.

■ **Note** An easy way to understand XAML is to inspect existing XAML files (preferably simple XAML files) and compare the XAML to the output in the designer. Enabling split mode (i.e., XAML and designer in the same view), which is available in both Expression Blend and Visual Studio 2010, is the ideal means for doing so, as this allows you to make a change in one and immediately see the result in the other.

Core XAML Syntax

XAML is based on XML, and thus inherits numerous concepts from it—the two most important being *elements* and *attributes*. Therefore, if you're familiar with XML, you'll already be familiar with the basic syntax of XAML.

Each element in a XAML file maps to a .NET class (which will often be a control). In most cases, an element will consist of both an opening and a closing *tag*. An opening tag consists of a *less-than* angle bracket (`<`), followed by the class name, followed by a *greater-than* angle bracket (`>`). The closing tag is similar; however, the less-than angle bracket is followed by a forward slash (`/`). For example, to insert a button into your XAML, you can use the following syntax:

```
<Button></Button>
```

As the button in the example does not have any content (i.e., nothing between the opening and closing tags), you may choose to use the shorter form, where the closing tag is omitted, but the closing `>` is preceded by a `/`, as demonstrated here:

```
<Button />
```

When you reference a control in XAML, you no doubt will want to assign values to its properties. There are a number of ways to do this in XAML (as will be described later in the “Assigning Property Values to Controls” section), but here we'll specifically look at the attribute syntax for doing so. Each attribute maps to a property on the control specified in the corresponding element.

Attributes are specified within the opening tag of an element. The control name should be followed by a whitespace character (i.e., space or new line), after which you can start assigning values to attributes. The syntax for an attribute is the name of the property you wish to assign a value to, followed by an equal sign (`=`), followed by the value surrounded by double quotes (`"`). For example, you can assign some text (Hello) to the Content property of the Button control in the previous example using attribute syntax like so:

```
<Button Content="Hello" />
```

While the standard character to surround the value of the attribute is the double quote (`"`), you can exchange it with single quotes (`'`) if necessary. You might want to do this, for example, if you wanted to

actually include a double quote as a part of the value, in which case any double quote character between the single quotes will be considered a part of the value and ignored as being a XAML markup token character.

■ **Note** Element and attribute names are case sensitive.

Creating an Object Hierarchy

When you create a XAML file, you are essentially defining a hierarchy of objects—also known as an *object graph* or *object tree*. A XAML file can have only a single root-level node, and elements are nested within each other to define the layout and content of the user interface. Most of the objects in the object tree in Silverlight will be controls.

■ **Note** The topmost object in the application hierarchy is called the *root visual*, and a reference to this object can be obtained in code using the `RootVisual` property of the `Application` object.

Let's create a simple XAML file and inspect its structure. For this we'll add a new file to our project using the Silverlight User Control item template, which will generate the following XAML file:

```
<UserControl x:Class="Chapter03Sample.SimpleXAMLFile"
    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"
    mc:Ignorable="d"
    d:DesignHeight="300" d:DesignWidth="400">

    <Grid x:Name="LayoutRoot" Background="White">

        </Grid>
    </UserControl>
```

Unfortunately, despite being a simple XAML file, it still covers a range of XAML concepts that we aren't quite ready to cover yet, including some namespace prefix declarations (the attributes starting with `xmlns:`) and some attached properties (such as the `d:DesignHeight` and `d:DesignWidth` properties, which differ from standard properties such as the `Background` property on the `Grid` element). Let's ignore these for the time being, strip this XAML right back to just its structure, and focus on that:

```
<UserControl>
    <Grid>
    </Grid>
</UserControl>
```


Here you have `UserControl` as your root element, with a `Grid` control element as its content. You could read this as, “The user control contains a grid control.” You cannot add any more elements to the `UserControl` element, as the `UserControl` control only accepts a single child element (as is the case with many controls). However, the `Grid` control *does* accept multiple child elements. This is a very simple object hierarchy, and a long way from defining anything useful; however, we’ll keep adding to this basic structure throughout this chapter to define a very simple user interface.

■ **Note** The object hierarchy is often referred to as the *visual tree*. You can traverse the objects in the visual tree in code using the `VisualTreeHelper` class, found in the `System.Windows.Media` namespace. It exposes methods to get the children of a XAML object (`GetChild` and `GetChildrenCount`), the parent of a XAML object (`GetParent`), and all the objects within a given area on the screen around an object (`FindElementsInHostCoordinates`).

Namespaces

Before we stripped the XAML file (introduced earlier) back to its core structure, you may have noticed that the opening tag of the root element contained a number of attributes that start with `xmlns:`. Let’s take another look at that element again:

```
<UserControl x:Class="Chapter03Sample.SimpleXAMLFile"
  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"
  mc:Ignorable="d"
  d:DesignHeight="300" d:DesignWidth="400">
```

A number of namespace prefixes are defined on this root node (in bold), each declared using `xmlns` (the XML attribute used for declaring namespaces). XAML namespace prefix declarations are conceptually similar to adding `using` statements at the top of a class in C#, but with a few notable differences. To use controls in your XAML file, the CLR namespace that they are defined in must be declared in the root element and assigned a prefix that is unique within the XAML file (which will be used when referencing a control in order to remove any ambiguity as to which control you are referring to).

■ **Note** Actually, although namespace prefixes are generally declared in the root element in the XAML file, they can in fact be declared at any level (i.e., in any element) in a XAML file (and be referenced anywhere below than that element in the hierarchy). However, it is generally standard practice to define them all on the root element of the XAML file.

Prefixes are a type of alias, reducing the verbosity of XAML by letting you use the prefix you have defined rather than qualifying a control when you use it in your XAML file with its whole namespace. The prefix is defined immediately following the colon after `xmlns`.

Note that the first namespace declared on the root element in the example isn't actually assigned a prefix:

```
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
```

This is because it is defining the default namespace for the file (i.e., the Silverlight namespace), enabling Silverlight controls to be used throughout the file without the need to qualify their names with a namespace prefix. However, the second namespace reference does define a prefix (`x`, used to reference the XAML namespace):

```
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
```

■ **Note** Unfortunately, these default namespace declarations aren't the best examples to use, as none of them (directly) reference CLR namespaces and assemblies (as will the namespaces you generally declare). Instead they map to URIs—known as *consolidated namespaces*. Consolidated namespaces combine multiple namespace declarations into one declaration, enabling you to declare a single URI namespace instead of needing to reference multiple namespaces. However, you don't need to worry about this concept—leave these declarations as they are and simply add additional declarations as required following them. If you do want to consolidate a number of namespaces yourself, you can use the `XmlnsDefinition` attribute, which will be described in Chapter 10. Another consolidated namespace that isn't referenced by default, but that you may find useful, is the SDK consolidated namespace—it brings all the namespaces in the Silverlight SDK together, such that instead of defining prefixes for all these namespaces in your XAML file you can simply define just the one namespace prefix (`sdk`) instead to reference all the controls in the SDK:

```
xmlns:sdk="http://schemas.microsoft.com/winfx/2006/xaml/presentation/sdk"
```

When adding your own namespace declarations, they will almost always begin with `clr-namespace::`, referencing a CLR namespace and the assembly that contains it. Prefixes can be anything of your choosing, but it is best to make them short (yet meaningful). For example, in the following declaration we are declaring the `System.Windows.Controls.Data` namespace in the `System.Windows.Controls.Data` assembly, and assigning `data` as its prefix:

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

Once you've declared a namespace prefix, you can use controls within that namespace like so:

```
<data:DataGrid></data:DataGrid>
```

■ **Note** IntelliSense in Visual Studio 2010 is very helpful when both declaring namespaces and when referencing controls in those namespaces. After you enter a namespace prefix and press = (e.g., `xmlns:data=`), IntelliSense will display a list of all the namespaces and their corresponding assemblies referenced in the project. You can then select a namespace from the list and press Tab to insert the namespace and assembly name into the declaration. When using the designer, dropping a control from the toolbox onto the design surface (from a namespace that is not currently declared) will automatically declare its namespace and assign a default namespace prefix. When using a control, starting the open tag of the element (with `<`) will display a list of the namespace prefixes defined in the file, along with the controls from the default namespace. After you select/type the namespace prefix, IntelliSense will then display a list of the controls in the mapped namespace for you to choose from. The XAML syntax for declaring namespaces and using controls from those namespaces may initially look intimidating, but IntelliSense makes writing this XAML completely effortless.

Assigning Property Values to Controls

After inserting a control element into a XAML file, you will in most cases want to set some property values on that control (e.g., the background color of a text box). You can of course use the Properties window in both Visual Studio and Expression Blend to do so (in most cases), but let's look at the different means for doing so in XAML. The means you use will depend on the nature of the property and the value you are assigning to it. The reasons you might use each syntax will be explained along with each means.

Attribute Syntax

The simplest way to assign values to properties in XAML is to set them inline, using *attribute syntax*. This syntax is used when the property has a primitive type (i.e., string, integer, Boolean, etc.) or when the property's type has an associated type converter that can accept a string and convert it to that type. Regardless of the type of the property, all values should be enclosed within single or double quotes (preferably double quotes). The following example demonstrates assigning the Content property of a Button control using attribute syntax:

```
<Button Content="Hello" />
```

Content Element Syntax

Some controls define a property whose content (i.e., the XAML between its beginning and end tags) will be assigned to the property's value. This is known as *content element syntax*, and the control defines this content property by decorating its class with the ContentProperty attribute. A further discussion of how this is achieved will be covered in Chapter 11. There's no clear way of determining which property (if any) is assigned the content of the control, but you will find it in the MSDN documentation for the control. As a general rule, however, most properties defined as content properties are named Content (as this property is provided by the ContentControl base class that most controls that accept custom content derive from). The following example has exactly the same output as the previous example; however, it

uses content element syntax to assign the value to the Content property (as the Content property is designated as a content property on the Button control):

```
<Button>Hello</Button>
```

■ **Note** No quotes are required around the value when using content element syntax.

You may wonder at this point why there are two ways to achieve exactly the same outcome. The reason is that rather than just simple values, many content properties accept complex values (such as XAML), which can't be assigned using attribute syntax (the value would be treated as plain text if you were to try). Instead, using content element syntax, the Content property of the Button control can accept any XAML object hierarchy that will be displayed as the content of the button. You may have noticed that unlike Windows Forms' Button control, there doesn't appear to be an Image property (or similar) on the Silverlight Button control. This may initially make you think the Button control in Silverlight is very limited in its capabilities, but you would be mistaken, because this is where the Content property demonstrates the flexibility of XAML. Because the Content property will accept XAML elements as values, you can specify some XAML to display both an image and some text as the content of the button, as the following example demonstrates:

```
<Button Width="80" Height="35">
    <StackPanel Orientation="Horizontal">
        <Image Source="accept.png" Width="16" Height="16" />
        <TextBlock Margin="7,0,0,0" Text="OK" />
    </StackPanel>
</Button>
```

Here we are using a StackPanel (discussed later in this chapter) containing both an Image control and a TextBlock control to achieve the result shown in Figure 3-1. Although this may initially appear to be more work than it would be to achieve the same outcome in Windows Forms, it does enable you to easily lay out the content of the button in whatever form you choose (rather than how the control chooses to implement the layout), and demonstrates the incredible flexibility of Silverlight and XAML. Using this method you could have any Silverlight control in the content of the Button, such as a ComboBox or even a DataGrid (practicality issues aside)!



Figure 3-1. A Button control with custom content

■ **Note** Unless the content property is a collection, the content you assign to it can only consist of a single control XAML element (although that element can have child elements as required). Otherwise you will get the error, “The property *XXX* is set more than once.” If the property is a collection then you can assign multiple elements, as detailed shortly, using collection syntax.

Property Element Syntax

However, what if you want to assign a complex value to a property that isn’t defined as a content property on a control? This is still possible using *property element syntax*. Property element syntax requires you to create a new element between the control element’s opening and closing tags, of the form `<control.property>`. The dot in the element’s name indicates to the XAML parser that this is a property element rather than an object element.

In this example, we’ll assign the same value to the Content property of the button as in the previous example (using content element syntax), but we’ll assign the property value using property element syntax (with the property element highlighted in bold):

```
<Button Width="80" Height="35">
    <Button.Content>
        <StackPanel Orientation="Horizontal">
            <Image Source="accept.png" Width="16" Height="16" />
            <TextBlock Margin="7,0,0,0" Text="OK" />
        </StackPanel>
    </Button.Content>
</Button>
```

You will find yourself often using property element syntax to assign resources to a control’s Resources property (as will be discussed later in this chapter in the “Resources and Resource Dictionaries” section), or basically any control property that accepts a complex value that can be defined in XAML. However, if that property is designated as a content property (like the Content property in the preceding example) then it’s standard to use the content element syntax instead, and omit the property element’s tags.

Collection Syntax

As stated earlier, properties that can accept XAML content can only be assigned a single control (although that element can have its own child elements), unless the property is a collection type. If the property is a collection type, then you can use collection syntax to assign items (i.e., generally controls) to that collection, with each (top-level) element defined in the value being added as an item to the collection. The Items property on the ComboBox control is an example of where collection syntax can be used, as it is a collection that maintains all the items in the combo box. Because the Items property is designated as the content property for the ComboBox control, we can simply use content element syntax to assign the controls to the Items property (this method is equally applicable using property element syntax):

```

<ComboBox>
  <ComboBoxItem Content="First item" />
  <ComboBoxItem Content="First item" />
  <ComboBoxItem Content="Third item" />
  <ComboBoxItem Content="Fourth item" />
</ComboBox>

```

■ **Note** You can also assign values to multiple properties on a control, maintain these in a central location (as a resource), and assign them to multiple controls using style resources (these will be discussed later in this chapter in the “Styles” section).

Attached Properties

Attached properties are special types of properties that are assigned a value on a control, but the properties are actually defined on and belong to another control (usually but not always higher up in the hierarchy). In essence, controls can register particular properties as attached properties, which are then able to be used by *any* control. This may seem a rather confusing concept initially; therefore, the best way to demonstrate this concept is by an example.

Say you have a control whose position is being controlled by a parent control (e.g., a Button control that is contained within a Canvas control). There are no positioning properties defined on the Button control, as the type of layout control the button is contained within determines how it is positioned and what positioning properties are required (the “Layout Controls” section later in this chapter discusses this topic further). The Canvas control needs to know where to position the Button control on its surface, so it needs to expose two properties (Left and Top) as attached properties, which can be used by the Button control to allow the Canvas to position the button accordingly. You may like to consider it that the Button control *inherits* the Left and Top properties from the Canvas control.

To use these Canvas properties on the Button, control you need to include the name of the control that the properties belong to (using property element syntax). The example following demonstrates the Button control setting values for the Left and Top properties that it “inherited” from the parent Canvas control:

```

<Canvas>
  <TextBox Canvas.Left="30" Canvas.Top="15" />
</Canvas>

```

■ **Note** When using attached properties, these values will actually belong to the control that you are setting them on. In this case, despite the Left and Top properties belonging to the Canvas control, the values themselves belong to the Button control.

The ToolTipService control is used to provide tooltip functionality to controls, and uses attached properties to do so. However, unlike the previous example where the control defining the attached

properties exists higher up in the control hierarchy from the control that the attached property values are being applied to, `ToolTipService` demonstrates that this does not necessarily have to be the case. As previously mentioned, attached properties can be used by any control, and not necessarily via inheritance.

In this example we are using the `ToolTipService.ToolTip` attached property to assign a tooltip to a `Button` control:

```
<Button ToolTipService.ToolTip="This is an example tooltip" />
```

■ **Note** Attached properties require a special technique to be used if you want to set their values in your code-behind. This will be covered in Chapter 10.

XAML Namespace Properties

The namespace assigned to the `x:` prefix in the simple XAML file discussed earlier in this chapter maps to the XAML namespace. This namespace contains a number of important attached properties that you will make extensive use of in your XAML files. Let's take a look at the most important properties in this namespace, and the scenarios in which you might use them.

x:Class

Looking at the first line of the root element in this XAML file, you will notice that it assigns a value to the `x:Class` attribute:

```
<UserControl x:Class="Chapter03Sample.SimpleXAMLFile" ... />
```

This property is used to associate this XAML file with its corresponding code-behind class. As this property is automatically configured when you create a new XAML file in Visual Studio, you will rarely need to worry about it—just be aware of its importance.

x:Name

You will need to give controls names in order to interact with them in the code-behind (or bind to them in XAML using `ElementName` binding, which will be discussed in Chapter 10). Most controls will have a `Name` property that you can assign a name for the control to, but for those that don't you can use the `x:Name` attached property from the XAML namespace instead, like so:

```
<TextBox x:Name="UserNameTextBox" />
```

However, for controls that implement the `Name` property, you can use it instead to assign the control a name:

```
<TextBox Name="UserNameTextBox" />
```

It doesn't matter which method you use (you can only use one), as they both serve exactly the same purpose, but a good standard to stick to is to use the `Name` property where it has been implemented, and use `x>Name` only where it hasn't. Alternatively, you can choose to standardize on `x>Name`, which is applicable to every control.

Regardless of which you choose, a member variable will be automatically created for the control once you have given it a name (in the hidden code-generated designer class) that you can use to refer to that control in code.

■ **Note** At times, the member variable will not be created in the hidden code-generated designer class until you compile the application.

x:Key

Instead of a name, resources are assigned a unique key using this property, which controls can use to refer to that resource. See the “Resources and Resource Dictionaries” section later in this chapter for more information.

x:ClassModifier and x:FieldModifier

By default, the backing class to your XAML file will have a scope of public, and (surprisingly) the corresponding backing fields for any controls that you give a name to will have their scope set to internal. The `x:ClassModifier` and `x:FieldModifier` properties work in the same way to enable you to explicitly set these access levels in the XAML; however, the `x:ClassModifier` property can only be applied to the root element, and the `x:FieldModifier` property can only be applied to controls. For example, the following XAML uses this property to set the button's scope to private (so it cannot be accessed from outside the XAML file's backing class):

```
<Button x>Name="OKButton" x:FieldModifier="private" Content="OK" />
```

■ **Note** If you change the access level of the class using the `x:ClassModifier` property, you will need to manually update the scope for your code-behind class accordingly; otherwise, you will receive a compilation error stating that partial declarations of the class have conflicting accessibility modifiers (i.e., the scope of the automatically generated partial designer class for the XAML file does not match that of the code-behind partial class).

Design-Time Properties

The namespace assigned to the `d:` prefix in the simple XAML file discussed earlier in this chapter contains some attached properties that can be used to improve the design-time experience when developing Silverlight applications. These will then be ignored at runtime.

Take another look at the root element of the simple XAML file that we have been using:


```
<UserControl x:Class="Chapter03Sample.SimpleXAMLFile"
  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"
  mc:Ignorable="d"
  d:DesignHeight="300" d:DesignWidth="400">
```

In bold are two attached properties (`DesignHeight` and `DesignWidth`) from the Expression Blend 2008 namespace (assigned a prefix of `d`), and another attached property (`Ignorable`) from the Markup Compatibility namespace (assigned a prefix of `mc`).

Try removing the `DesignHeight` and `DesignWidth` properties from the element (including their values). You will find that the design surface in the designer has shrunk to the size of its contents (which will be to nothing as there's currently nothing in the file). You could instead set the `Height` and `Width` properties of the `UserControl` instead—which will work, but will also affect how the user control will be rendered at runtime (where you may want it to expand to fill the browser window area). Therefore, the design-time properties can be very useful for sizing your design surface at design time, while still allowing the user control to behave as required at runtime.

If you were to look in the Expression Blend 2008 namespace, you would also find that there are other design-time data-related properties, including `DataContext`, `DesignData`, and `DesignSource`, in addition to the `DesignInstance` and `DesignData` markup extensions. It can often be useful (especially if you are a designer) to specify the structure of the data that you will be binding to, and populate your design surface with sample data in order to see how it might look at runtime. This is a concept often referred to as *blendability*. This concept will be explained in detail in Chapter 10.

In order for these design-time properties to be ignored at runtime, the `mc:Ignorable` attached property is used (from the Markup Compatibility namespace) to tell the XAML parser to ignore the namespace declaration `d` and any references to it. If you were to remove this property and attempt to compile your Silverlight application, you would find that the compilation fails, stating the following:

```
XAML Namespace http://schemas.microsoft.com/expression/blend/2008 is not resolved
```

Markup Extensions

Often you have scenarios where you want to set the value of a control's property, but this value is not known at design time (or should be determined at runtime). A typical example of this would be when you want to display data that has been retrieved from the server—a problem that the concept of data binding was designed to solve.

To support these types of scenarios and specify them in XAML (including but not limited to enabling data binding) is the concept of *markup extensions*. A markup extension returns a value that will be applied to the property it was applied to at runtime. Markup extensions are surrounded by curly brackets. The brackets indicate to the XAML parser that what is inside them is a markup extension, and should be handled accordingly. The open curly bracket is followed by the name of the markup extension to be used. Common markup extensions include

- **StaticResource**: Used to assign a resource as the value of the property (discussed later in this chapter in the “Resources and Resource Dictionaries” section).
- **Binding**: Used to specify a data binding expression (discussed later in this chapter in the “Data Binding” section).

- **TemplateBinding**: Used in custom control templates to assign a value from a property exposed by the control (discussed further in Chapter 11).
- **RelativeSource**: Used to bind the value of a property to that of another control relative to it in the object hierarchy (discussed further in Chapter 10).
- **x:Null**: Used to assign a null value to a property.

The simplest use of a markup extension is to use `x:Null` to assign a null value to a property, as demonstrated here:

```
<TextBox Background="{x:Null}" />
```

Markup extensions can have properties whose values can be assigned following the markup extension name (within curly brackets). For example, here we are using the `Binding` markup extension, and assigning `ListPrice` as the value of its `Path` property:

```
<TextBox Text="{Binding Path=ListPrice}" />
```

■ **Note** Notice the lack of double quotes around the property value. Since the entire value of the property (i.e., the markup extension) is surrounded by quotes, to use quotes within this value would result in invalid markup, and your application would fail to compile. Therefore, you must omit the double quotes from around the value, or use single quotes (') instead (which would be required if the value contains a comma).

Some markup extensions have specified a default property, enabling you to specify a value for that property without explicitly identifying the property. For example, since the `Path` property of the `Binding` markup extension is its default property, we can instead write the same expression like so:

```
<TextBox Text="{Binding ListPrice}" />
```

Note that the properties of a markup extension in turn can also be assigned a markup extension value. Here we are assigning a `StaticResource` markup extension to the `Converter` property of the `Binding` markup extension:

```
<TextBox Text="{Binding ListPrice, Converter={StaticResource CurrencyConverter}}" />
```

■ **Note** To support markup extensions, a new concept had to be introduced on the code side of things, with special types of properties called *dependency properties*. These will be discussed later in this chapter, but it's important to note that all properties to which you want to assign a markup extension *must* be dependency properties. This should, however, not be too much of an issue, since most of the important properties on the standard Silverlight controls are already registered as such.

Unfortunately, unlike in WPF, it's not currently possible to write your own markup extensions in Silverlight. In WPF you can create a class that derives from the `MarkupExtension` class (under the `System.Windows.Markup` namespace); however, this class is not exposed in Silverlight.

■ **Note** On occasion you may have a scenario where you want to assign a literal value to a property in XAML whose first character is an open curly bracket. However, the XAML parser will attempt to process this as a markup extension, and a runtime error will occur. This is especially likely when specifying tokens in a URI mapping with the navigation framework (covered in the next chapter), as these also use curly brackets as identifiers. To work around this issue you can use an escape sequence, which is an open bracket immediately followed by a closing bracket (`{}`), after which the rest of the value will be treated as a literal. For example, to assign a value of `{page}` to the `Uri` property of a `UriMapping` element in XAML, you would need to preface the value with `{}`, like so:

```
<navigation:UriMapping Uri="{}{page}" MappedUri="/Views/{page}.xaml" />
```

Namespaces

XAML namespaces are not a concept you will likely need to worry about much, as they are handled automatically by the XAML parser behind the scenes. However, they are worth mentioning as they are important in how named controls are managed in XAML.

If you were to create two controls in a view, assign them the same name (e.g., `ItemText`), and then attempt to compile your application, you would get the following compile error:

The type 'XXX' already contains a definition for 'ItemText'

The designer would also state the following:

Name 'ItemText' already exists in the current name scope.

This makes perfect sense, as when you name a control, the compiler creates a variable in the code-behind that you can use to reference that control in your code. If you had multiple controls with the same name in your view, Silverlight wouldn't know which of these controls you were actually referring to.

However, ensuring that all control names added to the object hierarchy are unique is simply impossible. For example, let's say you have a control (which itself is made up of a number of elements). If one of the elements making up the control had a name, then using the control multiple times in a single view would result in the named element being added to the object hierarchy multiple times. Thus, multiple controls with the same name would exist in the object hierarchy, colliding with one another. In order to permit this scenario without creating collisions, Silverlight segments the object hierarchy into *namespaces* at given intervals. Control names within each namespace still need to be unique, but controls in different namespaces can have the same name.

Your application will have a default namespace (created with the root visual), and other namespaces will be automatically created by the XAML parser where required. Examples of instances where a new namespace may be created include when you load in XAML from an external source (such

as when you use `XAMLReader.Load()`, when you use a `UserControl`, and when you use a template (`DataTemplate`, `ControlTemplate`, etc.).

Templates will be discussed later in this chapter, but let's look at a simple example of where new namespaces are created due to the use of a template. Say you have a `ListBox` control with a number of items, and you want to customize the appearance of each item by assigning a template to them. This template may have various controls within it. For example, it may have an `Image` control and a `TextBlock` control, like so:

```
<ListBox Name="FolderList">
  <ListBox.ItemTemplate>
    <DataTemplate>
      <StackPanel Orientation="Horizontal">
        <Image Source="folder.png" />
        <TextBlock Name="ItemText" Text="{Binding}" />
      </StackPanel>
    </DataTemplate>
  </ListBox.ItemTemplate>
</ListBox>
```

In the preceding XAML, the `TextBlock` in the template is assigned a name of `ItemText`. However, once the template is applied to multiple list items, there will be multiple text blocks having the name `ItemText` in the object hierarchy. Therefore, the XAML parser will create a new namespace each time the template is applied to a list item, and the controls created from that template will live in that (preventing the control names from colliding with one another in the same namespace used by the view).

There are two instances in which you are likely to find the effects of namespaces. When you use the `FindName` method on a control (to find an element with a given name below that control in the object hierarchy), it will only search for that name within the current namespace. Therefore, searching in the view containing the `ListBox` in the previous example for an element named `ItemText`, like so:

```
object namedElement = this.FindName("ItemText");
```

will not find the `TextBlock` control (because any instances of it will be in a different namespace).

In addition (as you will find out later in this book), it's possible to bind the property of a control to that of another named control (using `ElementName` binding), but you will only be able to bind to other controls that exist in the same namespace.

■ **Note** Any named control that forms a part of a template (such as the `ItemText` `TextBlock` in the recent `ListBox` item template example) will not have a variable created for it in the code-behind. This is because the control may be created multiple times in the view, and the Silverlight runtime wouldn't know which instance you are referring to.

Controls

Controls are the fundamental building blocks of user interfaces in XAML. Silverlight comes with an extensive set of core controls, quite similar what's available in Windows Forms (and WPF). Where controls are missing (such as the `TreeView` control), you may find that these additional controls are

available in the Silverlight Toolkit project (created and maintained by Microsoft) on the CodePlex web site.

In this section we'll specifically look at a subset of these core controls (which didn't exist in Windows Forms) used to lay out (i.e., manage the positioning of) controls on the user interface surface (which we'll call "layout controls"). We'll also look at some of the naming differences between control properties in Windows Forms and Silverlight. Then we'll take a look at specifying event handlers that can handle an event in the code-behind when it is raised by the given control. However, let's first look at the base control classes in Silverlight that all controls derive from.

Base Control Classes

While you will never use these directly, it's worth taking a cursory look at the `DependencyObject`, `UIElement`, `FrameworkElement`, `Control`, and `ContentControl` base control classes that other controls derive from. The inheritance hierarchy of these controls is shown in Figure 3-2.

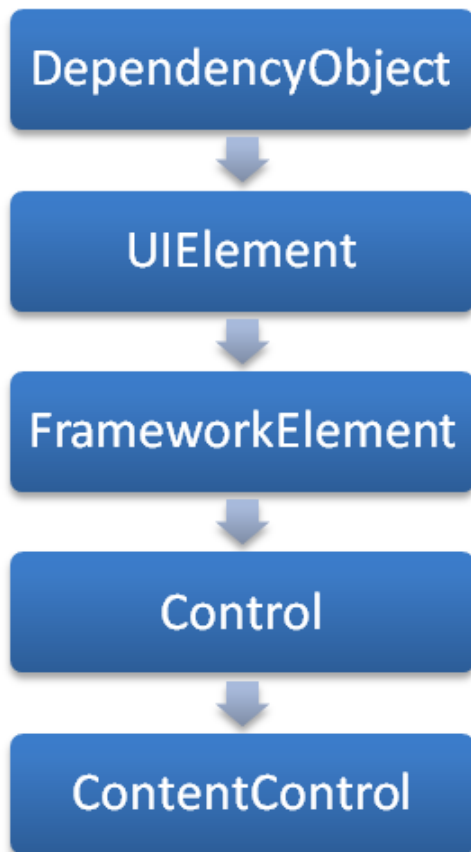


Figure 3-2. Inheritance hierarchy of the base control classes

DependencyObject

One of the concepts for supporting special XAML features in the code-behind is the dependency property. Dependency properties are special types of properties on XAML controls that are defined in a different way from normal class properties, and support additional features. Possibly the most important feature dependency properties support is the ability to accept an expression that must be resolved as its value at runtime. This feature enables markup expressions to be assigned to the property in XAML (hence enabling data binding, etc.), as the values of markup extensions need to be determined at runtime. Dependency properties are also used to enable attached properties.

In order for dependency properties to be supported on an object, that object must be a *dependency object* (i.e., inherit from the `DependencyObject` class). Dependency objects support dependency properties by enabling a class that inherits from it to register dependency properties with it, maintain the value for each of these dependency properties, and provide helper methods for setting/getting these values. All XAML controls derive from `DependencyObject` (although indirectly via the `Control` or `ContentControl` class as a general rule).

■ **Note** Whether or not an object is a dependency object or a property is a dependency property are generally not issues you will need to worry about, at least until you try to create your own user/custom controls. A more complete discussion of dependency properties and how to create them will be covered in Chapter 11.

UIElement

The `UIElement` class derives from `DependencyObject`, and adds functionality for handling user input (keyboard, mouse, and stylus) and control focus. It also adds the base layout functionality for controls.

FrameworkElement

The `FrameworkElement` class derives from `UIElement`, and adds additional layout functionality to that provided by `UIElement`, along with object lifetime events (primarily the `Loaded` event) and support for data binding (via the `GetBindingExpression` and `SetBinding` methods).

Control

The `Control` class derives from `FrameworkElement`, and is the class that most XAML controls inherit from. It adds functionality used to define the appearance of the control (most importantly by defining the `Template` property).

ContentControl

The `ContentControl` class derives from `Control`, and adds functionality to help derived controls define a content area in their control template (using the `ContentPresenter` control) into which custom content can be inserted (using the `Content` property that it defines). For example, the `Button` control indirectly inherits from this control via the `ButtonBase` class, enabling the content of the button to be customized

(such as displaying an image next to the text in the button, as demonstrated earlier in this chapter in the “Content Element Syntax” section).

Creating content controls will be detailed in Chapter 11.

Layout Controls

When you start experimenting with Silverlight, you’ll find that none of the controls have properties to enable you to specify how they should be positioned on the design surface. In Windows Forms, positioning controls was as easy as assigning values to a control’s `Left` and `Top` properties; however, these are nowhere to be found on Silverlight controls. How exactly can you then position controls on a Silverlight surface? The solution to this problem comes in the form of *layout controls*.

Layout controls are invisible container controls (i.e., controls designed specifically to act as a container for other controls) that handle the positioning of their child controls on their surface. So that controls can provide them additional information for where they need to go, layout controls have attached properties that they can use when required. There are a number of layout controls in Silverlight—each with its own logic and behavior for positioning the controls that they contain. These layout controls can be mixed and matched as required (i.e., nested within one another) in order to achieve the layout you are after.

This means that instead of designing your views using the fixed positioning of controls that you may be used to, you can design very flexible layouts that can automatically adjust to various conditions (such as browser/window resizing, etc.)—all controlled by the layout controls that you use.

Out of the box, Silverlight provides three layout controls for positioning controls in a view—the `Canvas` control, the `Grid` control, and the `StackPanel` control. These all inherit from the `Panel` class, which provides the base layout functionality to which they add their own specific layout functionality. The Silverlight Toolkit also includes some additional layout controls that you may find useful. Let’s take a look at each of these now.

Canvas

The `Canvas` layout control enables you to position controls on its surface using fixed coordinates. This layout control enables you to easily position controls, but doesn’t lead to particularly flexible layouts. It can be useful in some circumstances, however. Using it is quite simple. Here we define a `Canvas` control, and place a `TextBlock` control within it:

```
<Canvas Width="300" Height="300">
  <TextBlock Text="Hello" />
</Canvas>
```

The problem here is that the `TextBlock` will be positioned in the upper-left corner of the `Canvas`, but it has no properties enabling it to specify the coordinates where it should be positioned. Because the properties that would be required to position it depend on the type of layout control that it is contained within, the `TextBlock` control doesn’t have any properties defined on it to position it within a `Canvas` (or any other type of layout control). However, the `Canvas` can provide these properties to the `TextBlock` as attached properties, enabling the `TextBlock` to specify where it should be positioned within the `Canvas`. For this purpose, the `Canvas` control provides two attached properties: `Left` and `Top`. It can then use those values when given the task of positioning all the controls that it contains.

The following XAML demonstrates assigning values to the `Left` and `Top` attached properties provided by the `Canvas` control values on our `TextBlock` control, enabling the `Canvas` control to position the `TextBlock` control on its surface accordingly.

```
<Canvas Width="300" Height="300">
    <TextBlock Text="Hello" Canvas.Left="100" Canvas.Top="50" />
</Canvas>
```

Grid

The most commonly used layout control in Silverlight is the Grid control. Unlike the Canvas control, you do not specify the location of controls within a Grid using absolute coordinates. Instead, you divide the grid up into rows and columns (forming cells) that you can place controls into. Rows and columns can have a fixed height/width, can automatically size to fit their contents, can expand to fill all the remaining available space, and can be sized relative to the size of other rows/columns. This allows for very flexible control layouts, and thus the Grid control is generally preferred over the Canvas control for laying out controls.

When you create a new UserControl or page in your project, you will find that it already has a Grid control in place, named LayoutRoot (but with no rows or columns defined):

```
<Grid x:Name="LayoutRoot" Background="White">

</Grid>
```

The first thing you will need to do is divide it up into rows and columns. You can do so using the RowDefinitions and ColumnDefinitions properties of the Grid control. These are collections, to which you need to add RowDefinition and ColumnDefinition objects (using a combination of property element syntax and collection syntax in XAML). For example, this 300×300-pixel grid is divided into two columns and three rows:

```
<Grid Width="300" Height="300">
    <Grid.ColumnDefinitions>
        <ColumnDefinition />
        <ColumnDefinition />
    </Grid.ColumnDefinitions>
    <Grid.RowDefinitions>
        <RowDefinition />
        <RowDefinition />
        <RowDefinition />
    </Grid.RowDefinitions>
</Grid>
```

■ **Note** The XAML designer in Visual Studio makes the task quite easy, so you don't have to write all the XAML-defining rows and columns by hand. When a Grid control is selected, a light-blue margin will appear at the top left of the Grid. Clicking in one of these margins will create a row/column (depending on which margin you clicked in) at that position.

Weighted Proportion Sizing

When no column widths and row heights are explicitly set (as per the example), they will have a default value of *, which is used to specify a *weighted proportion*. Weighted proportions enable you to have columns or rows whose width/height are relative to one another. Any fixed/auto-sized columns or rows are allocated their requested space in the Grid, after which columns/rows with weighted proportions divide up the remaining available space according to their “weights.” Essentially, each column/row takes a percentage of this remaining space.

Looking at the columns in the preceding example, each column will have its default width of *. This will result in them dividing their sizes up evenly, with each column taking 50 percent of the width of the Grid. To make the first column take twice the space assigned to the second column, you can prefix the asterisk in the value of the width property with a multiplier. For example, when the columns are defined like so:

```
<Grid.ColumnDefinitions>
    <ColumnDefinition Width="2*" />
    <ColumnDefinition Width="*" />
</Grid.ColumnDefinitions>
```

the first column will take 66.67 percent of the Grid’s width, and the second column will take 33.33 percent. As the Grid is 300 pixels in width, the first column will therefore be 200 pixels wide, and the second column will be 100 pixels wide.

■ **Note** A width of * is the same as 1*.

If the Grid control doesn’t have a width/height specified, then it will expand to fill the space available to it. As its size is changed (such as when the browser/window is resized), its columns and rows whose width/height are assigned a value specifying a weighted proportion will also resize themselves accordingly. This enables you to design a layout that will automatically resize itself to fill the space available to it, resulting in a very flexible user interface layout that will adjust accordingly to whatever size is allocated to it.

Fixed-Width Sizing

You can give a column or row a fixed width/height by assigning it a numeric value (in pixels). For example, the first column in the following example will have a width of 200 pixels, and the second column will have a width of 100 pixels:

```
<Grid.ColumnDefinitions>
    <ColumnDefinition Width="200" />
    <ColumnDefinition Width="100" />
</Grid.ColumnDefinitions>
```

■ **Note** If you simply set the width of the second column to *, the remaining width of the Grid will be filled after the 200 pixels have been assigned to the first column. If the width of the Grid is 300 pixels, then the second column will be 100 pixels wide. If the width of the Grid is 500 pixels, then the second column will be 300 pixels wide.

Automatic Sizing

At times you may not know what width or height a column/row should be, but you'll want it to be calculated to expand to the size of its contents. To do this, set the column or row's Width/Height property to Auto.

Assigning Controls to Grid Cells

Once you've defined the rows and columns in your Grid, you can start assigning controls to its cells. As with the Canvas control, the Grid control provides attached properties that your controls can use to tell the Grid which row and column it should be positioned within. These attached properties are Row and Column. These properties specify an index in the RowDefinitions and ColumnDefinitions properties, and thus are zero-based (i.e., to put a control in the first row in the Grid, set its Grid.Row attached property to 0). If you want a control to span across more than one row or column in the grid, you can use the Grid's RowSpan and ColumnSpan for this purpose.

The following example demonstrates three controls being positioned within the Grid. The TextBlock control is being placed in the first column of the first row, while the TextBox control is being placed in the second column of the first row. The Button control is being placed in the first column of the second row, but will span both columns in the Grid:

```
<Grid Width="300" Height="300">
  <Grid.ColumnDefinitions>
    <ColumnDefinition />
    <ColumnDefinition />
  </Grid.ColumnDefinitions>
  <Grid.RowDefinitions>
    <RowDefinition />
    <RowDefinition />
    <RowDefinition />
  </Grid.RowDefinitions>

  <TextBlock Grid.Column="0" Grid.Row="0" Text="Name:" />
  <TextBox Grid.Column="1" Grid.Row="0" />
  <Button Grid.Column="0" Grid.Row="1" Grid.ColumnSpan="2" Content="OK" />
</Grid>
```

■ **Note** When you drop a control from the toolbox into a grid cell, a margin will be applied to the control to position it within that cell. To remove this margin easily (as you will rarely want it), right-click the control and select **Reset Layout ► Margin** from the menu. Otherwise, you can simply remove it from the XAML defining that control.

Stack Panel

The `StackPanel` control is used for stacking controls one after another, either horizontally or vertically (which you specify via a control's `Orientation` property). For example, you could put a `TextBlock` control and a `TextBox` control within a `StackPanel`, and have the `TextBox` positioned immediately following the `TextBlock`, like so:

```
<StackPanel Orientation="Horizontal">
    <TextBlock Text="Name: " />
    <TextBox Width="155" />
</StackPanel>
```

Silverlight Toolkit Layout Controls

As mentioned earlier, the Silverlight Toolkit contains some additional layout controls that you may find useful. In the `System.Windows.Controls.Toolkit.dll` assembly, you will find the `WrapPanel` and `DockPanel` layout controls.

The `WrapPanel` is much like the `StackPanel` (when stacking controls horizontally), but once the controls within it hit the right edge of the `WrapPanel`, new items will wrap and start being displayed on the next line. You might use this, for example, when you have thumbnails of photos and want to lay them out in much the same way as Windows Explorer does.

The `DockPanel` enables you to dock the controls contained within it to one of its edges, or fill the remaining space. If you miss the ability to dock controls from Windows Forms, then this will provide you the same ability in Silverlight.

XAML vs. Windows Forms Controls' Property Names

As discussed earlier, you will find that despite the core control set in Silverlight being quite similar to that available in Windows Forms, Silverlight's controls are actually implemented quite differently in places, having been developed from the ground up. In particular, some of the properties of controls in Windows Forms have different names in Silverlight. For example, most Boolean properties are now prefixed with `Is` (`IsEnabled`, `IsTabStop`, `IsHitTestVisible`, etc.). Also worth noting is that the `Visibility` property (which was named `Visible` in Windows Forms and used to show or hide controls) is now an enumeration with two possible values: `Visible` or `Collapsed`. This may seem rather strange and confusing at first, and in terms of Silverlight development it probably is! The reason for this is purely for compatibility with WPF, which supports an additional value of `Hidden` (which in WPF hides the control like `Collapsed` does, but unlike `Collapsed` still consumes space in the layout). However, since Silverlight doesn't support that value (as yet), we are left with an enumeration with just `Visible` or `Collapsed` as its possible values.

Assigning Event Handlers

Controls have the ability to raise events that the code-behind can handle to respond to something that has happened to the control (such as a Button control raising a Click event in response to being clicked). Assigning an event handler to a control in XAML is a case of simply using attribute syntax (in the same way as assigning a value to a property on the control) to assign the name of the event handler in the code-behind that should be called when the event is raised. For example, assigning an event handler called `OKButton_Click` to the Click event on a Button control is achieved like so:

```
<Button Name="OKButton" Content="OK" Click="OKButton_Click" />
```

■ **Note** As a general rule (though not mandatory), the name of event handler methods are of the form `ControlName_EventName`.

Assigning Event Handlers in XAML in Visual Studio

In Visual Studio, typing in the event name and pressing = will give you a list of event handler methods in the code-behind to choose from (that match the required method signature), and it will also give you the following option: `<New Event Handler>`. Clicking this (or simply pressing Tab) will automatically generate a name for the event handler, assigning it to the event in XAML and automatically creating an event handler method in the code-behind.

However, if you choose to type in the name of the event handler, then you will find that the event handler will not be automatically created in the code-behind. In this case, the easiest way to create the event handler is to right-click the name of the event handler in the XAML, and select the `Navigate to Event Handler` option from the context menu. This will open up the code-behind class and create the event handler method for you.

This option can also be useful if you've deleted the event handler in the code-behind and want to recreate it. It's also particularly useful when you simply want to navigate to an event handler in the code-behind for a given control in XAML.

■ **Note** Many controls define a default event, enabling you to simply double-click the control in Visual Studio's designer to automatically assign an event handler to that event on the control, and create the event handler method in the code-behind. For example, double-clicking a Button control will create a Click event handler in the code-behind, and assign that method to the Button's Click event in the XAML. You can also use the Events view of the Properties window to create event handlers without resorting to writing the XAML yourself.

Assigning Event Handlers in XAML in Expression Blend

Unfortunately, working with events is not so straightforward in Expression Blend. Events don't appear in Expression Blend's IntelliSense; you can't double-click a control to have it generate and assign the event

handler for the default event of the control, nor can you right-click an event handler name in XAML and navigate to it in the code-behind. The only way to automatically create an event handler for a control in the code-behind and assign it to the control (without performing the whole process manually) is to use the Events view in the Properties window.

Routed vs. Direct Events

Silverlight supports a special type of event called a *routed event*. Routed events are passed up the object hierarchy (known as “bubbling” the event), and the event is raised on each control until an event handler marks the event as “handled” or the root visual is reached. Although most events are *direct* events (i.e., the event handler is only called on the control that raised the event), many user interaction-related events are routed events (such as mouse/key input events and focus events), and these will bubble up the object hierarchy until they are handled.

■ **Note** Silverlight (unlike WPF) does not support any tunneling of events (where the event is raised on controls from the root visual object down to the event source).

Assigning Event Handlers in Code

Event handlers can be assigned to control events in the code-behind as per C#'s standard language syntax. For example:

```
this.Loaded += MainPage_Loaded;
```

Be particularly aware, however, that adding an event handler to a control can result in the control not being garbage-collected once it is no longer required (hence leading to memory leaks). This is because the control will still have a reference to it (via the event handler), and thus the garbage collector will assume the control is still required and not clean it up.

For example, say you add a button to a view and add a Click event handler to it, and then remove the button from the view. A reference to the button will still exist by way of the event handler, preventing it from being garbage-collected. For example:

```
// Add the button and handle its Click event
Button button = new Button();
button.Click += button_Click;
LayoutRoot.Children.Add(button);

// Now remove the button and clear the variable reference
LayoutRoot.Children.Remove(button);
button = null;
```

Even though at the completion of this code the button no longer exists in the object hierarchy and there is no variable referencing it any longer, it still will not be garbage-collected because there is still a subscription to the Click event being maintained. This will result in a memory leak. Although the impact of this example is small, if you did it many times (or with bigger objects that consume more memory),

the memory usage of your application could grow over time to potentially unacceptable levels. In order to avoid such a scenario, when you have a long-lived object with an event handler to a short(er)-lived object, remember to always remove that event handler once the control is no longer required:

```
button.Click -= button.Click;
```

■ **Note** Another solution is to use a weak event listener that subscribes to the event without maintaining a strong reference (allowing the control to be cleaned up when there are no other references to the control). The Silverlight Toolkit contains a `WeakEventListener` class that you could use for this purpose if you wish.

Creating a Simple User Interface

Now that we've gone through most of XAML's important concepts, let's put them together to create a simple XAML user interface. For this example we'll create a simple application where you enter some information about yourself (first and last name), which will then be displayed in a message box when you click a button. For this we'll be using a form containing two text boxes with corresponding labels, a combo box, and a button, all laid out using a Grid control. When in the designer, it should look similar to Figure 3-3.

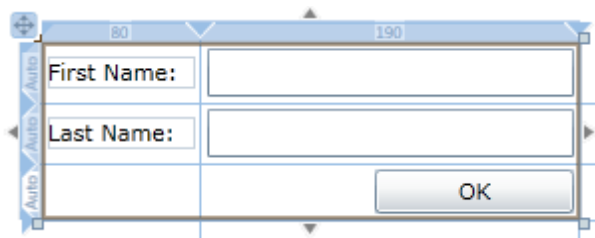


Figure 3-3. A simple user interface layout

■ **Note** It's best to do this workshop in a new project that you create using the Silverlight Application project template, rather than using the project created in the previous chapter using the Silverlight Business Application project template. Any XAML in this workshop should be added to the `MainPage.xaml` file in the project.

The first step is to configure your Grid's rows and columns (three rows and two columns). This will be used to lay out the controls in your view:

```
<Grid x:Name="LayoutRoot">
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="80" />
    <ColumnDefinition Width="190" />
```

```

</Grid.ColumnDefinitions>
<Grid.RowDefinitions>
    <RowDefinition Height="Auto" />
    <RowDefinition Height="Auto" />
    <RowDefinition Height="Auto" />
</Grid.RowDefinitions>
</Grid>

```

Add controls to the Grid's content area (two TextBlock controls for displaying labels, two TextBox controls for entering the data into, and a Button control to display the message box). Set the Name property on each user input control, and assign each control to a specific row/column in the Grid using the Grid's Row and Column attached properties:

```

<Grid x:Name="LayoutRoot">
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="80" />
        <ColumnDefinition Width="190" />
    </Grid.ColumnDefinitions>
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto" />
        <RowDefinition Height="Auto" />
        <RowDefinition Height="Auto" />
    </Grid.RowDefinitions>

    <TextBlock Grid.Column="0" Grid.Row="0" Margin="3" Text="First Name:"
        VerticalAlignment="Center" />
    <TextBox Grid.Column="1" Grid.Row="0" Margin="3" Name="firstNameTextBox" />
    <TextBlock Grid.Column="0" Grid.Row="1" Margin="3" Text="Last Name:"
        VerticalAlignment="Center" />
    <TextBox Grid.Column="1" Grid.Row="1" Margin="3" Name="lastNameTextBox" />
    <Button Grid.Column="1" Grid.Row="2" Margin="3" Name="OKButton" Content="OK"
        Width="100" HorizontalAlignment="Right" />
</Grid>

```

■ **Note** If you're typing this XAML out by hand, you will find that Visual Studio's IntelliSense is a big help, making it quite quick and easy to write. However, this project is available online for you to download if required.

Now add an event handler for the Click event to the Button control. As the Click event is the default event for the Button, the easy way to create the event handler and wire it up is to simply double-click the button in the designer. Alternatively, you can use the XAML here:

```

<Button Grid.Column="1" Grid.Row="2" Margin="3" Name="OKButton" Content="OK"
    Width="100" HorizontalAlignment="Right" Click="OKButton_Click" />

```

Add the following code to the event handler in the code-behind (to obtain the values from the controls and display them in a message box):

```
private void OKButton_Click(object sender, RoutedEventArgs e)
{
    MessageBox.Show("Hello " + firstNameTextBox.Text + " " + lastNameTextBox.Text);
}
```

And now you're done—your very first XAML user interface! Run the application and try it out. As another exercise, try building the same user interface using Visual Studio 2010's designer using a drag-and-drop approach.

Resources and Resource Dictionaries

Resources are essentially reusable objects and values, somewhat similar in concept to constants in code. Resources are ideal for when you want to define something (most commonly styles, discussed in the next section) in a location in the object hierarchy where it can be used and shared by the control it is defined on and all the controls further down the object hierarchy. The advantage of defining objects or styles as resources is that any changes you want to make can be done in the one place and automatically applied to all the controls that use it—making your application much more customizable and maintainable.

Resources can be defined in the `Resources` property of any control in your layout and assigned a key that can be used to reference it (using the `x:Key` attached property from the XAML namespace, discussed earlier in the chapter). The resource can then be used by any controls further down the object hierarchy, referencing it by the key you have given it.

■ **Note** The control you define a resource on cannot actually reference that resource. Only controls lower than the control in the object hierarchy on which a resource is defined will have access to that resource.

For example, if you wanted to define a resource available for use by any control within your XAML file, you could define it in `UserControl.Resources`. Or if you wanted it to be available project wide, you could define it in the `Application.Resources` property on the `Application` element in `App.xaml`. For example, here we'll create a `SolidColorBrush` resource in the resources of a `UserControl`, to define a color that can be assigned to control properties throughout that XAML file:

```
<UserControl.Resources>
    <SolidColorBrush x:Key="TextBoxBackgroundColor" Color="LemonChiffon" />
</UserControl.Resources>
```

Once you have defined a resource, you can assign it to control properties in XAML using the `StaticResource` markup extension, and referencing the resource by its key:

```
<TextBox Background="{StaticResource TextBoxBackgroundColor}" />
```

■ **Note** It is perfectly legal to specify more than one resource with the same key at multiple levels of the object hierarchy. This enables you to redefine a resource (and assign an alternative value to it) that had previously been defined higher up in the object hierarchy. When a resource is referenced, the runtime will traverse the object hierarchy starting at that control and working upward, until it finds a control that defines a resource with a matching key. It will use the first resource with a matching key that it finds.

Silverlight also supports defining resources in a *resource dictionary*. A resource dictionary is a separate XAML file whose root node is `ResourceDictionary`, and that defines a collection of resources that you can then *merge* into the resources of a control.

■ **Note** You can easily create a new resource dictionary file using the Silverlight Resource Dictionary item template from the Add New Item dialog in Visual Studio or Expression Blend.

You define resources in a resource dictionary in exactly the same way as the resources of a control (as shown earlier):

```
<ResourceDictionary
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">

  <SolidColorBrush x:Key="TextBoxBackgroundColor" Color="LemonChiffon" />
</ResourceDictionary>
```

Once you have defined your resource dictionary, you will need to merge it into the resources of a control (enabling those resources to be used by the controls lower than it in the object hierarchy). The best example of a resource dictionary is the `Styles.xaml` file in the Silverlight Business Application project template (in the `Assets` folder). You can merge it into your application's resources by adding a `ResourceDictionary` element to the `Resources` property (using property element syntax), and adding the resource dictionary file to its `MergedDictionaries` property:

```
<Application.Resources>
  <ResourceDictionary>
    <ResourceDictionary.MergedDictionaries>
      <ResourceDictionary Source="Assets/Styles.xaml"/>
    </ResourceDictionary.MergedDictionaries>
  </ResourceDictionary>
</Application.Resources>
```

One of the advantages of defining the style resources in a resource dictionary is that you can simply drop a different `Styles.xaml` file into the `Assets` folder (with the same resources defined by with different values) to change the theme of your application, overwriting the existing one. The next time you run your Silverlight application, it will be completely restyled according to the resources in the new `Styles.xaml` file.

■ **Note** Even strings can be defined as resources, which can be very useful to help you localize your application by defining the strings used in your application in resource dictionaries (one resource dictionary per language) and referencing those, rather than hard-coding the strings into your user interface XAML files. You can then merge the appropriate resource dictionary (based upon which language the application should use) into your application's resources.

Styles

Styles are collections of property values for a control that is stored as a resource; they can be defined once and applied to multiple controls. If you've done some web-based projects previously, you might like to think of styles as a type of CSS, but for XAML.

Rather than, for example, specifying on multiple TextBox controls that the text should be brown and bold and the background should be lemon chiffon, like this:

```
<TextBox Background="LemonChiffon" Foreground="Brown" FontWeight="Bold" />
```

you can instead create a style that defines these property values (which you create as a resource on a control somewhere higher up in the object hierarchy):

```
<UserControl.Resources>
  <Style TargetType="TextBox" x:Key="TextBoxStyle">
    <Setter Property="Background" Value="LemonChiffon" />
    <Setter Property="Foreground" Value="Brown" />
    <Setter Property="FontWeight" Value="Bold" />
  </Style>
</UserControl.Resources>
```

This style has been defined on the Resource property of the UserControl, using property element syntax (discussed earlier in the chapter). Note that the type of control that the style can be applied to is specified using the TargetType property of the style (this is mandatory). We also assign the style resource a key using x:Key. Then it's a case of creating a Setter element for each property you want to define a value for, providing it the name of the property and its corresponding value.

You can then apply this style to each control that should use it using the StaticResource markup extension and the resource key assigned to the style:

```
<TextBox Style="{StaticResource TextBoxStyle}" />
```

This example *explicitly* applies the style to the control (by assigning the style resource to its Style property), but a new feature in Silverlight 4 is the ability to *implicitly* apply a style to controls instead. Implicit styling enables you to create a style and have it automatically applied to all controls matching the style's TargetType. This makes it a lot easier to theme your application, and reduces the verbosity of your XAML. The downside is that implicit styling doesn't allow you to selectively apply the style to given controls matching the TargetType. Implicit styles are identical to explicit styles, except you omit the resource key when defining the style resource. This style will then be automatically used by all controls lower in the object hierarchy matching the specified TargetType, without their Style properties needing to be set.

■ **Note** Explicit and implicit styles will be discussed in detail in Chapter 9.

Templates

Templates enable you to completely customize the way a control (or part of a control) looks, and are arguably one of XAML's most powerful features. Whereas styles enable you to define a set of property values for a control (or part of a control), templates enable you to completely redefine its underlying XAML. Because XAML enables a clear separation between look and behavior (maintaining only a contract between the two), assuming you fulfill the needs of the contract you can modify the look of the control to whatever you desire. These templates can be stored as a resource and reused by multiple controls. Two of the most common types of templates are *control templates* and *data templates*.

A control template enables you to completely redefine the entire XAML for a control (hence completely changing the way a control looks). For example, you may want to customize a Button control to have a flat look. Then any Button controls that you apply this template to will have that look. Control templates are covered further in Chapter 9 and Chapter 11.

A data template enables you to define a custom way in which data should be displayed in a control. For example, you may want to customize the way each item in a ComboBox control is displayed (such as displaying the flag of a country next to its name in a ComboBox that displays a list of countries). Data templates are covered further in Chapter 6 and Chapter 7.

Data Binding

Data binding is primarily used for displaying data from an object in a user interface and enabling user input controls to automatically update the data on that object (essentially creating a link between a control and the source of the data it should display/update).

Rather than “pushing” data into the user interface from the code-behind, XAML's strong data binding capabilities enable the user interface to “pull” data in from the appropriate source and populate itself as required.

Silverlight supports a very rich and flexible data binding model, and this data binding model is one of Silverlight's key strengths (and one you will find extremely important when building business applications). In Silverlight you can bind to objects, resources, and even properties on other controls. As long as a property is a dependency property, you can bind it to almost anything. Data binding requires the use of the Binding markup extension.

Data binding consists of a *source* and a *target*. The source is usually an object (or collection) containing the data to which you are binding, and the target is a property on a control that will display and/or update some data exposed by that object.

For a control to bind to a data source (such as an object), generally that source will be assigned to the DataContext property of a control, enabling the properties of that control to bind to the source's properties. This value will also be inherited down the object hierarchy, meaning that any controls below that control will also have the object assigned to their DataContext property. This will be discussed further shortly. There are other ways of setting the source of a data binding, although this is the most common way of doing so. The other ways will be detailed in Chapter 10.

You may be familiar with data source controls such as the BindingSource control in Windows Forms, or the ObjectDataSource control in ASP.NET. These data sources often have the role of obtaining the data to be displayed in the user interface (handling the pulling process), which controls in the view

can then bind to. There is no data source control included in the standard control library in Silverlight; however, Chapter 6 will cover the `DomainDataSource` control (which is a part of RIA Services), and discusses data source control type behavior. For now we'll focus on more direct means of obtaining the source data.

Binding to an Object

For this example we'll use the simple user interface created earlier, and walk through instantiating an object (in XAML), assigning the object to the `DataContext` property of a control (to act as the source of any data bindings on that control), and binding the user input controls to it (which will display the object's property values and enable them to be updated by the user).

Say we have a class called `Person`, containing the properties `FirstName` and `LastName` (matching the fields in our user interface created earlier):

```
public class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
}
```

We could push a `Person` object into the user interface, or use a data source control to obtain it, but for this example we'll take a XAML-only approach, and instantiate the object as a resource and bind to that. In order to do so, we must first add a namespace declaration to our XAML file to reference the CLR namespace in which the `Person` class is defined, and assign a prefix that we can use to reference the namespace by (shown in bold):

```
<UserControl x:Class="DataBindingSample.MainPage"
    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:my="clr-namespace:DataBindingSample"
    mc:Ignorable="d"
    d:DesignHeight="300" d:DesignWidth="400" >
```

Then we can instantiate the object (as a resource) in the resources of the `UserControl`:

```
<UserControl.Resources>
    <my:Person x:Key="personObject" />
</UserControl.Resources>
```

■ **Note** You can assign the object initial values if you wish when instantiating the object, in the same manner as you would assign property values on any control in XAML:

```
<UserControl.Resources>
    <my:Person x:Key="personObject" FirstName="Chris" LastName="Anderson" />
</UserControl.Resources>
```

Now that the object is instantiated, we can assign it to the `DataContext` property of each control that wants to bind to it using the `StaticResource` markup extension. For example:

```
<TextBox DataContext="{StaticResource personObject}" />
```

However, assigning an object like this to the `DataContext` property of each control in the view that wants to bind to it would be a painful experience—especially when you have many controls. That said, as mentioned earlier, the value assigned to the `DataContext` property of a control is inherited down the object hierarchy. If all the controls that you wished to bind to the object were in a `Grid`, you could simply assign the object to the `Grid`'s `DataContext` property, and all controls within it would inherit its value (without requiring you to assign their `DataContext` properties individually, as you will see shortly).

The final step is to bind the controls to this data by writing a data binding expression (using the `Binding` markup expression). Here we want to associate the `FirstName` property of the person object with the `Text` property of the `TextBox` to show the person's first name (using the `Path` property of the `Binding` markup expression), and the `LastName` property of the person object with the `Text` property of the `TextBox` to show the person's last name:

```
<Grid x:Name="LayoutRoot" DataContext="{StaticResource personObject}">
    <TextBox Grid.Column="1" Grid.Row="0" Margin="3"
        Name="firstNameTextBox" Text="{Binding FirstName , Mode=TwoWay}" />

    <TextBox Grid.Column="1" Grid.Row="1" Margin="3"
        Name="lastNameTextBox" Text="{Binding LastName , Mode=TwoWay}" />
</Grid>
```

■ **Note** The default property of the `Binding` markup expression is the `Path` property, enabling you to specify a value for the `Path` property without needing to explicitly specify which property you are assigning it to in the binding expression. This means that

```
Text="{Binding Path=FirstName , Mode=TwoWay}"
```

can instead be written as

```
Text="{Binding FirstName , Mode=TwoWay}"
```

You may have noted that we are explicitly setting the `Mode` property on the binding expression. The binding mode specifies the direction of the data flow. There are three different binding modes:

- **OneTime:** This option enables the target to obtain a value from the source object, after which the binding is discarded (until a new object is assigned to the data context). Changes to the source object will *not* be propagated back to the target property, and changes to the target property's value in the user interface will *not* be propagated back to the source object. Use this option when the source object is guaranteed not to change.
- **OneWay:** This option enables the target to obtain a value from the source object and be notified when that value is updated (so it can update itself with the new value). Changes to the source object *will* be propagated back to the target property, but changes to the target property's value in the user interface will *not* be propagated back to the source object. Use this option when the source object may change, but the value cannot be changed in the user interface (such as when binding to a `TextBlock` control).
- **TwoWay:** This option enables the target to obtain a value from the source object, be notified when that value is updated (so it can update itself with the new value), and update that value with changes made to the target. Changes to the source object *will* be propagated back to the target property, and changes to the target property's value in the user interface *will* be propagated back to the source object. Use this option when any changes made to the target in the user interface (such as the `Text` property of the `TextBox` control) should update the source object accordingly.

By default, data binding expressions use the `OneWay` mode. Since changes made to the data in the user interface are not saved back to the source object with `OneWay` mode, it is therefore particularly necessary for data input controls to explicitly specify `TwoWay` mode on their binding expressions.

■ **Note** In order for changes made to a data-bound property in the code-behind to be propagated to the user interface, you will need to implement the `INotifyPropertyChanged` interface on your class, and call the `PropertyChanged` event in the property's setter. Changes to the values of automatically implemented properties will not notify the user interface that they have been updated (as they unfortunately do not support the `INotifyPropertyChanged` interface), and hence any changes to them will not be reflected on the screen. Note, however, that dependency properties *do* automatically notify any controls bound to them when their value changes. Further discussion of this topic and examples of implementing the `INotifyPropertyChanged` interface can be found in Chapter 7.

Any changes made in the text boxes in the user interface will update the bound properties automatically. You can prove this by updating the values in the text boxes and then displaying the corresponding property values from the bound object in a message box:

```
private void OKButton_Click(object sender, RoutedEventArgs e)
{
    Person person = LayoutRoot.DataContext as Person;
    MessageBox.Show("Hello " + person.FirstName + " " + person.LastName);
}
```

■ **Note** There are many advanced options you can use when binding, such as converting the data in the source object to something different before assigning it to the target (and vice versa) with a *value converter* (particularly useful for converting the data from one type to another), and specifying how invalid data exceptions are handled. These more advanced data binding concepts are covered in Chapter 10.

Binding to a Collection

When you are binding a control to a collection, you will find that it is a slightly different scenario to that of binding to a single object. Controls that expect a collection to bind to (e.g., `DataGrid`, `ComboBox`, and `ListBox`) have a different property (often named `ItemsSource`) that the items to be displayed in that control will be populated from; they don't obtain the collection from the `DataContext` property.

For example, say you have a `Title` property on the `Person` object:

```
public class Person
{
    public string Title { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
}
```

and a `Titles` collection containing a list of titles that the user can select from:

```
public class Titles : List<string>
{
    public Titles()
    {
        this.Add("Mr");
        this.Add("Mrs");
        this.Add("Ms");
        this.Add("Miss");
    }
}
```

The `DataContext` property of that `ComboBox` control will be assigned the `Person` object, and the `ItemsSource` property of the `ComboBox` control will be assigned the `Titles` collection. The `SelectedValuePath` property of the `ComboBox` control can then have a data binding expression assigned to link the selected item in the `ComboBox` control with the value of the `Title` property on the `Person` object:

```
<UserControl.Resources>
    <my:Person x:Key="personObject" />
    <my:Titles x:Key="titlesCollection" />
</UserControl.Resources>

<ComboBox DataContext="{StaticResource personObject}"
    ItemsSource="{StaticResource titlesCollection}"
    SelectedValuePath="{Binding Path=Title}" />
```

■ **Note** Visual Studio 2010 provides a very good data binding expression builder (from the Properties window), which can be very helpful when learning how to write data binding expressions. It also has a great way to easily create forms (complete with bindings) from an object. Simply add your object as a new data source to the Data Sources tool window, select the type of controls the object and each of its properties should use, and drag it from the Data Sources tool window onto your design surface. You can then change the generated form as you see fit.

Designing User Experiences

Although I can't go into too much depth here about designing user experiences (since it is such a big topic and beyond the scope of this book), it's worth touching on the concept anyway, as it's something you will often hear about when XAML-based technologies are being discussed.

Traditionally, developers have an inherent focus on function over form when it comes to designing applications (i.e., ensuring that they work as per the specifications, rather than focusing on how they look and behave). However, there is a big push nowadays for producing applications that are not only functional, but are also a pleasure for the user to use. From the wish to achieve this outcome, the process of *user experience* (often referred to as *UX* for short) design emerged.

Contrary to popular belief, designing user experiences is not about designing the look of the application (which is the role of a graphic designer), but instead designing the behavior of the application. The outcome of this process should be a user interface that is tailored to the needs of the user and how they interact with the application. Hence, the user experience design process may or may

not be the role of a graphic designer, depending on their skill set (user experience design is a whole skill in itself, and not necessarily related to the ability to create visually pleasing user interfaces). Instead, it has given rise to the specialized role of the user experience designer, whose responsibility is to design the user interface from a user-centric perspective.

The concept of user experience design is regularly used in conjunction Silverlight and WPF development because these technologies leverage the power of XAML for designing user interfaces. XAML is the key technology that enables the design of unique and customized user experiences in Silverlight. Because of the incredible flexibility provided by XAML to intricately customize the look and behavior of your application, unique user experiences (and hence unique user interfaces) can emerge.

Because the controls are highly customizable, you are no longer locked into only the functionality and appearance that a control provides (including its limitations). Hence, you have a lot of power to bend the user interface to your requirements.

In particular, there is a big debate over the overuse of the DataGrid control in business applications. The DataGrid control is generally used as a quick, easy-to-use all-purpose control (for the developer); however, this is often at the expense of the user's experience. So every time you use it, you should ask yourself whether you should be using that control and reinforcing the old paradigm (to the possible detriment of the user), or instead see if there is a way to achieve a better interaction with the user for that feature.

User experience design should be done before you actually start development on the application, as the outcome of the user experience design will have a big impact on the core design of the application. SketchFlow (a part of Expression Blend) is a tool that provides a lot of support for the user experience design process, and provides a basis from which you can build a fully functional application.

The Designer/Developer Workflow

Again, the designer/developer workflow is another topic that is beyond the scope of this book, but it's worth noting that XAML is the key enabler in providing a clear separation of concerns between the user interface and the code, and hence a clear separation between the roles of the designer and those of the developer. Generally, the designer will be focused on the look and behavior of the user interface, while the developer will be focused on writing the code that services the needs of the user interface.

The clear separation of these roles enabled by XAML reduces the friction between the designer and the developer, enabling a designer to work in Expression Blend on a feature for the user interface at the same time the developer is working in Visual Studio on the code requirements of the corresponding feature, without clashing.

XAML User Interfaces: A Different Perspective

XAML brings many benefits to developing user interfaces, but requires you to take a different approach to laying out the controls on the page and interacting with the code-behind. If you primarily develop Windows Forms applications, then you are probably used to simply dropping controls on a form and pushing data into them from the code-behind. Essentially, you are probably used to the code-behind controlling the user interface, with the user interface functioning in a subservient role.

While this is still possible, the power of XAML enables the user interface to take the controlling role, with the code-behind (or a ViewModel, which is a part of the Model-View-ViewModel [MVVM] design pattern) merely supporting it. This requires somewhat of a mental leap, as your user interface should now pull the data into itself—a concept enabled by XAML's advanced data binding capabilities.

This new perspective for designing and structuring an application requires a new supporting pattern, and this comes in the form of the new MVVM design pattern. The primary reasons you'd want

to use this design pattern are to help maintain a clean separation between the designer and developer components of a project, and to design a more testable application than was previously possible. I'll cover this pattern in depth in Chapter 12, but in the meantime, to keep the number of new concepts being demonstrated at any one time to a minimum, I'll discuss the more familiar and easier approach of simply using the code-behind to support and control the user interface.

Summary

As you've seen through this chapter, there is a lot learn about XAML, and yet this is just an introduction covering XAML's core features! Chapter 10 will cover many more advanced features of XAML, but this chapter should have given you enough to get started developing user interfaces in Silverlight, and you will use many of the concepts covered here in the following chapters.

The power and flexibility of XAML unfortunately results in a steep learning curve that even experienced and productive developers in other technologies will need to climb and overcome. This will take some time (and possibly lead to some frustration); and will require patience, willingness to learn, and the expectation that you will be less productive as a developer than normal while you get up to speed. However, once you do so you will really start to appreciate the benefits and new possibilities that the use of XAML provides. You will also be able to transfer many of these new skills to WPF development (which shares most of Silverlight's XAML concepts) as it gains in popularity as a technology for developing rich desktop applications.

The best way to learn XAML is probably to start with dragging controls from the toolbox onto the designer, setting properties in the Properties window, and observing the XAML that the designer produces. The designer, however, won't always be there for you, and you may find that you have to resort to writing XAML by hand at times. Therefore, a complete reliance on letting the designer write the XAML for you is probably unwise. Inspecting existing XAML files created by other developers or designers and cross-referencing the XAML in them with what's been discussed in this chapter is another method that might help you get up to speed.

In the end you may choose to use the designer for most tasks and let it produce the XAML, but it's important to at least understand the underlying concepts of XAML and have some familiarity with its syntax. If you choose to handwrite XAML (or find yourself needing to do so), you will find that once you understand the syntax, the IntelliSense in both Visual Studio 2010 and Expression Blend make writing XAML a breeze.

CHAPTER 4



The Navigation Framework

One of the issues (as a developer) in the early versions of Silverlight was that when you created a new Silverlight project, you were essentially given a blank slate—with no existing user interface structure/framework to help you get started. This created quite a barrier for creating business applications in Silverlight, which typically contain complex user interfaces with numerous views that the user must navigate between.

■ **Note** When you create a new project using the Silverlight Application project template, you get exactly the same amount of user interface structure that was available in Silverlight 2.

Therefore, to attempt to create a business application of any size in Silverlight 2, you would have to start by creating a user interface framework. However, Silverlight 3 introduced the navigation framework to act as the foundation and structure for your application's user interface, which helped this problem significantly. It provides a user interface framework for creating single document interface (SDI) applications, in which only a single screen of information (i.e., content) is presented to the user at a time, but users are enabled to navigate between these screens (which I will refer to as “views”).

Two project templates were added in Silverlight 3 (the Silverlight Navigation Application and Silverlight Business Application project templates discussed back in Chapter 2), which already implement the navigation framework and help you get started building an application.

In this chapter, we'll take a look at how you make use of the navigation framework in your applications—but let's first discuss the various types of application navigation patterns, and where the navigation framework fits into these.

Getting Started with the Navigation Framework

A typical business application consists of a number of views, which act as content to be displayed within an outer shell. This shell generally includes ways of navigating around the application (along with headers, footers, etc.). The project created back in Chapter 2 using the Silverlight Business Application project template is a good example of this. If you haven't done so already, create a new project using this template now. In this project you will have a shell and two views (Home and About). When you compile and run this project, you will get the user interface shown in Figure 4-1.



Figure 4-1. The default user interface created with the Silverlight Business Application project template

When you click the “home” and “about” buttons in the header, the view that is displayed in the body of the application’s user interface will change accordingly. The navigation framework provides the infrastructure for this user interface.

If you’ve previously worked on an ASP.NET Web Forms project, then you’ll no doubt be familiar with the concept of master pages and content pages. Silverlight’s navigation framework mimics this somewhat, providing a similar type of structure and functionality for your Silverlight application, even integrating with the browser navigation functions (i.e., the Back/Forward buttons) to enable the user to navigate through their navigation history in your application in much the same way as if it were a web site.

Looking now at the project structure (as shown in Figure 4-2) and how it relates to what you see on the screen, the `MainPage.xaml` file acts as the application shell, while the `Home.xaml` and `About.xaml` files in the Views folder are the individual views that are displayed as content within the shell.

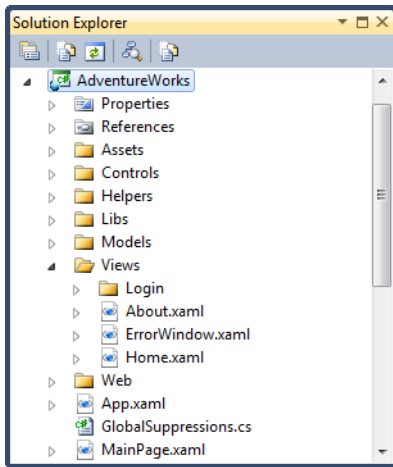


Figure 4-2. The project structure

■ **Note** WPF also has a navigation framework, and the structure of the navigation framework in Silverlight was designed to be closely aligned with its WPF counterpart.

Components of the Navigation Framework

The two core components of the navigation framework are the Frame control and the Page class. There are also two other helper objects that enable you to navigate between views and get information about a navigation operation—the *NavigationService* class and the *NavigationContext* class. Let's take a look at each of these components and what roles they play in the navigation framework.

The Frame Control

The Frame control is a host container for views. Therefore, it is normally used in a *master page* that provides the outer shell of the application, and hosts the views as content. When provided with a URI (specifying the view to load and display), it will pass that URI through a converter to obtain a path to the view that it can then navigate to.

The Frame control also provides browser journal (i.e., history) integration and handles navigation events raised by the browser (Back/Forward button clicks, new URIs entered via the address bar, etc.).

In the project you created from the Silverlight Business Application project template, the *MainPage.xaml* file (acting as the application shell) contains a Frame control, enabling it to host the views.

The Page Class

Views must inherit from the `Page` class, which enables them to be hosted in a `Frame` control.

■ **Note** It's worth noting that there's somewhat of an inconsistency in the use of terminology within the navigation framework (pages vs. views), which should be clarified to avoid confusion. In the Model-View-ViewModel (MVVM) design pattern (discussed in Chapter 12), we refer to each of these “content” pages as a *view*. In the default Silverlight Business Application project template these views reside in the `Views` folder, which makes perfect sense. However, the views inherit from the `Page` class, and you use the Silverlight `Page` item template when adding a new view to your project. The term “page” tends to conjure up the image of a web page, which is not really a valid comparison.

■ **Note** From its name, you might also expect the `MainPage` class to inherit from `Page`, but it actually inherits from `UserControl`. Ideally, the `Page` class would have been named `View` instead, which would have solved this inconsistency; however, this would put it out of alignment with WPF's navigation framework. If this bothers you, you may wish to consider creating a class called `View` that inherits from `Page` (which you can use in `Page`'s place). Generally, it's also a good idea to suffix the name of each view with `View` (e.g., `HomeView`, `AboutView`, etc.). Throughout this book, when we mention the term “view,” it will generally be in reference to one of these XAML files that inherits from `Page`.

In the project you created from the Silverlight Business Application project template, you will find that the `Home.xaml` and `About.xaml` files in the `Views` folder both inherit from the `Page` class.

You can create additional views by using the Silverlight `Page` item template when adding a new item to your project.

■ **Note** Ensure that you set the `Title` property on each view (this is a property inherited from the `Page` class) to the name of that view. The value of this property will be displayed in the browser's title bar (and the history) when the view has been navigated to.

The NavigationService Object

Each `Frame` control instance creates a `NavigationService` object that is shared with the views it hosts (which is available as a property on the `Page` class each view inherits from). This `NavigationService`

object is used to maintain an independence between the views and their host (i.e., the Frame control), and contains the methods that enable you to programmatically navigate around the application.

The NavigationContext Object

The NavigationContext object meanwhile contains information about the navigation operation that led to the current view being displayed (which amounts to the query string appended to the URI).

You define the layout/structure of your user interface (i.e., the outer shell) in the MainPage class, and this generally provides the means to navigate between the views. The MainPage class inherits from UserControl and the views each inherit from the Page class (which is a part of the navigation framework). By inheriting from the Page class, this enables the views to be hosted within a Frame control and handle the various navigation events provided by the navigation framework.

Navigating Between Views

In order to load and display a view in a Frame control, you need to *navigate* to it. Navigation operations can be initiated either by code, by a control (i.e., the HyperlinkButton control), by property bindings, or from the browser (via the user clicking the Back/Forward buttons or changing the URI in the address bar). How you invoke a navigation operation depends on where you are attempting to initiate the navigation from. In this section we'll take a look at the various ways that you can initiate a navigation operation in your application, but let's first look at how you specify a view to load and display using a URI.

View URIs

In much the same way that a URL can be used to specify the location of a web page to retrieve from a server, you can specify the location of a view to load and display in your application using a URI.

■ **Note** Essentially, a URL (Uniform Resource Locator) is a URI (Uniform Resource Identifier)—a URL is a subset of a URI, as is a URN (Uniform Resource Names). There is a lot of confusion as to the difference between the terms and when to use them. “URI” is the more generic term, and thus is generally the safest to use. More information on the differences can be found in the following Wikipedia article:

[http://en.wikipedia.org/wiki/Uniform Resource Identifier](http://en.wikipedia.org/wiki/Uniform_Resource_Identifier).

This URI will specify the path to the view in your project. For example, in the project you created from the Silverlight Business Application project template, the Home.xaml file would have a URI of /Views/Home.xaml. If you look again at Figure 4-2, you will see that the Home.xaml view can be found in the Views folder, which you can see is represented in the URI. You can therefore load and display this view by navigating to this URI.

In order to pass data to a view in the URI, you can append query string parameters to it, in much the same way that you might append query string parameters to the URL to a web page. You can even perform fragment navigation. (These will be discussed later in the chapter.)

Before attempting to load a view from a URI, the Frame control will run it through a URI mapper using a custom mapping scheme that you can define. This means that instead of specifying the path to the view, you can have a custom URI for each view that you use throughout your application. Then, before loading a view from a URI, the Frame control will convert the URI to the path of the view that it represents via the mapping scheme. The output from this conversion will specify the path to the corresponding view, which can then be loaded and displayed in the Frame control. This URI mapping will be discussed in detail later in this chapter.

■ **Note** It's even possible to navigate to a view in another assembly referenced by your project. The path to the view needs to be of the form `"/[assembly];component/[viewpath]"`. For example, if the `Home.xaml` view existed in a referenced assembly named `MyViews.dll`, the path to navigate to it would be `"/MyViews;component/Views/Home.xaml"`.

Navigation Methods on the Frame Control

When you are trying to navigate to a view and display it within a Frame control, you can simply use the navigation methods on the Frame control:

- `Navigate`
- `GoBack`
- `GoForward`
- `StopLoading`

As you can see, the `Navigate` method takes a URI as a parameter. For example, the following code instructs a Frame (named `ContentFrame`) to navigate to the view that `/Home` maps to:

```
ContentFrame.Navigate(new Uri("/Home", UriKind.Relative));
```

■ **Note** We are explicitly specifying that we are passing in a relative URI to the `UriKind` parameter. While the `Uri` class has a constructor where only the URI needs to be passed in, it will assume it is an absolute URI instead of a relative URI, and an exception will be raised.

Navigation Methods on the NavigationService Object

The previous example is fine when you are initiating the navigation from the page hosting the Frame control. However, you also might attempt to initiate a navigation operation in code from within a view itself, where you don't have a reference to the Frame control. In this case, the current view would be closed and the view being navigated to would be displayed in its place. This is where the `NavigationService` object (provided by the inherited `Page` class via a property) comes in. It has the

methods that you can use to request a navigation operation on the Frame control hosting the view. It contains the same navigation methods that you will find on the Frame control, and thus is used in the same way:

```
NavigationService.Navigate(new Uri("/Home", UriKind.Relative));
```

Using a HyperlinkButton Control

Another way to initiate a navigation operation is to use the HyperlinkButton control. The HyperlinkButton control contains built-in functionality to request a Frame to navigate to a given view. Here is some sample XAML to define a HyperlinkButton element to navigate to the Home view:

```
<HyperlinkButton Name="HomeLink" Content="home"
    NavigateUri="/Home" TargetName="ContentFrame" />
```

In this example we are doing the following:

- Giving the element a name (Name)
- Providing the URI to navigate to (NavigateUri)
- Specifying the name of the Frame to navigate within (TargetName)
- Setting the text to display on the screen (Content)

Note that if there is only one Frame on the page, then you can leave the TargetName property blank. If you are using the HyperlinkButton control within a view, then you can leave the TargetName property blank in that case too.

■ **Note** The HyperlinkButton control can also be used to navigate to standard HTML web pages, although these won't be displayed within a Frame control (attempts to do so will raise an exception). To navigate to a web page, set the TargetName property to `blank` (which will open the link in a new browser window), or set it to `self` (to open it in the current browser window/tab—closing your Silverlight application).

Using the Source Property of the Frame Control

The Frame control has a Source property to which you can assign a URI for the Frame control to navigate to and display. This is most useful when you want to bind to a data source containing the URI of the view to be displayed. Note, however, that the Source property expects a value of type `Uri`, so if you wish to bind it to a string, then you will need to use a `ValueConverter` to convert the value to a type of `Uri`.

User-Initiated Navigation

One of the features of the navigation framework is that the current view's URI is appended to the application's URL in the browser's address bar as a bookmark. The user can enter/ modify this

bookmark manually and have the application navigate to the view that they specify. The Frame control will note this, parse the URI in the bookmark using a URI mapping scheme (discussed later in the chapter), and then load the corresponding XAML view within the Frame. This is a concept known as *deep linking*, which will be discussed in more detail later in this chapter.

Passing Data Between Views

In standard web development, HTML pages are essentially stateless—i.e., when the server processes each page request, it is not aware of the requests that came before it. Session variables, cookies, and view state are workarounds to this issue in order to simulate statefulness as you navigate through a web site. To pass data to another page in a request, you would usually pass this as query string parameters appended to the end of the URL of the page that accepts the data. Alternatively, you may post complex state data containing state information (such as view state) back to the server to be processed.

By being entirely client side, Silverlight applications aren't subject to this problem of statelessness, and are inherently stateful (in the same way as a rich desktop application is stateful). The state of the application is maintained as the user navigates between the views. However, it would not be particularly good design to maintain state information belonging to a small set of views at a global (i.e., application) level simply as a means of passing state/data between them. Since each view (when using the navigation framework) has an associated URI, the navigation framework follows the standard URI pattern and enables you to include query strings in them, which you can harness to pass data between the views. Let's take a look at doing this now.

Passing Data Using Query String Parameters

Query string parameters enable you to pass simple data types (strings, integers, etc.) from one view to another as you navigate between them (such as passing the ID of a selected product from the product summary list view to the product details view).

The query string parameters are appended to the end of the bookmark in the URI when navigating to a view that will consume them. The format that these normally take is

```
/Views/ProductDetailsView.xaml?ProductID=879
```

In this example, we are passing in a value of 879 to the ProductID query string parameter of the ProductDetailsView view. A question mark (?) is appended to the URI, followed by the parameter name and its value (separated by an equal sign). Therefore, to pass the product ID to another view, you will need to specify that full URI as the URI to navigate to.

Multiple parameters can be passed, with each needing to be separated by an ampersand (&). For example:

```
/Views/ProductDetailsView.xaml?ProductID=879&SupplierID=437
```

Depending on your URI mapping scheme, the query string parameters don't need to take this extended form. Instead we could use a much friendlier URI (in place of the one from the first example), such as

```
ProductDetails/879
```

That will be mapped and parsed internally in the application using the specified URI mapping scheme to convert it to its standard form. How this is achieved will be discussed later in this chapter when we cover URI mapping.

Reading Query String Parameters

When a view needs to read the parameters passed to it in its query string, it can use the `QueryString` property on the view's `NavigationContext` object. This is an indexed collection, where the index is the name of the property you wish to retrieve the value for. For example:

```
object productID = NavigationContext.QueryString["ProductID"];
```

One of the problems with this code is that it returns an object that you will need to cast to the required type. However, if that parameter wasn't provided in the query string, then a null value will be returned, and there's no guarantee that it will be of the correct type (requiring additional code to check for this before attempting to convert it).

Parsing query string parameters each time you use them makes for messy code that is hard to read and maintain. This process generally requires you to

- Ensure the parameter exists
- Check that the value is of the correct type
- Convert the value from a string to the required type

The best way to handle this is to require that all expected query string parameters have a corresponding strongly typed property (getter only) in the view's code-behind, and return the value as the correct type. If a required parameter has not been passed in or is malformed (i.e., cannot be converted to the required type or contains an invalid value), then it should return a default value or throw an exception. Wrapping the retrieval of a query string parameter in a property this way makes for much cleaner and more maintainable code. For example, the following code is used to handle access to the `ProductID` parameter. As you can see it returns a typed value (an `int`), checks to see if the value has been passed in, and then parses out the value. Whenever you need the value of the `ProductID` parameter, you simply need to reference the `ParamProductID` property to obtain it.

```
private int ParamProductID
{
    get
    {
        int paramValue = 0;
        const string paramName = "ProductID";

        if (NavigationContext.QueryString.ContainsKey(paramName))
            int.TryParse(NavigationContext.QueryString[paramName], out paramValue);

        return paramValue;
    }
}
```

Passing Complex Data Types Between Views

Simple data types can be passed easily between views using query string parameters. However, what if you wanted to pass some more complex data types (such as a collection of objects)? What if, for example, you wanted to pass the entire product object to the product details view instead of just its ID? You might possibly be able to serialize the product object to a string and pass that as a parameter in the query string, but this is exceedingly messy and open to exceeding the maximum length of an acceptable URL (which depends on the browser—but the general recommendation is that it should not exceed approximately 2,000 characters, and the usual advice for user-friendliness is to keep it short). Of course, we could create an application-wide cache to store objects in and retrieve them from, but this isn't a particularly nice or robust design.

Unfortunately, the Silverlight navigation framework has no means built into it to “post” or pass complex data types between views without resorting to storing them globally in an application-wide cache (before navigating away from a view), and then retrieving (and removing) them from the cache when the destination view has been navigated to. Ideally, there would be an additional parameter to the `NavigationService.Navigate` method that takes an object that you could attach your state-based values to that can be passed directly to the view being navigated to, but sadly this is not the case. The navigation framework is a part of the Silverlight Toolkit (and is hence open source), enabling you to add the behavior yourself if you were so inclined. However, modifying a framework like this—where the change cannot be submitted back to the original project (since Microsoft maintains the project and is the only contributor)—is not generally good practice, as it makes your implementation nonstandard, preventing you from easily updating to new releases in your project. In Chapter 12 I'll discuss the MVVM pattern, which provides an alternative solution to global variables by maintaining the state in a `ViewModel` and making it accessible to multiple views.

■ **Note** You should be wary when using this method of passing complex data between views. Users may potentially alter the query string in the browser's address bar (or exclude it altogether). They also may potentially bookmark a view with data in the query string and come back to it later (where the data may no longer be valid). Checks should be made to ensure that expected data is available and valid, and situations in which it's not should be handled.

Deep Links

If your `Frame` control has its `JournalOwnership` property configured accordingly (it is by default—this feature is discussed later in the chapter), you will find that the URI used to navigate to the currently displayed view is appended to the application's URL in the address bar. For example, in the link shown in Figure 4-3, note the hash character (#) indicating the start of the bookmark, followed by `ProductDetails`. The URI mapping scheme will translate this and load the `ProductDetails` view.

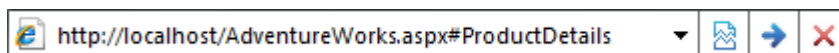


Figure 4-3. A sample URL containing a deep link

This is known as a deep link. Deep linking support is an important feature of the navigation framework, enabling URI-based navigation in your application. While every page in a web site has a URL that enables you to directly navigate and link to it, traditionally the URL for RIA applications could only point to the entry point of the application. The user would then need to navigate to where they wanted to go within the application from the entry point. Now, with deep linking support, you can provide a URI containing the location of a view deep within your application, and have the application automatically navigate there when it is loaded. This deep link can include query string parameters, enabling you to have a link that navigates to a page to display the details of a specific product, for example. Benefits of deep linking include the following:

- The user can bookmark a location within your application and come back to it later without needing to navigate through the application.
- The user can send a link to another user pinpointing a location within the application.
- Deep linking enables the user to go backward and forward (using the browser's Back and Forward buttons) through their navigation history (as will be discussed shortly).

URI Mapping to Enable Friendly URIs

As discussed earlier in this chapter, it's possible to set up a mapping scheme (also known as routes) to automatically transform the deep link that has been navigated to (visible to the user in the browser's address bar) into a form that the navigation framework can act upon (which typically contains the view to navigate to and a query string to pass to it). For example, the mapping could transform the following URI:

`ProductDetails/879`

to its actual representation internally:

`/Views/Inventory/ProductDetailsView.xaml?ProductID=879`

The feature of the navigation framework that makes this automatic transformation possible is the *URI mapper*, which is defined on the Frame control (or associated with it as a resource). When a navigation operation takes place in your application, the Frame control will first parse and transform the given deep link via its URI mapper, and use this output to determine what view to load and the query string to pass to it. You will be familiar with the concept of URI mapping if you've worked on an ASP.NET MVC project before, as it is quite similar to the URL routing functionality available there—although in this case we are only “rewriting” the deep link—not the entire URL, as with ASP.NET MVC URL routing.

■ **Note** It is by no means essential to set up URI mappings in your project, but one advantage of doing so is that it enables you to hide the internal structure of your project (with a side benefit of allowing you to change this internal structure without affecting the links to the views), and can make the URIs less complex and much more readable to end users.

Take a look at the definition of the Frame control in `MainPage.xaml` (as generated by the Silverlight Business Application project template), and note the following `UriMapper` property value definition:

```
<navigation:Frame.UriMapper>
  <uriMapper:UriMapper>
    <uriMapper:UriMapping Uri="" MappedUri="/Views/Home.xaml"/>
    <uriMapper:UriMapping Uri="/{pageName}" MappedUri="/Views/{pageName}.xaml"/>
  </uriMapper:UriMapper>
</navigation:Frame.UriMapper>
```

This mapping is used to transform the given URI to the path of its corresponding view, which the Frame control can then navigate to. Note the use of `{pageName}` in the second route in both the `Uri` and `MappedUri` properties. This is a parameter token that is used to create generic mapping routes. When the format of a URI maps to a route template that contains a token, the corresponding wildcard part of the match will be used to replace the instances of the token in the mapped URI.

To create a token, you simply need to define a name of your own choosing, surround it with curly brackets, and then insert it into the appropriate locations in both the URI route template (i.e., `Uri` property) and the mapped URI (i.e., `MappedUri` property), like so:

```
<uriMapper:UriMapping Uri="/{pageName}" MappedUri="/Views/{pageName}View.xaml"/>
```

So, if a URI with the value `#/About` were passed into the mapper, it would match with the preceding route template and map to `#/Views/AboutView.xaml`. This also comes in handy when mapping query string parameters. For example, in order to map the following URI from

`ProductDetails/879`

to

`Views/Inventory/ProductDetailsView.xaml?ProductID=879`

we could use the following route template:

```
<navigation:UriMapping Uri="ProductDetails/{id}"
  MappedUri="/Views/Inventory/ProductDetailsPage.xaml?ProductID={id}" />
```

So how does the URI mapper determine whether the given URI maps to a given template? It would be simple if it were just checking string equality, but the support for parameter tokens means that it must do pattern matching using regular expressions behind the scenes. The mapper matches the passed-in URI to the constant (i.e., nontokenized) parts of the route template, with wildcards in place of the tokens. When a match is found, the value of the tokens are obtained from the wildcard parts in the match.

It is entirely possible that a given URI matches more than one mapping template defined in the scheme. In this case, the first URI mapper that can successfully map the given URI will win, so you need to take care when ordering your routes. Take, for example, the following generic mapping route:

```
<uriMapper:UriMapping Uri="/{pageName}" MappedUri="/Views/{pageName}View.xaml"/>
```

Perhaps you want the deep link to read `#/AboutUs`, but the actual XAML view this corresponds to is in the Views folder and named `AboutView.xaml`. If this deep link were transformed by the preceding route, then it would attempt to look for a view called `AboutUsView.xaml` (which doesn't exist). In this case,

to retain the preceding route but still handle this scenario, you would have to ensure that you place the route defining this specific case *above* the generic mapping, like so:

```
<uriMapper:UriMapping Uri="/AboutUs" MappedUri="/Views/AboutView.xaml"/>  
<uriMapper:UriMapping Uri="/{pageName}" MappedUri="/Views/{pageName}View.xaml"/>
```

In summary, you should ensure that the “concrete” route mappings are placed above the generic (tokenized) mappings. Therefore, the order in which you define the route mappings is important; they should be ordered from least generic to most generic. Using the preceding mapping scheme, when the #AboutUs deep link is navigated to, it will be matched to the first route and immediately directed to the AboutView.xaml file as required.

■ **Note** The project won’t compile if you try to define a mapping that starts with a tokenized parameter (e.g., `Uri="{pageName}"`). If you start with an open curly bracket, the XAML parser will think you are specifying a markup extension as the value. As described in Chapter 3 (in the “Markup Extensions” section), in order to have this value treated as a literal, you’ll need to add open and closed curly brackets with nothing between them before your token, like so: `Uri="{{pageName}}"`—this will solve that problem.

When an URI is passed in that cannot find a route to map to, then the mapping process will fail (as you’d expect). When this occurs, the `NavigationFailed` event will be raised on the Frame control, which you can handle accordingly. The default project template will display an error window when this occurs, but you can change this, for example, to navigate to a different view instead.

You may not want your URI mappings defined directly on the Frame control (as they are in the default project template output) or even in the XAML file hosting the Frame control. If so, you can define your mapping scheme as a XAML resource in the `App.xaml` file or a standalone resource dictionary instead, and reference it as you would any other resource—using the `StaticResource` markup extension:

```
UriMapper="{StaticResource uriMapperKey}"
```

■ **Note** If you find that you need complex or custom mapping logic that can’t be implemented easily using the standard `UriMapper` URI mapping class, then you can write your own mapping class. Create a new class, inherit from `UriMapperBase`, and override the `MapUri` method (which will handle the mapping logic). Then, after you’ve written your logic, you can reference this class as your URI mapper rather than the standard one.

Integrating with the Browser History

You may notice the following when you navigate between views in your application (in addition to the view changing):

- The URI in the address bar changes.
- The status of the browser history buttons is updated.
- The browser's title bar is changed to display the title of the view.

These occur because the navigation framework integrates with the browser's journal (which maintains the browser's history). When each view is navigated to in your application, it will be added to the browser's journal, which enables you to navigate through this history using the browser's Back and Forward buttons as if the views were standard HTML web pages.

How the views that have been navigated to are maintained in the browser's history is dependent on the browser. For example, Internet Explorer's history only displays a single entry for the web page hosting the Silverlight application, while in Firefox each view within the Silverlight application is added to the history when it's navigated to.

■ **Note** In Chapter 14 I'll discuss having the application run outside the browser. In that situation, the browser Back/Forward buttons are not available, nor is the address bar (since it's not running within a browser). However, a navigation journal is still maintained within the application, and if it's important for the user to have the ability to navigate backward and forward through their navigation history, you can detect if the application is running outside of the browser (checking the value of the `App.Current.IsRunningOutOfBrowser` property) and display your own Back/Next buttons to provide this functionality (via the `GoBack` and `GoForward` methods). Note that you will want to make use of the `CanGoBack` and `CanGoForward` properties to check whether each operation is valid before attempting it, as an exception will be raised if the operation cannot be performed.

While generally you would only have one `Frame` control in a XAML file, you can actually host more than one `Frame` if you wish, and views can also host other subviews in a `Frame`. However, only the top-level `Frame` can integrate with the browser's journal.

You can use the `JournalOwnership` property on the `Frame` control to control whether or not the `Frame` control integrates with the browser's journal. Valid values for this property include

- `Automatic`: This will choose the correct option based upon whether a higher-level frame is already using the browser's journal.
- `UsesParentJournal`: This will use the browser's journal, as previously described.
- `OwnsJournal`: With this option, the `Frame` can still navigate between pages, but moving backward and forward through the history must be done via code. Frames not using the browser's journal will not respond to deep links entered into the browser's address bar, and the URI displayed in it will not change as pages are navigated between in that `Frame`. If the `Frame` needs to track the history of loaded views, it will need to track them itself.

■ **Note** You might want to have multiple Frame controls in your shell if it contains multiple content sections where the user might be navigating between views while drilling down on data, such as in a dashboard.

Handling Navigation Events

When navigation operations take place, you need to know what's happening so you can respond accordingly. The Frame control raises events that you can create event handlers for, while within a view you can handle the events on the `NavigationService` object or simply override the navigation methods on the view itself (which inherits these virtual methods from the `Page` class).

Frame Events

The Frame control provides the following events that enable you to respond to navigation operations and their current status:

- Navigating
- Navigated
- NavigationStopped
- NavigationFailed
- FragmentNavigation

The `Navigating` event is raised before a navigation operation actually takes place, while the `Navigated` event is raised after the operation is complete. If you want to stop the navigation operation from actually taking place, you can set the value of `e.Cancel` to true in the `Navigating` event handler.

The `NavigationStopped` event is raised when the `StopLoading` method on the Frame control or the `NavigationService` object is called. The `NavigationFailed` event is raised if an invalid deep link was included as a part of the URI that couldn't be mapped to a view by the URI mapper.

The `FragmentNavigation` event is raised instead of the `Navigating`/`Navigated` events in a couple of different scenarios. If you start a navigation operation for the view that is currently active in the Frame (and are passing in the same parameters that were passed to the view previously), then the `FragmentNavigation` event is raised (the view is not reloaded). An example of this is clicking on the Home button to navigate to the Home view while you are already there. This might be useful when you need to know that a navigation operation has taken place but the view was not reloaded. This event would also be raised if you were to start a navigation operation and have appended a fragment to the end of the URI to navigate to. For example, you might try this navigation operation:

```
ContentFrame.Navigate(new Uri("ProductDetails/879#StockLevels", UriKind.Relative));
```

Note the # used to indicate the start of the fragment. This might be initially confusing, as the start of the deep link in the resulting URI (which points to the view) is indicated by a # already. However, you will find that the resulting URI is as follows:

```
ProductDetails/879$StockLevels
```

As you can see, the # is converted to a \$, which now indicates the start of the fragment. The purpose for which you include a fragment in a URI depends on your needs, but you can access its value easily via the `e.Fragment` property, which is passed in as a parameter to the `FragmentNavigation` event handler.

You might want to hook into the navigation events on the `Frame` control when there is a toolbar on the `Frame` host page that needs to know when the active view has been changed in the `Frame` so that it can highlight the appropriate toolbar button—you would handle the `Navigated` event for this purpose. This is a scenario that you will find in the default project template, where the `MainPage` class handles the `Navigated` event and changes the styles on the buttons according to whether its corresponding view is currently open. Note also that it handles the `NavigationFailed` event and displays an error message window if that event is raised.

View Events

There are two methods of responding to a navigation operation within a view. The first is to add an event handler on the `NavigationService` object (obtained by the `NavigationService` property on the view), like so:

```
NavigationService.Navigating += NavigationService Navigating;
```

The same events that were available on the `Frame` control are available on the `NavigationService` object.

■ **Note** You cannot add an event handler for the `NavigationService` methods in the constructor, as the `NavigationService` object will be null at that point (and will result in a `NullReferenceException` being thrown). You should add the event handler in the `Loaded` event handler's method instead.

Alternatively, you can override any of the following virtual methods on the view object that you want to respond to:

- `OnNavigatedTo`
- `OnNavigatingFrom`
- `OnNavigatedFrom`
- `OnFragmentNavigation`

Note that there are no overridable methods corresponding to the `NavigatingTo`, `NavigationStopped`, or `NavigationFailed` events on the `NavigationService` object, but there are two new “events” that we can handle: `OnNavigatingFrom` and `OnNavigatedFrom`. One use for overriding the `OnNavigatingFrom` method is to check whether the current data form is dirty (i.e., has unsaved changes) and ask the user whether the changes should be saved before moving away from the view. The navigation operation can be cancelled from within this method by setting the value of `e.Cancel` to true.

■ **Note** The `NavigatedTo` event is raised *before* the `Loaded` event on the view.

Caching Views

When you navigate away from a view, by default all references to the view are removed, and the view will be cleaned up the next time the garbage collector runs. However, this is not ideal if you need to retain the state of the view, such as if the user has entered unsaved data and is navigating away temporarily, or the view has been populated with data that would need to be retrieved again and repopulated.

You can, however, change this default behavior in order to cache the view and maintain its state by setting the `NavigationCacheMode` property on the view appropriately (a property of the `Page` class that you can set in XAML). There are three options: `Disabled`, `Enabled`, and `Required` (with the default value being `Disabled`):

- **Disabled:** This will destroy the view once it has been navigated away from and recreate it when it's navigated back to.
- **Enabled:** This means that the view will be added to a cache (maintained by the `Frame` that hosts it) when it is navigated away from. When the user navigates to the view again, this previous instance will be used instead of creating a new instance.
- **Required:** Because caching views can be quite memory intensive, the cache stores only a given number of views (configurable by the `CacheSize` property on the `Frame` control), after which it removes the oldest entries to be cleaned up by the garbage collector in order to conserve memory. However, this is not ideal behavior in all scenarios, as you may want to have more control over whether the view can be removed from the cache. To stop a view from automatically being removed from the cache, set the `NavigationCacheMode` property of the view to `Required` (instead of `Enabled`). This will result in it remaining in the cache indefinitely, regardless of the size of the cache.

If you want to retain a view so that data doesn't have to be loaded repeatedly, a value of `Enabled` would be quite satisfactory. However, if you are retaining unsaved data that you don't want to lose, `Required` would be a better option in this scenario.

To force a view to be removed from the cache once it is no longer required to be cached, simply set its `NavigationCacheMode` property to `Disabled`. When it's navigated away from, it will be removed from the cache. The next time it is navigated to, a new instance of it will be created, and it will go back to using its default value for `NavigationCacheMode`.

■ **Note** When the view has been retrieved from the cache, rather surprisingly both the `Load` and `NavigatedTo` events for the view are still raised when it is navigated to, and there is no property that indicates whether it was retrieved from the cache or that it had previously been loaded. If you have code to populate the view in these events, then it would defeat the purpose of caching the view if you were to just populate the view again. The best method to work around this omission is to have a class-level private Boolean variable called `isContentLoaded` with a default value of `false` that you can set to `true` once the view has been populated. You can then check this variable the next time one of these events is raised.

Visual Transition Effects

Using a visual transition effect when navigating from one view to another can provide a nice touch to your application, and can help add some wow factor that will impress your client and the users (a step toward securing their acceptance of the new system). By default, Silverlight's navigation framework doesn't incorporate any transitions when navigating between views—it simply removes the current view from the Frame and displays the new one in its place. However, a nice alternative might be to slide the current view out to the left and bring the new view to be displayed in from the right, or fade the current view out as the new one fades in. The options are boundless (limited only by your skills in designing animations in XAML, really). The focus of this book isn't on animations in XAML, so we won't be focusing too much on the process of creating transition effects, but we'll take a quick overview of how view transitions can be implemented within the navigation framework. When designing animations, however, consider Expression Blend your best friend—it will make the process of designing the transition much easier.

■ **Note** Just a word of warning—in small quantities animations can improve the look of your application, but they can just as easily detract from your application and make it look amateurish. It's a fine line, and it's better to underuse animations than overuse them. Animations should be simple and fast in order to effectively add a professional feel to your application, and they should be used sparsely and not attempt to attract the user's attention. In this situation, less is more. What can amuse the first time can soon get old, especially if it slows the user down from the task at hand.

The easiest way to implement transitions is to make use of the `TransitioningContentControl`, which is a part of the Silverlight Toolkit. You can include this (along with an animation for the transition) as a part of the control template for the Frame. Here are the steps to do this:

1. Open the `Styles.xaml` file (in the Assets folder) in the default project and look for the `ContentFrameStyle` style that's defined there. The Frame control definition in `MainPage.xaml` already references this as containing its styling information, although it doesn't define a custom template for the control.
2. Create a copy of the default control template of the Frame control into this style definition (see Chapter 9 for details on how to do this).
3. Change the `ContentPresenter` reference in the control template to `TransitioningContentControl` (remembering to prefix it with a namespace reference) and create this reference in the namespace reference definitions at the top of the XAML file that points to the Silverlight Toolkit layout controls assembly. Keep the other properties as they were defined for the `ContentPresenter` on the `TransitioningContentControl`.

4. As it is now, the transitioning will work without any further changes, using a default transition of fading the old view out and the new one in. To use your own transition, you will need to set the `Style` property for the `TransitioningContentControl` to use a new style that you will create for it. This style will contain a control template for the `TransitioningContentControl` that defines the animation to use when navigating from one view to another.

■ **Note** Some example transition animations can be found in the sample project for this book.

Alternative User Interface Frameworks

The navigation framework isn't the only user interface framework available for Silverlight (although it is the only one with support out of the box). Other alternatives that you may wish to consider include the following:

- *Prism*: An open source framework for creating modular applications, and corresponding guidance dictating patterns and practices for doing so. You can download it from <http://compositewpf.codeplex.com>.
- *nRoute Framework*: An open source framework created and maintained by Rishi Oberoi that describes itself as a “composite application framework for creating MVVM-style applications in Silverlight, WPF, and Windows Phone 7.” You can download it from <http://nroute.codeplex.com>.
- *Silverlight.FX*: An open source framework created and maintained by Nikhil Kothari that describes itself as a “light-weight application framework for building Rich Internet Applications with Silverlight.” You can download it from <http://projects.nikhilk.net/SilverlightFX>.
- *Caliburn*: An open source framework created and maintained by Rob Eisenberg that describes itself as a “client framework for WPF and Silverlight.” You can download it from <http://caliburn.codeplex.com>.
- *ClientUI Framework*: A commercial framework from Intersoft Solutions that describes itself as a “next-generation user interface library for the client platform.” You can find out more here: www.clientui.com.

Summary

In this chapter you learned how to use the navigation framework as the structural foundation of your application's user interface. As the Silverlight Business Application project template uses this by default, as soon as you've created the project you will have something that you can compile and run, and can use as a solid basis from which to build your application.

CHAPTER 5



Exposing Data from the Server: Using WCF RIA Services

By their very nature, business applications are highly data-centric applications. Therefore, one of the primary concerns you will have when designing your application is to how it will consume data from a server.

If you're used to writing applications that communicate directly with a database (such as SQL Server), you may be surprised to find that there is no object in Silverlight to enable you to do so (`SqlConnection`, `OdbcConnection`, etc.). This is because Silverlight is a client platform, designed to be run from within a browser anywhere in the world, and thus it does not make sense for it to be able to access databases directly (as databases are generally hidden behind a firewall). Instead, data must be explicitly exposed externally from the server, using something such as a web service to sit on this boundary and provide a conduit between the data in the database and external applications.

There are numerous means to expose data from a server and consume it in a client application (many of which Silverlight supports); therefore, it can often be difficult to determine the best and most appropriate technology to use. Windows Communication Foundation (WCF) is generally the most popular technology for creating services supported by the .NET Framework.

Although you can consume WCF services in Silverlight (with a few limitations detailed later in this chapter), you will probably find that creating and maintaining your services becomes somewhat of a chore—it will require a lot of repetitive boilerplate code, it will become more complex when you add support for filtering/sorting/grouping/paging, it will require you to constantly update your service references, and it will force you to write/maintain logic (such as validation and business rules) on both the server and the client.

To try to combat these problems and simplify access to data in Silverlight applications, Microsoft has introduced a new technology called WCF RIA Services (commonly referred to as RIA Services), specifically designed for use by Silverlight applications (but with a view to support additional presentation technologies in the future).

RIA Services is a layer on top of WCF that provides a prescriptive pattern and framework that you can use when building data-driven applications that consume data from a server. Another way of saying that is that RIA Services simplifies the process of exposing data from a server, consuming it in your Silverlight application, and sharing business logic between the two—which it does by providing a set of patterns to follow in creating your services, and providing a framework to support them.

In this chapter, we'll be focusing on how to use RIA Services to expose data from the server, and write logic that can be shared between the server and the client (consuming these from your Silverlight application will be covered in the following chapters). I'll also discuss some of the alternative means and technologies for doing so if you find that RIA Services does not suit your needs.

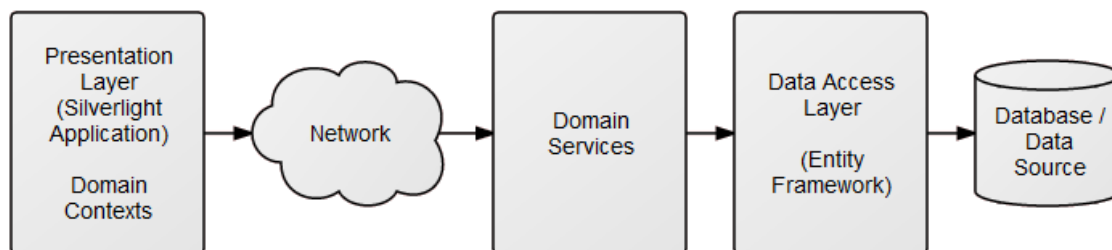
What Is WCF RIA Services?

In a typical server/client scenario, you need to pass data back and forth between the server and the client, with data being passed back and forth over a boundary (such as the Internet, a network, or even applications on the same machine). For example, the client requests data from the server (which queries a database for the requested data), possibly makes changes to it, and then passes the changes back to the server (which updates the database accordingly).

To accommodate this scenario, a typical system consists of a number of layers/tiers that provide a clean separation of concerns between each logical module. Often you will find three tiers in an n-tier design; this design will separate the presentation tier from the middle tier, and the middle tier from the data tier (with the presentation and data tiers having no direct connection with each other).

The problems that this structure creates, however, is that it requires an often complex underlying framework to effectively transfer data between the tiers. Also, a duplication of the same code is required on multiple tiers (such as data classes, and validation/business rules/logic). Duplication of code requires additional effort to write, test, and maintain, and code can easily get out of sync between the tiers, potentially leading to problems that are difficult to track down. RIA Services has been designed to solve these problems, and simplify the scenarios of architecting and developing an end-to-end Silverlight application. Therefore, developers can focus on solving the actual business problem rather than focusing on architectural concerns.

RIA Services is partly a data-centric design pattern for structuring your application; partly a framework that provides advanced data management, authentication/authorization management, and querying features; and partly a code generator that generates code in one project based on the code written in another project (so common code to be used in multiple tiers can be written in one place). It piggybacks on top of WCF Services, which provides the communication mechanism between the middle tier and the presentation tier.



The focus of the RIA Services pattern is very much directed toward the middle tier. It's not tied to a specific data access technology, although it works particularly well with both the Entity Framework and LINQ to SQL (the latter requiring the WCF RIA Services Toolkit to be installed, as discussed later in the chapter). It is also not designed to be tied to a specific presentation technology, although the focus is currently primarily on Silverlight, with limited support for other presentation technologies such as ASP.NET, WPF, and Windows Forms.

RIA Services is an end-to-end technology; therefore, you need to have control over both the server and the client aspects of the application. If you need to access existing services (without routing the requests through the server), or to use services being developed by a separate team, RIA Services may not be suitable.

Although this book focuses on the use of RIA Services to expose data and logic on the server, Silverlight is by no means limited to this technology. Some of these other technologies that you can use for client/server communication will be discussed at the end of this chapter, and most of the concepts

discussed in this book will still apply to your business application even if you choose not to use RIA Services.

The framework portion of RIA Services provides a host of functionality out of the box that vastly simplifies communicating between the presentation tier and the middle tier. Included is a data source control that can be used in XAML to communicate with a service, management of changes made to data on the client and processing these changes on the server, the ability to write a LINQ query on data exposed by a service on the client and have that query executed on the server, and an out-of-the-box working authentication service.

How the WCF RIA Services Code Generator Works

RIA Services requires both the server and the client projects to be within the same solution and linked together. One contains services that expose data (acting as a server), and the other communicates with the services and consumes the data (acting as a client).

When a service is written in the server project that follows the RIA Services design pattern, the RIA Services build task in Visual Studio generates corresponding code in the client project (i.e., a code generation process) that simplifies the process of communicating with the service. It also generates corresponding proxy data object classes (which are generally referred to as “entities”) in the client project for those that are exposed by the service, applies attributes (such as those that denote validation logic) on the entities and their properties, and copies code marked as shared code from the server project into the client. This code generation is generally referred to as “projection,” because code is projected from the server project onto the client project.

Let’s investigate what is happening under the covers when you link the two projects via RIA Services. For this, we’ll use the solution that we created back in Chapter 2. This solution was created using the Silverlight Business Application project template, and is already configured to use RIA Services, with the two projects in the solution already linked with an RIA Services link.

Start by toggling the Show All Files option (the second button from the left in the Solution Explorer window’s toolbar) to “on” for the Silverlight project. There is a Generated Code folder (as shown in Figure 5-1) into which code generated by the RIA Services build task is placed, as code is written in the web project on the fly and when the Silverlight project is compiled. When you make a change on the server (such as adding a property to a class), the change will be almost immediately propagated to the Silverlight project so you can use it (if not, try compiling your project to force the code generation). The code that is generated enables you to access all the operations and data from the client that has been exposed on the server via RIA Services.

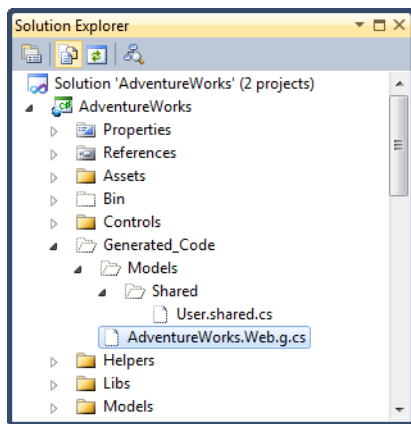


Figure 5-1. The Silverlight project structure, showing the hidden `Generated_Code` folder

■ **Note** You can have a look at the code that has been generated for the client, but don't change anything because any changes you make will be overwritten when the Silverlight project is next compiled (which regenerates the code in this folder). The generated classes are partial classes, which enables you to extend them on the client with client-only logic if you find the need. We will take a closer look at this generated code later in the chapter, in the "Inspecting the Generated Code in the Silverlight Project" section.

The code generator is intelligent enough to ascertain the nature of operations on the services. The logic it uses is discussed in the "Domain Operations" section later in this chapter.

You can govern how entity classes are generated in the client project (what properties are included or excluded) by decorating the entity's properties with attributes (also known as data annotations). How you do this is discussed in the "Decorating Entities" section later in this chapter.

Code that should be shared between the server and the client (copied from the server project into the client project by the RIA Services build task) must be placed into files that have the `.shared.cs` extension, so that the build task knows to copy these files from the server project to the client project. Sharing code is discussed in the "Sharing Code Across Tiers" section later in this chapter.

How Do You Use WCF RIA Services?

This section goes through a brief overview of the process of working with RIA Services before drilling down into the details.

Linking Your Silverlight and Web Projects

Using RIA Services requires you to follow a pattern in structuring your application. First, your server (web) project and your client (Silverlight) project must be linked so that the RIA Services build task/code generator can project code from the server project onto the client project and create the supporting objects used to interact with the server. This means that both your client and server projects must exist in the same solution.

■ **Note** There is support to have your entities and metadata classes in a separate project from your web project (using the WCF RIA Services Class Library project template discussed later in this chapter).

Creating Your Domain Services

The next step is to create one or more *domain services*. Think of a domain service as much like a standard WCF service, but following a given pattern (and consisting of some base functionality).

■ **Note** In fact, each of your domain services is actually a WCF service. RIA Services generates code in the client project to automatically communicate with the domain service, meaning that you don't have to worry about adding service references to your domain services. However, because domain services are standard WCF services, you can add them as service references to your projects if you wish (including Windows Forms, WPF, and ASP.NET projects).

The RIA Services build task automatically generates code in the client project that enables it to interact with the domain service on the server. It creates a *domain context* for each domain service, and proxy classes for each of the entities exposed by the domain service.

Creating Domain Operations on Your Domain Services

On your domain service you create *domain operations*, which will typically expose data to the client and accept changes in a create, read, update, and delete (CRUD) pattern.

The entities in the server project that are exposed by the domain operations can be decorated with attributes (such as specifying data validation rules), either directly or by applying the attributes to associated classes (known as metadata classes) that enable you to apply attributes to the entities without modifying them directly (important when the entities are code generated, such as Entity Framework entities that can't or shouldn't be modified directly). You can share code between the server and the client projects by simply placing the code to be shared in one or more files with the `.shared.cs` extension.

Consuming the Domain Services

Once this is set up on the server and you are exposing data from your domain services, you can turn your attention to the client project in order to call the operations on the domain service and consume the data. You can do this in code by creating an instance of a domain context and calling the exposed operations. Alternatively, you can make use of the `DomainDataSource` control to interact with a domain service declaratively in XAML.

Entities are generally exposed from a domain operation via an `IQueryable<T>` expression or as an `IEnumerable` collection. By returning an `IQueryable<T>` expression, the power of the RIA Services framework is demonstrated in that you are able to write a LINQ query on the client specifying how you want the resulting entity collection filtered, sorted, grouped, and paged. RIA Services serializes the LINQ query and sends it to the server, where it is appended to the query expression returned from the query operation before it is executed. This enables the collection to be filtered/sorted/grouped/paged on the server without requiring the entire collection to be sent back to the client first (minimizing the required data traffic).

As this collection is updated on the client, the domain context that exposes it maintains a *changeset*, which includes the entities that have been inserted into, deleted from, or updated in the collection. *Changeset* management is another extremely powerful feature of the RIA Services framework that greatly simplifies managing the collection and sending updates back to the server. The `SubmitChanges` method on the domain context will send the changes made to an entity collection back to the server (which based on the changes made will automatically call the insert/update/delete operations to update the back-end database accordingly, plus any custom operations that were called) as appropriate.

This discussion should have given you a broad overview of the process of working with RIA Services and introduced you to some of the key concepts. The next section looks at how to actually put these concepts into practice.

Getting Started

The project we created in Chapter 2 using the Silverlight Business Application project template is set up to use RIA Services by default, and already provides some basic functionality using RIA Services. Therefore, we will continue working from that base and the minor modifications we made in Chapter 2. Figure 5-2 displays the structure of this solution again.

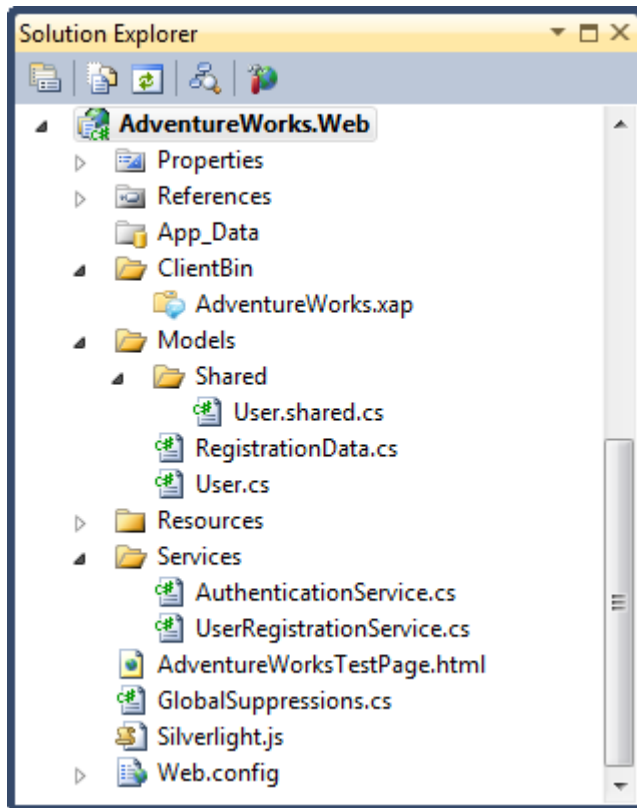


Figure 5-2. The structure of the Web Application project

Note from Figure 5-1 that there is a `Services` folder and a `Models` folder in the project. The `Services` folder already contains two domain services (`AuthenticationService` and `UserRegistrationService`) for providing authentication and user registration operations to the client (these will be discussed further in Chapter 8). The `Models` folder contains two data classes (`User` and `RegistrationData`) that are passed between the server and the client. You will also find a `Shared` folder under the `Models` folder, which has a file called `User.shared.cs` that contains code to be shared between the server and the client projects.

As demonstrated in Chapter 2, the Silverlight and web projects are linked together. The purpose of this link is to copy the compiled XAP file generated by your Silverlight application to a location under the web project when the project/solution is compiled. However, RIA Services requires another link to be created between the projects. The purpose of this link is to configure what project contains the services that the Silverlight application will be consuming data from. This will enable the RIA Services build task to generate the required code to communicate with those services in your Silverlight project accordingly.

Whereas the project link to copy the Silverlight application's XAP file into the web project was configured in the web project's project properties, the RIA Services link is configured in the project properties of the Silverlight project, as shown in Figure 5-3.

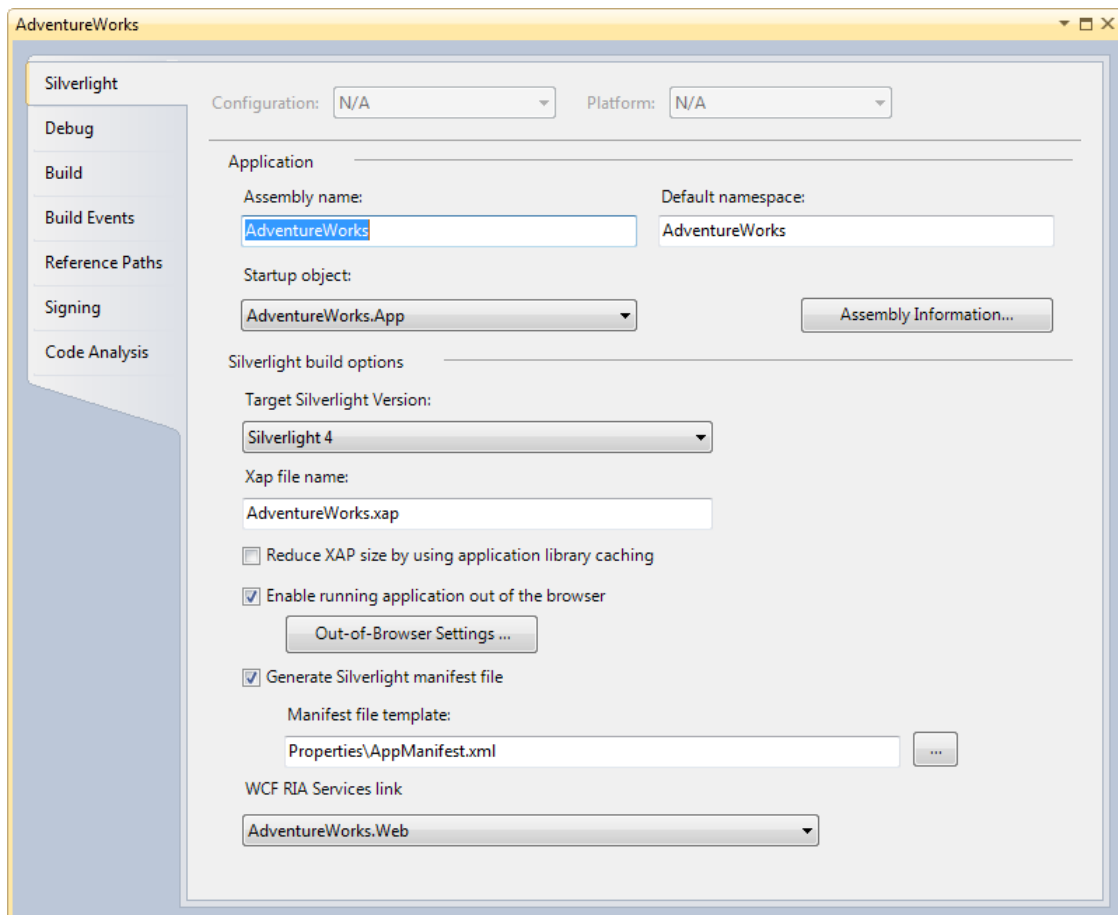


Figure 5-3. The project properties for the Silverlight project, with the WCF RIA Services link configuration at the bottom

This link will already be set up by using the Business Application project template; however, if you have an existing Silverlight project or web application that you want to use RIA Services with, you can manually link the projects together by linking the Silverlight project to an ASP.NET project with this option.

Now you are ready to start writing some code. The example you'll work through in this chapter demonstrates some of the key concepts of the RIA Services pattern. Your aim will be to expose data and operations from the server, and make them accessible from the client. In doing so you will also be sharing some validation logic and code between the server and the client, which will be executed on both sides. First, however, you need to set up a data source/layer from which you will expose data to the client.

Creating a Data Access Layer Using the Entity Framework

As this book is focused on Silverlight as a technology, I won't be going into too much depth on database-related concepts such as database terminology and principles, creating and managing databases in SQL Server Management Studio, or methods of accessing data (such as ORMs, the Entity Framework, LINQ to SQL, ADO.NET, etc.). These are considered prerequisite knowledge for this book, and instead I'll focus here on how to actually expose the data publicly via a service, and how to consume it in a Silverlight application.

■ **Note** If you're not familiar with the Entity Framework (which we'll be using throughout this book), you can find more information and videos on using it here: <http://msdn.microsoft.com/en-us/data/videos.aspx>.

Configuring the Database

Throughout this book you'll be creating a business application in Silverlight for a fictional bicycle sales chain called Adventure Works Cycles, which will enable employees to manage the various aspects of the business, including product inventory, sales, and purchasing. For this we'll be using SQL Server 2008 as our database server (the free Express version installed with Visual Studio will be sufficient), and the database that will be supporting this scenario will be the AdventureWorks2008 OLTP sample SQL Server database from Microsoft, which you can download from <http://msftdbprodsamples.codeplex.com>. This database has been chosen because it is an example of a typical business database with a suitably complex structure, and is also well populated with sample data.

■ **Note** The AdventureWorks2008 database comes with an installer that includes a number of other databases. The only database that you need to install is the AdventureWorks2008 OLTP database. Ensure that the check box next to this database is selected in the installer so that it will actually be created as a database in the selected SQL Server instance (otherwise the database installer script will be installed but not executed).

About the Entity Framework

In order to retrieve and persist data to the database, we'll be using the Entity Framework as our data access layer. The Entity Framework is an object-relational mapper (ORM) that enables you to create a conceptual object model (known as an entity model) and map it to the database schema (e.g., tables, views, and stored procedures). You can then query this object model in code using *LINQ to Entities* (LINQ specifically designed for querying an entity model).

The Entity Framework takes care of translating the LINQ to Entities queries to SQL queries in the database, and returns the data as the objects that you've defined in your model. Essentially, the Entity Framework makes working with database data in code a much more pleasant and robust experience, as it enables you to work with the data as objects and collections (instead of rows and tables), and to write strongly typed queries against the entity model.

The Entity Framework has been subjected to some controversy since its initial release, including a vote of no confidence petition signed by many prominent members in the developer community (including a number of Microsoft MVPs). It has undergone numerous improvements since then (with the .NET Framework version 4) to counter the issues that were raised with the first version, and is working its way toward becoming a robust yet easy-to-use ORM.

One of the benefits of using the Entity Framework for this project is that RIA Services has been designed to work very smoothly in conjunction with it, enabling services with the standard CRUD operations to be created very easily for given entities.

However, the Entity Framework is by no means the only data access technology supported by RIA Services—it also has support for LINQ to SQL models (which can be found in the WCF RIA Services Toolkit, discussed later in the chapter), and in fact it can work with any data source (although doing so involves a degree of additional work when creating a domain service). So if you have an existing data access layer to access your database, or you want to use a different ORM (such as nHibernate), RIA Services can also support these scenarios.

In fact, employing the built-in support for exposing entities directly from your Entity Framework model could in some ways be considered bad practice. While taking advantage of this support to expose given entities to the client makes creating and maintaining services incredibly easy, the entities are essentially a model of your data layer, and ideally these entities should not be exposed to the presentation layer. Whether or not you pass entities or plain-old CLR objects (referred to as POCOs, but also known as presentation model types in RIA Services) of your own design back and forth is a decision you will have to make, dependent on many factors. Using entities will make development much faster, but will also be less flexible than using custom presentation model types. RIA Services works just as well using presentation model types as it does with entities, despite more work being involved in initially creating the domain services. Therefore, the best practice would be to use presentation model types as the data transfer mechanism (which can be populated with data from your entity model), although we will take a look at both of these methods (starting with directly exposing entities because they provide the easiest and fastest means to get started).

Creating an Entity Model

Once you've installed the database you can create your entity model. This is a very straightforward process, as the Entity Data Model Wizard will guide you through this process.

Add a new item to your Web Application project, select the ADO.NET Entity Data Model item template, and name it `AdventureWorksModel.edmx`. Following through the wizard steps, you want to create a model from a database, create a connection to the database, and select the tables in the database to be included in the model. This chapter will demonstrate exposing product data, so select the Product table to be added to your entity model. You can add other tables from your database to your entity model at a later time if you wish.

Ensure that the Include Foreign Key Columns in the Model check box is selected, as this will greatly simplify dealing with foreign keys in scenarios where an entity is being updated and combo boxes are enlisted to select an item from a dictionary and assigned to a property on the entity.

When you click the Finish button, the entity model will be generated and displayed in the entity model designer.

Domain Services

Once you have connected your Silverlight and web projects via RIA Services and configured a data source on the server, the next task is to expose some data and operations from the server (which you will

consume from your client in the following chapters). This is achieved through the use of a domain service. A domain service typically consists of the CRUD operations and any other custom operations you wish to expose for a given entity that will be available to the client.

Understanding the Domain Service Life Cycle

Let's take a quick look at how a domain service works and the various phases in its life cycle. The core interaction that a client has with a domain service is to retrieve a collection of objects (or a single object) from the domain service, and return any of the objects that have been modified, or additional objects that have been inserted into the collection back to the server to update the database accordingly (i.e., the create/update/delete in CRUD). The objects in the collection that have been changed (inserted/updated/deleted) are tracked on the client and returned back to the domain service, together with the domain operation that should be applied to each object on the server as a *changeset* (when the `SubmitChanges` method is called on the domain context associated with the domain service that was created in the client project).

This calls the `Submit` method on the domain service, initiating the start of the domain service's life cycle. The domain service will work through the following phases in order:

- The *authorize changeset* phase essentially verifies that the user passing through the changeset is actually authorized to execute the operations in that changeset. It checks that each operation to be called from that changeset is permitted according to the security-related rules defined on each of these operations in the domain service (by decorating the operations with security rule attributes). Configuring security rules on domain operations is covered in Chapter 8.
- The *validate changeset* phase performs server-side validation of the entities being passed through in the changeset according to the validation rules applied to each object and its properties (by decorating them with validation rule attributes) within the changeset. Although the objects and their properties are validated on the client side, the domain service validates them again according to the validation rules to verify that someone didn't circumvent the validation rules on the client (as is proper practice). Configuring validation rules on objects and their properties is covered shortly in this chapter.
- Once the changeset has been fully authorized and validated, the *execute changeset* phase can begin. This phase enumerates through each entry in the changeset and calls the specified domain operation (i.e., the insert, update, delete, or custom operation on the domain service), passing in the given object to perform that operation upon.
- The final phase in the domain service's life cycle is the *persist changeset* phase. When you are using a data access layer that maintains its own changesets (such as the Entity Framework or LINQ to SQL), the final task is to persist that changeset to the database. For example, when using an Entity Framework model as your data access layer, this phase would be used to call the `SubmitChanges` method on the data context (and complete the transaction). If changes are immediately committed to the database in the domain operations, then this phase will usually be ignored.

Each of these phases is automatically performed by the domain service; however, there is a corresponding method on the domain service that you can override and hook into if you need to provide

some custom behavior in addition to what is provided by the base domain service class. These methods are

- `AuthorizeChangeSet`
- `ValidateChangeSet`
- `ExecuteChangeSet`
- `PersistChangeSet`

You can also override the `Submit` method, which encompasses each of these phases in order to hook into that, too. While in a phase, the logic for that phase will be run over every object in the changeset. If an exception is thrown on any of these objects (during any of these phases), then the entire changeset submission will be cancelled.

Creating a Domain Service

Let's work through the process of exposing a `Product` entity from the entity model. The first step is to add a domain service to your web project.

■ **Note** After creating an entity model that you want to expose data from (and before attempting to create a domain service), make sure that you compile your solution. If you don't, you will find that the Domain Service Class Wizard won't display your entity model in the available `DataContexts/ObjectContexts` drop-down list.

Add a new item to the `Services` folder in your web project, using the Domain Service Class (under the `Web` category) as the item template. We will be using this service to serve up product data, so we'll call it `ProductService.cs`. This will initiate the Domain Service Class Wizard, as shown in Figure 5-4.

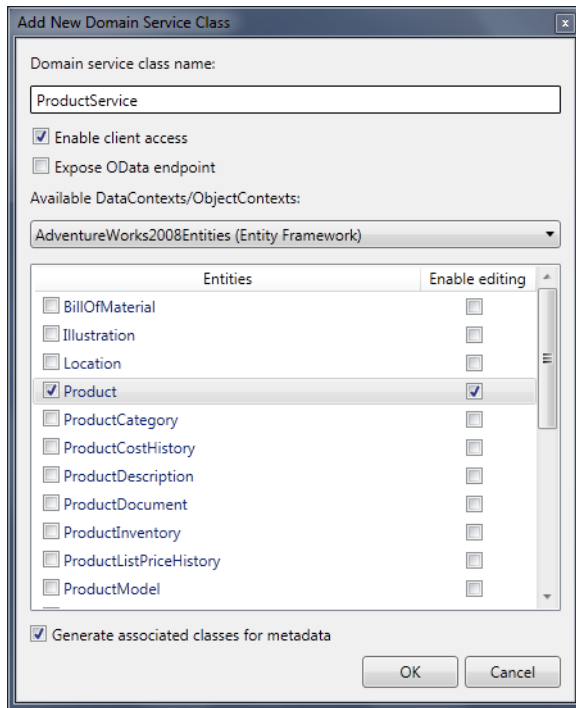


Figure 5-4. The Domain Service Class Wizard dialog

Select your AdventureWorks2008 entity model from the Available DataContexts/ObjectContexts drop-down list (if it isn't already selected). This will display each entity in your entity model in the list.

■ **Note** If you're using POCO types instead of Entity Framework or LINQ to SQL types, you will need to select the <empty domain service class> option from the Available DataContexts/ObjectContexts drop-down list, and implement the domain operations yourself. In addition, if you want to use a LINQ to SQL model as your data access layer, you will need the WCF RIA Services Toolkit installed, which will be discussed later in this chapter.

Let's now take a look at each element of this dialog and what it does.

Entity Selection

The entity list enables you to select the entities that you want to expose via this domain service (using the check boxes). A domain operation will be created in the domain service for each selected entity, and this operation will return a collection of the corresponding entity to the client. If you want to also be able

to modify an entity from the client (i.e., add, update, and delete entities), select the Enable Editing option next to it. For each entity marked as editable, the wizard will create insert, update, and delete domain operations for that entity on the domain service.

Enabling Client Access

Make sure that the Enable Client Access check box is checked before moving on from this dialog. This will decorate the domain service class with the `EnableClientAccess` attribute, which will indicate to the RIA Services code generator that it should generate code in the Silverlight project to enable it to access the domain service on the server.

Exposing an OData Endpoint

The Expose OData Endpoint check box enables your domain service to expose data using the Open Data (OData) protocol. OData is a set of extensions for the ATOM protocol to standardize the creation of HTTP-based data services, somewhat similar to REST data services—returning the data as an ATOM feed and providing CRUD operations on the data. While this is a new protocol, it has Microsoft backing, and Microsoft is integrating support for OData into a number of its products (such as SharePoint and PowerPivot). Selecting this check box will enable your domain service to be consumed by products supporting this OData protocol.

Generating Associated Classes for Metadata

The Generate Associated Classes for Metadata check box creates a metadata class for each entity selected in the entity list. Metadata classes will be covered later in this chapter, but in summary they enable you to decorate the properties on your entities with attributes (such as those specifying data validation rules) without actually needing to modify the entities themselves. Entities are generated code (created by the Entity Framework modeler), and modifying generated code is not a good idea. Doing so would prevent you from regenerating the code again at a later stage—if you did regenerate the code after manually altering it, then you would lose the changes you had made.

Metadata classes provide the solution to this problem. Each entity can have a corresponding metadata class, with each property on the entity having a corresponding field in the metadata class. You can then decorate the fields in the metadata class with attributes that RIA Services will map to the entity and use at design time when generating the corresponding entity on the client, and at runtime when persisting entities passed from the client to the database (to revalidate the data).

The Domain Service Class Wizard can generate these metadata classes for you for the entities you have selected from the list. These will go into a `.metadata.cs` file—for example, the `ProductService` domain service will have a corresponding `ProductService.metadata.cs` file created, containing metadata classes for all the entities that the `ProductService` domain service exposes.

It's not essential that you generate metadata classes for the entities that you expose from your domain service. However, if you choose not to create them when creating the domain service, and then decide you wish to use them at a later time, you will need to either create them manually or create another temporary domain service for the entities (from which you can get the generated metadata classes, and then delete the domain service).

Creating the ProductService Domain Service

To finish creating the ProductService domain service (which we'll use throughout this chapter and the following two chapters), take the following steps:

1. Select the Product entity and its corresponding Enable Editing check box.
2. Ensure that the Enable Client Access check box is selected.
3. Ensure that the Generate Associated Classes for Metadata check box is selected.
4. Click the OK button.

This will then create the domain service and metadata class(es) for you in your web project.

Domain Operations

Domain operations are methods on a domain service that can be invoked from a client. These operations will typically be CRUD operations, but you can also have invoke operations and custom operations.

In order for the RIA Services code generator to ascertain what type of operation a given method is (so that it can correctly generate the corresponding code in the client project), it uses a convention-based approach (by default) to do so. The conventions prescribe how the operation methods should be named and what pattern their signature should follow.

Alternatively, you can decorate the operation's method with an attribute that identifies what type of operation it is. In fact, you may even wish to explicitly decorate the operation's method with an attribute specifying the type of operation regardless. The naming/signature conventions (and alternative attribute decorations) for each operation will be included in this section's discussion.

■ **Note** Some people prefer to decorate their operations even when they follow the naming/signature convention in order to explicitly define what type of operation is being represented.

Let's now take a look at each of the various types of domain operations that you might have in a domain service.

Query Operations

A query operation enables you to expose data to a client. This data may be either a collection of entities/objects or a single entity/object. Let's look at each of these cases separately.

Returning a Collection of Entities

In the ProductService class created by the wizard, you will find a query operation that returns a collection of Product entities to the client, as follows:

```
public IQueryable<Product> GetProducts()
{
    return this.ObjectContext.Products;
}
```

The `GetProducts` operation returns an expression (via an `IQueryable<T>` interface type) that when executed by a LINQ provider will return a collection of the `Product` entity. Throughout this book, the LINQ provider we are using is the Entity Framework, although any LINQ provider can be used. Although you could return an actual collection from the query operation (such as a `List<T>`), returning an `IQueryable<T>` expression provides numerous benefits, and enables one of the most powerful features that RIA Services provides.

By returning an expression (that hasn't been executed against the LINQ provider as yet), you enable RIA Services to add additional query criteria to the expression, which it can then execute, and can subsequently return the results to the client. As you will see in Chapter 6, the client can specify filtering, sorting, grouping, and paging criteria as a LINQ query when calling a query operation on the server. This LINQ query will then be serialized and sent to the server when calling the query operation, and these criteria will be appended to the existing query before it is executed and the results are returned to the client. This provides a neat, easy, and elegant way of passing criteria from the client to the server, and having them included them in the query to the database.

Query operations can accept parameters as part of their signature, which you can use as an alternative means to pass filter and other criteria to the query operation. For example, the following query operation accepts a string that it will use to filter the results (and must be provided when calling the operation from the client):

```
public IQueryable<Product> GetProducts(string nameFilter)
{
    return this.ObjectContext.Products.Where(p => p.Name.StartsWith(nameFilter));
}
```

Returning a Single Entity

In addition to returning a collection of entities, you will often want to add a domain operation that returns a single entity matching the key you pass through to it as a parameter (this will support the drill-down-type scenario that will be covered in Chapter 7). To demonstrate this, we'll add a new domain operation to the `ProductService` domain service called `GetProduct`, which will return the `Product` entity with the matching `ProductID` passed through to it as a parameter:

```
public Product GetProduct(int productID)
{
    return ObjectContext.Products
        .Where(p => p.ProductID == productID)
        .FirstOrDefault();
}
```

■ **Note** `GetProduct` (which returns a single `Product` entity) is very similar in name to the default `GetProducts` domain operation, which returns a collection of `Product` entities. It's best not to have such similarly named operations on your domain service (it will cause confusion and lead to mistakes being made on both the server and the client); therefore, it's best to rename the `GetProducts` domain operation to something such as `GetProductCollection` or `GetProductList`, for example.

Naming/Signature Convention

A query operation has no convention for its naming, and can accept any or no parameters. For it to be implicitly identified by the RIA Services code generator as a query operation, however, it must return an `IQueryable<T>`, `IEnumerable<T>`, or an entity. Otherwise, decorate the method with the `Query` attribute.

■ **Note** You can configure the query operation to cache the results in order to save calls back to the server (from the client), or calls to the database (depending on how you configure the caching). Data caching is ideal when you have data that rarely (or never) changes. To utilize output caching, decorate the query operation with the `OutputCache` attribute, and configure the caching behavior by specifying values for its constructor parameters. These parameters include where the data should be cached (most commonly either on the server and/or the client), and how long the data should be cached (in seconds).

Insert/Update/Delete Operations

When a client application submits a changeset to the server, RIA Services will call the appropriate insert/update/delete operation for each entity in the changeset, based on the action specified for that entity in the changeset. These operations cannot be called explicitly from the client.

Naming/Signature Convention

The insert/update/delete operations must accept an entity as their parameter, upon which it will perform the corresponding operation (with no additional parameters) and not return a value. When naming the methods, note the following:

- *Insert* operation method names must start with `Insert`, `Create`, or `Add`. Otherwise, you will need to apply the `Insert` attribute to the method—for example, `public void InsertProduct(Product product)`.
- *Update* operation method names must start with `Update`, `Change`, or `Modify`. Otherwise, you will need to apply the `Update` attribute to the method—for example, `public void UpdateProduct(Product currentProduct)`.

- *Delete* operation method names must start with Delete or Remove. Otherwise, you will need to apply the Delete attribute to the method—for example, `public void DeleteProduct(Product product)`.

Example Insert/Update/Delete Operations

If you have enabled editing on the Product entity in the Domain Service Class Wizard, you will find that insert, update, and delete operations have been created and implemented for you in the ProductService domain service.

The following code is performs an insert operation:

```
public void InsertProduct(Product product)
{
    if (product.EntityState != EntityState.Added)
    {
        if (product.EntityState != EntityState.Detached)
        {
            this.ObjectContext.ObjectStateManager.ChangeObjectState(product,
                                                                    EntityState.Added);
        }
        else
        {
            this.ObjectContext.AddToProducts(product);
        }
    }
}
```

The following code performs an update operation:

```
public void UpdateProduct(Product currentProduct)
{
    if (currentProduct.EntityState == EntityState.Detached)
    {
        this.ObjectContext.Products.AttachAsModified(currentProduct,
                                                    this.ChangeSet.GetOriginal(currentProduct));
    }
}
```

The following code performs a delete operation:

```
public void DeleteProduct(Product product)
{
    if (product.EntityState == EntityState.Detached)
    {
        this.ObjectContext.Attach(product);
    }
    this.ObjectContext.DeleteObject(product);
}
```


Invoke Operations

An invoke operation is a nonspecific domain operation, and behaves the same way as a service operation in a standard WCF service. As a general rule, invoke operations don't return or act upon entities. They either initiate an action on the server (e.g., requesting the server to send an e-mail) or request data from the server, which returns a simple value or data for which changeset tracking is not required (e.g., requesting the current currency exchange rate).

Like query operations, invoke operations are methods that the client application can call directly. These operations are called immediately (unlike custom operations, which are queued and only called when a changeset is submitted to the server).

■ **Note** Unfortunately (at the time of writing), despite invoke operations being conceptually identical to service operations in WCF, they are not able to return complex types (other than entities). This is possibly the biggest limitation you will face when using RIA Services, as it means that slightly more complicated (although common) requirements such as returning a variety of objects to the client as a batch is sadly not possible (even if they don't require change management). The only way to work around this limitation is to drop back to plain WCF communication, which is hardly ideal. Hopefully this limitation will be removed in the near future.

Naming/Signature Convention

Invoke operations do not need to adhere to any naming or method signature convention, and essentially any operation in a domain service that does not fall into any of the other operation categories (via its name or method signature) is considered by RIA Services to be an invoke operation. To explicitly define a method as an invoke operation, apply the `Invoke` attribute to the method.

An Example Invoke Operation

The following code is an example of an invoke operation:

```
public decimal GetExchangeRate(string fromCurrency, string toCurrency)
{
    return ExchangeRates.GetRate(fromCurrency, toCurrency);
}
```

Custom Operations

Custom operations are operations that perform an action on an entity. For example, you might have an operation that discontinues a product, with all the logic to do so to be performed on the server. However, whereas an invoke operation is called immediately from the client, custom operations have their execution on the server deferred until the client submits a changeset to the server (hence treated in the same way as insert/update/delete operations).

When a custom operation is generated in the client project, it will be created as a method on the entity that it acts upon. In addition, it will also be created as a method on the domain context associated with the domain service.

■ **Note** Custom operations are called *after* any insert or update operations are called on the server. If a custom operation is called on an entity that is later deleted (before the changes are submitted to the server), the custom operation will not be called and will be discarded when the changes are submitted.

Naming/Signature Convention

Custom operations do not have a naming convention; however, they must not return a value, and they must accept an entity as their first parameter (any number of additional parameters is permitted). There is no specific attribute for identifying a custom operation, but instead you should use the `Update` attribute and set its `UsingCustomMethod` property to `true`.

An Example Custom Operation

The following code is an example of a custom operation:

```
public void DiscontinueProduct(Product product)
{
    // Logic to discontinue the product...
}
```

Decorating Entities

Decorating entities is the process of applying attributes to entity classes and properties to convey a certain intention. There are three primary reasons why you would want to decorate your entities with RIA Services attributes and data annotations:

- So you can control how the entity is created on the client (i.e., the properties from the server entity class that will be created on the client entity), and hence what data is passed over the wire
- To apply validation rules to the entity and its properties (validating what values are acceptable)
- To apply miscellaneous data annotations (such as whether the property is a key that uniquely identifies the entity, whether the property should be used for concurrency checks, display options for how the property should be presented on the client, etc.)

You can decorate entities with the required attributes directly or via metadata classes.

■ **Note** If you have used the ASP.NET Dynamic Data scaffolding framework, then you will be quite familiar with these attributes, as RIA Services uses the same attributes that it does (defined in the `System.ComponentModel.DataAnnotations` namespace).

Metadata Classes

As previously described, metadata classes enable you to decorate your entity with attributes without actually modifying the entity class itself (useful when you code-generate your entities and a regeneration re-creates the entity class). A metadata class is associated with your entity via an attribute on the entity class that points to it. In order to not have to modify the entity class to apply this attribute, the usual method is to create a partial class for the entity and apply the attribute to it instead.

Open up the metadata class that was created for the Product entity (from your Entity Framework model) to see how it is structured. The file name for metadata class files are generally of the form `<DomainServiceName>.metadata.cs`.

This file contains a partial class that extends the Product entity. This is merely so the `MetadataType` attribute can be applied to the entity, designating the corresponding metadata class.

```
[MetadataTypeAttribute(typeof(Product.ProductMetadata))]  
public partial class Product  
{  
    internal sealed class ProductMetadata  
    {  
        // Field definitions (removed for the purposes of brevity)  
    }  
}
```

This attribute requires the type for the metadata class associated with this entity as its constructor parameter. As you can see from the example, it is associating the `ProductMetadata` class contained within the `Product` class (i.e., the `ProductMetadata` class is a nested, or inner, class of the `Product` class). This provides the link between the entity class and the metadata class, which is used by RIA Services to associate the two together. It's not essential to use a nested class as your metadata class—you can actually use a class anywhere that can be referenced, but by default the metadata classes created by RIA Services are created as nested classes. You can then treat the metadata class as if it were the actual entity class, and decorate it with the required attributes.

Note that the metadata class that was created by the Domain Service Class Wizard contains a public field representing each property on the associated entity, which you can decorate with attributes as required.

Controlling Client Entity Generation

You may not necessarily want the entity that is created (by the RIA Services code generation) in the client project to be the same as the entity that is in your server project. For example, the entity may consist of properties that shouldn't be transferred to the client and/or made accessible to it (such as password properties, images, calculated properties, credit card numbers, etc.). Let's look at customizing what's generated on the client-side entity and what data is transferred to the client from the server.

Including/Excluding Properties

You can explicitly state that a given property is created on the client using the `Include` attribute, or not created by using the `Exclude` attribute (both found in the `System.ServiceModel.DomainServices.Server` namespace).

By default, all public properties on an entity that return simple types (such as `string`, `int`, and `bool`), or a small predefined group of complex types (such as byte arrays), will be automatically created as properties (i.e., included) on the client entity (unless they have the `Exclude` attribute applied). However, this means that other complex values, such as an entity collection associated with each requested entity being returned (i.e., entities associated with the associated entities via a foreign key relationship), will not be transferred to the client when the entity is transferred, and the collection property providing the association will not be created on the client's corresponding entity.

Including/Excluding Associated Entities

For example, by default the collection of product inventory entities for each product entity is not returned or accessible in the client project when the `Product` entity (or entity collection) is requested. You can, however, explicitly specify that these associated product inventory entities should be returned to the client along with the product entities, and a property to access them should be created on the product entity on the client.

You need to make two changes to enable this: one to the product's domain service that was created for you, and one to the metadata class for the product entity.

In the query operation that returns a collection of products in the domain service, you will need to request (from the Entity Framework) that the product inventory records associated with each product entity are retrieved from the database (using the `Include` method in the query):

```
public IQueryable<Product> GetProducts()
{
    return ObjectContext.Products.Include("ProductInventory");
}
```

■ **Note** You can chain additional `Include` methods to the statement in order to include additional associated entities in the request to the database.

The other change that needs to be made is to decorate the `ProductInventory` field (in the metadata class) with the `Include` attribute to indicate that it should be included and created on the entity generated in the client project (being a complex property and thus not included by default):

```
[Include]
public EntityCollection<ProductInventory> ProductInventory;
```

This property represents an association between the `Product` entity and the `ProductInventory` entity. However, RIA Services needs to know how these two entities relate (i.e., what property on each entity represents the relationship). The `Association` attribute is used for this purpose; however, because we are returning an entity from our entity model, this attribute will have already been applied to the property.

You do, however, need to explicitly decorate properties representing entity relationships such as this when you are exposing presentation model types. An example of using this attribute can be found in the Presentation Model Types section later in this chapter.

Enabling Associated Entities to Be Editable

If the `ProductInventory` entities are to be updatable, you will need to create the insert/update/delete operations for the `ProductInventory` entity on the `ProductService` domain service. Otherwise, you will receive an exception when you attempt to insert, update, or delete an inventory entity in the collection on the Silverlight client. Alternatively, you can configure your entity association as a compositional hierarchy by decorating the property with the `Composition` attribute. This will cause the parent entity to be marked as updated when one of its children's entities is updated, and will send all the children entities (regardless of whether they have been modified or not) back to the server when performing an update—effectively treating both the parent and child entities as a single unit. You can then handle updates of the children in the parent entity's update operation rather than requiring them to have their own update operations.

The `ExternalReference` attribute is designed to be used in conjunction with the `Association` attribute, and is used to specify that the related entity in an association relationship should not be created on the client side. The reason for using this attribute is that RIA Services only permits an entity to be exposed from a single domain service. By configuring the association, you are actually exposing the related entity type from this domain service. If you already expose the related entity from another domain service, then you will receive an error stating, "Entity types cannot be shared across DomainServices" when you attempt to compile the application, as it is attempting to create the same entity twice on the client. Decorating the property representing the association with the `ExternalReference` attribute prevents the domain service from exposing the related entity and circumvents the issue.

Validation

One of the important problems that RIA Services tries to resolve is where validation logic needs to be written and executed on both the server and the client. Not only is it a duplication of code and effort to write the same logic in both places, but this logic can easily get out of sync (with a change being made in one place but not the other). It's important to execute the validation logic in both places because you want to have a responsive user interface (without waiting for the server to tell the user that something is invalid), but you also want to validate the data on the server because there is no guarantee that the data came from your own client application (and that its rules weren't circumvented). By enabling you to define your validation logic on your entities on the server, and automatically applying the same validation attributes when it generates the corresponding entities in the client project, RIA Services overcomes this problem. Your validation logic only needs to be written (and maintained) in a single location, and is kept synchronized by the code generation of the entities in the client project.

Predefined Validation Attributes

You can apply a number of basic validation rules to your properties via attributes (from the `System.ComponentModel.DataAnnotations` namespace), including the following:

- The `Required` attribute enables you to specify that the property must have a value (i.e., nonnull). It also enforces that a string must not be empty (by default—

although the `AllowEmptyStrings` named property allows you to disable this behavior).

- The `Range` attribute can be used to specify the minimum and maximum values permissible for the property.
- The `StringLength` attribute can be used to set a minimum and maximum number of characters permissible for a string property. The default constructor accepts a maximum length (with a default minimum length of 0 characters), but you can also specify the minimum length using the `MinimumLength` named property).
- The `RegularExpression` attribute can be used to validate the value of the property with a regular expression. For example, you may want to validate that an e-mail address conforms to a valid e-mail address format using one of the many commonly used regular expressions for that purpose (of which many aren't perfect but are considered acceptable).

Examples of these rules are shown in the following code:

```
[Required]
[StringLength(50)]
[RegularExpression(@"(?<user>[^\s@]+)@(?<host>.+)" )]
public string EmailAddress;

[Range(0, 150)]
public int Age;
```

Custom Property Validation Attributes

Of course, these are only basic validation rules, so what if you need a validation rule that isn't included in the ones provided? In that case you can write your own. However, in writing the code for a validation rule, you will need it to be available on the client too. Luckily, RIA Services helps because it contains code-sharing functionality (between the server and the client), where code written on the server can be duplicated on the client by the RIA Services code generator. This feature is covered later in this chapter, but in the meantime let's take a look at what's required to write a validation rule. You have two ways of creating your own validation rule. One way uses the `CustomValidation` attribute to decorate the property (or class) or be validated with, to which you can pass a type (i.e., class name) and the name of the method in the class that does the validation. This method isn't ideal, because it requires you to pass the method name as a string, which is likely to cause problems when refactoring. A better and much neater way is to create your own validation attribute, which is the method we'll use here.

Let's create a simple rule that validates the entries in the `Class` property to limit it to a small set of predefined values. The valid values for this property are validated in database using check constraints; however, ideally we'd like to validate the property value before it gets to this stage. The first step is to create a class for your validation rule that inherits from the `ValidationAttribute` class (in the `System.ComponentModel.DataAnnotations` namespace). For example, we will call our class `ProductClassValidationAttribute`. The next step is to override the `IsValid` method from the base class. The value of the property to be validated will be passed in as a parameter to this method, along with a validation context—which contains information about the property being validated, the entity instance that the property belongs to, and so on. This method must return either a result of `ValidationResult.Success` (if the validation rule is passed) or an instance of the `ValidationResult` class to which you've passed an error message as the constructor parameter value. An example of a custom

validation rule to validate the Class property on the product entity to a set of predefined classes is as follows:

```
public class ProductClassValidationAttribute : ValidationAttribute
{
    protected override ValidationResult IsValid(object value,
                                                ValidationContext validationContext)
    {
        string productClass = value.ToString();
        bool isValid = false;

        string[] validClasses = new string[] { "H", "M", "L", "" };
        isValid = validClasses.Contains(productClass.ToUpper());

        return isValid ? ValidationResult.Success : new ValidationResult(
                                                                    "The class is invalid");
    }
}
```

Note that this validation class should be included in a shared code file (which will be described later in the chapter).

Now you simply need to apply the attribute to the property to be validated—very easy!

```
[ProductClassValidation]
public string Class;
```

Custom Class Validation Attributes

Now, what if you need to compare two properties in a validation (e.g., if the finish date property's value must be later than the start date property's value)? Again, you can create a custom validation rule (in the same way as before) and apply it to the entity as a whole (rather than at the individual property level). In this scenario you don't want the validation rule to run as soon as you modify one of the properties, because it will prove frustrating to users to raise a validation error when they make a change to one but have yet to make a change to the other. Specifying a validation rule at the entity level solves this problem by only validating the entity when it is saved (or navigating to another one). Creating a validation rule to validate an entity is the same as creating the ProductClassValidation attribute, but instead of a property value being passed in to be validated, the entire entity is passed in. For example, here is a custom validation rule for ensuring that the sell finish date on a product is later than the sell start date:

```
public class SellDatesValidationAttribute : ValidationAttribute
{
    protected override ValidationResult IsValid(object value,
                                                ValidationContext validationContext)
    {
        Product product = value as Product;
        return product.SellEndDate > product.SellStartDate ?
            ValidationResult.Success : new ValidationResult(
                                                                    "The sell end date must be greater than the sell start date");
    }
}
```

Then you can apply the validation rule to the entity like so:

```
[SellDatesValidation]
internal sealed class ProductMetadata
```

Presentation

Decorating your entities (which are essentially middle-tier components) with information for the presentation tier is somewhat controversial—it can make creating forms in your application very easy (using the drag-and-drop features in Visual Studio 2010), but it blurs the line between the middle tier and the presentation tier. Ideally, the middle tier should know nothing about how the presentation tier will display the data. In any case, some attributes are available that enable you to provide presentation information on your entities—whether you use these attributes or enforce a strict separation of concerns is up to you.

The primary attribute used for this purpose is the `Display` attribute. The `DataForm` control (from the Silverlight Toolkit) uses this attribute (if available) when it is automatically creating a data entry form from an entity, and the `DataGrid` control uses it when creating a data grid with automatically generated columns. The `Display` attribute has a number of properties that suggest to the presentation tier how the property should be displayed. Some of these properties control the following:

- Whether a field should be created in a data form or data grid for this property (`AutoGenerateField`)
- A “friendly” name for the property that will be used as the label of a corresponding field/column in a data form or data grid (`Name/ShortName`)
- A description for the property that will be displayed in a data form as a tooltip when the user moves the mouse cursor over an information icon next to the field (`Description`)
- The order (in relation to the other properties) in which a property will be displayed in a data form or data grid (`Order`)
- What grouping the field for this property should be displayed under in a data form (`GroupName`)

Miscellaneous Data Annotations

A number of important miscellaneous data annotations don’t fit in either of the previous annotation type categories, but are used to control how RIA Services manages the entity, or to provide information to the presentation tier on how this entity should be displayed.

Some of the more important data annotations used by RIA Services are as follows:

- The `Key` attribute is used to specify that a property (or multiple properties) on an entity uniquely identifies that entity (i.e., represents a primary key).
- The `Editable` attribute is used to specify whether a property should be editable or not. By default, properties are editable, but you can decorate a property with this attribute, passing a value of `false` to its `allowEdit` constructor parameter to specify that the value of the property should not be able to be altered. Any attempt

to do so will raise an exception on the client. You would typically apply this to the property that contains the entity key.

- The `ConcurrencyCheck` attribute is used to specify that a property should be used when performing concurrency checks back on the server before updating (or deleting) the entity in the database. The original value of the property (as retrieved from the server) will be passed back to the server when performing an update/delete operation so that it can be used in the concurrency check. Identifying and handling concurrency violations is discussed later in the chapter in the “Handling Data Concurrency Violations” section.
- The `RoundtripOriginal` attribute is used to specify that the original value of the property (as retrieved from the server) will be passed back to the server when performing an update/delete operation.
- The `Timestamp` attribute is used to specify that the property is a binary number (note that it is not a date/time) that is being used to version the entity, and is often used in concurrency checks. The original values of properties with this attribute will (like those with the `ConcurrencyCheck` or `RoundtripOriginal` attributes) be passed back to the server when performing an update/delete operation.

Presentation Model Types

The examples so far have passed entities from an Entity Framework model on the server to the client, but what if you’re not using an Entity Framework (or LINQ to SQL) model in your application? In fact, because entities from the model could be considered data access layer objects, it could be considered bad practice to pass these to your client and consume them in the presentation tier. Alternatively, perhaps the entities in your model just don’t match one-to-one what is required to be displayed in the presentation tier (i.e., the objects to be displayed need data aggregated from multiple entities), requiring you to design an alternative and more effective means of transferring the data. Or perhaps you just aren’t using the Entity Framework or LINQ to SQL as your data access layer, yet still want to take advantage of the benefits that RIA Services provides.

There are many additional reasons why you wouldn’t want to pass entities between the server and the client—although it is very easy to do so, the practice often creates problems due to the tight coupling between the data access layer and the presentation layer.

You’ll therefore often find the need to transfer custom objects (designed to cater to the needs of the presentation layer) rather than entities between the server and the client—populating these objects on the server from your data access layer, and then (if these objects are editable) manually persisting the changeset returned from the client back to the data access layer. Implementing custom objects rather than entities will help you maintain a clean separation of concerns between your data access layer and presentation layer, which is generally considered best practice despite requiring additional work to wire up this structure.

You may think of these objects in traditional terms where they might be called data transfer objects (DTOs) or POCO types. Generally, when using them with RIA Services, they are termed *presentation model* types.

■ **Note** Once upon a time, a common practice was to return data from web services in `DataSets`. Despite being a “heavy” way of exposing data, many developers liked to use `DataSets` for their simplicity and ability to modify the data structure being returned without needing to update references to the services exposing them. However, time has shown that the use of `DataSets` is a source of numerous problems in projects, and their use is now considered bad practice. Thus, many developers have turned to exposing data using lightweight, strongly typed DTOs instead. If no other reason compels you to stop using `DataSets`, the fact that Silverlight has no support for `DataSets` puts the final nail in their coffin. Therefore, don’t try to expose `DataSets` from your domain service in an attempt to return an untyped object whose structure you can modify at will. Neither RIA Services nor Silverlight has any support for `DataSets`—consider them finally dead!

Although RIA Services works very nicely with the Entity Framework and LINQ to SQL models (by creating domain services with operations for the selected entities automatically for you with the Domain Service Class Wizard, and implementing some core behavior in the base class to work with these types), fortunately it’s not limited to just transferring entities from these technologies. Let’s take a look at how to implement RIA Services that pass presentation model types between the server and the client.

Creating the Presentation Model Class

The first step is to create a standard class containing just the properties that should be transferred between the client and the server. As with the examples provided earlier with entities from the Entity Framework, you can decorate this class (either directly or using a metadata class) with validation rule attributes and such.

Note that one property (or more as required) must be decorated with the `Key` attribute to indicate that that value of the property uniquely identifies the object (otherwise you will receive a compilation error, because the changeset manager on the client cannot manage the changes without it).

For this example we will create a presentation model class named `ProductPM` with similar properties (although we’ll use only a subset for the purpose of brevity) to our `Product` entity, but demonstrate exposing presentation model objects from a domain service instead of Entity Framework entities. This object will not have any knowledge of how it is populated or persisted—it will be handled in the domain service.

Here is the code for the presentation model class:

```
public partial class ProductPM
{
    [Key]
    [Editable(false)]
    public int ProductID { get; set; }

    public string Name { get; set; }
    public string ProductNumber { get; set; }
    public decimal ListPrice { get; set; }
    public DateTime ModifiedDate { get; set; }
}
```

■ **Note** You can also use metadata classes with presentation model objects in the same way as was demonstrated with entities. For simplicity's sake, however, we are simply decorating the `ProductPM` class's properties directly with the attributes required by RIA Services without going to the trouble of creating a separate metadata class for it. Therefore, you will need to add a `using` directive to the `System.ComponentModel.DataAnnotations` namespace at the top of the file. Also worth noting is that RIA Services does not require your presentation model class to be defined in the web project (or even in a WCF RIA Services Class Library project, as described later in this chapter). For example, you may have the objects you want to expose from your domain service in a referenced assembly for which you don't have the code, and this is possible because RIA Services simply reflects over the given object and generates the code it requires accordingly in the client project—no source required.

Populating and Exposing Your Presentation Model Types

Now we need to create a domain service so the client can retrieve a collection of these objects, and update/insert/delete objects in the collection if appropriate. Unfortunately, the Domain Service Class Wizard only generates the CRUD operations in your domain service when you are using entities from the Entity Framework or objects from a LINQ to SQL model—it doesn't allow you to select objects of other types (because the CRUD logic for these objects does not necessarily follow a specific model). Therefore, we must create the domain service ourselves from scratch.

Add a new item to your project (in the Services folder), named `ProductPMService`, using the Domain Service Class item template, and select <empty domain service class> from the Available DataContexts/ObjectContexts drop-down list in the Domain Service Class Wizard dialog. This creates an empty domain service to which we can add the required operations. Notice that the domain service inherits from `DomainService`, instead of the generic `LinqToEntitiesDomainService` class that the previous domain services (that served up entities from the Entity Framework model) inherited from (providing the base functionality for how the domain service would handle the entities).

```
[EnableClientAccess()]
public class ProductPMService : DomainService
{
}
```

Now we need to add the query operation to the domain service. Because the data is coming from a query to the Entity Framework model, we can still take advantage of the benefits of returning an `IQueryable` expression by projecting data from the Entity Framework model onto a collection of the presentation model objects, and returning this query from the operation. Therefore, the client can add any additional sorting/filtering/grouping/paging that it requires. The following code demonstrates how this is done:

```
[EnableClientAccess()]
public class ProductPMService : DomainService
{
    private AdventureWorks2008Entities context = new AdventureWorks2008Entities();
```

```

public IQueryable<ProductPM> GetProducts()
{
    return from p in context.Products
           select new ProductPM()
           {
               ProductID = p.ProductID,
               Name = p.Name,
               ProductNumber = p.ProductNumber,
               ListPrice = p.ListPrice,
               ModifiedDate = p.ModifiedDate
           };
}

```

When executed, this expression will populate and return a collection of our presentation model object (ProductPM) to the client.

Updating Your Presentation Model Types

You can also implement the insert, update, and delete operations by taking the ProductPM object that will be passed into each one as a parameter, and manually updating the Entity Framework model with the changes accordingly.

Particularly when inserting (and sometimes updating) data on the server, you will want to return values that have been generated on the server back to the client (such as a primary key value generated by the database when inserting a row, or a timestamp value being used for row versioning in the database). You can do this using the Associate method on the domain service's ChangeSet object. Pass this method the presentation model object, the entity that was updated, and a callback function. Once the database has been updated, the callback function will be called, at which point you can update the presentation model object as required from the entity.

The following code demonstrates updating the ProductPM object with the ProductID assigned to the Product entity when inserting a new record into the database:

```

public void InsertProduct(ProductPM productPM)
{
    Product product = new Product();
    product.Name = productPM.Name;
    product.ProductNumber = productPM.ProductNumber;
    product.ListPrice = productPM.ListPrice;
    product.ModifiedDate = DateTime.Now;
    context.Products.AddObject(product);
    context.SaveChanges();
    ChangeSet.Associate(productPM, product, UpdateProductPMKeys);
}

private void UpdateProductPMKeys(ProductPM productPM, Product product)
{
    productPM.ProductID = product.ProductID;
}

```

■ **Note** If your presentation model object contains a property exposing a collection of another object that you want to be passed to the client, as discussed earlier in this chapter (see the “Controlling Client Entity Generation” section), you must explicitly specify that the property should be generated on the client by decorating it with the `Include` attribute. If the objects in the collection are to be updatable, you will need to add corresponding insert/update/delete operations to the domain service for the object type. For example, you may want to expose the collection of product inventory objects associated with each product object (with the `ProductPM` object containing a property representing a collection of related `ProductInventoryPM` objects). However, if you attempt to compile the project, you will find that you receive a compile error stating that you have an invalid `Include` specification. This is because you need to provide an association between the child object and its parent by decorating the property that exposes the collection on the parent with the `Association` attribute, which takes as its parameters a name for the association (of your own choosing) and the property names that link the parent and the child together. The following example demonstrates implementing a property that provides an association between the `ProductPM` and `ProductInventoryPM` objects:

```
[Include]
[Association("ProductPM ProductInventory", "ProductID", "ProductID")]
public IEnumerable<ProductInventoryPM> ProductInventory { get; set; }
```

Sharing Code/Logic Across Tiers

At times you may want to share custom code between the middle tier and the presentation tier. For example, you might want to share custom validation logic (and some types of business logic) between the server and the client. This may be a case of extending an entity with a partial class (such as adding additional methods and computed properties), or you may want to share completely new classes (such as one of the custom validation attribute class discussed earlier in the “Validation” section).

The RIA Services code generator enables this by automatically copying any files in the server project with a `.shared.cs` extension into the client project untouched. Therefore, you can write the code in one place (the server) and know that the client will be updated accordingly. Of course, because the shared code will be running on both platforms, you should ensure that the code you write will successfully run on both platforms, and that it doesn’t use any platform-specific functions.

For example, say you want to extend your `Product` entity to add a calculated property called `ProfitMargin` to it (which determines the expected profit to be made on the product by subtracting the `StandardCost` from the `ListPrice`) that will be used on both the server and the client.

To do so, you simply need to create a partial class of the `Product` entity somewhere in the server project, and be sure to include `.shared` in the file name. For example, create a file called `Product.shared.cs` in the `Models\Shared` folder in the server project. Then add the following code to extend the `Product` entity with the `ProfitMargin` property:

```
namespace AdventureWorks.Web
{
    public partial class Product
    {
```

```

        public decimal ProfitMargin
        {
            get { return ListPrice - StandardCost; }
        }
    }
}

```

This file will automatically be copied to the Silverlight project (due to its file name), and thus extend the Product entity accordingly in both the server and client projects.

■ **Note** If the property or method you are sharing is only required on the client (not the server), shared code may not be the most appropriate means of extending the entity. In this case you would be better off writing the partial class in the Silverlight project directly to extend the entity with that functionality on the client only.

Inspecting the Generated Code in the Silverlight Project

We'll look at how to actually consume a domain service in your Silverlight project over the following chapters; however, for the time being let's take a look at the code that RIA Services generates in the Silverlight project. As discussed earlier in the chapter, RIA Services automatically generates code in a hidden folder (named `Generated_Code`) under the Silverlight project that enables it to communicate with the domain services in your web project. You can view the code generated in this folder (as shown in Figure 5-5) by selecting the Silverlight project in Solution Explorer and toggling the Show All Files button to "on" (using the second button from the left in the Solution Explorer window's toolbar). The core file code generated by RIA Services is named `[webprojectname].g.cs`. For example, in the project structure shown in Figure 5-5, this file is named `AdventureWorks.Web.g.cs`.

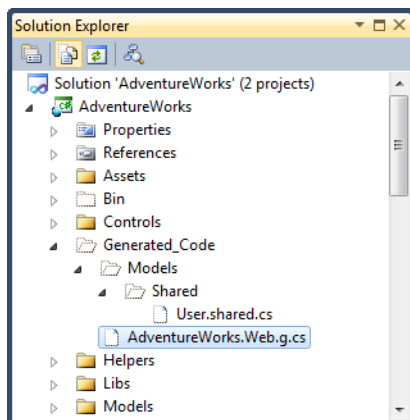


Figure 5-5. The Silverlight project structure, showing the hidden `Generated_Code` folder

When you open this file, you will find that each domain service in the web project will have a corresponding domain context class created in this file (that can then be used to communicate with the domain service from the client). If you follow the standard naming convention for naming domain services (XXXService, where XXX can be anything of your own choosing), the RIA Services code generator will name the domain context XXXContext. For example, a domain service named ProductService in the web project will have a corresponding domain context created in this file named ProductContext.

Each query, invoke, and custom operations defined on the domain service will have a corresponding method created on this domain context class that enables it to be called from the client. However, the insert/update/delete operations do not have corresponding methods created on the domain context, as they are never explicitly called from the client. Instead, as changes are made to an entity collection that's returned from a query operation, the domain context will maintain a changeset—consisting of the actions performed on the entities in the collection. When the SubmitChanges method is called on the domain context, RIA Services will send the changeset to the server and call the corresponding insert/update/delete operations on the domain service as required.

In addition, each entity (or presentation model class) exposed by a domain service in the web project will have a corresponding class created in this file. Note that any property whose corresponding field has been decorated with attributes in the metadata class for that entity on the server will have those same attributes applied directly to this generated client-side entity.

■ **Note** Inspecting the code generation files can also be useful when attempting to identify code generation problems. When debugging, you can also step into these files and set breakpoints to help identify issues. Just be sure not to modify these generated files, because any changes you make will be overwritten by the RIA Services code generator the next time it updates that file. If you want to add additional functionality to a generated class, create a separate file defining a partial class and extend the class in that manner.

Encapsulating Logic in a Separate WCF RIA Services Class Library

By default, when using RIA Services, your Silverlight project is linked directly to your web project (which is what the examples so far have demonstrated). You are defining your business objects and business logic in your web project, and this is being copied directly into your Silverlight project. However, from a design perspective this is possibly not an ideal structure for your solution (depending on the nature and scale of your web application), because this tightly couples both your Silverlight project and your business objects/logic to your web project (and limits their reusability).

It is possible, however, to move the business logic out of the web project and into a separate class library, using the WCF RIA Services Class Library project template. Unfortunately, the process to do so is not particularly straightforward, but let's first look at the structure that using RIA Services class libraries gives us.

The WCF RIA Services Class Library project template was designed to be added to an existing Silverlight project solution (rather than providing a full solution template). When you add a new project to your solution using this template, it will actually add two projects to your solution (one with .Web appended to the name you specified)—both placed into a solution folder that the project template creates (as shown in Figure 5-6). These two projects will share an RIA Services link, meaning that you can

now remove the link directly between the web project and the Silverlight project, removing the tight coupling between the two. Now you can move your domain services, entity model, metadata classes, presentation model classes, and shared code into the .Web class library, and RIA Services will generate the corresponding code into the Silverlight class library (in the same way it previously did into the Silverlight application itself).

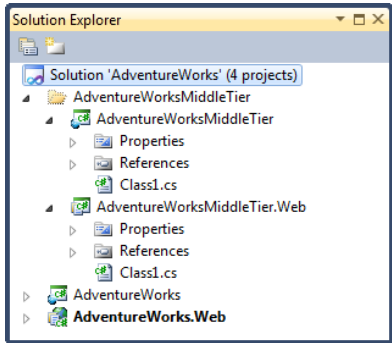


Figure 5-6. The structure of the RIA Services Class Library projects

Now it *should* simply be a case of adding a reference to the .Web class library to your web project, and adding a reference to the other class library to your Silverlight application. Unfortunately, however, this will most likely not be the case. If you are using the authentication features of RIA Services (which will be discussed in Chapter 8), then you will find that by putting the AuthenticationService in the class library, the WebContext object will not be generated by RIA Services in the Silverlight application (which is required to access the authentication service). You can work around this problem, however, by manually implementing the WebContext class and instantiating it when the application starts.

■ **Note** The sample project associated with this book uses this class library approach, and is configured with the authentication service workaround that you can work from.

Handling Server Errors

When errors occur on the server, it's useful to log them (such as to the event log, the database, or a text file) in order to help track down the problem that led to the exception, and often you may wish to add some additional handling behavior such as sending an e-mail notification to the system administrator. You could handle this in each domain operation, but that would mean repetitive exception handling code in each operation. An easy way to handle errors with RIA Services is to simply override the `OnError` method on the domain service:

```
protected override void OnError(DomainServiceErrorInfo errorInfo)
{
    // Log exception here
}
```


This method will be called each time an exception is thrown in the domain service. You can then log the error and add in an additional error handling logic. Useful information that you should include with any logged error includes when the error occurred (i.e., the date and time), who encountered the error (if the user is logged in), where the error occurred (i.e., the method), the error message, a stack trace, and any data associated (such as the current changeset, which can be obtained simply using the `ChangeSet` property on the domain service itself) that may help identify the cause of the issue. You will find most of these as properties on the `errorInfo` object.

To test your error logging and simulate something going awry in your domain service, you can simply throw an exception in one of the domain operations, which will then call your `OnError` override method, enabling you to log the error.

■ **Note** You can choose whether to directly propagate these exceptions to the client, or sanitize them first (in order to avoid exposing the potentially sensitive details of the inner workings of your service and the stack trace to the client). The steps to configuring your web project to sanitize the exceptions are detailed in Chapter 8.

Handling Data Concurrency Violations

When you have multiple users updating data in your application, you are sure to eventually be faced with data concurrency violations. Say you have two users who retrieve the entity representing the same product in the database at roughly the same time (such that the `Product` entity returned to both is identical). If both of those users decide to update that product, however, then you face a conundrum. If one user were to submit their changes, followed later by the other user, the first user's changes would be overwritten by the second user's changes, leading to the first user's updates being lost, and creating potential data integrity issues in your database.

This scenario is what is known as a *concurrency violation*. Ideally, you want the second user to be informed that the product has changed on the server, since they retrieve the data when they attempt their update, and to be able to respond to that situation accordingly (which may differ from application to application, or even based upon the particular data being modified).

■ **Note** Although concurrency violations are rare, you must cater for them accordingly in your code. Otherwise, you will find people complaining that the system is “losing data,” with no idea why.

Methods of Identifying Concurrency Violations

To appropriately handle this type of scenario, you will need to create (and design) a strategy to identify, handle, and resolve these concurrency violations, on both the server and the client. RIA Services uses optimistic concurrency (i.e., the record in the database isn't locked from being updated by other users after being retrieved), so you will need to design your strategy within these boundaries. Optimistic concurrency violations are generally identified by comparing given properties on the original version of an entity (before it had been modified on the client) with those on the stored version of the entity (as it

currently exists in the database), immediately before persisting data to the database. Therefore, concurrency checks generally involve three versions of the entity:

- The *original* version containing the state of the entity as originally retrieved from the server
- The *current* version, which generally consists of updates made on the client to the original version of the entity that are now being sent back to the server to be committed to the database
- The *stored* version, which can be currently found in the database prior to being updated

For example, a common strategy to identify concurrency violations is to have a column on your database table that is updated whenever the row is changed. Common column types for this purpose include

- A timestamp/rowversion column
- A uniqueidentifier (i.e., GUID) column
- A datetime column

When performing an update operation on an entity, RIA Services will retrieve the entity as it is currently stored in the database and compare the original value of the given concurrency check property sent back from the client with its stored value as found in the database. If the values are different, then a concurrency violation has been identified. A possible reason the value of this property might have changed since it was retrieved from the database is if another user has updated the entity since the original version had been retrieved from the server.

Alternatively, if you don't have a single property on the entity that you can use for concurrency checks, then you may compare a number of (or all) properties on the entity between its original version and the stored version, and raise a concurrency violation if any of the values are different. This method is far more cumbersome than the single-column method, although it does let you pick and choose which properties whose changes you don't want to be lost, while ignoring changes to others whose changes may be unimportant.

Configuring the Entity Model to Check for Concurrency Violations

By default, RIA Services (and the Entity Framework) will simply update an entity without performing any concurrency checks. You will need to configure it to perform these checks before updating/deleting data in the database.

The first part of the puzzle to implement concurrency checks is to send the original values of given properties back from the client that you want to compare for concurrency violations before an update operation. In order to minimize data traffic passing between the client and the server, only the original values of the entity's properties that are decorated with the `ConcurrencyCheck` attribute, the `TimeStamp` attribute, or the `RoundtripOriginal` attribute will be sent back to the server when performing an update operation. Therefore, you need to specifically identify these properties to RIA Services so their original values are returned to use in the concurrency check. However, despite providing the framework to identify the fields to use on concurrency checks, RIA Services doesn't actually provide the logic to implement the concurrency checks themselves. Instead, it leaves the task of identifying concurrency violations to the data access layer.

If you are using the Entity Framework as your data access layer (as we are in this book), then you can simply configure the entities in your entity model to perform concurrency checks prior to an update, specifying which of the entity's properties will participate in the comparison check. In your entity model, select one of these properties on the entity, and from the Properties window change its Concurrency Mode property from None to Fixed.

If you need to compare more than one property on the entity for a concurrency violation, then repeat this for each property that should participate in the comparison. Now, prior to any update of that entity, the stored version of the entity will be loaded from the database, and the stored values of the specified properties retrieved will be compared to their values as returned from the client, and a concurrency violation exception will be thrown if any of the values being checked are different.

■ **Note** If exposing entities from your entity model directly to the client, the entity properties identified in the entity model to be used in concurrency checks will already be decorated with the appropriate attributes to send their original values back from the client, removing the need to do so in your entity metadata class. Therefore, in this case, implementing concurrency checks is as simple as configuring them in your entity model. However, a little more work is required when you are using presentation model objects, as will be discussed shortly.

For example, the Product entity contains a property called `ModifiedDate` (with a type of `DateTime`), which we can use to perform concurrency checks. When updating the entity, we want the original value of this property to be sent back to the server along with the updated entity, and if its value has changed in the database since the entity was originally retrieved, then a concurrency exception should be thrown. To specify that this property should be used for the concurrency check, find the Product entity in your entity model, select the `ModifiedDate` property, and from the Properties window set its Concurrency Mode property to Fixed.

■ **Note** The `ModifiedDate` field on the Products table in the AdventureWorks2008 database is not automatically updated by a database trigger when the record is updated. Therefore, you will need to explicitly update this property when the Product entity is being added/updated. You should set this value on the server, not the client side, as doing so on the client side would assign the client PC's date/time (which can vary greatly based upon the user's time zone, or if they have the incorrect date/time set on their machine)—potentially misrepresenting the actual time that the record was updated. The easiest way to update this property is in the insert/update domain operations to the Product entity passed in as a parameter before continuing on with the add/update. The value you set here will be automatically passed back to the client.

Testing Your Solution

Assuming you have implemented functionality in your Silverlight application to retrieve and update Product entities, you can test it using two copies of your Silverlight application that are open at the same

time in different browser windows. Retrieve a given Product entity from the server in one instance of the application, and retrieve exactly the same entity in the other. Make a change to the entity in one instance, and send it to the server to update the database with the change. Now make a change to the entity in the other instance and send the update to the server. Because you updated the entity in the first instance, the ModifiedDate property of that entity on the server will have changed from its original value that is being sent through with the update in the second instance. Therefore, when the Entity Framework compares these values and finds that they are different, it will throw a concurrency exception.

Resolving Conflicts in the Domain Service

In this sort of scenario your domain service will inherit from `LinqToEntitiesDomainService`, which provides the `ResolveConflicts` method that you can override, which will be called when a concurrency conflict has occurred, and enable you to handle it using some custom logic on the server side if you wish. This method returns a Boolean value—if it returns false, then the client will simply be notified of the conflicts. If your method returns true, then the changeset will be resubmitted to the database once again (however, this will happen only once).

As mentioned earlier, a little more work is required if you are exposing presentation model objects to the client instead of entities. In this case you will need to specify which properties should have their original values sent back to the server when updating an object, and then handle the concurrency check logic yourself when updating the database.

The first step will be to decorate the properties on the presentation model object that will participate in the concurrency check (generally in the object's metadata class) with one of the attributes that ensure that the original value of that property will be sent back with any update of the object to the server (i.e., `ConcurrencyCheck`, `TimeStamp`, or `RoundtripOriginal`). For example:

```
[ConcurrencyCheck]
public DateTime ModifiedDate { get; set; }
```

You then need to compare the original value of this property with its stored value in the update operation of the domain service. However, you will find that only the updated version of the entity is passed into the operation as a parameter, whereas you also need the original version of this entity for your comparison. You can retrieve the original version of the object using the `GetOriginal` method of the `ChangeSet` object. Note that on this “original” object, only the properties specified to be round-tripped will have values.

```
ProductPM originalProduct = ChangeSet.GetOriginal<ProductPM>(currentProduct);
```

The next step depends on how you have implemented your data access layer. If your data access layer already performs concurrency checks, then no additional work is required. Otherwise, your update domain operation will need to manually retrieve the stored version of the entity from the database and compare the values of the properties between this and the original version to identify a concurrency violation. If one is found and you want to notify the client, you will need to get the `ChangeSetEntry` object for the object being updated, and assign values to its `ConflictMembers`, `StoreEntity`, and `IsDeleteConflict` properties. For example:

```
ProductPM originalProduct = ChangeSet.GetOriginal<ProductPM>(currentProduct);
ProductPM storedProduct = GetProduct(currentProduct.ProductID);

if (storedProduct.ModifiedDate != originalProduct.ModifiedDate)
```

```

{
    ChangeSetEntry entry =
        ChangeSet.ChangeSetEntries.Single(p => p.Entity == currentProduct);

    List<string> conflicts = new List<string>();
    conflicts.Add("ModifiedDate");
    entry.ConflictMembers = conflicts;

    entry.IsDeleteConflict = false;
    entry.StoreEntity = storedProduct;
}
else
{
    // Save ProductPM...
}

```

■ **Note** The `OnError` method (as discussed in the “Handling Errors” section of this chapter) is thrown when a concurrency violation has been identified. If you are logging errors, you will probably want to ignore the `OptimisticConcurrencyException` exception, as the errors do not indicate a failure in the system and thus are of little value being logged.

Handling concurrency violations on the client side will be covered in Chapter 7.

Transactions

Transactions are often used when updating data in a database, such that if an update fails (such as if a concurrency violation is identified), the updates made since the transaction was started will be rolled back, leaving the database in the same state as it was before the update started. The use of transactions ensures that the database doesn’t get into an unknown or inconsistent state because some changes but not others have been written to the database, potentially compromising its integrity.

Often you will want to encapsulate the changes being submitted to the database within a transaction, particularly when using a data access layer other than the Entity Framework or LINQ to SQL (as these both already automatically implement transactions when submitting changes). You can do so in a domain service by overriding the `Submit` method from the base class, and encapsulating the call to the `Submit` method in the base class in a transaction scope, as demonstrated here:

```

public override bool Submit(ChangeSet changeSet)
{
    bool result = false;

    TransactionOptions transactionOptions = new TransactionOptions();
    transactionOptions.IsolationLevel =
        System.Transactions.IsolationLevel.ReadCommitted;

    using (TransactionScope transaction = new TransactionScope(

```

```

        TransactionScopeOption.Required, transactionOptions))
    {
        result = base.Submit(changeSet);
        transaction.Complete();
    }

    return result;
}

```

■ **Note** You will need to add a reference to the `System.Transactions` assembly and a `using` statement to the `System.Transactions` namespace in your class in order to implement the preceding example.

WCF RIA Services Toolkit

The WCF RIA Services Toolkit consists of a number of out-of-band features that are (currently) not included in the core RIA Services framework. These typically are features that haven't quite reached the maturity required to be full-fledged members of the core RIA Services framework, but are updated frequently and made available for you to use while they gain maturity (without any guarantees that they will work as required, or won't radically change and/or break your code with future updates). By making them available in the toolkit, Microsoft is able to garner feedback from developers and update them accordingly in a timely manner without needing to wait for a new release of the full framework. In turn, Microsoft is providing new features on a regular basis for you to use, much earlier than you would otherwise be able to.

At the time of writing, the following features can be found in the WCF RIA Services Toolkit:

- *LinqToSqlDomainService*: Unfortunately, there is currently no support for LINQ to SQL models out of the box in the RIA Services framework. However, you can find the `LinqToSqlDomainService` class in the toolkit (similar to the `LinqToEntitiesDomainService` in the framework, but for LINQ to SQL) to provide this support.
- *SOAP Endpoint*: This enables you to expose a SOAP endpoint for your domain services.
- *JSON Endpoint*: This enables you to expose a JSON endpoint for your domain services.
- *ASP.NET DomainDataSource*: This is a data source control that enables you to communicate with your domain services in an ASP.NET application or web site.

A link to download the WCF RIA Services Toolkit can be found on the WCF RIA Services page on the Silverlight web site, at www.silverlight.net/getstarted/riaservices.

Alternative Communication Technologies

If you have developmental control of the server-side aspects of your application (i.e., you are building the services that will expose data to the application), RIA Services will probably (in most cases) be the most appropriate technology to use for consuming data in a Silverlight application. RIA Services has been specifically designed for Silverlight applications, and with its built-in changeset management functionality, in conjunction with its ability to return an `IQueryable` expression (such that filters/sorting/grouping/paging can be applied on the client but executed on the server), it is undeniably the most powerful communication framework available to Silverlight applications.

That said, it's not the only means to consume data from a Silverlight application—in fact, there are a plethora of technologies available for you to use—so much so that choosing the right one can be rather difficult.

You may be forced to consume data using another technology if you need to consume data from services that you have no control over, need to use a feature not supported by RIA Services (such as duplex communication), or simply find that the architecture of RIA Services does not fit in with the architecture of your application. Although this book specifically focuses on consuming data in Silverlight using RIA Services, let's take a look at some of these alternative technologies that you can use to consume data in a Silverlight application.

■ **Note** All server communication must be performed asynchronously. The Silverlight team made the decision to not support synchronous communication in order to avoid service/HTTP/socket calls blocking the Silverlight application's user interface (and in turn, the browser, as Silverlight runs in the browser's user interface thread) while waiting for a response. Regularly hanging the browser would not make Silverlight a good plug-in citizen. Therefore, you will need to make these calls asynchronous instead—starting the call and adding an event handler to the `Completed` event to handle the result.

If you're a developer only used to writing synchronous code, this will require you to think a little differently in how you structure the flow of your code. For example, if you wanted to queue service calls (in instances where the next service call to be made relies upon the result of the previous call), the asynchronous-only behavior complicates matters somewhat.

Despite the service/HTTP/socket call being performed on a background thread, the `Completed` event is actually raised on the user interface thread. Therefore, you don't need to worry about the cross-thread synchronization issues often experienced after making an asynchronous call (due to attempting to update the user interface from an event raised on a background thread). Therefore, there is no need to update the user interface via thread-safe calls using the `Invoke` method. This also applies to the other technologies listed here, with some exceptions. As a general rule, if the call raises an event when it completes, it will usually be raised on the user interface thread. However, if it uses asynchronous callback delegate (taking an `AsyncCallback` object as a parameter instead), then the callback method will be called on the background thread.

WCF Services

As previously mentioned, RIA Services is built on top of WCF. You will no doubt be familiar with WCF if you have ever written an application that consumes data from a server. In essence, WCF is a unified communication layer that implements a single programming model that can target different communication technologies. It enables you to write the messaging features for an application in a technology-agnostic manner, with the actual technology to be used (referred to as *bindings*) specified through configuration files. In other words, you can write the communication functionality for your application without worrying about what technology it will use and how that communication will take place, and without tying your application to a specific messaging technology. This is achieved by WCF providing a common model for you to write services against, which it can then map to the configured messaging technology. This model is broken into three components—the service's *address*, its *bindings*, and its *contract*, with a clear separation maintained between each. You define the contract (i.e., the functionality to be exposed by your service), and then you can configure the address and the bindings that the service supports to form a service *endpoint*.

Configuring Bindings

The biggest limitation to consuming WCF services in Silverlight is that not all bindings (i.e., messaging technologies) supported by WCF in the full .NET Framework are supported by Silverlight (for consumption). Therefore, you must ensure that the WCF service that you wish to consume is actually configured to use a binding that Silverlight supports. The bindings supported by Silverlight are

- **basicHttpBinding**: Uses the XML-based SOAP 1.1 protocol when communicating with the server. This is the protocol used by the older-style ASMX web services, and sends messages in clear text over HTTP.
- **pollingDuplexHttpBinding**: Enables the server to push messages to the client in addition to simply handling client requests. As its name suggests, this binding works over HTTP, with the client using HTTP long-polling the server (behind the scenes), waiting for it to send it a message. This is useful when you want the server to send notifications to the clients, although the TCP binding is now a better and more performant choice to implement this type of behavior.
- **netTcpBinding**: Used for similar purposes as **pollingDuplexHttpBinding**; however, instead of working over HTTP, it uses the **net.tcp** protocol. The performance of this binding is much better than **pollingDuplexHttpBinding** (primarily because there is no polling of the server); however, it is better suited to intranet scenarios, as it uses ports other than HTTP port 80 (although it's restricted by Silverlight to ports 4502 to 4534), which are often blocked by firewalls. Issues that you may face when using this binding include the fact that you cannot use transport-level security (it's not supported by Silverlight with the **net.tcp** protocol), there are a number of features you need to configure on your machine to support **net.tcp** communication, and you need to expose a socket policy file on port 943 in order to permit TCP connections from Silverlight applications.
- **customBinding**: Enables you to tailor a binding to suit your requirements where a standard binding matching these requirements does not exist. Defining a custom binding enables you to customize the configuration of the various layers forming the binding, which includes the encoding of the messages (such as clear text,

binary, etc.) and the transport over which they will be transferred (such as HTTP, HTTPS, TCP, etc.).

■ **Note** The `wsHttpBinding` often used by the default endpoints configured by Visual Studio for WCF services is not supported in Silverlight. Therefore, you will need to explicitly define an additional endpoint (or modify the existing configuration of an endpoint) that implements a binding supported by Silverlight. If you are creating the WCF service yourself, use the Silverlight-Enabled WCF Service item template (instead of the standard WCF Service item template), which will be configured in a manner friendly to being consumed by a Silverlight application (using a binding supported by Silverlight, and allowing the service to run in ASP.NET compatibility mode).

You may have noticed that neither RIA Services nor WCF services created from the Silverlight-Enabled WCF Service item template use any of the standard bindings. Instead, they define a custom binding to binary-encode the messages before sending them over an HTTP connection. The reason they need to define a custom binding is because none of the bindings supported by Silverlight implement binary encoding of messages. The big advantage achieved by binary-encoding messages (instead of sending them as plain text) is that you have much smaller messages passed between the server and the client.

Consuming a WCF Service

Once you have the service configured, you can then add a reference to it (using the Add Service Reference context menu item when right-clicking the Silverlight project in the Solution Explorer window), and consume it in exactly the same way as you would consume a service (asynchronously) in any .NET application.

If you attempt to add a reference to a service that has no endpoint with a binding supported by Silverlight, the service reference will still be added to your project, but a warning will appear in the Error List window stating that the endpoint is not compatible with Silverlight. Your project will compile, but attempting to make a call to the service will result in an `InvalidOperationException` exception.

Common Pitfalls

There are a couple of pitfalls to be aware of when calling WCF services in Silverlight. As both RIA Services and Data Services (discussed shortly) are built upon WCF, these issues also apply to those technologies.

Maximum Message Size

The WCF service on the server has a limit to the maximum message size it will return as a response, and the Silverlight client also has a maximum message size that it will accept. Although the default maximum message size accepted by the Silverlight client is configured at an impossibly large 2GB, the service in the web project has a default maximum message size of 64KB. Attempting to return a response larger than this from your service (common when returning large lists of data) will result in an exception being thrown. You can modify this limit by customizing your binding properties in the `web.config` file, setting an alternative maximum message size (in bytes). That said, it's probably better to design your

server/client communication to transfer smaller messages (where possible). Unless the user requires all that data at the one time, you would be better aiming to minimize data traffic (and hence the time that the user is left waiting for a response from the server), only sending the data back to the client that the user requires at that time. For example, when displaying a list of data to the user, request the data in pages rather than as a single unit.

ASP.NET Session State

Despite Silverlight enforcing that server calls must be performed asynchronously, if you have ASP.NET session state turned on in your web project, you will find that calls are executed on the server sequentially, resulting in the calls appearing slower than expected.

You can tell if ASP.NET session state is turned on by using a tool such as Fiddler (which you can download from www.fiddler2.com) to intercept the responses from the server. If a cookie with the name `ASP.NET_SessionId` appears in the header, then ASP.NET session state is turned on. Generally, ASP.NET session state will be turned on automatically when you add a `Global.asax` file to your project.

This behavior only occurs when using the browser HTTP stack (the default when the application is running inside the browser). To work around this behavior, you can either stop using ASP.NET session state or simply use the client HTTP stack instead (discussed later in this chapter).

Sharing Code, Business Objects, and Logic Between the Server and Client

One of the great features of RIA Services is the ability to share code, business objects, and business logic between the server and the client. However, there is a little-known method for sharing these when using plain WCF services. The trick is to put the classes and shared code in a Silverlight Class Library project, and then add a reference to this assembly in both your Silverlight and web projects. Specifically create this project as a Silverlight class library (rather than one targeting the full .NET Framework), as you can only add a reference to an assembly targeting the Silverlight runtime to a Silverlight project; however, projects targeting the full .NET Framework can reference assemblies that target the Silverlight runtime. Now you have any code and business objects that you include in this assembly shared between both the client and the server.

Note that this still leaves you the task of manually populating your business objects after data has been retrieved from the server. This leads to trick number two. When you create your WCF service in the server project, expose data using the business objects from this shared assembly. Usually, adding a service reference to a project creates a proxy containing objects with the structure matching the objects being exposed by the service. It then populates these proxy objects with data returned from the server, and only accepts objects of these types when sending data back to the server. However, you can configure your service reference to populate the types defined in one or more given assemblies (instead of populating the proxy objects). After selecting the service you want to add a reference to (in the Add Service Reference dialog), click the Advanced button to open the Service Reference Settings dialog. Ensure that the “Reuse types in referenced assemblies” check box is selected. You can then select specific assemblies for it to use, or have it search for matching types in all the assemblies referenced by the project. This means that it will automatically populate objects from the shared assembly when you request data from the server, and accept objects of those types when you pass data back to the server—hence saving you the need to copy the data between the service’s proxy objects and the business objects from your shared assembly when communicating with the WCF service.

WCF Data Services

WCF Data Services (or Data Services for short) is another data-centric communication technology built on the WCF stack that exposes data to clients as JSON- or ATOM-serialized RESTful resources using the OData protocol. The OData protocol specifies how a client can query, navigate, and update data exposed by a service. Using this protocol, the client can communicate with the service in a RESTful manner by retrieving data using a standard HTTP GET call (using specially constructed URIs), and sending updates to the server using a standard HTTP POST call.

Data Services vs. RIA Services

Data Services shares a number of similarities with RIA Services, and in a way you could consider the two to be sibling technologies, at times both jostling for attention and attempting to outdo each other. Both are built on top of WCF and conform to a pattern to expose data to clients, but each has a somewhat different focus. RIA Services (at least for the time being) specifically targets Silverlight clients, whereas Data Services is client agnostic. While RIA Services generates code in the client project in order for the client to communicate with the services (an aspect that concerns some developers who are against “magic” code generation that they have no control over), Data Services does not need a link to the client project, and no code generation is required. Instead, Data Services provides a client library (discussed shortly) to help the client interact with the service.

It can be difficult to identify whether you should use RIA Services or Data Services in your project, and how the two differ. However, you can break it down to the following:

- *RIA Services* has a rich model designed for the development of end-to-end Silverlight applications (with the server tightly linked with its client via the code generation process).
- *Data Services* is designed simply to expose data as RESTful resources in a standalone manner that a range of clients can consume (such as reporting tools that can consume data exposed over HTTP and serialized as either ATOM or JSON).

That said, RIA Services can also expose data as RESTful resources that can be consumed by a range of clients (by enabling OData or exposing a JSON endpoint using the functionality provided in the WCF RIA Services Toolkit). Also, together with its ability to share code and validation logic with the client (among the numerous other features it provides), you will find it the best solution when writing an end-to-end Silverlight application. However, if you want your service to support a wide variety of clients without having a tight relationship to your Silverlight project, Data Services is a viable alternative.

Using Data Services

When you add a data service to your project using the WCF Data Service item template, you configure it to expose your entire entity model. You can then configure permissions for each entity in that model (if you wish) to restrict which entity sets can be read, modified, and so on, and which service operations are available.

Once your service is configured, add a service reference to it in your Silverlight project. This will also automatically add a reference to the Data Services client library (`System.Data.Services.Client.dll`). Then you can start interacting with the service and performing CRUD operations on the entities exposed by the service.

In a similar manner to RIA Services, Data Services exposes queryable endpoints (by returning `IQueryable` expressions from operations). By using the client library, you can write LINQ queries on the client against these endpoints and have them executed on the server. This was one of the most powerful features provided by RIA Services, and it's also available (with a slightly different implementation) in Data Services.

Data Services does manage a changeset on the client; however, by default the changeset entries are processed individually (i.e., each operation is processed one at a time when you submit the changes). This creates an issue if one of these operations fails, potentially leaving your database in an unknown or inconsistent state. You can send these changes to the server as a batch; however, they will still be processed individually on the server side, with no transaction and rollback if one of the updates fails. This is one of the downsides of using Data Services, although you can implement transactions by hooking into the processing pipeline.

HTTP Requests

Despite being considerably more work (since you will have to parse the results yourself), another means of communicating with a server is through the use of standard HTTP requests. For example, you may wish to download an XML file from the server, or even consume RESTful services (aka WebHTTP services). You can also use this method to download files from a server (such as images).

Making an HTTP Request

There are two ways to make HTTP requests in Silverlight. Silverlight has both a `WebClient` and a `WebRequest` class that can be used for uploading and downloading data over HTTP to/from the server. `WebRequest` gives you finer control over the request; however, `WebClient` is the easiest to use for simple HTTP download tasks. It provides two means of downloading data (`DownloadString` and `OpenRead`) and two means of uploading data (`UploadString` and `OpenWrite`). `DownloadString/UploadString` simply downloads/uploads the data as a string (making communicating plain text easy), and `OpenRead/OpenWrite` uses a stream (best for binary data). For example, this is all it takes to download a text file from the server and load the contents into a string variable:

```
private void StartDownload(string url)
{
    WebClient client = new WebClient();
    client.DownloadStringCompleted += client DownloadStringCompleted;
    client.DownloadStringAsync(new Uri(url));
}

private void client DownloadStringCompleted(object sender,
                                           DownloadStringCompletedEventArgs e)
{
    string result = e.Result;
    // Do something with the result
}
```

Choosing a Networking Stack

By default, Silverlight uses the browser's networking stack (when running inside the browser). This places some limits on what you can do when making HTTP requests from Silverlight, as the browser's stack has limitations, and it sometimes modifies the response before passing it back to your application (particularly in regard to HTTP status codes). In order to support Out Of Browser (OOB) mode, the client networking stack was introduced in Silverlight 3, and you can invoke this networking stack instead to work around the browser's networking stack's limitations. Using this stack enables you to access HTTP headers and cookies, to use HTTP verbs other than GET and PUT, and to obtain the actual HTTP status codes returned from the server. It also provides various other benefits. There are some downsides, however, such as the lack of content caching and authentication support. Simply call the `RegisterPrefix` method on the `WebRequest` (or `HttpWebRequest`) class, passing in a prefix (which can be the scheme, or even a domain) to which future calls will use the given networking stack, and passing in the networking stack that those calls will use (with `ClientHttp` representing the client networking stack):

```
WebRequest.RegisterPrefix("http://", WebRequestCreator.ClientHttp);
```

Sockets

Sockets enable you to communicate bidirectionally with the server (i.e., the client can send messages to the server, and the server can send messages to the client). This is particularly useful when you want the server to send the clients notifications, or if you are writing a chat client, for example. Silverlight supports socket communication (with classes in the `System.Net.Sockets` namespace for support. For scenarios where you want to broadcast messages to all clients, IP multicasting is also supported.

As discussed earlier, you are restricted to only using ports 4502 to 4534 unless you are running in OOB mode with elevated privileges. Sockets are best used in intranet applications (rather than public-facing applications), as the ports will possibly be blocked behind a firewall. In this case you may be better off using a WCF service using the `pollingDuplexHttpBinding` binding to implement bidirectional duplex communication with the server.

Summary

This chapter taught you how to use WCF RIA Services to expose data and operations from the server to your client Silverlight application, and how to implement all the server-based requirements associated with doing so (such as handling errors, identifying and handling concurrency violations, and sharing code across tiers). The following chapters will show you how to consume this data and call these operations to create a functional Silverlight application.

CHAPTER 6



Implementing Summary Lists

Summary lists (see Figure 6-1) are designed to enable users to quickly locate and identify a record (of a given type) among the existing data in the system. Once they have found the record, they can then drill down to the record to view it in detail (and potentially edit it).

Instead of requiring the full record for each item in the list to be sent back to the client, summary lists should only request and display enough data from the server to enable the user to identify the record they wish to view/edit. Once the user has selected the record, the application will go back to the server and retrieve the remaining data for that record (and any associated/supporting data) to populate a details screen or data entry form. This is a common strategy for working with data in business applications, which minimizes the amount of data being sent to the client from the server, and facilitates a drill-down approach for interacting with the detailed data.

In this chapter we'll expose a collection of summary data for products from the server, which will then be used to populate a DataGrid or a ListBox on the client. The user will then be able to manipulate the data in the list to suit their needs (with the actual manipulation of the data taking place on the server), and drill down on a particular item (such that they can view and edit its details).

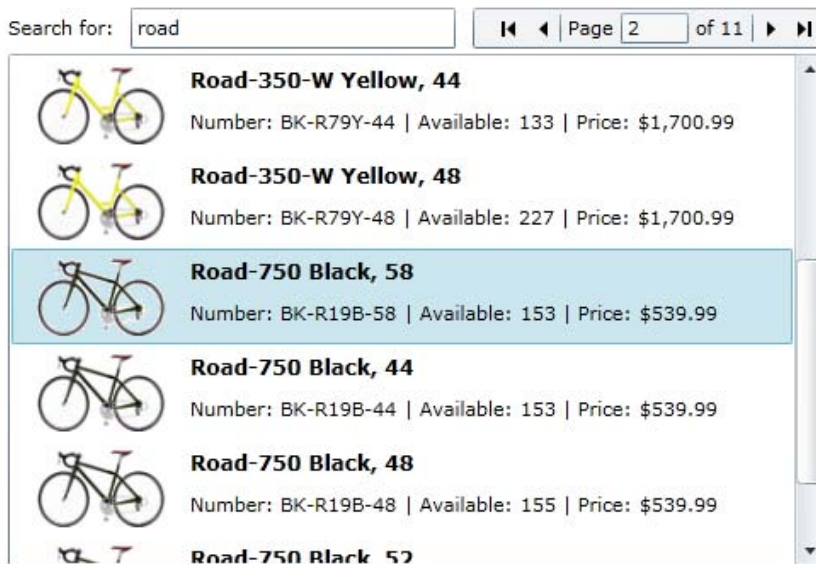


Figure 6-1. A summary list

Exposing a Summary Collection from the Server

In most cases, you will want to return your summary list data as a collection of presentation model objects (a concept discussed in Chapter 5). The benefits of specifically using presentation model objects in this scenario is that they

- Allow you to specify just the particular properties you want passed back from the server
- Combine data from multiple entities (from your entity model) into the one object
- Summarize/total related data
- Denormalize the data suitable for display to the user

All of these benefits will be demonstrated in the example we'll work through in this section. In this example we will return a summary list of product records that will:

- Return a key that will be used to drill down on the record
- Return various properties of the product record (name, number, etc.)
- Return an thumbnail image of the product (which is obtained from a different entity in the entity model on the server than the core Product entity, hence demonstrating combining data from multiple entities into the one object)
- Return the total quantity of that product available from all the inventory locations (demonstrating summarizing/totaling related data)
- Return the denormalized descriptions of the product's category, subcategory, and model (which are each stored in their own table, and referred to by a foreign key on the Product entity)

Create a presentation model class in the Models folder in your web project named ProductSummary that will be used to return this data to the client (note that, as a general rule, we'll suffix the name of the summary classes with Summary).

```
public partial class ProductSummary
{
    [Key]
    [Editable(false)]
    public int ID { get; set; }
    public string Name { get; set; }
    public string Number { get; set; }
    public decimal ListPrice { get; set; }
    public byte[] ThumbnailPhoto { get; set; }
    public int? QuantityAvailable { get; set; }
    public string Category { get; set; }
    public string SubCategory { get; set; }
    public string Model { get; set; }
    public bool MakeFlag { get; set; }
}
```

■ **Note** You will need a using directive to `System.ComponentModel.DataAnnotations` at the top of this file.

Once you have created the presentation model class (with the required properties and metadata attributes) you will need to create a domain operation to return a populated collection of those objects. This primarily consists of writing a LINQ query to collate and summarize the required data from the entity model into a single summary object, and using this to populate the collection.

■ **Note** When creating the presentation model class, remember that at least one property must be decorated with the `Key` attribute; otherwise, you will receive a compilation error.

Ideally, you should return the results from a domain operation as an `IQueryable` expression, which will project data from the Entity Framework model onto a collection of the presentation model objects and return the results. As described in Chapter 5, returning an `IQueryable` expression enables the client to request the results to be filtered/sorted/grouped/paged, with these criteria being added to the `IQueryable` expression before executing it. In other words, the client can query the entire collection being exposed by the domain operation, with the query being run on the server, and only the results returned to the client (reducing unnecessary data traffic between the server and the client). Here is the code to populate and return an `IQueryable` expression that will populate our `ProductSummary` objects with data from the entity model:

```
public IQueryable<ProductSummary> GetProductSummaryList()
{
    return from p in this.ObjectContext.Products
           select new ProductSummary()
           {
               ID = p.ProductID,
               Name = p.Name,
               Number = p.ProductNumber,
               ListPrice = p.ListPrice,
               ThumbnailPhoto = p.ProductProductPhotos.FirstOrDefault().
                               ProductPhoto.ThumbnailPhoto,
               QuantityAvailable = p.ProductInventory.Sum(pi => pi.Quantity),
               Category = p.ProductSubcategory.ProductCategory.Name,
               Subcategory = p.ProductSubcategory.Name,
               Model = p.ProductModel.Name,
               MakeFlag = p.MakeFlag
           };
}
```

■ **Note** Depending on your architecture, you may like to create a standalone domain service to host this domain operation, or otherwise simply add it to the same service that contains the product-related CRUD operations.

Note that we're returning an image from the database in the `ThumbnailPhoto` property of the `ProductSummary` object. Returning an image from the database for each item to display in a summary list results in increased network and bandwidth usage, so it's recommended to only return images if they add value to your application, and make sure that they are small (preferably less than 2KB each).

Collections and Collection Views

Since the data being exposed from the server is a collection of objects, we should take a look at the support Silverlight has for collections and working with them.

Silverlight supports many of the common generic collection types you may be used to from the full .NET Framework, including `List`, `Dictionary`, `LinkedList`, `Stack`, and `Queue`. Other nongeneric collections such as the `ArrayList` and `Hashtable` aren't implemented in Silverlight, but you should be able to find an appropriate alternative among the generic collections.

Silverlight has some additional types of its own, however, which we'll discuss here.

ObservableCollection<T>

An important collection type unique to Silverlight and WPF is `ObservableCollection`. This is a generic collection type that implements the `INotifyPropertyChanged` interface (exposing a `CollectionChanged` event), which is raised when items are added or removed from the collection. Bound user interface controls automatically handle this event and update themselves when the collection has changed. One of the biggest benefits obtained from this feature is that you can update the collection in the code-behind (or in a `ViewModel`, as discussed in Chapter 12), and the changes will be automatically propagated to the user interface. In essence, the code that is modifying this collection doesn't need to know anything about the user interface, or that the collection is even bound to a user interface that needs updating. This feature makes the `ObservableCollection` type a key part of Silverlight's data binding ecosystem.

Views

Another important data binding concept related to collections is that of *views*. Views enable collections to be manipulated in the user interface, but without altering the underlying collection. Typical view manipulations you might want to perform include filtering, sorting, grouping, and paging the collection's items. For example, you might want to bind a collection to a `ListBox`, and enable the user to filter the items by entering some text into a `TextBox`. Instead of adding/removing items from the bound collection, you can simply filter the view of that collection. Alternatively, the control itself may manipulate its view of the collection. A good example of this is the `DataGrid` control—it allows you to sort its rows by clicking a column header, and it does this without modifying the bound collection—simply by manipulating its view.

Whenever you bind a collection to the property of a user interface control, a view is automatically wrapped around that collection (internally within the control) and bound to the property instead.

Alternatively, if you want control over the view, you can create it yourself, manipulate the data in the view, and bind to that instead. There are three primary types of views that you will deal with in Silverlight: the `CollectionViewSource`, the `PagedCollectionView`, and the `DomainDataSourceView`.

The `CollectionViewSource`

The `CollectionViewSource` isn't actually a view as such, but a view proxy. You assign a collection to its `Source` property, and it will provide you a corresponding view from its `View` property (as a `ListCollectionView`). You can then manipulate the view as you see fit, via properties such as `Filter`, `GroupDescriptions`, and `SortDescriptions`. For example, the following code creates a view of a collection called `productCollection`, sorts the items by their `Name` property, and assigns the view to the `ItemsSource` property of a `ListBox` control named `ProductListBox`:

```
CollectionViewSource view = new CollectionViewSource();
view.Source = productCollection;
SortDescription sortBy = new SortDescription("Name", ListSortDirection.Ascending);
view.SortDescriptions.Add(sortBy);
ProductListBox.ItemsSource = view.View;
```

The `PagedCollectionView`

Unlike the `CollectionViewSource`, the `PagedCollectionView` is an actual view, and can be created directly (rather than via a proxy like the `CollectionViewSource`, which then exposes a `ListCollectionView`). It has all the features of the `CollectionViewSource`, but also enables you to page the data (which the `ListCollectionView` has no support for). If you want to manually create a view, this is probably your best choice. The following example is similar to the preceding one, but uses the `PagedCollectionView` instead:

```
PagedCollectionView view = new PagedCollectionView(productCollection);
SortDescription sortBy = new SortDescription("Name", ListSortDirection.Ascending);
view.SortDescriptions.Add(sortBy);
ProductListBox.ItemsSource = view;
```

■ **Note** Later in this chapter (in the “Manipulating the Summary List” section), we’ll look at how to use the various features of the `PagedCollectionView` to manipulate the view of a collection.

`DomainDataSourceView`

As with the `ListCollectionView`, you can't create an instance of a `DomainDataSourceView` object in your code. Instead, this is a view exposed by the `DomainDataSource` control (discussed shortly) by its `DataView` property, enabling you to manipulate the underlying collection that it manages. Generally, you will use this view when manipulating the contents of the collection managed by the `DomainDataSource` control in code.

Consuming Data from the Server

Now that we have exposed a collection of summary objects from the server, we can request this data and consume it in our Silverlight application. The key component to retrieving the data from the server is the domain context, so we'll take a look at how that operates first. Then we'll look at the two approaches of interacting with the domain context (using either a XAML-based approach or a code-based approach), and discuss when it is most appropriate to use each one.

Understanding the Domain Context

As discussed in Chapter 5 (in the “Inspecting the Generated Code in the Silverlight Project” section), a domain context class is automatically generated in the Silverlight project by RIA Services for each domain service in the web project. Your code interacts with it, and it talks to the server (via the `DomainClient` class). In other words, it essentially acts as a proxy to facilitate communication with the domain service from the client. However, the way it is structured can be a little confusing. Query operations on the domain service are exposed on the corresponding domain context object as methods (their name suffixed with `Query`) that simply return an `EntityQuery` object.

For example, the domain operation we created on our `ProductService` domain service called `GetProductSummaryList` will have a corresponding method on the domain context called `GetProductSummaryListQuery` on the domain context object. However, this method doesn't actually initiate the request to the server. Instead, it returns an `EntityQuery` object that represents the domain service query. How you actually initiate the server request now depends on which of the two approaches (XAML based or code based) you decide to take to interact with the domain context in your application—so let's take a look at each of these now.

■ **Note** The concept of obtaining an `EntityQuery` object in order to query the server probably seems a little strange, but it makes more sense when you want to refine the query in order to have the data filtered/sorted/grouped/paged on the server before it is returned to the client. We will look at this feature later in this chapter in the “Manipulating the Summary List” section; however, until then we will simply work with the query as it is—essentially requesting the full collection from the server.

XAML-Based Approach

The key component in the XAML-based approach is the `DomainDataSource` control. The `DomainDataSource` control is a part of the RIA Services framework, and provides a bridge that enables you to declaratively interact with a domain context in XAML. You configure it by assigning it a domain context (which handles the communication with the server) and providing it the method on this domain context that returns the domain operation query you want to call. You can then consume the data from the server by binding the `DataContext` or `ItemsSource` property of controls in your view to the `Data` property of the `DomainDataSource` control.

Although you could drag and drop a `DomainDataSource` control from the toolbox (or write the XAML by hand) and configure it manually, the easiest way to set up and configure one is to enlist the Data Sources window to help you. When you open the Data Sources window in Visual Studio, you will

find that a data source has already been created for each domain context created by RIA Services in the Silverlight project, with the entities that they expose beneath them (as shown in Figure 6-2). Dragging and dropping one of these onto your design surface will create both a `DomainDataSource` control and a bound control to display the data.

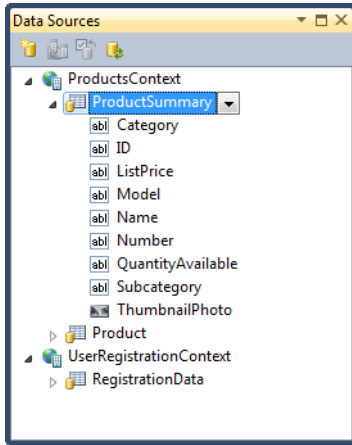


Figure 6-2. The Data Sources window

Note The icon next to the entity in the Data Sources window will indicate the type of control that will be created when the entity is dragged and dropped onto the design surface. You can change the type of control created by selecting the entity, clicking the drop-down button that appears (as you'll note in Figure 6-2), and selecting an alternative control type or layout format from the menu.

For the time being, ignore the control that will be created and bound for you. Drag the entity (in this case the `ProductSummary` entity) from the Data Sources window and drop it onto the design surface. This will result in the following XAML:

```
<riaControls:DomainDataSource AutoLoad="True"
    d:DesignData="{d:DesignInstance my1:ProductSummary,
        CreateList=true}"
    Height="0" Width="0"
    LoadedData="productSummaryDDS LoadedData"
    Name="productSummaryDDS"
    QueryName="GetProductSummaryListQuery">
    <riaControls:DomainDataSource.DomainContext>
        <my:ProductContext />
    </riaControls:DomainDataSource.DomainContext>
</riaControls:DomainDataSource>
```

Let's take a quick walkthrough of the important aspects of this XAML:

- The `AutoLoad` property is set to `True`, meaning that the call to the server will be made as soon as the view is loaded.
- An event handler is assigned to handle the `LoadedData` event. The event handler contains code to display a message box if an error occurs while attempting to retrieve the data. This will be discussed further in the “Handling Errors” section later in this chapter.
- The `QueryName` property is set to the name of the query domain operation that you want to call on the domain service. Note that we didn't need to select in the Data Sources window which method on the domain context should be used—instead, the Data Sources window was able to infer the correct one, as (in most cases) there will be only one query domain operation on a domain service that returns the selected entity. However, if you do happen to have two or more query domain operations on a domain service that return the entity (or collection of entities) that you want to bind to, you can select which domain operation you want to be called by selecting the entity in the Data Sources window, clicking the drop-down button, and selecting the required domain operation from the menu.
- The `DomainContext` property is assigned an instance of the domain context that will handle obtaining the entity collection from the server. The `my` namespace prefix is automatically defined for the namespace that the domain context can be found in if it isn't already defined.

These are the core requirements for configuring a `DomainDataSource` control, and once these properties are configured, your user interface controls can consume the data retrieved from the server by binding their `DataContext` or `ItemsSource` property to its `Data` property.

Code-Based Approach

Whereas a XAML-based approach using the `DomainDataSource` control implicitly implements a pulling strategy to populating the user interface with data, taking a code-based approach enables you to decide whether you push or pull data into the user interface.

Back in Chapter 3, we discussed how XAML's powerful binding capabilities enable the user interface to take a controlling role by pulling data into itself. Whereas a push approach involves the code-behind handling the request to the server and assigning the results once they are retrieved to a control property in the user interface (usually the `DataContext` or `ItemsSource` property of a control), a pull approach involves binding that control property in XAML to a property defined in either your code-behind class or on a separate class (where that class automatically requests data from the server and assigns the results to that property).

The pull approach is commonly used in conjunction with a `ViewModel` (part of the MVVM pattern), and requires knowledge of some advanced binding concepts that haven't been covered as yet in this book. I'll cover these binding concepts in Chapter 10. However, the techniques shown in this section to retrieve data from the server using a code-based approach are equally applicable to both strategies.

The following code demonstrates calling the `GetProductSummaryList` operation on our `ProductService` domain service. Start by adding the following using statement to the top of your class:

```
using System.ServiceModel.DomainServices.Client;
```

The next step is to instantiate the corresponding `ProductContext` domain context object, and request the `EntityQuery` object that represents the domain operation you want to call:

```
ProductContext context = new ProductContext();
EntityQuery<ProductSummary> qry = context.GetProductSummaryListQuery();
```

The `EntityQuery` object (`qry`) is then passed to the `Load` method on the domain context, which initiates the server request for the data:

```
LoadOperation<ProductSummary> operation = context.Load(qry);
```

Note that the `Load` method doesn't actually return you the data. Instead, it returns a `LoadOperation` object that contains an `Entities` property (consisting of a collection of the requested objects); however, this collection will currently be empty. As with calling standard WCF services in Silverlight, all calls to domain services using the RIA Services framework are asynchronous; therefore, the `Load` method will not wait for the data to be returned before returning control back to your code. The collection will remain empty until the data is returned from the server, at which stage the collection will be populated with the data.

The `LoadOperation` object has a `Completed` event, which it raises when the data has been retrieved from the server, or if an error has occurred that has prevented the successful retrieval of the data. Although you can wait to do something with the collection returned by the `Entities` property on the `LoadOperation` object until the `Completed` event is raised, there's nothing stopping you from immediately assigning the collection to the `DataContext` or `ItemsSource` property of a control, which will bind to and display the data in your user interface (assuming you choose to push the data into the user interface). The collection inherits from the generic `ObservableCollection` type (discussed earlier in this chapter), which raises an event when items are added or removed from the collection that can be handled by any bound user interface controls (hence enabling them to update themselves accordingly). Therefore, despite initially being currently empty, the collection will be automatically populated once the data is retrieved from the server, at which stage the user interface will know to update itself.

Putting all this together (using the default query to return all the products from the server) and assigning the results to the `ItemsSource` property of a `DataGrid` control called `productsDataGrid` results in the following code:

```
ProductContext context = new ProductContext();
EntityQuery<ProductSummary> qry = context.GetProductSummaryListQuery();
LoadOperation<ProductSummary> operation = context.Load(qry);
productsDataGrid.ItemsSource = operation.Entities;
```

This whole process may all seem an overly convoluted means of obtaining data from the server, and it is in a way, but it is actually quite an elegant way of both supporting custom queries and handling the asynchronous nature of server communication when requesting data from the server.

Choosing the Right Approach

With two approaches available to choose from, you may be wondering how to decide which approach you should use. This is a controversial topic that a number of developers have strong opinions about, and can lead to much debate.

Data source controls provide a quick-and-easy means to populate your user interface with data. Being a declarative data-pull mechanism, a data source control requires no code. Also, as was demonstrated, the Data Sources window makes configuring a data source control and a form to

consume data from the server even easier. The XAML-based approach is often used in presentations for this very reason; however, in practical applications, this approach has its drawbacks.

The biggest concerns come from a design/architectural perspective. The use of data source controls leads to a tightly coupled application, where you are mixing the user interface definition and data access logic. Any form of tight coupling in software is generally considered bad practice, and something to be avoided in your design. Data source controls also lead to a loss of control over the process of retrieving/saving data, and also make any issues you have in communicating with the server difficult to debug.

The XAML-based approach is also not compatible with the MVVM design pattern popularly used in Silverlight applications (discussed further in Chapter 12). Therefore, if you are following this design pattern, using the `DomainDataSource` control isn't really applicable in any case.

In summary, use caution when deciding to take a XAML-based approach and using the `DomainDataSource` control to declaratively consume data in your user interface—it may be quick and easy, but there may also be hidden dangers to deal with. It will provide an easy way to get started with RIA Services, but in practice you would be better off taking a code-based approach in conjunction with using the MVVM design.

Explicitly Specifying a Domain Service Host

Note that in neither approach did we explicitly specify the URI to the domain service, and you will find that there is no configuration file in the project where these can be configured (as you would find for referenced WCF services). Fortunately, in the standard scenario, where the site of origin for the Silverlight application also hosts the domain services, this is already taken care of for you. The domain context assumes that you will be communicating with a domain service from the site of origin (i.e., the web site that the Silverlight application was downloaded from), and automatically determines the correct address of the domain service to communicate with. This means that you don't need to worry about manually reconfiguring the domain service addresses each time you deploy the application to a different host—the client will always look to the server from which it was obtained for the services.

However, it is possible to override this behavior by explicitly specifying an alternate URI to locate the services. If you have a scenario where the application should communicate with domain services located on a different host from where your application was downloaded, you can specify the address of the corresponding domain service by passing in the new URI as a constructor parameter when instantiating the domain context. Assuming that any cross-domain issues have been taken care of with a client access policy file on that server (which will be discussed in Chapter 8), the domain context will communicate with the domain service at the specified URI instead. Note that unfortunately, specifying a domain service address is only possible with the code-based approach—this option is not available when using the `DomainDataSource` control.

Using the `BusyIndicator` Control

It's always a good idea to let the user know that the application is retrieving (or submitting) data. Usually, you can do this by providing an animation that runs while the communication is being completed. You can design your own animation or take advantage of the `BusyIndicator` control (a part of the Silverlight Toolkit), which provides a simple message and animation for you (as shown in Figure 6-3). This control (in source form) has already been included in your project by the Silverlight Business Application project template, and is implemented in the user login and registration functions.

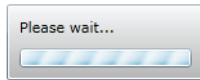


Figure 6-3. *The BusyIndicator control*

The key property on the `BusyIndicator` control is the `IsBusy` property. Setting this to `True` will show the animation, and setting it to `False` will hide the animation. If you decide to take the XAML-based approach to consuming the data and use the `DomainDataSource` control in your view, you can simply bind the `IsBusy` property of the `BusyIndicator` control to the `IsBusy` property of the `DomainDataSource` control (which will be set to `True` when the `DomainDataSource` control is waiting for the server request to complete) using `ElementName` binding. `ElementName` binding will be covered properly in Chapter 10, but in summary it binds the value of one control's property to the value of another control's property. When the property value is changed on the source control, the bound property's value will be changed accordingly. Here we want the `IsBusy` property of the `BusyIndicator` control to have the same value as the `IsBusy` property of the `DomainDataSource` control, so we bind the two control properties using the following binding syntax:

```
<controlsToolkit:BusyIndicator
    IsBusy="{Binding ElementName=productSummaryDDS, Path=IsBusy}" />
```

If you are taking a code-based approach instead, you can manually set the `IsBusy` property of the `BusyIndicator` control to `True` when you initiate the server request in your code, handle the `LoadOperation's Completed` event, and set the property value back to `False`. Alternatively, you can bind the `IsBusy` property of the `BusyIndicator` control to a property in your code-behind, and change that property's value instead.

■ **Note** You can determine whether the `DomainDataSource` control is loading data or submitting changes by its `IsLoadingData` and `IsSubmittingChanges` properties, which you can use to alter the message displayed in the `BusyIndicator` control accordingly.

By default, the text reads "Please wait . . ." but you can change that to something more meaningful by assigning the text you want displayed to the `BusyContent` property:

```
<controlsToolkit:BusyIndicator BusyContent="Retrieving products..." />
```

The `BusyContent` property is actually a content property, and can accept any control as its value (using the content element syntax, as discussed in Chapter 3). Therefore, this provides you with a simple means to customize the layout of the `BusyIndicator` control to your liking. For example, you may wish to add a button to cancel the communication, or add your own animation.

However, one thing may get in the way of you laying out the contents of the `BusyIndicator` control completely the way you want it: the animated bar. The animated bar is not affected by setting the content of the `BusyContent` property, and remains where it is. This bar is actually just a progress bar, and its style is exposed to you by the `ProgressBarStyle` property on the control. This means you can assign a

style to modify its properties, including its `Visibility` property. Therefore, you can set its value to `Collapsed` and hide it completely if you so wish:

```
<controlsToolkit:BusyIndicator.ProgressBarStyle>
  <Style TargetType="ProgressBar">
    <Setter Property="Visibility" Value="Collapsed"/>
  </Style>
</controlsToolkit:BusyIndicator.ProgressBarStyle>
```

■ **Note** Rather than just modifying the content of the `BusyIndicator` control, you can completely change its look (including changing the gray gradient background) by assigning it a new control template. Control templates are covered in Chapter 9.

In addition to displaying an animation to let the user know something is happening, the `BusyIndicator` control can also be used to automatically disable an area of your user interface while the server communication is taking place. The `BusyIndicator` control inherits from `ContentControl`, and therefore has a `Content` property that can contain other controls in the same way as the `Grid` or the `Canvas` controls can. These content controls (and any other controls within the area consumed by the control) will be disabled when the `IsBusy` property is set to `True`. It disables the controls by overlaying a semitransparent `Rectangle` control over its area so that they cannot be clicked. Note that if used in a `Grid`, the `BusyIndicator` control actually fills the area of its cell, and just shows the indicator in the middle. Hence the overlay will fill the entire grid cell, overlaying any other controls within that same cell. You can modify the properties of this `Rectangle` control via the `OverlayStyle` property on the `BusyIndicator` control.

■ **Note** Disabling user interface controls is not so important when retrieving data for summary lists, but can be useful when you have a data entry form to stop the user from attempting to resubmit or alter the changes they've made while communicating with the server.

To stop the `BusyIndicator` control from flashing on and off when a very short server call is made, or when multiple server calls are made in succession, you can control how long the control should be in the busy state before actually displaying the indicator to the user with the `DisplayAfter` property. By default it's set to 0.1 seconds, but you can change this as required.

Displaying the Summary List

Now that we have the mechanisms in place to actually consume the data in our view, it's time to display that data. The most commonly used controls to present a summary list to the user are the `DataGrid` control and the `ListBox` control, so we will investigate how to populate both these controls with data,

and compare and contrast the two so you can determine the most appropriate control to use for your purposes.

Retrieving the Data from the Server

Both the DataGrid control and the ListBox control essentially contain a list of items, and both have an ItemsSource property that can be used to populate the list. This actually means that populating both controls is done in the same way—by assigning a collection of objects to their ItemsSource property. How you do so depends on whether you have decided to take a XAML-based approach to consuming the data, or a code-based approach.

The XAML-Based Approach

If you have chosen a XAML-based approach, you would bind the ItemsSource property of the given control to the Data property of the DomainDataSource control using ElementName binding (as covered briefly in the “Using the BusyIndicator Control” section of this chapter, and will cover be fully in Chapter 10). For example, this XAML data binds the ItemsSource property of a ListBox control to the Data property of a DomainDataSource control called productSummaryDDS:

```
<ListBox ItemsSource="{Binding ElementName=productSummaryDDS, Path=Data}" />
```

The Code-Based Approach

If you have chosen to take a code-based approach, you can simply assign a collection to the control's ItemsSource property in code, like so (where operation is a LoadOperation object returned by a call to the Load method on a domain context object):

```
productsDataGrid.ItemsSource = operation.Entities;
```

Alternatively, you could use the more MVVM-friendly method of binding the ItemsSource property of the given control to a property in the ViewModel (or the code-behind).

■ **Note** As discussed in the “Data Binding” section of Chapter 3, if you have a scenario where you want to bind the ItemsSource property to the control's DataContext property (which may have been inherited down the control hierarchy), you can do so by assigning {Binding} to the control's ItemsSource property.

Configuring and Customizing a Control for Displaying Data

Although you populate the DataGrid and ListBox controls in the same manner, the ways you configure and customize them to display the data are quite different. Let's take a look at how to do so for each of these controls now.

Configuring and Customizing a DataGrid Control

Configuring a DataGrid control essentially involves setting up the columns to be displayed. If you happen to have presentation layer information (using the `Display` attribute) in the metadata of the entity being displayed, then simply assigning the collection to the `ItemsSource` property of the DataGrid and setting the `AutoGenerateColumns` property to `True` is all that is necessary. However, assuming that you don't have this metadata available (as having presentation layer information in the middle tier isn't generally recommended), you will need to manually configure the columns to be displayed. Let's look at how to produce the output shown in Figure 6-4 using a DataGrid control.

Name	Qty Available	Made In-House	
Adjustable Race	1085	☐	
All-Purpose Bike Stand	144	☐	
AWC Logo Cap	288	☐	
BB Ball Bearing	1352	☒	
Bearing BallE	1109	☐	
Bike Wash - Dissolver	36	☐	
Blade	1361	☒	
Cable Lock	252	☐	
Chain	589	☐	
Chain Stays	1629	☒	
Chainring	1684	☐	

Figure 6-4. A simple summary list displayed in a DataGrid control

■ **Note** If you want to use a DataGrid in your application, but the native DataGrid doesn't serve your needs, there a number of third-party DataGrids available that you could use instead, from vendors such as Telerik, ComponentOne, Infragistics, ComponentArt, DevExpress, and Syncfusion.

Configuring DataGrid Columns

Each column will need to be defined on the `Columns` property of the DataGrid.

There are three different column types available for the DataGrid control in Silverlight:

- **DataGridTextColumn:** This is the most commonly used column type in summary list scenarios. In read-only mode it shows the bound property value in a TextBlock, and in edit mode it shows it in a TextBox.
- **DataGridCheckBoxColumn:** This column type is used for showing Boolean values. It displays the bound property value in a CheckBox in both read-only and edit modes (it simply disables it when it's in read-only mode).
- **DataGridTemplateColumn:** This column type is for use when neither of the other two column types are appropriate. Essentially, you can configure this column to work the way you want by defining a cell template for the column (providing the CellTemplate property with a *data template*), and if you wish, an alternative cell-editing template (providing the CellEditingTemplate property with a data template). In other words, you can define two templates for a column—one for read-only scenarios, and one for editing scenarios. If only one of these templates is provided, then it will be used in both scenarios. In these cell templates you can define the control(s) that you want displayed (and their layout if required), providing practically limitless possibilities for the cell contents.

■ **Note** Data templates were defined and briefly described back in Chapter 3.

The following example XAML demonstrates a simple DataGrid configuration with three columns, each demonstrating one of the preceding column types. Each column has a binding (to a property on the entity), and assigns the text to be displayed in the column header. Note that both a cell template and an edit template are being defined for the quantityAvailableColumn column—using a TextBlock control to display the data for read-only scenarios, and a NumericUpDown control to enable the value to be altered in edit mode:

```
<sdk:DataGrid AutoGenerateColumns="False">
  <sdk:DataGrid.Columns>
    <sdk:DataGridTextColumn Binding="{Binding Path=Name}" Header="Name" />

    <sdk:DataGridTemplateColumn Header="Qty Available">
      <sdk:DataGridTemplateColumn.CellTemplate>
        <DataTemplate>
          <TextBlock Text="{Binding Path=QuantityAvailable}" />
        </DataTemplate>
      </sdk:DataGridTemplateColumn.CellTemplate>
      <sdk:DataGridTemplateColumn.CellEditingTemplate>
        <DataTemplate>
          <toolkit:NumericUpDown
            Value="{Binding Path=QuantityAvailable, Mode=TwoWay}" />
        </DataTemplate>
      </sdk:DataGridTemplateColumn.CellEditingTemplate>
    </sdk:DataGridTemplateColumn>
  </sdk:DataGrid.Columns>
</sdk:DataGrid>
```

```

        <sdk:DataGridCheckBoxColumn Binding="{Binding Path=MakeFlag}"
                                   Header="Made In-House" />
    </sdk:DataGrid.Columns>
</sdk:DataGrid>

```

■ It's worth noting that when you bind a collection to the `ItemsSource` property of a `DataGrid` control (or a `ListBox`), the `DataContext` property of each row/item in the `DataGrid/ListBox` will be assigned the corresponding entity from the bound collection (which you are then binding to).

Creating Column Definitions the Easy Way

There is a trick that can save you time when manually configuring the columns of a `DataGrid` control. Even if you have chosen not to take the XAML-based approach to consuming data in the view, use the method described to get started with this approach (ensure the `DataGrid` control is selected for the given entity in the Data Sources window, and then drag and drop it onto the design surface). This will automatically define a column in the `DataGrid`'s `Column` property for each property in the entity, and bind it to that property. You can then customize and reorganize/delete the columns that it defined for you to your heart's content, saving you from having to configure each of these manually.

In addition, this method can save you a lot of time when you are using template columns to display/edit various properties on the entity (where the control to display/edit the value isn't available as a part of a standard `DataGrid` column—i.e., `DataGridTextBoxColumn` or `DataGridCheckBoxColumn`). For example, you may want a numeric property to be able to be edited using a `NumericUpDown` control instead of a plain-old `TextBox` control. Select the control that should be used to edit the data for a property in the Data Sources window from the drop-down menu that appears when the property is selected. If the control doesn't appear in the list, click the `Customize` menu item and select the control from the list of all available controls. Remember to also select the data type that the control should appear for! Selecting this control type for a property will automatically create a template column for that property in the `DataGrid` that is created, with a preconfigured cell template containing that control bound to the property.

Now, if you aren't using the XAML-based approach, once you've dropped the entity onto the design surface, you can simply delete the `DomainDataSource` control that was created, and delete the binding that was created on the `ItemsSource` property of the `DataGrid`.

■ **Note** Often, you will want the last column in the `DataGrid` to simply fill the remaining space (especially in summary list-type scenarios). You can achieve this by simply setting the `width` property of the column to `*`.

Displaying an Image in a Column

We'd also like to display an image of each product against the corresponding item in the `DataGrid`. This is another scenario where a template column can be used, with a bound `Image` control in the cell template. However, properties that expose images on your entity will in most cases be represented by a simple byte array (`byte[]`). Unfortunately, the `Image` control cannot accept a byte array, and cannot be

bound to it. The Image control can accept a `BitmapImage`, so we need to find a way to convert the byte array to a `BitmapImage` so it can be bound. There are a number of ways to achieve this, including

- Defining a partial class for your entity, and creating a property that converts and exposes the byte array as a `BitmapImage` (which you can bind to)
- Creating your own image control that can accept an image as a byte array
- Creating a value converter that can be used as a part of a binding to convert the bound value before it is applied to the property of the control

All of these are viable solutions, and have their own pros and cons. As a general rule, using a value converter is the better option. Value converters are a powerful feature of the binding mechanism in Silverlight, enabling the properties of two objects to be bound together, even if their types don't match. The value converter sits in the middle of the binding, translating the value of the source property and returning the result so that it can be applied to the destination property (and vice versa). You simply design a class that implements the `IValueConverter` interface, and insert the logic to translate one value to another—in this case a byte array to a `BitmapImage`. Value converters will be covered in full in Chapter 10, so I won't go into details here—but in summary we'll use a value converter to convert the byte array to a `BitmapImage` (which the Image control can understand) so that it can display a thumbnail image for the product.

■ **Note** When it comes to exposing images from the `AdventureWorks2008` database, unfortunately we hit a bit of a hurdle. The images are stored in the database as GIFs—an image format not actually supported by Silverlight. Therefore, we need to either convert them to an image format Silverlight supports on the server before sending them to the client (such as PNG or JPG), or find a way to do so on the client. Converting images for each item on the server when the client requests a summary list would put an excessive load on the server and reduce the scalability of your application—therefore, it would be better to try to leave that task to the clients. Luckily, there is a CodePlex project called `ImageTools` for Silverlight (<http://imagetools.codeplex.com>) that provides a GIF decoder that you can use. This library is used in the sample project by calling it in our value converter (used when binding the image property to an Image control in XAML) to convert the byte array returned from the server to a `BitmapImage`. The GIF decoder from the library is used to decode the byte array from the GIF format, which is then reencoded using the PNG encoder from the library to a `BitmapImage` (which can then be bound to an Image control). This conversion process will be performed in the value converter, which will read in the byte array, convert the image to a PNG, and then assign it to a `BitmapImage` (which it will then return). However, I won't go into details of the conversion process here.

The following code demonstrates defining a template column containing an Image control whose `Source` property is bound to the `ThumbnailPhoto` property of the `ProductSummary` object. Note how the `Converter` property of the binding is assigned a custom value converter called `GifConverter` (which is defined as a resource higher up in the object hierarchy). This is the converter being used to handle the translation of a GIF image in a byte array to a PNG image in a `BitmapImage` object.

```

<sdk:DataGridTemplateColumn>
  <sdk:DataGridTemplateColumn.CellTemplate>
    <DataTemplate>
      <Image Margin="2"
        Source="{Binding ThumbnailPhoto,
          Converter={StaticResource GifConverter}}" />
    </DataTemplate>
  </sdk:DataGridTemplateColumn.CellTemplate>
</sdk:DataGridTemplateColumn>

```

Creating Calculated Columns

Note that you can use the same value converter technique to implement a calculated column. Bind the property to your whole object (so that the whole object is passed into the value converter), and then you can calculate a value to display in that column from the property values on the object (e.g., *total amount* = *quantity* × *price*, where *total amount* is the calculated value, and the *quantity* and *price* values are retrieved from the object). However, in this particular case you are probably best off defining a partial class for that object and creating a property to bind to that performs the calculation in that instead.

Editing Data in the DataGrid

The DataGrid control is primarily designed for editing data, and thus grid cells are in edit mode by default; however, you can disable the editing behavior to use the control more like a list by setting the `IsReadOnly` property of the DataGrid to `True`.

■ **Note** Being primarily designed for editing data, by default the DataGrid control uses a cell selection-type behavior, whereas in a summary list-type scenario, you really want the selection behavior to involve the full row. You'll find that when a cell is selected, the full row does have a selected appearance, but the cell has an additional (and quite prominent) selection square around it. There is no easy way to hide this, unfortunately, but you can use Silverlight's styling support to hide it yourself. This process involves applying a custom style to the `CellStyle` property of the DataGrid and retemplating the `DataGridCell` control to hide the `Rectangle` (named "FocusVisual") that it incorporates. Retemplating a control using a control template will be covered in Chapter 9.

Additional Built-In Behaviors

When you use the DataGrid, you get a lot of additional built in features and behaviors "for free," which is one of the primary reasons that the DataGrid control has such a wide appeal to many developers. Some of these features are listed in the following subsections.

Sorting

Simply click the header of a column to sort the DataGrid by its values, and click it again to reverse the sorting. This feature is automatically turned on, but can be turned off by setting the `CanUserSortColumns` property of the DataGrid to `False`. It can also be toggled off/on selectively by setting the appropriate value to the `CanUserSort` property of a column.

Grouping

One of the most powerful features of Silverlight's DataGrid control is its ability to display grouped data. Most summary lists in business applications have the need to group data, and the DataGrid control has the ability to display grouped data (as configured on a collection view) built in. This feature alone makes Silverlight's DataGrid more powerful and functional than that available natively in any other Microsoft platform. Multiple levels of grouping can be displayed, meaning that you can group items by category, and then subcategory, for example.

Note that grouping cannot actually be configured directly on the DataGrid control itself. Instead, you must bind it to a view and configure the grouping on it instead. This will be discussed further in the "Grouping the Summary List" section later in this chapter.

Resizing of Columns

Users can increase or decrease the size of a column by clicking and dragging the right edge of the column header to the size they want. They can also double-click the right edge of the column header to make the column automatically size itself to a width at which all its content is visible. This feature is automatically turned on, but can be turned off by setting the `CanUserResizeColumns` property of the DataGrid to `False`. It can also be toggled off/on selectively by setting the appropriate value to the `CanUserResize` property of a column.

Reordering of Columns

Users can rearrange the columns to their liking by dragging a column header to a new position. This feature is automatically turned on, but can be turned off by setting the `CanUserReorderColumns` property of the DataGrid to `False`. It can also be toggled off/on selectively by setting the appropriate value to the `CanUserReorder` property of a column.

Displaying Additional Row Details

The DataGrid has a `RowDetailsTemplate` property that enables you to define a data template containing a layout of additional details you want to display about a row beneath the main row. For example, you may want to display only some of the details on the entity in the columns, and then display more details when the row is selected. You can even have quite a complex layout where selecting a row displays another DataGrid below it showing a list of related items (e.g., the line items on an invoice). You can choose to have the row details displayed for each row automatically, only display the details for a row when it is selected, or only display them manually (via the `RowDetailsVisibilityMode` property on the DataGrid).

Validation Support

When the DataGrid is in edit mode, any validation rules you've defined in your entity's metadata will be run over the entered values, and a validation rule failure message will appear so the user knows what the problem is so that they can fix the issue.

DomainDataSource Integration

A big advantage of the DataGrid control is that it integrates particularly well with the DomainDataSource control (if you have decided to take a XAML-based approach to consuming the data). Often various data manipulations (covered later in this chapter) require the manipulation to be performed on the server and the list to be repopulated with the results. In collaboration with the DomainDataSource control, the DataGrid handles all of this for you, with no code required.

Configuring and Customizing a ListBox Control

One of the big advantages of using a ListBox control (over a DataGrid) is its flexibility. By default, a ListBox simply displays a line of text for each item in the list, as shown in Figure 6-5.

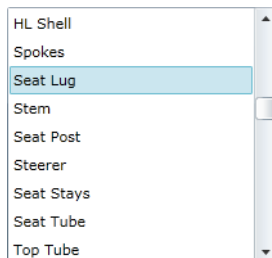


Figure 6-5. A ListBox control (with no customizations)

However, the ListBox control is actually quite a versatile control, with a lot more potential than you may initially assume. Let's take a look at the techniques you can use to customize the ListBox control to your needs.

Templating List Items

You can define a data template that will be applied to each item, enabling you to completely customize the appearance of items in the list. To do this, simply define a data template and assign it to the `ItemTemplate` property of the ListBox. For example, take the following data template:

```
<DataTemplate>
  <Grid Height="50">
    <Grid.ColumnDefinitions>
      <ColumnDefinition Width="100" />
      <ColumnDefinition />
    </Grid.ColumnDefinitions>
    <Grid.RowDefinitions>
      <RowDefinition />
      <RowDefinition />
    </Grid.RowDefinitions>

    <Image Margin="2"
      Source="{Binding ThumbnailPhoto,
        Converter={StaticResource GifConverter}}"
      Grid.RowSpan="2" />
    <TextBlock Name="NameField" Text="{Binding Name}" Margin="2"
      Grid.Row="0" Grid.Column="1" FontWeight="Bold" FontSize="12" />

    <StackPanel Orientation="Horizontal" Grid.Row="1" Grid.Column="1">
      <TextBlock Text="Number:" Margin="2" />
      <TextBlock Text="{Binding Number}" Margin="2" />
      <TextBlock Text="| Available:" Margin="2" />
      <TextBlock Text="{Binding QuantityAvailable}" Margin="2" />
    </StackPanel>
  </Grid>
</DataTemplate>
```

```

        <TextBlock Text="| Price:" Margin="2" />
        <TextBlock Text="{Binding ListPrice, StringFormat=C}" Margin="2" />
    </StackPanel>
</Grid>
</DataTemplate>

```

Applying this data template to the `ItemTemplate` property of the `ListBox` results in the output shown in Figure 6-6.



Figure 6-6. A `ListBox` control (with a custom item template)

Customizing the Layout of the Items

Items don't have to appear vertically in a list when using the `ListBox`. For example, you may choose to have items appear in a horizontal layout instead, or implement a thumbnail for each item in the list (as you find in Windows Explorer), or even display items in the form of contact cards. These effects can be achieved by assigning a new items panel template to the `ItemsPanel` property of the `ListBox`. This property enables you to customize how the items are laid out in the `ListBox`.

Let's take a look at how to implement the contact card-type layout. For this scenario we want the items to stack horizontally until they reach the right-hand edge of the `ListBox`, at which point they should wrap around to the next line. This is a scenario that the `WrapPanel` control from the Silverlight Toolkit is ideally suited for, as it was designed specifically to achieve this effect. All you need to do is assign this to the `ItemsPanelTemplate` property of the `ListBox`'s `ItemsPanel` property. For example, applying this template to the `ItemsPanel` property of the `ListBox`:

```

<ItemsPanelTemplate>
    <controlsToolkit:WrapPanel />
</ItemsPanelTemplate>

```

and then applying this data template to the `ItemTemplate` property:

```

<DataTemplate>
    <Grid Width="270" Margin="5">
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="100" />
            <ColumnDefinition />
        </Grid.ColumnDefinitions>

```

```

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

<Rectangle Stroke="Black" RadiusX="3" RadiusY="3"
  Grid.RowSpan="5" Grid.ColumnSpan="2">
  <Rectangle.Fill>
    <LinearGradientBrush StartPoint="0,0" EndPoint="0,1">
      <GradientStop Color="White" Offset="0" />
      <GradientStop Color="#6D6D6D" Offset="1" />
    </LinearGradientBrush>
  </Rectangle.Fill>
  <Rectangle.Effect>
    <DropShadowEffect ShadowDepth="2" />
  </Rectangle.Effect>
</Rectangle>

<Border BorderBrush="Black" Background="White" Margin="8" Height="70"
  VerticalAlignment="Top" CornerRadius="3" Grid.RowSpan="5">
  <Border.Effect>
    <DropShadowEffect ShadowDepth="2" />
  </Border.Effect>
  <Image Source="{Binding ThumbnailPhoto,
    Converter={StaticResource GifConverter}}"
    VerticalAlignment="Center" HorizontalAlignment="Center" />
</Border>

<TextBlock Name="NameField" Text="{Binding Name}" Margin="2,8,2,2"
  Grid.Row="0" Grid.Column="1" FontWeight="Bold" FontSize="12" />

<StackPanel Orientation="Horizontal" Grid.Row="1" Grid.Column="1">
  <TextBlock Text="Number:" Margin="2" />
  <TextBlock Text="{Binding Number}" Margin="2" />
</StackPanel>

<StackPanel Orientation="Horizontal" Grid.Row="2" Grid.Column="1">
  <TextBlock Text="Available:" Margin="2" />
  <TextBlock Text="{Binding QuantityAvailable}" Margin="2" />
</StackPanel>

<StackPanel Orientation="Horizontal" Grid.Row="3" Grid.Column="1">
  <TextBlock Text="Price:" Margin="2" />
  <TextBlock Text="{Binding ListPrice, StringFormat=C}" Margin="2" />
</StackPanel>
</Grid>
</DataTemplate>

```

and then setting the `ScrollViewer.HorizontalScrollBarVisibility` property on the `ListBox` to `Disabled` (disabling its horizontal scroll bar so that the items will wrap) results in the output shown in Figure 6-7.

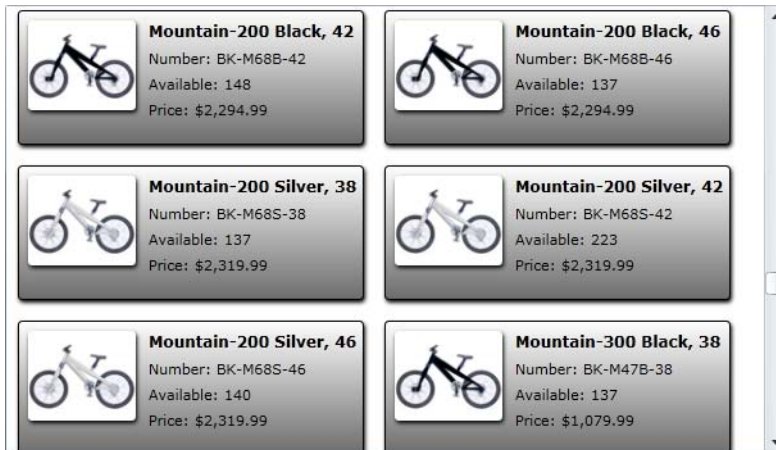


Figure 6-7. A `ListBox` control (with a custom item panel and item template)

Limitations of the `ListBox` Control

Despite all its flexibility, the `ListBox` control does have its limitations. One big issue comes when you want to group the items in the list (which is covered later in this chapter). Unlike the `DataGrid` control, the `ListBox` unfortunately doesn't have any built-in grouping functionality. Data can be grouped within the `ListBox`; however, there is no support for displaying group headers. You could attempt to implement group headers yourself, but this would require quite a bit of code and could become a messy solution. If you require grouping in your summary list, then you could potentially add some custom styling and behavior to the `TreeView` control (to make it look and behave like a list), as it is fundamentally designed to support hierarchical data (which grouped data could be considered to be). However, a fair amount of manual effort is still required to achieve the correct effect. Therefore, if you need to display grouped data beneath headers in your summary list, then the `DataGrid` control may be your best option.

■ **Note** The `ListBox` control has both an `Items` and an `ItemsSource` property. You can't assign a collection to the `Items` property (it is read-only), but you can add/remove items to/from it. The `ItemsSource` property enables you to assign an entire collection to it, and hence it can be bound to another control or source of data. You cannot bind the `Items` property, and hence the `ItemsSource` property is more commonly used. Also note that if you assign a value to the `ItemsSource` property, you cannot add or remove items from the collection exposed by the `Items` property.

Choosing the Right Control

With these two controls in Silverlight suitable for displaying summary lists, you are left with the potentially difficult decision of which one you should use.

The DataGrid is a powerful and versatile control, revered by developers but detested by most user experience designers. Developers tend to like to use the DataGrid, as it provides a lot of functionality out of the box, with little work required to enable it. Hence, developers treat it like a “god” control that can do everything and make their lives easier. By simply binding a collection to the DataGrid control, you get display and editing behavior, sorting, header resizing, header reordering, multilevel grouping, data validation, and more—all with little or no work required on the part of the developer. However, because of this, it is sometimes overused (e.g., implemented in scenarios where a ListBox would be a more appropriate choice), and often at the expense of the application’s user experience and user interface design.

■ **Note** Ease of implementation should not provide the basis for your decision on which control you should use. Your decision should solely be based upon what will provide the best experience for the user, and often you will find that the easiest-to-use control (for the developer) is not always the easiest to use (or understand) for the user. Keep this in mind when making your choice.

The manner in which the DataGrid has its columns spread out over its width (and beyond) is not always the most efficient way to display the data—especially when the columns extend beyond the width of the DataGrid (where horizontal scrolling would be required). It’s important to design your user interface based upon the needs of the user—not simply using what a control provides you.

As a general rule, the DataGrid control is best suited to data entry scenarios; but again, it should be selected with care for use in these scenarios too. Generally, you would use the DataGrid (as opposed to something like the DataForm control) for data entry purposes when the user needs to enter the details for a number of records associated with a parent record. For example, the DataGrid is very useful for enabling users to enter line items in an invoice, since there are typically only a few fields required for each invoice line item. However, for general data entry purposes, laying out controls in a form may be more appropriate. (This topic will be discussed further in Chapter 7.)

While the ListBox control is row driven by design, the DataGrid is primarily cell driven, which becomes an issue when using it to display a summary list. The purpose of a summary list is to locate and drill down on the data, and as a general rule the user will use it to drill down upon the data at the row level (not the cell level). Therefore, cell selection behavior in this scenario can be somewhat confusing to the user. If you do use the DataGrid for displaying summary lists, it is recommended that you change the default cell style to remove the selection rectangle from around the selected cell (hence, the cell won’t appear selected, but instead the whole row will, as it already exhibits highlighting behavior).

As discussed earlier, the DataGrid does have a big advantage over the ListBox (excluding data entry scenarios) when you need to implement grouping in a summary list, as the ListBox simply doesn’t have that capability. Implementing grouping in a DataGrid will be discussed later in this chapter in the “Grouping the Summary List” section.

One of the reasons why few developers tend to use the ListBox control is simply because they don’t realize the possibilities opened by templating its items; or if they do, then they think it’s too much work. You may have to design a data template first to lay out any data to be displayed that involves more than one field.

Ultimately, the most appropriate control for displaying summary lists is the `ListBox`, even if it does require a little more work to lay out and to implement various behaviors that are built into the `DataGrid` (such as sorting the list). It's much more lightweight than the `DataGrid`, and it's very flexible when templated. The `DataGrid` does have its place, but should be used only when the user experience design calls for it.

Manipulating Summary Lists

Displaying a summary list is one thing, but it only partly fulfills the objective of helping the user find the record(s) that they are after. This is particularly relevant when the list contains many items, especially if they aren't sorted in such a manner to help the user in this task. Therefore, summary lists typically provide a number of data manipulation actions that the user can apply to the list in order to browse it more efficiently, including

- Filtering
- Sorting
- Grouping
- Paging

In this section I will detail how to implement each of these data manipulation actions when you are using the `DomainDataSource` control to consume data, when you are working directly with the domain context object in code to consume data, and when you simply have the data in a collection on the client (using a `PagedCollectionView`).

■ **Note** Both the `DomainDataSource` and domain context methods detailed here request the data manipulation to be performed on the server, while the `PagedCollectionView` method uses a client-based means of manipulating the data. Performing the data manipulation on the server is extremely useful when you don't have (or don't want) the entire summary list returned to the client (such as when you want to page the list). Only perform the action on the server if you don't have all the data on the client, as otherwise you will be making the server do unnecessary work (by transferring the entire list back to the client again). When binding actions (filtering/sorting/grouping/paging) directly to the `DomainDataSource` control, you are asking it to perform those actions on the server—which is not the most efficient implementation if you are already transferring all the data to the client anyway. If you are already transferring the entire list to the client and then want to perform an action on it, handle the action manually using the `PagedCollectionView` method instead.

Querying Data Exposed by a Domain Operation

When the entire summary list is sent to the client, then filtering/sorting/grouping/paging can be performed on the client. However, when the client has only a part of the full list (such as when paging is

implemented), any filtering/sorting/grouping will need to be performed on the server. This is so that the given action can be applied to the entire list, as it will often affect which records the client will have.

Passing additional clauses to the server to be appended to the database query is generally a messy process to implement with standard web services. Often you need to add additional parameters to the query operations to accept the clauses, and then manually append these clauses to the main query to be run before returning the results to the client.

This is where the ability for a domain operation to return an `IQueryable` expression (enabled by using RIA Services in conjunction with the Entity Framework) comes in handy. This enables us to easily query the exposed collection from the client without needing to manually accept and apply the additional clauses that the client is requesting to the main database query. Instead, RIA Services automatically handles transferring any query clauses requested by the client and applies them to the `IQueryable` expression returned by the domain operation on the server. Assuming the `IQueryable` expression was in the form of a LINQ to Entities query, these additional clauses will be appended to that query and included when it is actually executed against the database. Nothing is required on your part on the server to enable this feature, except to return an `IQueryable` expression—making it one of the most powerful benefits of using RIA Services (you can also find a similar feature in WCF Data Services).

■ **Note** The ability to specify additional query clauses from the client does not allow the client to circumvent any filters you may have applied at the server on your query (e.g., if you've applied a `Where` clause to filter out data that the user does not have permission to access). This feature *does not* open you up to SQL injection type attacks in any way.

From the client side, the `DomainDataSource` control contains the functionality to enable each of the data manipulation actions listed earlier to be applied to any data that it is configured to consume. This is achieved by defining descriptors on the `DomainDataSource` for these actions. These actions can be defined either declaratively at design time (in XAML) or at runtime (in code). This makes performing those actions easy when using a XAML-based approach to consuming data. (Implementing these actions will be detailed throughout this section.)

However, if you are using a code-based approach to consuming data from the server, then this client-side functionality needs to be implemented manually. Let's take a look at how you can add clauses to a query domain operation call when using the code-based approach.

Until now, the code-based approach examples have simply accepted the entire collection returned by the domain operation. However, the `EntityQueryable` object provides you with a number of extension methods enabling you to append clauses to be added to the query to be run on the server. The methods it gives you are

- `OrderBy`: Orders the results, with the records sorted by the given property in ascending order
- `OrderByDescending`: Orders the results, with the records sorted by the given property in descending order
- `Select`: Does nothing; only empty selections are accepted
- `Skip`: Usually used when paging data to skip a given number of records before "taking" a given number

- **Take:** Usually used to limit the maximum number of records to be returned, or when paging data to return a given number of records after skipping a given number
- **ThenBy:** Adds an additional clause to order the records, with the records sorted by the given property in ascending order
- **ThenByDescending:** Adds an additional clause to order the records, with the records sorted by the given property in descending order
- **Where:** Used when filtering to restrict the records being returned based upon given criteria

■ **Note** There is no `GroupBy` method on the `EntityQueryable` object. Grouping in the context of returning data to display is actually a type of data sorting, so you would use the `OrderBy` and `ThenBy` methods instead.

Using the code from a previous example:

```
ProductContext context = new ProductContext();
EntityQuery<ProductSummary> qry = context.GetProductSummaryListQuery();
LoadOperation<ProductSummary> operation = context.Load(qry);
productsDataGrid.ItemsSource = operation.Entities;
```

let's add a simple `Where` clause to the query to limit the results being returned from the server. This process is the same as appending additional clauses to a LINQ to Entities query (using lambda expressions). You simply add additional clauses to the `EntityQuery` object returned from the query method on the domain context before passing it to the `Load` method:

```
ProductContext context = new ProductContext();
EntityQuery<ProductSummary> qry = context.GetProductSummaryListQuery();
qry = qry.Where(p => p.QuantityAvailable == 0);
LoadOperation<ProductSummary> operation = context.Load(qry);
operation.Completed += new EventHandler(operation Completed);
```

■ If you don't see the query methods on the `EntityQuery` object, make sure you have a `using` statement in your code file referencing `System.ServiceModel.DomainServices.Client`.

Various combinations of these clauses can be appended one after the other in a fluent manner using the standard query operators and lambda expressions to achieve the results that you require. For example:

```
qry = qry.Where(p => p.QuantityAvailable == 0).OrderBy(p => p.Name);
```

Alternatively, rather than using this fluent syntax, you can write a declarative query expression against the query method, like so:

```
EntityQuery<ProductSummary> qry = from p in context.GetProductSummaryListQuery()
                                   where p.QuantityAvailable == 0
                                   select p;
```

Now when you run this query from the client, the clauses you added will be automatically applied to the query on the server and included in the SQL query to the database that returns the results.

■ **Note** You can easily prove that a query has been built using the clauses provided by the client and run against your database on the server (and see the query that was generated) using SQL Profiler. Simply monitor SQL queries against your database and make a request for data (with additional clauses applied) from the client—the resulting SQL statement will be displayed in the profiler. If you don't have the full version of SQL Server, you can use a free tool with similar functionality to the SQL Profiler called SQL Server Express Profiler, created by AnjLab. You can download this tool from here: <http://sites.google.com/site/sqlprofiler>.

Filtering the Summary List

Let's enable the user to filter the summary list by product name (as shown in Figure 6-8). Add a TextBox control to the view into which the user can enter some text to filter the list, and set its name to SearchTextBox:

```
<TextBox Name="SearchTextBox" Width="100" />
```

We'll implement this feature such that as the user types the text, the list will automatically filter the list's contents, displaying only the products whose name contains the text that the user has entered into the text box.

Filter:

Name	Qty Available	Made In-House
Chain	589	☐
Chain Stays	1629	☒
Chainring	1684	☐
Chainring Bolts	1136	☐
Chainring Nut	1750	☐

Figure 6-8. A summary list with a filter

DomainDataSource Method

The `FilterDescriptors` property on the `DomainDataSource` control enables you to add a new `FilterDescriptor` object specifying the property to filter the list by (`PropertyPath`), bind to the `Text` property of a `TextBox` containing the search text (`Value`), and provide an operator for how the comparison will be performed (`Operator`).

```
<riaControls:DomainDataSource AutoLoad="True" Height="0" Width="0"
                             LoadedData="productSummaryDDS LoadedData"
                             Name="productSummaryDDS"
                             QueryName="GetProductSummaryListQuery">
  <riaControls:DomainDataSource.DomainContext>
    <my:ProductContext />
  </riaControls:DomainDataSource.DomainContext>

  <riaControls:DomainDataSource.FilterDescriptors>
    <riaControls:FilterDescriptor PropertyPath="Name"
                                Operator="Contains"
                                Value="{Binding ElementName=SearchTextBox, Path=Text}" />
  </riaControls:DomainDataSource.FilterDescriptors>
</riaControls:DomainDataSource>
```

Alternatively, you can add filter descriptors in the code-behind like so:

```
FilterDescriptor descriptor = new FilterDescriptor("Name", FilterOperator.Contains,
                                                "Road");
productSummaryDDS.FilterDescriptors.Add(descriptor);
```

■ **Note** In this example we are using the `Contains` operator, such that the query will match on any part of the product name, with the user not needing to know the entire name of the product in order to get a match. This is great for the user, but doesn't allow the database engine on the server to make use of the column index (if it has one) to perform the search. If you find that your database is struggling, you may be better off using the `StartsWith` operator instead, which will be able to use the index. However, this will only return products whose name starts with the given search text.

Domain Context Method

Simply add an event handler to the `TextBox`'s `TextChanged` event (which will be raised whenever the user changes the text in the `TextBox`), and in the event handler add the code that queries the server, adding a `Where` clause to the default query like so:

```
private void SearchTextBox_TextChanged(object sender, TextChangedEventArgs e)
{
    EntityQuery<ProductSummary> qry = context.GetProductSummaryListQuery();
```

```
// Add a Where clause to filter the results
qry = qry.Where(p => p.Name.Contains(SearchTextBox.Text));

LoadOperation<ProductSummary> operation = context.Load(qry);
operation.Completed += new EventHandler(operation Completed);
}
```

■ **Note** We aren't creating a new instance of the domain context here. When implementing filtering in this manner, you should declare your domain context at the class level and use it instead of instantiating a new one in the TextBox's TextChanged event. Otherwise, each time the user presses a key in the TextBox, a new domain context instance will be created, and an additional call to the server will be made—potentially resulting in multiple queries being active with the server simultaneously. This way, if an existing call is in process, it will be cancelled as the next one is made, leaving only one call in progress at a time.

PagedCollectionView Method

To filter the items in a PagedCollectionView, you can simply apply a lambda expression to its Filter property. For example, use the following code to limit the view to only items that contain “Road” in their name:

```
PagedCollectionView view = new PagedCollectionView(op.Entities);
view.Filter = p => ((ProductSummary)p).Name.Contains("Road");
```

■ **Note** You have to cast the function argument (p) to the type of object contained in the collection, as the PagedCollectionView collection is not restricted to your object type (i.e., any type of object can be added to the collection), so the argument being passed in (p) is of type object, not ProductSummary.

You can also build a more complex filter by simply appending additional conditions to the lambda expression.

Sorting the Summary List

If you are using the DataGrid to display the summary list, note that it contains its own action to enable the user to initiate a sorting action (i.e., clicking a column header). It also sorts its contents accordingly with no code required, including automatically refreshing its data from the server when bound to a DomainDataSource control. If you are using a different control to display the data (such as a ListBox), you will need to implement your own mechanism to enable the user to specify how they want the list to be sorted. Let's take a look at how to sort the list (regardless of how the user will initiate the sort).

DomainDataSource Method

The `SortDescriptors` property on the `DomainDataSource` control enables you to add a new `SortDescriptor` object specifying the property to sort the list by (`PropertyPath`) and the direction to sort by (`Direction`):

```
<riaControls:DomainDataSource AutoLoad="True" Height="0" Width="0"
                             LoadedData="productSummaryDDS LoadedData"
                             Name="productSummaryDDS"
                             QueryName="GetProductSummaryListQuery">
  <riaControls:DomainDataSource.DomainContext>
    <my:ProductContext />
  </riaControls:DomainDataSource.DomainContext>

  <riaControls:DomainDataSource.SortDescriptors>
    <riaControls:SortDescriptor PropertyPath="Name" Direction="Ascending" />
  </riaControls:DomainDataSource.SortDescriptors>
</riaControls:DomainDataSource>
```

Alternatively, you can add sort descriptors to a `DomainDataSource` control programmatically like so:

```
SortDescriptor descriptor = new SortDescriptor("Name", ListSortDirection.Ascending);
productSummaryDDS.SortDescriptors.Add(descriptor);
```

■ **Note** If you bind a `DataGrid` to the `DomainDataSource` control, whenever the user clicks a column header in the `DataGrid` it will automatically update the sort descriptors on the `DomainDataSource` control to sort by that column and requery the server for the reordered list. The reason why it requires the server to be requered is that if the results are paged, then the results to be displayed in the page will be (more than likely) different to the current set of items. Therefore, the sort operation must be performed on the server against the entire set of results, before returning the new results for the current page.

Domain Context Method

To sort the summary list when using the domain context method, simply add an `OrderBy` or `OrderByDescending` clause to the default query like so:

```
ProductContext context = new ProductContext();
EntityQuery<ProductSummary> qry = context.GetProductSummaryListQuery();

// Order the list by the product name
qry = qry.OrderBy(p => p.Name);

LoadOperation<ProductSummary> operation = context.Load(qry);
operation.Completed += new EventHandler(operation Completed);
```

You can sort by multiple properties if you wish, simply by appending additional `ThenBy/ThenByDescending` clauses to the query:

```
qry = qry.OrderBy(p => p.Model).ThenBy(p => p.Name);
```

If you are using the domain context method of loading a sorted collection from the server and populating a `DataGrid` control with the results, when the user clicks a column header it will only automatically perform client-side sorting of the list. However, if you have paged results and need the sorting to be done on the server, you will have to capture this event yourself and requery the server (and then repopulate the `DataGrid` with the results).

That said, you will find that there is no event on the `DataGrid` that enables you to catch this action. Instead, you need to

1. Wrap the collection in a `PagedCollectionView`
2. Assign this view to the `DataGrid`'s `ItemsSource` property
3. Handle the view collection's `CollectionChanged` event (so you are handling the event on the source collection, not the `DataGrid`)

The `CollectionChanged` event will be raised when the user clicks a column header in the `DataGrid` (the `Action` passed in on the event arguments object will have a value of `Reset`), and in the event handler you can requery the server—being sure to manually apply the same sort descriptions that are now on the `PagedCollectionView` collection to the `EntityQuery` object that you are passing to the domain context's `Load` method).

PagedCollectionView Method

To sort the items in a `PagedCollectionView`, simply instantiate a `SortDescription` (passing it the property to sort the collection by and a sort direction), and then add it to the `PagedCollectionView`'s `SortDescriptions` collection property. For example, use the following code to sort the collection by the item's `Name` property (in ascending order):

```
PagedCollectionView view = new PagedCollectionView(productCollection);
SortDescription sortBy = new SortDescription("Name", ListSortDirection.Ascending);
view.SortDescriptions.Add(sortBy);
```

Multiple sort descriptors can be added as required. If you bind the view collection to a `DataGrid`, clicking a column header in the `DataGrid` will automatically update the sort descriptions in this underlying collection view accordingly.

Grouping the Summary List

Grouping is a feature often requested in business applications. It essentially enables you to group related data together and display that data under a heading in the list. Often this extends beyond a single level of grouping to two or even more levels. As discussed earlier in this chapter, the `DataGrid` has built-in grouping functionality (as shown in Figure 6-9), but the `ListBox` has none at all. Data can be sorted in the `ListBox`, but not under headers separating record groups, so the `ListBox` isn't really applicable in this scenario.

Name	Qty Available	Made In-House
Model: ML Mountain Tire (1 item)		
ML Mountain Tire	669	☐
Model: ML Road Frame (5 items)		
ML Road Frame - Red, 44		☒
ML Road Frame - Red, 48		☒
ML Road Frame - Red, 52		☒
ML Road Frame - Red, 58		☒
ML Road Frame - Red, 60		☒
Model: ML Road Frame-W (5 items)		
ML Road Frame-W - Yellow, 40		☒
ML Road Frame-W - Yellow, 42		☒
ML Road Frame-W - Yellow, 44		☒

Figure 6-9. Grouped items in a DataGrid control

DomainDataSource Method

The GroupDescriptors property on the DomainDataSource control enables you to add a new GroupDescriptor object specifying the property to group the list by (PropertyPath):

```
<riaControls:DomainDataSource AutoLoad="True" Height="0" Width="0"
    LoadedData="productSummaryDS LoadedData"
    Name="productSummaryDDS"
    QueryName="GetProductSummaryListQuery">
  <riaControls:DomainDataSource.DomainContext>
    <my:ProductContext />
  </riaControls:DomainDataSource.DomainContext>

  <riaControls:DomainDataSource.GroupDescriptors>
    <riaControls:GroupDescriptor PropertyPath="Model" />
  </riaControls:DomainDataSource.GroupDescriptors>
</riaControls:DomainDataSource>
```

You can create multiple levels of groupings by adding additional GroupDescriptor objects to the DomainDataSource control's GroupDescriptors property as required. For example:

```
<riaControls:DomainDataSource.GroupDescriptors>
  <riaControls:GroupDescriptor PropertyPath="Model" />
  <riaControls:GroupDescriptor PropertyPath="Category" />
  <riaControls:GroupDescriptor PropertyPath="Subcategory" />
</riaControls:DomainDataSource.GroupDescriptors>
```

Alternatively, you can add group descriptors to a `DomainDataSource` control programmatically like so:

```
GroupDescriptor descriptor = new GroupDescriptor("Model");
productSummaryDDS.GroupDescriptors.Add(descriptor);
```

Domain Context Method

As mentioned earlier, grouping is primarily a presentation issue—the underlying collection only needs to be sorted in order to support it. Strangely, however, despite the fact that the `DataGrid` has built-in support for grouping similar rows under headers, there is no property on the `DataGrid` itself to configure the grouping. However, it can be configured by using a `PagedCollectionView` (as will be demonstrated in the next section) on the data once it has been retrieved from the server. Once the data is returned from the server, instantiate a `PagedCollectionView` (passing in the collection to the constructor) and configure the grouping on that in the handler for the `Completed` event of the `LoadOperation` (returned from the `Load` method on the domain context).

If the data being returned from the server is being paged, however, you will need to sort the data appropriately on the server so that the items will be grouped list-wide (rather than just within the page itself). Therefore, you will need to sort the data by the properties to be grouped first, and then add any additional sorting criteria for how you want the data to be sorted within these groups. For example, if you wanted to group products by category and subcategory and sort them by name, you would actually add sort criteria for these grouped properties to your server query like so:

```
ProductContext context = new ProductContext();
EntityQuery<ProductSummary> qry = context.GetProductSummaryListQuery();

// Order the list by the product category, subcategory, and name
qry = qry.OrderBy(p => p.Category)
          .ThenBy(p => p.Subcategory)
          .ThenBy(p => p.Name);

LoadOperation<ProductSummary> operation = context.Load(qry);
operation.Completed += new EventHandler(operation Completed);
```

■ **Note** The sort clauses for the properties to group by have precedence over those defining how the items are sorted within those groups.

Then you can use the `PagedCollectionView` to specify the grouping that should be applied in the `DataGrid` once the results have been returned from the server.

PagedCollectionView Method

To group the items in a `PagedCollectionView`, simply instantiate a `PropertyGroupDescription` (passing it the property to group the collection by), and then add it to the `PagedCollectionView`'s `GroupDescriptions`

collection property. For example, use the following code to group the collection by the item's Category property:

```
PagedCollectionView view = new PagedCollectionView(productCollection);
PropertyGroupDescription groupBy = new PropertyGroupDescription("Category");
view.GroupDescriptions.Add(groupBy);
```

Multiple group descriptors can be added as required.

Note that you can also supply a value converter as a parameter to the constructor of the `PropertyGroupDescription` object to convert the property value before it is grouped. This is particularly useful in scenarios where you want to group by a property that is a foreign key to data located elsewhere (such as in another collection). For example, say you want to group the products by category, but the product object only contains an ID representing the category, not its name. You could group it by ID, but the ID would appear in the group header instead of the category name. You would generally also want the groups sorted alphabetically by category name rather than category ID. This feature enables you to write a value converter that converts the category ID into a category name (assuming you have a collection of category names and their corresponding IDs on the client), and use the output value of this converter as the value to display as the item's group.

■ **Note** Value converters will be discussed in Chapter 10.

Paging the Summary List

When you have hundreds or even thousands of records on the server, generally you don't want to transfer all of them to the client. Not only does it overwhelm and confuse the user, but it also consumes excessive network traffic and requires the user to wait for it all to be downloaded from the server (making your application appear slow). If you have (or may have in the future) a large number of records returned from the server, then paging the list will help solve these issues. Paging data essentially returns only a small subset of the full list, and with each additional "page" that the user wants to view, the application will query the server again to request the data for that page. Alternatively, if all the data is located on the client, you can page the list to simply avoid overwhelming the user with a huge list.

To help implement this feature, Silverlight has a `DataPager` control (as shown in Figure 6-10) that keeps track of the current page number, and provides an interface to enable the user to navigate backward and forward through the pages. Declaring a `DataPager` control in XAML is as simple as

```
<sdk:DataPager PageSize="30" />
```

and binding its `Source` property to the collection to be paged. Note that we set the `PageSize` property to the number of items/rows to appear in each page. The control handles the rest.

The `DataPager` supports different display modes (with options such as a page number button for each page, first and last buttons plus page number buttons, first/previous/next/last buttons, etc.), which you can specify using its `DisplayMode` property. It also has an `IsTotalItemCountFixed` property that if set to `true` will stop the user from navigating beyond the calculated number of pages (by disabling the `Next` button). If set to `false`, it will keep the `Next` button enabled even when the last page is reached (useful when the list may have an uncertain or rapidly changing size).

■ **Note** When querying the server for paged data, the data must be sorted (with an `OrderBy` clause in the query) in order to be paged. Otherwise, you will get an error message stating that “The method ‘Skip’ is only supported for sorted input in LINQ to Entities” once you attempt to load any pages after the first.

Filter:

Page 23 of 51

Name	Qty Available	Made In-House
▲ Model: Full-Finger Gloves (3 items)		
Full-Finger Gloves, S	72	☐
Full-Finger Gloves, M	108	☐
Full-Finger Gloves, L	144	☐
▲ Model: Half-Finger Gloves (3 items)		
Half-Finger Gloves, S	324	☐
Half-Finger Gloves, M	0	☐
Half-Finger Gloves, L	36	☐
▲ Model: Headlights - Dual-Beam (1 item)		
Headlights - Dual-Beam	180	☐
▲ Model: Headlights - Weatherproof (1 item)		
Headlights - Weatherproof	216	☐

Figure 6-10. A summary list with an associated `DataPager` control

DomainDataSource Method

The `DomainDataSource` integrates very nicely with the `DataPager` control to provide server-based paging to the summary list. In fact, paging isn't even configured on the `DomainDataSource` control itself—instead you simply bind the `Source` property of the `DataPager` control to the `DomainDataSource` control's `Data` property and set its `PageSize` property, and the `DomainDataSource` control will automatically implement the paging behavior accordingly.

```
<sdk:DataPager PageSize="30"
    Source="{Binding Data, ElementName=productSummaryDDS}" />
```

The `DataPager` provides you with another feature when used with the `DomainDataSource` control—setting a different number of items to be retrieved from the server than what's actually displayed in the list. For example, if users are regularly navigating through pages, then it can be useful to retrieve more

data from the server in a single call than is displayed in a single page. For example, you may only want to display 20 items/rows to the user at a time, but load 5 pages worth (100 items/rows) to save the user from waiting for the server to return the data for each page as they navigate through the list. The (optional) `LoadSize` property on the `DomainDataSource` control enables you to set the number of items/rows that should be retrieved from the server at a time. The `PageSize` property will continue to define how many items/rows are presented to the user at the one time on the client, with the additional items/rows remaining cached in the background. The `DomainDataSource` control will then automatically go back to the server to retrieve the next set of results when the user attempts to view a page whose items have not been retrieved yet.

Domain Context Method

When breaking a list up, paging is implemented by taking the whole list, ignoring all the items/rows for the previous pages (i.e., skipping them), and then returning the following number of items/rows until the page size has been reached. Let's say you have 100 items/rows in your list and a page size of 10 items/rows. Getting the first page is easy—you simply take the first 10 items/rows from the list, and ignore the rest. When getting subsequent pages, however, you need to skip as many items/rows as the page size multiplied by the page number, take the number of items/rows up to your page size, and then ignore the rest.

Say you have a `DataPager` control in your view named `ListPager`. To page the summary list when using the domain context method, you will need to add a `Skip` clause and then a `Take` clause to the default query, like so:

```
ProductContext context = new ProductContext();
EntityQuery<ProductSummary> qry = context.GetProductSummaryListQuery();

// Get the items/rows for the given page
qry = qry.OrderBy(p => p.Name)
    .Skip(ListPager.PageSize * ListPager.PageIndex)
    .Take(ListPager.PageSize);

qry.IncludeTotalCount = true;

LoadOperation<ProductSummary> operation = context.Load(qry);
operation.Completed += new EventHandler(operation Completed);
```

Note that we are supplying an `OrderBy` clause (which as mentioned earlier is mandatory in order to enable paging—although it can be done either on the client or the server). Also note that we are setting the `IncludeTotalCount` property of the `EntityQuery` object to `true`. In order to show the user how many pages of items/rows there are in the list, we need to know the total number of items/rows that there are (since we only have a small subset of the full list). This number can then be used to calculate how many pages there will be. Setting the `IncludeTotalCount` property of the `EntityQuery` object to `true` means that the server will return the total count of items/rows in the list, along with the page of items/rows requested.

The total entity count can be obtained from the `LoadOperation` object passed into its `Completed` event handler:

```
LoadOperation<ProductSummary> op = sender as LoadOperation<ProductSummary>;
int itemCount = op.TotalEntityCount;
```

The problem that you will encounter is that the `DataPager` control's `ItemCount` property (which this value needs to be assigned to) is actually read-only. It needs this value so it can calculate the total number of pages, but it obtains the value from a property on the source that it binds to (if that source implements the `IPagedCollectionView` interface). The `DomainDataSource` control returns a `DomainDataSourceView`, which implements this interface and enables the correct paging behavior:

- Handling requerying the server as the user navigates between pages
- Caching the paged results so that once retrieved they don't need to be requested from the server again
- Exposing the total number of items from its `ItemCount` property so that the `DataPager` control can handle the page navigation (even though it doesn't have all the items currently available client side)

However, this is all done within the `DomainDataSource` control—the domain context (which we are using directly) simply returns a collection (instead of a view), and the `DomainDataSource` control wraps that collection in a `DomainDataSourceView` itself.

Unfortunately, we can't make use of the `DomainDataSourceView` ourselves, as its constructor scope is marked as `internal` (so we can't create an instance of it). You can't use the `PagedCollectionView` either, as its `ItemCount` property is read-only, too. Instead, you will have to create your own view that implements from the `IPagedCollectionView` interface, and implement the same behavior provided by the `DomainDataSourceView` yourself.

You will then be able to assign the collection returned from the server to the `ItemsSource` property of the `DataGrid/ListBox`, and assign the same collection to the `Source` property of the `DataPager`.

PagedCollectionView Method

Paging the items/rows in a `PagedCollectionView` requires no work on your part when using the `DataPager` control. Simply assign the view to both your `DataGrid/ListBox` and your `DataPager`, and the `DataPager` will manage what items should appear in the list.

If you aren't using the `DataPager` control, note that the `PagedCollectionView` has a `PageSize` and a `PageIndex` property, along with the following methods to navigate between pages:

- `MoveToFirstPage`
- `MoveToLastPage`
- `MoveToNextPage`
- `MoveToPreviousPage`
- `MoveToPage`

Drilling Down on a Record

Summary lists are generally used as a means to find and drill down on a record to get more details and/or edit the record. There are a number of methods you can use to implement this type of behavior, including

- Opening the details in a new view
- Opening the details in a pop-up window
- Using the row details feature of the DataGrid to display the details
- Using a master/details view

Which method you choose really depends on the scenario you are implementing. Each has its own pros and cons, and your choice will largely depend upon the user experience design for your application. In this section we will take a look at each of these methods and how to implement them.

Opening Details in a New View

How to implement this behavior isn't particularly straightforward, however. You have probably implemented this type of behavior in the past by having the user double-click an item in the list, but there are no `DoubleClick` events in Silverlight. In fact, there's no `RowClick` event on the `DataGrid` either, nor is there an `ItemClick` event on the `ListBox`. So how can you enable the user to drill down on a record? Let's take a look at some approaches you might like to take.

■ **Note** If you still want to use a double-click behavior, you can implement it yourself using a behavior or a trigger. You can also find various double-click behaviors/triggers already created by other developers with a quick search on the Web. Behaviors will be covered in Chapter 10.

If you are using a `DataGrid`, then you can use a template column and add a control such as a `Button` or `HyperlinkButton` to it that the user can click to navigate to the details view. Alternatively, if you are using a `ListBox`, then you can simply include one of these controls in your item template.

Unfortunately (as was detailed in Chapter 4), the navigation framework doesn't allow you to pass complex objects between views. Therefore, in order to enable the details view to know which record it should load and display, you will need to pass a unique identifier to that view as a query string parameter. You could use the `HyperlinkButton` control (which can be configured to automatically navigate to the details view with no code required); however, it unfortunately does not have the ability to automatically include the unique identifier of the record as a parameter in the URI to navigate to the details view. Your options in this case are to either handle the `HyperlinkButton`'s `Click` event in the code-behind and build the URI to navigate to manually prior to initiating the navigation, or to bind the `NavigateUri` property of the `HyperlinkButton` to the property representing the unique identifier on the bound object and use a value converter to build the full URI. The value converter approach is probably the most appropriate choice (especially if you are using the MVVM design pattern); however, it does not apply if you are using a `Button` control (as the `Button` control has no `NavigateUri` property). Therefore, we'll focus on the code-behind approach here instead, which you can use with either a `HyperlinkButton` control or a `Button` control.

The following XAML demonstrates the definition of a `HyperlinkButton` control (which can be added to the cell template of a `DataGrid` template column or to a `ListBox` item template) that displays the name of the product as its text, has a `Click` event handler, and binds the whole `ProductSummary` object to its `Tag` property (which it does by assigning `{Binding}` to its property value):

```
<HyperlinkButton Content="{Binding Name}" Tag="{Binding}"
Click="NameButton_Click" />
```

Now, in the Click event handler for the HyperlinkButton, we can get the HyperlinkButton that raised the event (which is passed into the event handler as the sender parameter), and then get the ProductSummary object corresponding to its summary list item/row from its Tag property. The structure of this URI will depend on how your URI mapping has been configured (see Chapter 4 for more information on URI mapping). The following example demonstrates building a URI that navigates to the product details view, and passes it the ID of the product to display (which it can obtain from the query string and retrieve the corresponding details from the server for).

```
private void NameButton_Click(object sender, System.Windows.RoutedEventArgs e)
{
    HyperlinkButton button = sender as HyperlinkButton;
    ProductSummary productSummary = button.Tag as ProductSummary;

    Uri detailsUri = new Uri("ProductDetails/" + productSummary.ID.ToString(),
                            UriKind.Relative);
    NavigationService.Navigate(detailsUri);
}
```

■ **Note** You should also handle the KeyDown event of the control to capture the Enter key being pressed, and navigating to the details view corresponding to the selected item/row. This helps enable users to use the keyboard to navigate around the application instead of having to switch between using the keyboard and the mouse.

Opening Details in a Pop-Up Window

Silverlight has a neat class built in called ChildWindow that enables you to display a modal pop-up window with content that you define. This makes it perfect for use in scenarios in which you want the user to be able to select a record in a summary list, and display the related details in a pop-up window.

To implement a child window, add a new item to your project and select the Silverlight Child Window item template from the dialog. This will create a new XAML file (and corresponding code-behind class) inheriting from the ChildWindow class. As shown in Figure 6-11, you have a nice window-type layout (which you can style to suit your user interface design if necessary) already set up with OK/Cancel buttons, which you can add your own control layout to so that the user can view and edit the details of the selected record in the summary list.

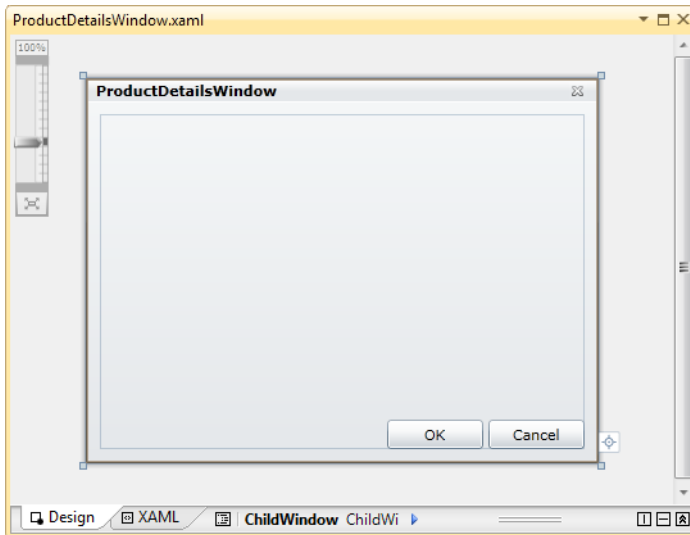


Figure 6-11. A ChildWindow in design mode

Once you have defined the layout of the child window, you now need to be able to display it when the user clicks a record in the summary list. Using the methods described in the previous section, instead of navigating to a new view you can simply handle the Click event of the hyperlink or button defined for an item/row and display the child window. All that is required to display a child window is to instantiate the class and call the Show method, like so:

```
ProductDetailsWindow window = new ProductDetailsWindow();  
window.Show();
```

■ **Note** While the child window is displayed modally, unlike displaying modal windows in Windows Forms or WPF, the method returns immediately (instead of waiting for the window to be closed). If you need to perform an action (such as updating the summary list) once the window is closed, you will need to handle either the Closing or Closed event of the child window and put your logic to do so in there.

However, in this scenario, where we want to show the details of the selected record in the summary list, we need to pass in an identifier or an object so it knows what data to load and display. Unlike the previous method of navigating to a new view where we could only pass a simple string to the new view to tell it which record to load, we can actually pass complex objects into a child window by modifying the constructor of the class to accept a simple type or object with the required details as a parameter. For example, change the constructor to

```

public ProductDetailsWindow(ProductSummary productSummary)
{
    InitializeComponent();

    // Logic to load product data corresponding to
    // the passed-in productSummary object can go here
}

```

You can now pass it the selected object in the summary list when instantiating the window, like so:

```

HyperlinkButton button = sender as HyperlinkButton;
ProductSummary productSummary = button.Tag as ProductSummary;

ProductDetailsWindow window = new ProductDetailsWindow(productSummary);
window.Show();

```

Figure 6-12 demonstrates displaying the details of a product in a child window.

Figure 6-12. Product details in a child window

■ **Note** The `ChildWindow` class only supports being displayed modally, but the source code for this class can be found in the Silverlight Toolkit, so you can modify its behavior by modifying the source code if you wish. The modality of the child window is provided by applying a semitransparent `Rectangle` control overlaying the application's user interface, positioning the child window control on top of this, and disabling the root visual. Therefore, any mouse clicks outside the child window will be ignored, effectively providing a modal behavior. With slight modifications to the code you can enable multiple child windows to be displayed simultaneously. Microsoft Silverlight program manager Tim Heuer has actually already done this with a control called the `FloatableWindow`, which includes additional behavior such as the ability for it to be resized by the user. It is available on CodePlex here: <http://floatablewindow.codeplex.com>.

Displaying Details Using the DataGrid's Row Details

The row details feature in the `DataGrid` enables you to display additional data related to a row in the `DataGrid`—useful when you only want to show limited details from the bound object in the main row, and display additional details in an area below it (as demonstrated in Figure 6-13). This is especially useful when you want to show additional details about a row without displaying them in a pop-up window or navigating away from the summary list view.

Name	Number
Road-150 Red, 48	BK-R93R-48
Road-150 Red, 52	BK-R93R-52
 Model: Road-150 Available: 128 Price: \$3,578.27	
Road-150 Red, 56	BK-R93R-56
Mountain-100 Silver, 38	BK-M82S-38
Mountain-100 Silver, 42	BK-M82S-42
Mountain-100 Silver, 44	BK-M82S-44

Figure 6-13. Row details in a `DataGrid` control

This is achieved by assigning a data template defining the layout of the row details area to the `RowDetailsTemplate` property of the `DataGrid`. For example:

```
<sdk:DataGrid>
  <sdk:DataGrid.RowDetailsTemplate>
    <DataTemplate>
      <!-- Row details layout goes here -->
    </DataTemplate>
  </sdk:DataGrid.RowDetailsTemplate>
</sdk:DataGrid>
```

You can assign a value to the `RowDetailsVisibilityMode` property to specify whether the row details are displayed only when the row is selected (`VisibleWhenSelected`), displayed for every row in the `DataGrid` (`Visible`), or never displayed (`Collapsed`). You can use the `RowDetailsVisibilityChanged` event to handle when the row details are shown/hidden, enabling you to implement behaviors such as loading related data from the server into the row details when the row is selected and the row details section is displayed.

By default, if the `DataGrid` has columns extending beyond the width of the `DataGrid`, its horizontal scrolling functionality will also allow horizontal scrolling of the row details area. However, you can freeze the row details such that they remain fixed in position regardless of any horizontal scrolling motion by setting the `AreRowDetailsFrozen` property to `True`.

Implementing a Master/Details View

The master/details view is a relatively common user interface pattern in which the summary list (acting as the master) occupies one part of the view, and when the user selects an item from the list, the associated records (acting as the details) are displayed in another part of the view. For example, you may have a list of invoices (i.e., the master), and when the user selects an invoice from this list, the line items associated with the selected invoice (i.e., the details) will be displayed in another list.

Alternatively, you may have a list of records that when selected will immediately populate a data entry form in another part of the view with the details of that record for editing. For example, let's say you have a list of products. When the user selects a product from the list, they can then edit that record in a data entry form in another part of the view (modifying the same object as being displayed in the list). This is actually quite easy to implement, thanks to Silverlight's binding functionality, and no code is actually required. Let's say you are using a `DomainDataSource` control in your view that retrieves the list of products (as has already been demonstrated) and displays them in a list (using either a `DataGrid` or a `ListBox` bound to the `DomainDataSource` control). Since building data entry forms won't be discussed until the next chapter, we'll simply bind a `TextBox` in a manner to display the name of the selected product in the list and enable you to edit it. Add a `TextBlock` control to your view and bind its `DataContext` property to the `DomainDataSource`'s `Data` property (as the `DataGrid`/`ListBox` does). Now bind its `Text` property to the `Name` property of the bound object.

```
<TextBlock DataContext="{Binding ElementName=productSummaryDDS, Path=Data}"
    Text="{Binding Name}" />
```

Now run your project. When you select a product in the list, note how the text in the `TextBlock` automatically changes to display the corresponding name of that product. Considering that the `TextBlock` isn't even bound to the list, how does it know what item has been selected? This is because the `DomainDataSource` control returns a `DomainDataSourceView` from its `Data` property (which the `TextBox` control is binding to). In addition to filtering/sorting/grouping functions (which it enables without the underlying collection needing to be modified), this view also contains current record management functions (including properties such as `CurrentItem` and `CurrentPosition`, and methods such as `MoveCurrentToNext` and `MoveCurrentToPosition`). This current record view can then be used by other associated data bindings.

Handling Errors

Because of the asynchronous nature of calls to the server in Silverlight, handling errors that are raised when a call to the server fails is a somewhat different process than you may be used to. For example, if

you are using the code-based approach to consuming the data, you will find that putting try/catch blocks around a call to the server are of little use, as those calls immediately return control back to your code while the call happens on a background thread. Therefore, any errors that are raised on the server or in communicating with the server will not be caught by these exception handlers.

To actually catch these errors, you need to handle the event that is normally raised when the server communication is complete. When using the XAML-based approach, you will need to handle the `DomainDataSource` control's `LoadedData` event, and when using the code-based approach, you will need to handle the `LoadOperation`'s `Completed` event.

As mentioned earlier in the chapter, if you drag and drop an entity from the Data Sources window onto the design surface, the `DomainDataSource` control that is created will already handle the `LoadedData` event and check whether an exception occurred:

```
private void productSummaryDDS_LoadedData(object sender, LoadedDataEventArgs e)
{
    if (e.HasError)
    {
        System.Windows.MessageBox.Show(e.Error.ToString(), "Load Error",
                                         System.Windows.MessageBoxButton.OK);
        e.MarkErrorAsHandled();
    }
}
```

■ **Note** Simply dropping a `DomainDataSource` control from the toolbox will not automatically create an event handler to handle its `LoadedData` event. Only using the method of dropping an entity from the Data Sources window onto the design surface will do so.

If you are using the code-based approach, handling the `LoadOperation`'s `Completed` event is very similar to handling the `DomainDataSource`'s `LoadedData` event, except you need to cast the sender parameter to a generic `LoadOperation` object:

```
private void loadOperation_Completed(object sender, EventArgs e)
{
    LoadOperation<ProductSummary> op = sender as LoadOperation<ProductSummary>;

    if (op.HasError)
    {
        System.Windows.MessageBox.Show(op.Error.ToString(), "Load Error",
                                         System.Windows.MessageBoxButton.OK);
        op.MarkErrorAsHandled();
    }
}
```

Note how in both cases we are calling the `MarkErrorAsHandled` method. If you don't call this method, the domain context will throw the exception (which will become an unhandled exception). Unhandled exceptions are caught by the `Application.UnhandledException` event handler method in the `App` class

and handled there (showing a pop-up error window displaying the details of the error) in order to prevent your application from crashing.

You can check the exception to find out what type of exception occurred and handle it accordingly. The exception is returned simply as an `Exception` type, but you can cast it to a `DomainOperationException` to get more useful information about it. Of most interest is the `Status` property that it exposes. This will tell you what type of exception occurred (using the `OperationErrorStatus` enumeration) so you can handle it accordingly. The values in this enumeration of most interest when loading data are

- **ServerError:** An exception was raised on the server, or the application could not reach the server.
- **Unauthorized:** The user does not have permission to perform the operation. Restricting access to domain operations based upon the user's role is discussed in Chapter 8.

Therefore, you can check this value to determine the category of the error, and handle it accordingly. The following example demonstrates a structure for handling the errors in the `LoadedData` event handler of the `DomainDataSource` control (change `e` to the instance of the `LoadOperation` if using the domain context method):

```
if (e.HasError)
{
    DomainOperationException error = e.Error as DomainOperationException;

    switch (error.Status)
    {
        case OperationErrorStatus.ServerError:
            // Handle server errors
            break;
        case OperationErrorStatus.Unauthorized:
            // Handle unauthorized domain operation access
            break;
    }
}
```

■ **Note** Alternatively, if you are using the code-based approach but don't want an exception to be thrown when an error occurs, use one of the overloads of the domain context's `Load` method that accepts the `throwOnError` parameter, and set it to `false`.

You can test your exception handler by throwing an exception in your query domain operation on the domain context. For example, add the following code to the domain operation method (before returning the query results) to see the effect of that exception in your Silverlight application:

```
throw new Exception("Text exception handling exception!");
```

Summary

You should now have the ability to retrieve data from the server (using RIA Services), and the knowledge to choose the most appropriate approach to take in doing so. You should also be able to apply this knowledge to retrieve and display a summary list, and enable users to manipulate it to suit their requirements and drill down upon a record. In the next chapter, we'll look at enabling users to edit the details of a record and submit the changes back to the server.

CHAPTER 7



Building Data Entry Forms

Now that you've learned how to retrieve data from the server and display it in the form of a read-only list, you need to learn how to display the data in a form such that the user can modify it and submit the changes back to the server. This chapter will take you through creating a data entry form that enables the user to add, edit, and delete records from the system, and submit the changes back to the server.

Creating the Data Entry User Interface

A typical data entry form consists of a number of editable fields—such as text boxes, drop-down lists, and check boxes, each with a corresponding label displaying the name of that field, and a tab order assigned such that users can tab between fields in a logical fashion. Ideally, the users should be able to enter data into the form and submit it without ever having to use the mouse (i.e., completely keyboard driven), making the data entry process smoother and more efficient. Most developers have created countless data entry forms in their careers, although not all fulfill these requirements. Let's take a look at how to create a data entry form that achieves these goals in Silverlight, and then move on to refine its functionality.

Laying Out the Data Entry Form

In the past, you may have been used to laying out data entry forms using fixed positioning, with each control assigned a Left and a Top value. Although you can do this in Silverlight (by laying out the controls on a canvas), it's usually better to create a flexible layout using the Grid control and lay out the controls in the Grid's cells. This helps you keep the controls aligned, and enables you to have the controls dynamically resize according to the size of the application's window.

There are actually a number of different methods you can use to create a data entry form. Although you can lay out the form yourself (manually), there are a couple of time-saving means available to help you create a data entry form from the object that it will be bound to. Let's take a look at those now.

Using the Data Sources Window

As shown in the previous chapter, you can open the Data Sources tool window in Visual Studio 2010, and drag an entity that is exposed from a domain service from this window onto your design surface. A DataGrid will be created in the view, with a column defined for each property on the entity.

Instead of creating a DataGrid, you can select Details from the drop-down menu when you select the entity in the Data Sources window (as shown in Figure 7-1) before dragging it onto the design surface.

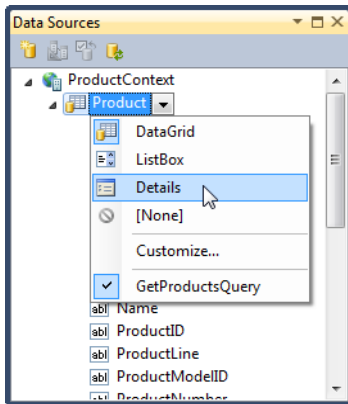


Figure 7-1. Selecting what control will be generated for an entity

Now when you drag the entity onto the design surface, a field and corresponding label will be created for each property on the entity, laid out using a Grid control (as shown in Figure 7-2), with the field's binding to the source property already configured.

 The image shows a 'Details' form for the 'Product' entity. It consists of a grid with labels on the left and input controls on the right. The labels are: 'Class:', 'Color:', 'Days To Manufacture:', 'Discontinued Date:', 'Finished Goods Flag:', 'List Price:', 'Make Flag:', 'Modified Date:', 'Name:', 'Product ID:', and 'Product Line:'. The corresponding controls are: empty text boxes for 'Class:', 'Color:', 'List Price:', 'Name:', 'Product ID:', and 'Product Line:'. A date picker control is used for 'Discontinued Date:' and 'Modified Date:', showing a date of '15'. Checkboxes are used for 'Finished Goods Flag:' and 'Make Flag:'.

Figure 7-2. Details form generated by the Data Sources window

The control created for each property will default to the one best suited to the property's type (i.e., TextBox for strings, CheckBox for Booleans, DatePicker for dates, etc.), but you can change these as required by expanding the entity in the Data Sources window and selecting different control types for the properties that you want to change (as shown in Figure 7-3). The labels will contain the name of the property, but with some spaces intelligently inserted into the name, and a colon appended.

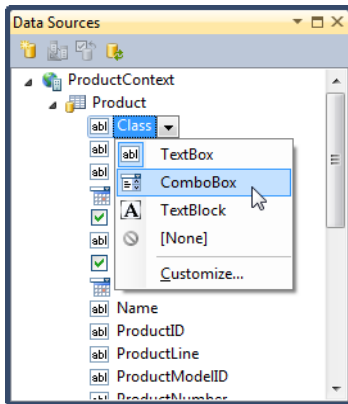


Figure 7-3. Selecting the control that will be generated for a property on the entity

■ **Note** Unfortunately, there is no way to order the properties of an entity in the Data Sources window before dragging it onto the view. This means that the fields will be created on the form in alphabetical order rather than the actual logical order that you want for them. Therefore, you will need to reorder the fields once they have been created in the form, assigning the fields to the actual grid row in which you want them to appear.

A `DomainDataSource` control will be automatically configured for you (if one doesn't already exist in the view), and the `DataContext` property of the Grid control that is created will be bound to it (and will be inherited and available to all the controls within the Grid).

This is an easy way of creating the fields required for a data entry form (you will generally need to rearrange them to form a logical layout); however, it's not the only (or best) way, as you'll soon find.

Using XAML Power Toys

Another method you can use to create a data entry form is to use the XAML Power Toys add-in for Visual Studio 2010, created by Karl Shifflet. After you download and install this add-in (from <http://karlshifflett.wordpress.com/xaml-power-toys>), it will add a new submenu to your right-click context menus in the XAML and code editors (as shown in Figure 7-4).

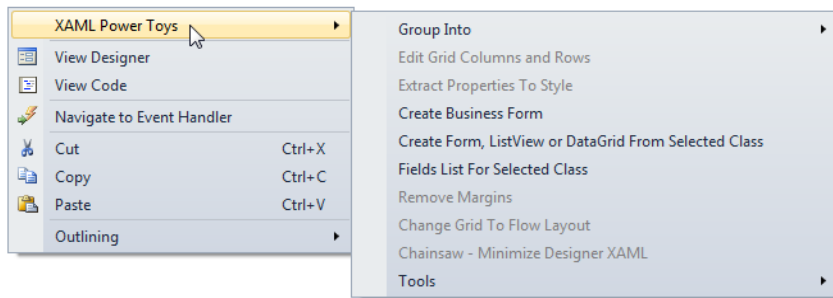


Figure 7-4. The XAML Power Toys menu

■ **Note** The context menu doesn't appear in the right-click context menu in the XAML designer view—it only appears in the XAML editor view.

Right-click in the XAML editor, and select the Create Form, ListView, or DataGrid From Selected Class menu item from the XAML Power Toys submenu.

From the dialog that appears, select the object that you want to create the data entry form from (i.e., the object that the form will bind to), and click the Next button. The screen shown in Figure 7-5 will be displayed.

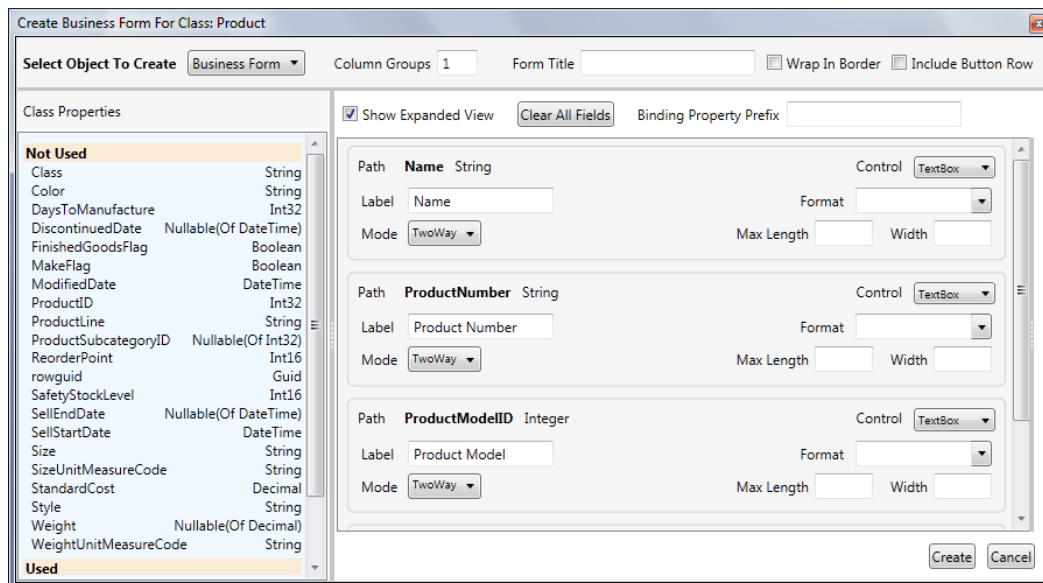


Figure 7-5. The XAML Power Toys add-in's Create Business Form For Class dialog

Start by ensuring that the Select Object To Create drop-down box at the top left of the dialog is set to Business Form. Now you can drag and drop properties that you want to create as fields in the form from the Class Properties section on the left side of the screen into the right side. You can then set any of the given properties for that field as required, including

- The type of control
- Its display format, maximum length, and width
- Its binding mode (which needs to be set to `TwoWay` in order for the updated value to be propagated back to the bound object, as discussed back in Chapter 3)
- Its label

Note how the tool splits the property name into separate words for the label by inserting a space before each capital letter in the property name, which (assuming the text to be displayed is similar to the property name) should save you time in renaming many of the labels. You can also reorder the fields if required by simply clicking and dragging the item up and down to a new position in the list.

There are numerous other features that I won't go into here, but essentially this method enables you to have a lot more control over the configuration of the data entry form before creating it than the previous method described.

Once you've completed configuring your form, you can click the Create button. However, this won't actually insert anything into your XAML as yet. The XAML that was generated has been copied to the clipboard and is ready for you to paste in the appropriate location in your XAML file. Place the cursor at the position in the XAML file to insert the form, and paste the contents of the clipboard to that location (press `Ctrl+V`). Your data entry form has now been created, and you can modify the XAML further as required. Note that you may have to declare various namespace prefixes used in the generated XAML—these are included when you paste the XAML into the XAML editor in commented-out form. You can then add them to your root element if they aren't already declared.

You can also configure other default property values to apply to each control in the form by selecting the XAML Power Toys ► Tools ► Set Control Defaults menu item from the right-click context menu.

■ **Note** There are a number of other useful features included in XAML Power Toys that you will probably find useful when working with XAML (in both Silverlight and WPF). The source code is also available for download, enabling you to make modifications and add additional features that will help you create data entry forms faster.

Using the DataForm Control

In addition to the previous two methods shown, there is another (and arguably better) method to create a data entry form, and that's using the `DataForm` control.

Creating a DataForm Control with Automatically Generated Fields

The `DataForm` control is a part of the Silverlight Toolkit, and provides a couple of different ways to create data entry forms. The first method is simply to let it do all the work in creating the data entry form for you at runtime. All you need to do is to drop a `DataForm` control onto the design surface, and bind

the object for the user to edit to its `CurrentItem` property (or a collection of objects to its `ItemsSource` property if you want the user to be able to edit multiple objects). For example, the XAML to create a `DataForm` and bind it to a `DomainDataSource` control (which retrieves the collection to be modified from the server) is

```
<riaControls:DomainDataSource AutoLoad="True" QueryName="GetProducts"
                             LoadedData="productDomainDataSource LoadedData"
                             Name="productDomainDataSource" Height="0" Width="0">
    <riaControls:DomainDataSource.DomainContext>
        <my:ProductContext />
    </riaControls:DomainDataSource.DomainContext>
</riaControls:DomainDataSource>

<toolkit:DataForm Header="Product Data Entry"
    ItemsSource="{Binding ElementName=productDomainDataSource, Path=Data}" />
```

The drag-and-drop approach will declare the toolkit namespace prefix for you if it hasn't already been defined:

```
xmlns:toolkit="http://schemas.microsoft.com/winfx/2006/xaml/presentation/toolkit"
```

This will generate a data entry form similar to that shown in Figure 7-6.

Figure 7-6. A `DataForm` control

■ **Note** This data entry form is generated at runtime, so none of these fields will appear at design time unless you bind it to sample data (as discussed in Chapter 10).

Note that the fields are created in alphabetical order (of bound property name) rather than the required logical order, using the property names as their labels. In addition, a field will be created for *all* the properties on the object, and these fields won't necessarily generate the control that you want to be used for editing the property value.

This automatically generated form doesn't create a particularly satisfactory user experience in its current form, but there is a way you can have more control over the process and customize it to your needs.

In Chapter 5, you learned how to decorate the properties of your entities/objects with attributes—either directly or via metadata classes. You can use the `Display` attribute that was described to control the generation of the corresponding fields in the `DataForm` control. For example:

```
[Display(Name="Product Number:", Order=3)]
public string ProductNumber;
```

When the corresponding field for the `ProductNumber` property is generated in the data entry form, its label will have the text that is assigned to the `Name` property of the `Display` attribute, and will appear third in the list of fields, based on the value assigned to the `Order` property. Other useful properties of the `Display` attribute that you can use to control the field generation include `AutoGenerateField` (which can be used to stop a field from being generated for the property) and `Description` (which can be used to assign a tooltip description to the field, as will be described shortly). In scenarios where your application will be localized for different languages, instead of hard-coding these strings you can use the `ResourceType` property of the `Display` attribute and assign it the type created for the resources file (`.resx`) where the resources can be found. Then assign the name of the string resources to use to their corresponding properties, like so:

```
[Display(ResourceType = typeof(ProductResources), Name="ProductNumberLabel",
        Description="ProductNumberDescription")]
public string ProductNumber;
```

■ **Note** You will need to add a link in your Silverlight project to the resource file in your web project so the resources can be used on the client side. If you are using the RIA Services class libraries for your middle tier, then the resource file should be linked between the server and client-side class library projects instead.

Now when the fields are generated at runtime in the `DataForm`, they will use this metadata and be laid out accordingly. However, there are some issues with generating the data entry form in this manner. The first is the still-limited control you have over how the data entry form is generated. For simple forms, this won't be too much of a problem; however, for more complex forms where you need to use other controls such as `ComboBoxes`, assign custom properties to each field control, or have less of a structured layout (e.g., with multiple fields in a single row), then this method really isn't satisfactory. Some of these issues can be worked around by handling the `DataForm` control's `AutoGeneratingField` event (which is raised when each field is being created, enabling you to make changes to the field being generated).

The most important issue, however, is really conceptual. Decorating your entities/objects with presentation-related attributes violates the separation-of-concerns principle. How the data entry form is laid out is a presentation tier/layer issue, not a middle tier/layer issue (where the entities/objects exist). By decorating your entities/objects with presentation-related attributes, you are creating a murky boundary between the two tiers/layers, which is against good design principles. Therefore, it's highly

recommended that you avoid using this method when creating your data entry forms, and explicitly lay out the data entry form using the method described next instead.

■ **Note** If you are binding the `DataForm` control to a `ViewModel` (a concept from the MVVM design pattern, which will be discussed in Chapter 12) and decorate its properties with the `Display` attribute instead, then the use of the `Display` attribute is more acceptable (with a `ViewModel` being a presentation-related object). However, even in this case it's recommended that you explicitly lay out the data entry form using the method that will now be described.

Creating a `DataForm` Control with Explicitly Defined Fields

Instead of having the fields in the data entry form dynamically generated at runtime as per the previous method, you can instead explicitly define the fields within the `DataForm` control in XAML instead. You'll then have complete control over the `DataForm`'s fields, and will be able to lay out the fields exactly as required. However, what then can the `DataForm` control offer you over the previous two methods?

- The ability to display a icon (next to the field) that will show a description for that field in a tooltip when the user hovers their mouse over it. This uses the `DescriptionViewer` control from the Silverlight Toolkit (and can be used outside of the `DataForm` control if you wish).
- The ability to navigate through each entity (if bound to a collection) without needing to leave the view or implement another control (such as a `List` or a `DataGrid`) to select the current item. In the header of the control are navigation buttons that you can use for this purpose. In conjunction with RIA Services, the user can modify various entities and add entities to the collection. Also, when the user sends the changes back to the server (as will be discussed later in this chapter), only the entities that have changed or been added server will be sent.
- The ability to add a new item or delete the current item if bound to a collection.
- Support for objects that implement the `IEditableObject` interface (discussed later in this chapter) to enable changes to an entity to be committed, or cancelled with the bound properties reverted to their original values.
- A summary of all the validation errors on the bound entity automatically displayed at the bottom of the control.
- The `DataForm` control already containing a `ScrollViewer`, so that if the fields extend past the area of the data entry form, then a scroll bar will automatically be displayed.
- The ability to define a different data template for each state that the `DataForm` control can be in. There are three data templates that you can define:
 - `ReadOnlyTemplate`: Used when the `DataForm` control is in read-only mode. This will be the default mode when the `DataForm` control is displayed and its

AutoEdit property is set to False, and it will also be used whenever the DataForm control's IsReadOnly property is set to True.

- **EditTemplate:** Used when the DataForm control is in edit mode. This will be the mode when the DataForm control's AutoEdit property is set to True (and the user hasn't clicked the Edit button for the current item), and when its IsReadOnly property is set to False.
- **NewItemTemplate:** Used when adding a new item to the bound collection.

You might notice some similarities between the structure of the DataForm control and the FormView/DetailsView controls in ASP.NET (with different templates for different modes, etc.). With these controls in ASP.NET, you could simply bind them to an ObjectDataSource control at design time, and the control would automatically generate the HTML for all the fields for you.

Unfortunately, the DataForm control does not have this same feature. The XAML for the fields in a DataForm control is a little different than what we used earlier, and there is also unfortunately no option in the Data Sources window to create the XAML for a data entry form from an object using the DataForm control. However, the XAML Power Toys add-in does have this ability, so you can use the method described in the previous section to create a data entry form, but with one difference. Instead of selecting Business Form from the Select Object To Create drop-down box at the top of the dialog, select Silverlight Data Form. This will create the XAML required for a data entry form using a DataForm control (which you can then bind to a data source such as the DomainDataSource control), and will save you a lot of time over building it manually.

■ **Note** You'll also need to declare the toolkit namespace prefix when using this method (as defined earlier in this section).

Configuring the Labels

As mentioned, the XAML for the fields in a DataForm control needs to conform to the DataForm's requirements, so let's take a look at how it is structured. We'll take a look at a very simple example of the XAML required to configure two fields in the EditTemplate of a DataForm control:

```
<toolkit:DataForm AutoGenerateFields="False" Header="Product Data Entry">
  <toolkit:DataForm.EditTemplate>
    <DataTemplate>
      <Grid>
        <Grid.RowDefinitions>
          <RowDefinition Height="Auto" />
          <RowDefinition Height="Auto" />
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
          <ColumnDefinition />
        </Grid.ColumnDefinitions>

        <toolkit:DataField Grid.Row="0" Grid.Column="0"
          Label="Name:" LabelPosition="Auto">
          <TextBox Text="{Binding Path=Name, Mode=TwoWay}"
```

```

        Name="NameTextBox"
        HorizontalAlignment="Stretch" VerticalAlignment="Top" />
    </toolkit:DataField>

    <toolkit:DataField Grid.Row="1" Grid.Column="0"
        Label="Product Number:"
        LabelPosition="Auto">
        <TextBox Text="{Binding Path=ProductNumber, Mode=TwoWay}"
            HorizontalAlignment="Stretch" VerticalAlignment="Top" />
    </toolkit:DataField>
</Grid>
</DataTemplate>
</toolkit:DataForm.EditTemplate>
</toolkit:DataForm>

```

Note the following from this XAML:

- The fields are laid out in a Grid, but there is only a single column defined in the Grid (yet the DataForm control ensures that they will remain aligned regardless of the differing lengths of the labels).
- There is no Label or TextBlock control defined for each field to display the field label. Instead, each field is defined using a DataField control, to which you assign the input control as its content. The DataField control is then assigned to a cell in the Grid, and the text to use as its label is assigned to its Label property. If you wish, you can assign a string to the DataField control's Description property to display an icon next to the field that shows the string when the user hovers their mouse over it.

When the DataForm control in the preceding XAML is bound to an entity or collection of entities, the output shown in Figure 7-7 will be produced.

Figure 7-7. A DataForm control with a custom edit template

Note that the labels for the fields in Figure 7-7 are bold, indicating to the user that these fields require a value. This is because the bound properties are decorated with the Required validation attribute.

Although labels are by default positioned to the left of the fields, you can also position them above the fields by setting the DataForm control's LabelPosition property to Top. Top-aligned labels require less horizontal space, and some research has suggested that they enable users to complete forms faster because they can see both the label and the field together as they scan the form. In addition, you don't

have to worry so much about how much room the label requires when a form is to be localized. Figure 7-8 demonstrates top-aligned labels.

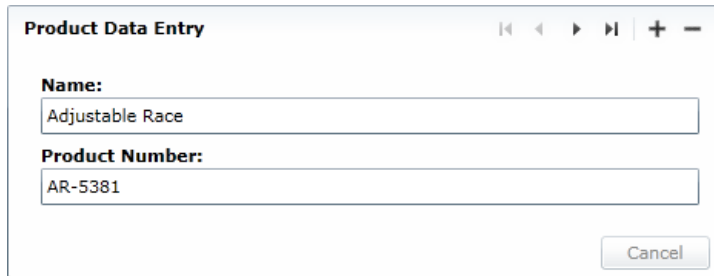


Figure 7-8. A DataForm control with top-aligned labels

More information on this can be found at these two URLs:

- www.uxmatters.com/mt/archives/2006/07/label-placement-in-forms.php
- www.lukew.com/ff/entry.asp?504

The Cancel Button

You may have noticed that a Cancel button has also been automatically been created at the bottom of the data entry form. This is because the bound object implements the `IEditableObject` interface (as do all entities/objects exposed by a domain service with RIA Services), and enables any changes made to the class to be committed or rolled back. If the bound object does not implement this interface, then the Cancel button will remain disabled. The `IEditableObject` interface will be discussed in more detail later in this chapter.

■ **Note** You can use the `CancelButtonContent` and `CancelButtonStyle` properties to customize the Cancel button if required.

The AutoEdit and AutoCommit Properties

You can make the editing of a record more explicit by turning off the automatic edit feature of the DataForm (set its `AutoEdit` property to `False`), and by turning off the feature that automatically commits the changes to the bound entity/object when moving to another record (set its `AutoCommit` property to `False`). Turning off the automatic edit feature means that whenever the user navigates to a record, it will be in read-only mode (and use the read-only template, if defined). A pencil icon is added to the header of the DataForm control, which will put it in edit mode when clicked. Turning off the automatic commit feature will add an OK button to the bottom of the DataForm control (next to the Cancel button). The user will not be able to navigate to another record (or add/delete a record) once they are in edit mode (and have started making changes to the data) until they've committed the changes (by clicking the OK button) or cancelled them (by clicking the Cancel button).

■ **Note** You can use the `CommitButtonContent` and `CommitButtonStyle` properties to customize the OK button if required.

Customizing the Header

If the `DataForm` control's header doesn't suit the look of your application, you can retemplate it by assigning a new data template to its `HeaderTemplate` property, or remove it completely by changing its `HeaderVisibility` property to `Collapsed`. If you do hide the header, you can still invoke the corresponding functions using the `DataForm`'s methods. For example, it has the following methods:

- `AddNewItem`: Adds a new item to the bound collection and navigates to it
- `DeleteItem`: Deletes the current item from the bound collection
- `BeginEdit`: Changes the mode to edit mode
- `CommitEdit`: Commits the current changes to the entity
- `CancelEdit`: Cancels the changes made since the editing began and restores the original property values of the control to what they were when `BeginEdit` was called

The `DataForm` control also raises events when the user invokes one of its functions (such as when an item is added or deleted, or a new item is navigated to), which you can handle if required in your code.

You may have noticed that there were no methods to navigate through the items in the bound collection. You can, however, use the `CurrentIndex` property for this purpose, incrementing and decrementing it to move to the next and previous items, setting it to 0 to move to the first item in the collection, or setting it to the item count minus one to move to the last item in the collection. You can also choose which buttons you want to appear in the header. The `DataForm` control's `CommandButtonsVisibility` property can be used for this purpose, accepting an integer value that can be calculated by ORing the required values from the `DataFormCommandButtonsVisibility` enumeration together (in code), or simply including all the values separated by a comma in XAML. The available values are

- `None`: Don't show any buttons.
- `Add`: Show the Add button.
- `Delete`: Show the Delete button.
- `Edit`: Show the Edit button.
- `Navigation`: Show the first/previous/next/last item buttons.
- `Commit`: Show the OK button (at the bottom of the `DataForm` control).
- `Cancel`: Show the Cancel button (at the bottom of the `DataForm` control).
- `All`: Show all the buttons (as appropriate).

For example, here's how to show only the Add and Delete buttons (in code):

```
ProductsDataForm.CommandButtonsVisibility = DataFormCommandButtonsVisibility.Add |  
                                           DataFormCommandButtonsVisibility.Delete;
```

And here's how to do the same in XAML:

```
CommandButtonsVisibility="Add,Delete"
```

■ **Note** The `DataForm` control is designed to be bound to an object. For simple unbound data entry scenarios (such as a login window), it's best not to use the `DataForm` control—simply lay out standard controls in a `Grid` instead.

Refining the Data Entry Form's Functionality

It's extremely important that you create data entry forms in your applications that are easy to navigate and use. You want to minimize the amount of work that the user has to do to enter the data, and preferably enable power users to not have to even take their hands off the keyboard (increasing their productivity). In this section, we'll look at the most popular data input controls available in Silverlight, and tips for making it easier for the user to enter data into the application.

Data Input Controls

There are many data input controls available in Silverlight and the Silverlight Toolkit that you can use in your data entry forms. You can actually use any control you like in a `DataForm` control (including third-party controls), so you can choose the most appropriate controls for your needs. Let's take a brief look at some of the more common data input controls that you may wish to use.

The TextBox Control

The `TextBox` control enables users to enter free-form text (as shown in Figure 7-9).

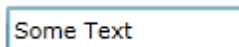


Figure 7-9. A `TextBox` control

You can get/set the text in the `TextBox` via its `Text` property:

```
<TextBox Text="{Binding Path=Name, Mode=TwoWay}" />
```

Thanks to a new binding feature in Silverlight 4, you can format the bound value before displaying it. This is very useful when binding to a non-string property (such as a `DateTime` or a decimal). Use the `StringFormat` property on the binding to set the format (using the same types of formatting strings you would use if you were to format the value using its `ToString` method in code (using either standard or custom formats). For example, the following binding displays the value formatted as currency (as shown in Figure 7-10):

```
<TextBox Text="{Binding TotalCost, Mode=TwoWay, StringFormat=C}" />
```

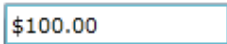


Figure 7-10. A bound TextBox control with a custom string format

■ **Note** You will find more information about the available standard formatting strings and creating custom formatting strings at the following MSDN page (and in related topics): <http://msdn.microsoft.com/en-us/library/26etazsy.aspx>. If you need to perform a more complex value format or conversion, then you may be better off using a value converter, as will be discussed in Chapter 10.

Unfortunately, the TextBox control doesn't have any masked edit functionality to restrict the input into the TextBox from the user. There are a number of third-party masked edit controls, as well as an open source one on CodePlex, which you can get here: <http://silverlightinputctrl.codeplex.com> (note the misspelling of Silverlight in the URL name).

Alternatively, you can assign a behavior that adds masked edit functionality to a text box. For example, you will find one (written by Jim Fisher) in the Microsoft Expression Gallery, here: <http://gallery.expression.microsoft.com/en-us/CMEditMaskBehavior>. How to use behaviors will be discussed in Chapter 10.

You can set the maximum number of characters accepted by the TextBox control using its `MaxLength` property.

If you want to stop the user from being able to enter text into the TextBox control (but still be able to select and copy the text, which can't be done when the TextBox control is disabled), you can set its `IsReadOnly` property to `True`. Doing so will change the background of the TextBox control to a light gray to indicate it cannot be edited.

You can horizontally align the text in the TextBox control using its `TextAlignment` property. You can left-align the text (the default), right-align it, or center it. Note that although `Justify` appears in IntelliSense and the Properties window as an alignment option, it will throw an exception when used.

By default, when more text is entered into the TextBox control than it can display, the text will remain on a single line and the beginning of the text will disappear off the TextBox's left edge. However, if you make your text box tall enough to display multiple lines, then you can get the text to wrap onto the next line by setting the TextBox control's `TextWrapping` property to `Wrap` (the default being `NoWrap`). If you want the user to be able to start new lines in the text box by pressing the Return/Enter key, then you will need to set the TextBox control's `AcceptsReturn` property to `True`.

By default, the vertical and horizontal scroll bars in the TextBox are hidden/disabled. You can enable them via the `VerticalScrollBarVisibility` and `HorizontalScrollBarVisibility` properties. Setting their value to `Auto` will cause the scroll bars to display only when the text exceeds the corresponding dimension of the text box, and setting them to `Visible` will cause them to display them regardless of whether they're required.

■ **Note** Setting the horizontal scroll bar to `visible` will disable text wrapping.

The CheckBox Control

The CheckBox control enables the user to enter a True or False value (or a null value in the case of a three-state check box), as shown in Figure 7-11. You can get/set the value of the check box via its `IsChecked` property, and set its label using its `Content` property:

```
<CheckBox IsChecked="{Binding FinishedGoodsFlag, Mode=TwoWay}"
          Content="Is Finished Goods" />
```

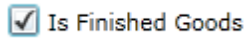


Figure 7-11. A bound CheckBox control

You can turn it into a three-state check box by setting its `IsThreeState` property to `True`. This enables an additional state (with a value of `null`) for the check box, which you may use to indicate that a value has not been set. Figure 7-12 shows what the third state looks like.

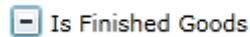


Figure 7-12. The third state in a three-state check box

The RadioButton Control

Radio buttons (also known as option buttons) enable the user to select one option from a number of options, as shown in Figure 7-13.

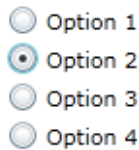


Figure 7-13. Option buttons

Much like the CheckBox control, you can get/set the value of a radio button via its `IsChecked` property, and set its label using its `Content` property:

```
<RadioButton Content="Option 1" IsChecked="{Binding Option1, Mode=TwoWay}" />
```

To bind the radio buttons to an object/entity, you will need to have each option exposed as a separate Boolean property on the object/entity that you are binding to, and bind each radio button to the property representing its corresponding option. Alternatively, you could create a value converter (discussed in Chapter 10) that compares the value of a property with a given value, returning `True` if the values match and `False` if they don't.

All radio buttons are implicitly "linked," such that when you select one radio button, any currently selected radio button will be deselected. If you want to have multiple groups of radio buttons in your view that you don't want to interact, then you can separate them by assigning each radio button to a group via their `GroupName` property. You can give each group whatever name you wish, and only the radio

buttons with the same group name will interact. When you group radio buttons, ensure that they are visually separated also in the view so that the user recognizes which options belong to which groups.

The ComboBox Control

The ComboBox control in Silverlight is not strictly a combo box as such, but a drop-down list (proper ComboBox controls accept text entry, where the Silverlight ComboBox does not). You can bind its `ItemsSource` property to a collection of objects to display (in much the same way as the List control), or declare the items in XAML using the `Items` property. You can then get/set the selected item using its `SelectedItem` property.

The following XAML demonstrates creating a ComboBox in XAML, with the items to be displayed in the list also declared in XAML. This gives the output shown in Figure 7-14.

```
<ComboBox>
  <ComboBox.Items>
    <ComboBoxItem Content="Option 1" />
    <ComboBoxItem Content="Option 2" />
    <ComboBoxItem Content="Option 3" />
    <ComboBoxItem Content="Option 4" />
  </ComboBox.Items>
</ComboBox>
```

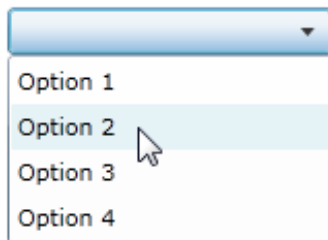


Figure 7-14. A ComboBox control

Prior to Silverlight 4, data binding to a ComboBox control was a rather painful experience, as the control didn't have a `SelectedValue`, `SelectedValuePath`, or `DisplayMemberPath` property—all of which are needed when binding a property of the selected object to your entities/objects. Luckily, these properties are finally here for us to use. Say, for example, that we have a `Product` entity with a property named `ModelID`, and also a collection of `Model` objects, each with a `ModelID` and a `ModelName` property. What we want to do is to enable the user to select the model of the product from a drop-down list, assigning the `ModelID` of the selected `Model` to the `ModelID` property of the `Product` entity. However, in the list, we want to display the names of the models, not their IDs. For this scenario, we can bind the collection of `Model` objects to the ComboBox's `ItemsSource` property, set its `SelectedValuePath` to `ModelID`, and set its `DisplayMemberPath` to `ModelName`. The ComboBox will then be bound to our collection of `Model` objects, so we just need to bind its `SelectedValue` property to our `Product`'s `ModelID` property (so that the `ModelID` property of the selected item in the ComboBox is assigned to the `Product`'s `ModelID` property, and vice versa):

```
<ComboBox ItemsSource="{StaticResource modelsResource}" SelectedValuePath="ModelID"
  DisplayMemberPath="ModelName" SelectedValue="{Binding ModelID}" />
```

■ **Note** As with the List control, you can customize the data template of each item by assigning a custom data template to the ComboBox's `ItemTemplate` property. This will allow you to show images, have multiple columns, and so on for your items.

The List Control

You've already seen how to use the List control to display a summary list and enable the user to drill down on a record in the previous chapter. You could also use it in a data entry scenario in much the same way as described for the ComboBox control—enabling the user to select an item from a collection. Another common use for the List control in a data entry scenario is to have two lists, with the user able to move items from one list to another (via dragging and dropping an item between lists, or simply selecting an item and clicking a button to move that item to the other list).

The Date Input Controls

There are two controls for entering/selecting dates in the Silverlight SDK: the Calendar control and the DatePicker control.

The Calendar Control

The Calendar control (shown in Figure 7-15) enables the user to select a date from a calendar. You can get this selected date from the Calendar control's `SelectedDate` property:

```
<sdk:Calendar SelectedDate="{Binding StartDate, Mode=TwoWay}"/>
```

You can also allow the user to select a range of dates by changing the control's `SelectionMode` property from `SingleDate` to `SingleRange` or `MultipleRange`. You can then get the selected dates from the control's `SelectedDates` property. Alternatively, you can use the Calendar control for display purposes only by setting its `SelectionMode` property.



Figure 7-15. A Calendar control

You can narrow down the range of the date(s) that can be selected by assigning a start date for the range to the `DisplayDateStart` property, and the end date for the range to the `DisplayDateEnd` property. All dates outside this range will be blanked out and nonselectable. You can also use the `BlackoutDates` property to set additional dates that the user cannot select. Instead of being blanked out, these dates will

display a gray cross over them. This feature can be used to prevent a weekend or a public holiday from being selected. To assign dates to the `BlackoutDates` property, you will need to add `CalendarDateRange` objects to the collection (with a from and to date), or bind the property to a collection of these objects in XAML.

By default, the current date is always highlighted in the calendar. You can disable this by setting the `IsTodayHighlighted` property to `False`. You can change the first day of the week using the `FirstDayOfWeek` property.

The month view isn't the only type of view supported by the `Calendar` control. Using the control's `DisplayMode` property, you change the mode between `Month`, `Year`, and `Decade`.

The DatePicker Control

The `DatePicker` control has a lot of features in common with the `Calendar` control (as shown in Figure 7-16). Clicking the button next to the text box (or pressing `Ctrl+down arrow` when the text box has focus) will pop up a calendar that the user can select the date from. Unlike the `Calendar` control, only a single date can be entered into/selected from the `DatePicker` control. However, you can still restrict the date entered to be within a range like the `Calendar` control, and black out various date ranges. It does have one additional property: the `SelectedDateFormat` property. Instead of using the default short date format used, you can change the date format to the long date format via that property.

```
<sdk:DatePicker SelectedDate="{Binding StartDate, Mode=TwoWay}" />
```

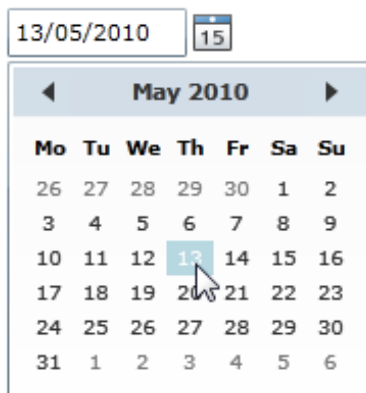


Figure 7-16. A DatePicker control

The Time Input Controls

In addition to the date data input controls in the Silverlight SDK, there are two controls for entering times in the Silverlight Toolkit: the `TimeUpDown` control and the `TimePicker` control.

The TimeUpDown Control

The `TimeUpDown` control (shown in Figure 7-17) lets the user either enter a time or select one using the up/down buttons. The control enables the user to enter a part of the time, and guesses the rest of it when the user tabs away (e.g., the user can simply enter `9`, and the control will fill in `9:00 AM`). The user

can also select a particular part of the time (hour, minute, etc.) and use the up/down buttons to change it accordingly.

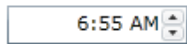


Figure 7-17. A TimeUpDown control

You can get/set the time via its Value property:

```
<toolkit:TimeUpDown Value="{Binding StartTime, Mode=TwoWay}" />
```

You can change the format of the time using the TimeUpDown control's Format property, using the standard or custom date formatting strings. For example, you could specify HH:mm:ss as the format, which uses 24-hour time and includes the seconds.

You can force the user to only enter the time via the up/down buttons by setting its IsEditable property to False. You can also stop the user from going past 11:59 p.m. or before 12:00 a.m. (and cycling through this period) by setting the IsCyclic property to False.

The TimePicker Control

The TimePicker control is similar to the TimeUpDown control, except it also has a drop-down list that the user can select a time from, as shown in Figure 7-18.

```
<toolkit:TimePicker Value="{Binding StartTime, Mode=TwoWay}" />
```

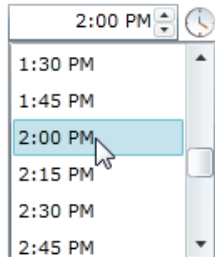


Figure 7-18. A TimePicker control

By default, the drop-down list contains an item for every half-hour, but you could change this to every quarter of an hour, for example, by setting PopupMinutesInterval to 15.

The Up/Down Controls

You've seen how the TimeUpDown and TimePicker controls have up/down buttons to enable the user to modify the time using only the mouse. There are also two other controls in the Silverlight Toolkit that implement this same feature for other data types: the NumericUpDown control (for numbers) and the DomainUpDown control (for selecting an item from a collection).

The NumericUpDown Control

The `NumericUpDown` control only allows the entry of numbers—attempting to enter a nonnumeric value will result in the value being reverted to its original value when the field loses the focus. You can set the number of decimal places that are displayed via the `DecimalPlaces` property, and how much the value will increment/decrement when using the up/down buttons with the `Increment` property. You can get/set the value via its `Value` property. The following code demonstrates creating a `NumericUpDown` control in XAML, and Figure 7-19 shows the result:

```
<toolkit:NumericUpDown DecimalPlaces="2" Increment="0.5"
    Value="{Binding ListPrice, Mode=TwoWay}" />
```



Figure 7-19. A `NumericUpDown` control

The DomainUpDown Control

The `DomainUpDown` control is a bit like the `ComboBox` control in that it enables you to enter or select an item from a list of items (using the up/down buttons). You bind a list of items to the `ItemsSource` property to select from, which could be an array of strings or a collection of objects. The user is constrained to only entering values from that collection. You can choose from one of two options as to what happens if the value entered by the user does not match an item in the list via the `InvalidInputAction` property. The `UseFallbackItem` option will result in the value being reverted to the value of the `FallBackItem` property when the field loses the focus. If a value for the `FallBackItem` property has not been specified, then entering a value not in the list will result in the previous value being reinstated when the field loses the focus. Alternatively, the `TextBoxCannotLoseFocus` option forces the user to enter/select a valid item from the list before they can move on.

When binding the `ItemsSource` property to a collection of objects, you can use either the `ValueMemberPath` property or the `ValueMemberBinding` property on the `DomainUpDown` control to specify the property on the objects that will be displayed in the control. Generally, you will use the `ValueMemberPath` property, to which you simply assign the name of the property identifying that item. Alternatively, you can use the `ValueMemberBinding` property, to which you can assign a binding expression pointing to the property to use instead. This enables you to specify a value converter as a part of the binding if necessary. As an example, you might have a collection of objects representing US states, each with two properties: one for the name of the state, and one for its short code (e.g., NY). You can display the name of the state in the control (I'll discuss how to do this shortly), but set the `ValueMemberPath` property to the name of the short code property. If you do this, when the user edits the value of the field, they can enter the short code for the state, and upon tabbing away the corresponding state name will be displayed instead.

Unfortunately, unlike the `ComboBox` control, the `DomainUpDown` control has no `DisplayMemberPath` or `SelectedValuePath` property (nor does it have a `SelectedValue` property). When binding to a collection of objects (which aren't strings), the `DomainUpDown` control will simply call the `ToString` method on the object to get the text to display. Unless you have overridden the `ToString` method in your classes, you will therefore need to create a data template and assign it to the control's `ItemTemplate` property to customize what is displayed in the control. This also means that you can create a somewhat more complex custom layout to display in the control for each item if you wish.

The index of the currently selected item in the bound collection can be obtained via the `CurrentIndex` property. Much like the `TimeUpDown` control, this control also has `IsCyclic` and `IsEditable` properties.

The following code demonstrates creating a simple `DomainUpDown` control in XAML, and Figure 7-20 shows the result:

```
<toolkit:DomainUpDown ItemsSource="{StaticResource StatesResource}" />
```



Figure 7-20. A `DomainUpDown` control

The `AutoCompleteBox` Control

The `AutoCompleteBox` control can be used to enable the user to start typing a value to filter a list of items (appearing below the control when they begin to type), and select an item from the filtered list to complete that field's data input (as shown in Figure 7-21). Under the covers, configuring the `AutoCompleteBox` control is similar to the `DomainUpDown` control. You bind its `ItemsSource` property to a collection that will be used to populate the list of potential values. Usually this will be a collection of strings, but you can also bind it to a collection of objects and specify the property on these objects to be displayed using the control's `ValueMemberPath`/`ValueMemberBinding` property (in the same way as you do for the `DomainUpDown` control). You can also assign a data template to the `AutoCompleteBox` control's `ItemTemplate` property to customize how items appear in the list.

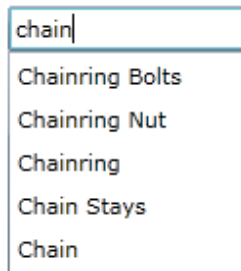


Figure 7-21. The `AutoCompleteBox` control

■ **Note** You may wish to use this control in some instances instead of a `ComboBox` control to speed up data input for power users. Note, however, that this control does not restrict you from entering a value not in the bound collection.

By default, items only appear in the list that start with the entered text. However, you can alter this behavior using the `Filter` property. There are a number of different options, but the ones you will generally use are the `StartsWith` option (the default) and the `Contains` option. Alternatively, you could

use the `Custom` option and create your own item or text filter, assigning it to the `ItemFilter/TextFilter` property accordingly. Further discussion of custom filters, however, is beyond the scope of this book.

You can have the text in the text box part of the control automatically complete with the first matching entry. Rather than having the user get partway through entering the value and then selecting an item from the list (which requires further effort), you can set the `IsTextCompletionEnabled` property to `True` to have the first matching item in the list already populate the text box (although the user can still keep typing to further refine the results). When the item within the text box is the one the user is after, they can simply press `Tab` to move to the next field.

The list of matches appears as soon as the user enters the first character, but if you have a very long list of items being filtered, then it may be best to increase the `MinimumPrefixLength` property, which specifies the number of characters that the user has to enter in order for the drop-down list to appear with the matching filtered items. Alternatively (or in addition), you could assign a value to the `MinimumPopulateDelay` property, which specifies a period of time (in milliseconds) to wait before populating this list.

A very simple example of using the `AutoCompleteBox` control in XAML is provided here:

```
<sdk:AutoCompleteBox ItemsSource="{StaticResource StatesResource}" />
```

The Label Control

Although it's not a data input control as such, the `Label` control has been designed for use alongside data input controls. You set some text to display (a label for a field) via its `Content` property, and then associate it with a data input control via its `Target` property (using `ElementName` binding, discussed in Chapter 10). By associating it with a data input control, when the data entered into that control is invalid, the label will turn red to indicate that the field has an error. In addition, if the data input control is bound to a property marked as `Required` (using the validation attributes), then the label will use bold text. The following XAML demonstrates associating a `Label` control with a `TextBox` control named `ProductNameTextBox`:

```
<sdk:Label Content="Name:" Target="{Binding ElementName=ProductNameTextBox}" />
```

Setting the Tab Order

In order for the user to navigate between the data input controls using the `Tab` key (and `Shift+Tab` to move back to the previous control) in a logical fashion, you should set the tab order for each data input control in the user interface. You do this by assigning a numeric value to each control's `TabIndex` property. When the user tabs away from a control, the control with the same or next highest `TabIndex` value will receive the focus. The values do not have to be sequential, so it is a good idea to separate `TabIndex` values in increments of five or ten (so that you can change the tab order of a control and insert it between two other controls without having to update the tab order of all the other controls).

You can prevent a control from getting the focus when the user is tabbing between controls by settings its `IsTabStop` property to `False`.

You can also customize the scope and behavior of tabbing using the `TabNavigation` property. The default value is `Local`, which tabs through all the data input fields, and then the controls outside the current view, the browser's address bar, and, so on, before coming back to tab through the controls again. If you change this property at the container level (such as the view/page level) to `Cycle`, then the focus will cycle between the controls in that container only (enabling you to restrict tab stops to only the data input controls in the data entry form). If you set this property to `Once`, the container and all its data input controls will only be able to be tabbed to once.

Setting Focus

In addition to enabling the user to logically tab between the controls in your data entry form, you'll want to set the initial focus to a given data input control when a record is loaded. You will need to do this by handling the `ContentLoaded` event on the `DataForm`, as the edit template is not instated in the `DataForm` control until the data is bound to it. The next issue is that even though you give a data input control a name in XAML (within the `DataForm` control), it won't have a corresponding variable in the code-behind. This is because the control is within a data template, which can be repeated. Because this template (and its content) can be repeated, to avoid possible name collisions, each data template is created in its own namespace (a concept discussed back in Chapter 3). If you were to refer to a control within a data template in code, it wouldn't know which instance to use. Since a variable is not created for the control in the code-behind, you will need to search for it inside the data template. The `DataForm` control actually makes this procedure easier by providing a `FindNameInContent` method. Pass it the name of the control and it will return you its current instance. You can then set the focus to this control, as demonstrated here:

```
private void dataForm ContentLoaded(object sender, DataFormContentLoadEventArgs e)
{
    TextBox NameTextBox = ((DataForm)sender).FindNameInContent("NameTextBox")
                                                as TextBox;
    NameTextBox.Focus();
}
```

■ **Note** At the time of writing, this method of setting focus to the first field only works properly when binding the `DataForm` control to a single object. When attempting to use it when binding to a collection, it will work for the first record, but subsequent attempts to navigate between the objects in the collection will fail. The `ContentLoaded` event will be raised in response to the navigation; however, for some reason, setting the focus in the event handler for this event prevents the current item being changed (and the `CurrentItemChanged` event is consequently never raised). Hopefully this problem will be fixed in a future version of the `DataForm` control.

Getting Focused Control

You can obtain a reference to the control that currently has the focus by using the `FocusManager` object (which you can find in the `System.Windows.Input` namespace):

```
object element = FocusManager.GetFocusedElement();
```

Checking Whether Items Have Changed

It's generally a good idea to notify the user if they are navigating away from a data entry form where data has been entered or modified, but has not been saved (to prevent them from accidentally losing changes to the data). This is often referred to as checking whether the data is "dirty." If the data is dirty, you can then prompt the user to save or discard their changes.

If the data entry form has been configured to enable only a single item to be edited, and you are using the `DataForm` control as the container, then you should be able to check its `IsItemChanged` property to see if any changes have been made to the current item. However, this doesn't appear to work at the time of writing (only ever returning `false`). Alternatively, if your bound object was retrieved from the server via RIA Services, then you can cast the `CurrentItem` property of the `DataForm` control to a type of `Entity` and check its `HasChanges` property, like so:

```
bool hasChanges = ((Entity)ProductsDataForm.CurrentItem).HasChanges;
```

If the user can edit multiple items in the data entry form, then you will need to check whether the collection has changes. If the collection was returned from the server via RIA Services, then you can simply use the `HasChanges` property of the domain context instance that returned the collection to find out whether unsaved changes have been made to the collection.

Using the DataGrid for Data Entry

In some scenarios, it can be useful to use the `DataGrid` control for data entry purposes instead of using a form-based layout. As described in the previous chapter, the `DataGrid` is more suited for data entry purposes than simply displaying data (in which a `List` control would typically be more appropriate). It's best used when multiple objects in a collection need to be added or maintained, each of these objects relate to the parent entity, and there are few fields to be displayed/modified.

One good example of when to use the `DataGrid` is when creating a data entry form for an invoice, for displaying and modifying the related invoice lines. In this scenario, there are multiple lines to be entered/displayed, all relating to the same invoice, and typically few fields involved (e.g., description, quantity, unit cost, tax, and line total). These features make this an ideal scenario for using a `DataGrid` control.

Configuring and customizing a `DataGrid` control was discussed in the previous chapter; however, in this scenario you should set the `IsReadOnly` property of the `DataGrid` to `False`. As was also discussed in the previous chapter, you can customize the types of input control used for each field (using the `DataGridTextBoxColumn`, `DataGridCheckBoxColumn`, or `DataGridTemplateColumn` column types). Note that when you assign bindings to the `DataGridTextBoxColumn` and `DataGridCheckBoxColumn` types, the bindings for the corresponding input controls that the `DataGrid` creates (when a row is in edit mode) will automatically use the `TwoWay` binding mode (regardless of the mode assigned to a column's binding). Input controls in a `DataGridTemplateColumn` column will, however, need their binding mode explicitly set to `TwoWay` as usual.

Adding a Row

How you handle adding/deleting of rows really depends on your requirements. Note that there are no `AddRow` or `DeleteRow` methods (or similar) on the `DataGrid`—instead, you will need to work directly with the underlying bound collection.

The simplest way to implement this behavior (although not particularly the most user-friendly) is to simply add a button to your view that adds (or inserts) a new item into the collection that the `DataGrid` is bound to. If you have a reference to the bound collection (or can cast the value of the `DataGrid`'s `ItemsSource` property to that type), then you should be able to simply call its `Add` or `Insert` method. However, if the `DataGrid` control is bound to a `DomainDataSource` control, then you will need to cast the value of the `DataGrid`'s `ItemsSource` property to a `DomainDataSourceView` (similar to the collection views discussed in the previous chapter). For example:

```

DomainDataSourceView view = productDataGrid.ItemsSource as DomainDataSourceView;
Product newProduct = new Product();
view.Add(newProduct);

// Scroll the first cell of the new row into view and start editing
productDataGrid.Focus();
productDataGrid.SelectedItem = newProduct;
productDataGrid.CurrentColumn = productDataGrid.Columns[0];
productDataGrid.ScrollIntoView(productDataGrid.SelectedItem,
                                productDataGrid.CurrentColumn);
productDataGrid.BeginEdit();

```

This code will add a new item to the collection, select this item in the DataGrid, scroll the item into view, and put the first cell of the item's row in edit mode.

■ **Note** If any of the properties have default values that fail one or more validation rules, the validation summary will be automatically displayed at the bottom of the DataGrid as soon as the user starts editing a row, which isn't a particularly nice user experience. This is unfortunately the case with all the methods of adding items to the DataGrid described here. Rather annoyingly, the validation summary often covers the row being edited—hopefully this issue will be resolved in the near future. You can avoid this problem by assigning default values to the new object that will pass the validation rules.

Inserting a Row

Unfortunately, the `DomainDataSourceView` class doesn't have an `Insert` method, but if you have a reference to the bound collection that does support this method (such as an `ObservableCollection`) and want to insert a new row after the currently selected row, you can use the following code:

```

ObservableCollection<Product> collection =
    productDataGrid.ItemsSource as ObservableCollection<Product>;

int insertIndex = productDataGrid.SelectedIndex + 1;

// To counter some strange behavior by the DataGrid when the
// first row is selected, but not reported as such by the
// SelectedIndex property (right after the DataGrid is populated)
if (productDataGrid.SelectedIndex == -1 && collection.Count != 0)
    insertIndex = 1;

collection.Insert(insertIndex, new Product());

// Select and scroll the first cell of the new row into view and start editing
productDataGrid.SelectedIndex = insertIndex;
productDataGrid.CurrentColumn = productDataGrid.Columns[0];
productDataGrid.ScrollIntoView(productDataGrid.SelectedItem,
                                productDataGrid.CurrentColumn);
productDataGrid.BeginEdit();

```

Spreadsheet-Like Editing

Another way to let the user enter multiple rows in a spreadsheet-like fashion is to populate the bound collection with items, and remove the unused items from the collection before submitting the data back to the server. This is useful when you have a maximum number of rows that you will support. However, when the number of rows is undetermined, setting a fixed maximum number of rows isn't an ideal solution.

Maintaining an Empty Row for the Entry of New Records

The final (and generally most user-friendly) method is to automatically maintain an empty row as the last row in the DataGrid. The user can enter data for a new item in this row, and the DataGrid should automatically add a new empty row once they do. Sadly, there is no built-in feature like this in the core Silverlight DataGrid. You could add it yourself (all the source code for the DataGrid control is available in the Silverlight Toolkit), but it's not a great idea, as you'd be tying yourself to that particular version of the DataGrid control, and wouldn't be able to (easily) take advantage of new features and bug fixes in future releases of Silverlight and the Silverlight Toolkit. You can, however, handle a number of events raised by the DataGrid control and manage this automatically. The steps are as follows:

1. Maintain a class-level variable that will reference the new item object:

```
private object addRowBoundItem = null;
```

2. Add a new item to the bound collection before (or shortly after) binding, and assign this item to the class-level variable. If binding the DataGrid directly to a DomainDataSource control, you would do this in the LoadedData event of the DomainDataSource control, like so:

```
DomainDataSourceView view = productDataGrid.ItemsSource as DomainDataSourceView;
addRowBoundItem = new Product();
view.Add(addRowBoundItem);
```

3. Handle the RowEditEnded event of the DataGrid control. If the row being committed is the empty row item that was edited (you can get the item in the collection that it is bound to from the DataContext property of the row), then it's time to add a new item to the end of the bound collection, ensure it is visible, select it, and put it in edit mode. For example:

```
private void productDataGrid RowEditEnded(object sender,
                                           DataGridRowEditEndedEventArgs e)
{
    if (e.EditAction == DataGridEditAction.Commit)
    {
        if (e.Row.DataContext == addRowBoundItem)
        {
            DomainDataSourceView view =
                productDataGrid.ItemsSource as DomainDataSourceView;

            addRowBoundItem = new Product();
            view.Add(addRowBoundItem);

            productDataGrid.SelectedItem = addRowBoundItem;
```



```

        productDataGrid.CurrentColumn = productDataGrid.Columns[0];

        productDataGrid.ScrollIntoView(addRowBoundItem,
                                       productDataGrid.CurrentColumn);
        productDataGrid.BeginEdit();
    }
}

```

4. Remember to always delete the last item in the collection before submitting the changes back to the server (as it will always be the item representing the new row):

```

DomainDataSourceView view = productDataGrid.ItemsSource as DomainDataSourceView;
view.Remove(addRowBoundItem);

```

Deleting the Selected Rows

Deleting the selected row(s) requires you to get the bound collection or collection view and enumerate through the `SelectedItems` property of the `DataGrid`, removing the corresponding item from the bound collection/collection view. This can be somewhat of a messy process due to removing items in an enumerated collection, but the use of LINQ can simplify the code somewhat:

```

ObservableCollection<Product> collection =
    productDataGrid.ItemsSource as ObservableCollection<Product>;

// Convert the SelectedItems property to an array so its enumerator won't be
// affected by deleting items from the source collection
// Note that this line requires the System.Linq namespace to be declared
var removeItems = productDataGrid.SelectedItems.Cast<Product>().ToArray();

foreach (Product product in removeItems)
    collection.Remove(product);

```

■ **Note** At the top of your file, you'll need to have a `using` directive to the `System.Linq` namespace to be able to use the `Cast<T>` method in this code.

Alternatively, you can force only one row to be selected at a time in the `DataGrid` by setting its `SelectionMode` property to `Single` (rather than the default value of `Extended`), meaning that you only need to worry about removing a single selected item. In this case, the following code would suffice:

```

ObservableCollection<Product> collection =
    productDataGrid.ItemsSource as ObservableCollection<Product>;

collection.Remove(productDataGrid.SelectedItem as Product);

```

Adding a Delete Button to Each Row

Adding a delete button to each row is reasonably easy. Simply use a template column and define a data template for the cell containing a button:

```
<sdk:DataGridTemplateColumn Width="80">
    <sdk:DataGridTemplateColumn.CellTemplate>
        <DataTemplate>
            <Button Content="Delete" Click="DeleteButton Click" />
        </DataTemplate>
    </sdk:DataGridTemplateColumn.CellTemplate>
</sdk:DataGridTemplateColumn>
```

Now handle the Click event of the button in code to delete the corresponding item. The bound item in the collection is assigned to the data context of the row, and this is inherited down the hierarchy to the cell and then the button. Therefore, you can get the object/entity that the row is bound to from the Button control's DataContext property, and remove it from the collection/view that the DataGrid is bound to:

```
private void DeleteButton Click(object sender, RoutedEventArgs e)
{
    DomainDataSourceView view = productDataGrid.ItemsSource as DomainDataSourceView;
    view.Remove(((FrameworkElement)sender).DataContext);
}
```

■ **Note** This delete button is generally more attractive and less intrusive if you retemplate the button to simply contain an image (such as a red cross).

Structuring Objects for Use by Data Entry Forms

In order to create effective data entry forms, there are a few things you should do to make your objects friendlier to the user interface controls that will bind to them. RIA Services entities automatically implement these features by default, but if you are binding to objects whose classes are defined on the client (such as a ViewModel when using the MVVM design pattern, as discussed in Chapter 12), or you're simply not using RIA Services at all, then you will need to implement these features manually. Let's take a look at the most important features you should add to your objects for the benefit of the user interface.

Implementing the INotifyPropertyChanged Interface

One of the key interfaces that your objects should implement is the INotifyPropertyChanged interface (found in the System.ComponentModel namespace). This is a simple interface, requiring your class to implement a single event named PropertyChanged. Bindings automatically listen for this event, and when it is raised, any binding associated with that property will update itself accordingly (and hence the control associated with that binding will be updated as well).

Basic Implementation (with “Magic Strings”)

To notify listeners of this event that a property value has been updated, you simply raise the `PropertyChanged` event, passing it a `PropertyChangedEventArgs` object that has the name of the property being updated. The following example demonstrates the code that you would use to notify the user interface that the value of a property named `Name` has changed:

```
if (PropertyChanged != null)
    PropertyChanged(this, new PropertyChangedEventArgs("Name"));
```

One thing that you must be aware of is that the `PropertyChanged` event is never raised when you use automatically implemented properties. For example, properties implemented in the following fashion:

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

do not raise the `PropertyChanged` event on the object, even when that object implements the `INotifyPropertyChanged` interface. Therefore, the user interface will not be aware of any changes to the value of that property (that it did not make itself), and will not be updated when the underlying property value is changed. This unfortunately means that you must implement the getter and setter for the property, and maintain the property’s value in a member variable. You can then raise the `PropertyChanged` event in the setter as required:

```
private string name = "";

public string Name
{
    get { return name; }
    set
    {
        name = value;

        if (PropertyChanged != null)
            PropertyChanged(this, new PropertyChangedEventArgs("Name"));
    }
}
```

■ **Note** Unfortunately, this means additional code (and hence additional work). Some Visual Studio extensions enable you to easily turn an automatically implemented property into a property with a backing store (i.e., a member variable that maintains the property’s value), and raise the `PropertyChanged` event for that property. This is one option to save you from having to hand-code the getters and setters. Alternatively, try searching the Web for “`INotifyPropertyChanged` snippets,” and you will find a multitude of snippets that will help you create your properties accordingly (and that you can customize to suit your needs).

The `INotifyPropertyChanged` interface is very commonly used, and many developers add an `OnPropertyChanged` (or similarly named) method that raises the `PropertyChanged` event (rather than

actually raising the event in the property setters). This simplifies the property setters, as you do not have to check that the `PropertyChanged` event is not null (which it will be if nothing is handling the event) before calling it.

```
protected void OnPropertyChanged(string propertyName)
{
    if (PropertyChanged != null)
        PropertyChanged(this, new PropertyChangedEventArgs(propertyName));
}
```

In the property setter now, you can simply use a single line of code to raise the event:

```
OnPropertyChanged("Name");
```

This makes your property setters somewhat neater, although there is still the unfortunate requirement to pass the method the name of the property as a string. This is very open to issues in which the property name is different from the string. For example, the string may misspell the property name, or the property name may have been refactored but the string was not updated accordingly. These issues can lead to particularly difficult bugs to identify and track down. Let's look at some other alternatives that can avoid this issue. Which you should choose really comes down to personal taste.

Using Reflection Instead of Magic Strings

You can use reflection to get the name of the property in its setter (ignoring the first four characters, `set`) and pass that to the `OnPropertyChanged` method:

```
OnPropertyChanged(MethodBase.GetCurrentMethod().Name.Substring(4));
```

■ **Note** You will need a `using` directive to the `System.Reflection` namespace at the top of your file in order for this code to compile.

Using Reflection to Validate Magic Strings in Debug Mode

Alternatively, you may wish to continue with hard-coding the property name as a string, and check in your `OnPropertyChanged` method that the property exists. The best idea is to perform this check in a separate method (and call that method from the `OnPropertyChanged` method), as you can then decorate it with the `Conditional` attribute, and only have it run when in debug mode:

```
[Conditional("DEBUG")]
private void EnsurePropertyExists(string propertyName)
{
    PropertyInfo propInfo = this.GetType().GetProperty(propertyName);
    Debug.Assert(propInfo != null, "The property " + propertyName + " does not " +
        "exist on this class");
}
```

■ **Note** For this method to work, you will need a `using` directive to both the `System.Reflection` namespace and the `System.Diagnostics` namespace.

Using Lambda Expressions Instead of Magic Strings

Yet another method is to use a lambda expression to specify the property name (instead of a string), and extract the name of the property from that. This is a popular method, is faster than reflection, and is refactoring-safe (no magic strings). Start by adding the following `using` directive to the top of your class:

```
using System.Linq.Expressions;
```

Now define the following method in your class (or a base class):

```
private string GetPropertyName<T>(Expression<Func<T>> property)
{
    MemberExpression expression = property.Body as MemberExpression;
    return expression.Member.Name;
}
```

■ **Note** Generally, you would put this method in your base class. Alternatively, you might like to define it as an extension method, as described by Jeremy Likeness here:

<http://csharperimage.jeremylikness.com/2010/06/tips-and-tricks-for-inotifypropertychan.html>.

You can then call the `OnPropertyChanged` method from your property's setter using the following code:

```
OnPropertyChanged(GetPropertyName(() => Name));
```

where `Name` is the name of the property. Emiel Jongerius has written up some other alternatives (including another way of implementing this method that also sets the property's value), and how they compare (performance-wise) here: www.pochet.net/blog/2010/06/25/inotifypropertychanged-implementations-an-overview. Einar Ingebrigtsen also has a blog post with an extended version of this code that provides additional functionality, here: www.ingebrigtsen.info/post/2008/12/11/INotifyPropertyChanged-revisited.aspx.

■ **Note** Oren Eini has another interesting method of handling the `INotifyPropertyChanged` problem that you may like, where he has created an `Observable<T>` class. He has blogged about it here:

<http://ayende.com/Blog/archive/2009/08/08/an-easier-way-to-manage-inotifypropertychanged.aspx>.

Justin Angel has yet another interesting method where he automatically implements `INotifyPropertyChanged` using `Mono.Cecil` and `PostSharp`.

Implementing the IEditableObject Interface

`IEditableObject` is another useful interface to implement in your objects (also found in the `System.ComponentModel` namespace). It is designed to enable any changes made to an object to be able to be cancelled and rolled back (essentially an undo action). Any changes made to the object are tracked using a start/complete-type transaction that can be cancelled, any changes made to the object since the beginning of the transaction are rolled back, and the object's original state is reinstated.

Implementing the `IEditableObject` interface requires your class to implement three methods: `BeginEdit`, `EndEdit`, and `CancelEdit`. The `BeginEdit` method should be called before any changes are made to the object (generally via a data entry form), and the `EndEdit` method should be called once those changes have been made and are to be committed.

Implementing the BeginEdit Method

In the `BeginEdit` method you will need to write code to capture the state of the object at that time (i.e., generally all the property values). The easiest way to implement this is using the `MemberwiseClone` method on the object to take a shallow copy of the state of the object:

```
private Product originalState = null; // Member variable
public void BeginEdit()
{
    originalState = this.MemberwiseClone() as Product;
}
```

Implementing the EndEdit Method

You can then discard this state in the `EndEdit` method, because any changes made to the object since the `BeginEdit` method was called will have been accepted (i.e., committed):

```
public void EndEdit()
{
    originalState = null;
}
```

Implementing the CancelEdit Method

In the `CancelEdit` method, you need to restore the original state of the object as it was taken when the `BeginEdit` method was called, copying that state back to the current instance of the object. You can implement this method in one of two ways—either by manually assigning the values from the “backup” object or by using reflection.

■ **Note** The reflection method is more reusable, as it takes a more generic approach, but it is slower to execute.

Manually Assigning Property Values

You can have full control over what property values are reinstated by assigning the values back to the object's properties manually, via the property setters (which will execute any code within that setter, including the property-changed notifications and validation rules—behavior you may or may not want), or directly to each member variable backing the corresponding property being reinstated.

```
public void CancelEdit()
{
    if (originalState != null)
    {
        Name = originalState.Name;
        ProductNumber = originalState.ProductNumber;
        // Etc...

        originalState = null;
    }
}
```

■ **Note** If you choose to bypass the property setters, you may need to revalidate your object after reinstating the property values (validation is discussed later in this chapter).

Assigning Property Values Using Reflection

Alternatively, you can use a generic method that copies the values of all the public properties on the `originalState` object back to the current instance using reflection:

```
public void CancelEdit()
{
    if (originalState != null)
    {
        PropertyInfo[] objectProperties =
            this.GetType().GetProperties(BindingFlags.Public |
                                         BindingFlags.Instance);

        foreach (PropertyInfo propInfo in objectProperties)
        {
            object originalValue = propInfo.GetValue(originalState, null);
            propInfo.SetValue(this, originalValue, null);
        }

        originalState = null;
    }
}
```

■ **Note** This method executes any code in the property setters.

Using the IEditableObject Methods

Both the DataForm and DataGrid have built-in support for calling the BeginEdit/EndEdit/CancelEdit methods on bound objects that implement the IEditableObject interface.

The DataForm control will call the BeginEdit method when changes are first made to the bound object via the bound controls, and commit them by calling the EndEdit method when the user navigates away from the current object, or when they user explicitly commits the changes (by clicking the OK button when the DataForm's AutoCommit property is set to False). As discussed earlier in the chapter, when a DataForm control is bound to an object that implements the IEditableObject interface, a Cancel button will appear in the data entry form. Clicking this Cancel button will call the CancelEdit method on the object to return it to its original state.

The DataGrid also calls the BeginEdit method when changes are first made to a row, and commits them by calling the EndEdit method when the user navigates away from the current row. Pressing the Esc key when editing a row will call the CancelEdit method on the object to return it to the state it was in prior to its being edited.

Adding Calculated Properties to Your Classes

Often you may wish to display a calculated value in your user interface based upon the values of properties on an object. For example, you may have an object/entity representing an invoice line and want to display the total amount of that line (quantity × unit price). Alternatively, you may wish to have a property that displays the full name of a person, built by concatenating their first name and surname (with a separating space).

If you have created the class on the client side (such as a ViewModel), then you can simply add a new property to the class that performs the calculation that your user interface controls can bind to and display the calculated value.

```
public decimal LineTotal
{
    get
    {
        return quantity * unitPrice;
        OnPropertyChanged("LineTotal");
    }
}
```

This example uses the previously described method of notifying any bindings that the property's value has been updated (by implementing the INotifyPropertyChanged interface), so the user interface can update itself accordingly and display its current value. Of course, when you update the values of the properties that represent the quantity or unit price, and notify their bindings that they have changed, the user interface will not know that the value of the LineTotal property has also changed accordingly. Therefore, you should also raise the PropertyChanged event for the LineTotal property in the setters for both the properties representing the line's quantity and unit price.

If you are binding directly to entities exposed from the server via RIA Services, then you can add calculated properties to these entities too. You should *never* modify the code generated by RIA Services, but each of the generated entities is a partial class, so you can simply create a corresponding partial of your own (in the client-side project) that extends the entity and adds the required calculated properties in the same way as was previously demonstrated. Unfortunately, there is still a problem. If you shouldn't modify the code generated by RIA Services (which is where you will find the source properties for the calculation), how will you notify the bindings that the value of the calculated property will have also changed? There are a number of options, but the best method is to extend one of the two partial methods created in an entity for each property (which are called before and after the value of the corresponding property has changed). For example, a `UnitPrice` property on an entity will have two corresponding methods: `OnUnitPriceChanging` (called before the value has changed) and `OnUnitPriceChanged` (called after the value has changed). You can extend the `Changed` method and notify the bindings of the property value change like so:

```
partial void OnListPriceChanged()
{
    OnPropertyChanged(new PropertyChangedEventArgs("LineTotal"));
}
```

■ **Note** Alternatively, you may wish to consider the more elegant method that Emiel Jongerius has written up for managing dependent properties here: www.pochet.net/blog/2010/07/02/inotifypropertychanged-automatic-dependent-property-and-nested-object-support.

Data Validation

One of the most important roles in a typical data entry form is to notify the user of any problems with the data that they have entered. Invalid data in a business system can have major consequences for the business, so effective validation of data is vital to reduce the number of data entry mistakes that make it into the system (it's impossible to catch all errors, but you want to minimize the number of them as much as possible), and ensure that almost all the data entered into the system is correct and accurate. In Chapter 5, we looked at decorating entities and their properties exposed by RIA Services with validation attributes, which are replicated by the RIA Services code generator on the corresponding client-side entities that it creates. However, we need to execute these rules as the data is being entered into the data entry form, and notify the user of any invalid data that they've entered (before submitting the data back to the server).

In this section we'll revisit implementing validation via validation attributes (from a client-side perspective this time), and then look at the means to expose validation errors publicly from your objects and notify the user interface of the errors accordingly. First, however, we'll take a look at the support that Silverlight has for displaying data validation errors to the user.

Displaying Validation Errors

Data entry forms require a friendly and nonintrusive way to notify users when they've entered invalid data, including the reason why the data is invalid (so they can fix it accordingly). Silverlight has very

strong support for displaying validation errors, without much work required on the part of the developer to implement it.

Data Entry Controls

Data entry controls can be notified by their bindings of any problems with their bound properties, and will automatically restyle themselves accordingly (adding a red border) to notify the user as such. In addition, a red tooltip will appear next to the control, displaying the validation error message so that the user knows what the problem is. If a label control is bound to a data entry control that has a validation error, then it will also turn red. Figure 7-22 demonstrates what happens when the user enters invalid data into a TextBox control.



Figure 7-22. A TextBox control displaying a validation error

■ **Note** You can modify how validation errors are displayed by restyling the control's template. Customizing control templates is discussed in Chapter 9.

In order for the control to be notified of related validation errors by its bindings, the bindings need to be configured to listen for and handle validation errors. The appropriate property depends on the method you are using to expose the validation errors to the user interface, and will be discussed later in this chapter as we detail these methods.

■ **Note** If you are binding directly to entities exposed by RIA Services, the method used to notify the user interface of validation errors is already configured by the bindings to listen for and handle these validation errors, reducing the configuration required when creating the binding.

The ValidationSummary Control

The ValidationSummary control can be used to display a list of all the data validation errors associated with an object. It does so automatically via its DataContext property (which is inherited from its container), meaning that you can in many cases simply place it in a view like so:

```
<sdk:ValidationSummary />
```

and it will automatically display any validation errors on the object inherited by its DataContext property, as shown in Figure 7-23.

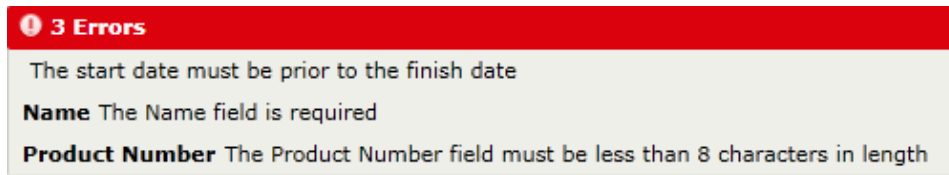


Figure 7-23. A ValidationSummary control

■ **Note** The ValidationSummary control remains hidden until it has validation errors to show, and returns to the hidden state when the errors are fixed.

If you want to take a more manual approach to assigning errors for it to display, you can also manually assign errors to its Errors property in the code-behind (or bind this property to a collection of validation errors that you maintain), like so:

```
var error = new ValidationSummaryItem("The Name field is required", "Name",  
                                     ValidationSummaryItemType.PropertyError,  
                                     null, null);  
validationSummary.Errors.Add(error);
```

The DataForm Control

The DataForm control has a built-in ValidationSummary control in addition to its standard validation error notification behavior. The ValidationSummary control will automatically appear at the bottom of the DataForm control when the bound object reports any data validation errors (as shown in Figure 7-24). The DataForm control also automatically validates the bound entity/object (at both the property and object level) before committing the changes, preventing the user from committing the changes or moving away from the record until the validation errors have been resolved.

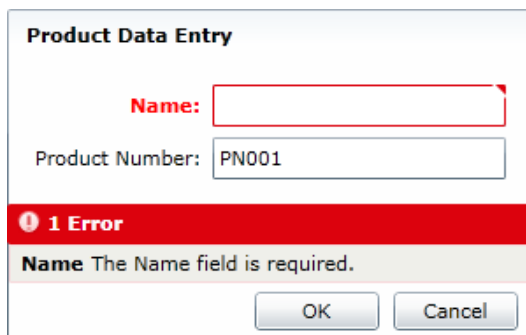


Figure 7-24. A DataForm control displaying a field in error and a validation summary

The DataGrid Control

Like the DataForm control, the DataGrid control also has a ValidationSummary control built in, which appears when the current row has validation errors reported by the bound entity/object. Validation errors will result in the background color of the row changing to a light red. Also, cells with associated validation errors will display a red border and a tooltip explaining the reason for the validation error, when they have the focus (as shown in Figure 7-25).

Name	Product Number
Adjustable Race	AR-5381
Bearing Ball	BA-8327
Headset Ball Bearings	BE-2008

1 Error
Name The Name field is required.

Figure 7-25. A DataGrid control displaying a cell in error and a validation summary

■ **Note** Until all validation errors are fixed on a row, the user will not be able to modify/add any other rows. If the bound object implements the `IEditableObject` interface, then the user can press the Esc key (twice if a field is currently in edit mode, otherwise once) to cancel the changes. The DataGrid control will respond by calling the `CancelEdit` method on that object, and return the object to its state before editing began.

Types of Data Validation

There are three types of data validation that you usually need to perform in business applications:

- *Property-level validation*, in which there are one or more validation rules associated with a property. Property-level validation rules are generally confined to validating the value of a single property, without being reliant on the values of other properties on the object. For example, a typical property-level validation rule may ensure that a numeric value assigned to a property is within a given range. Property-level validation is performed whenever the value of a property is updated (in its setter), and when validating the object as a whole.
- *Object-level validation*, in which each validation rule generally relates to the values of multiple properties on a single object. For example, a typical object-level validation rule may ensure that the end date is after the start date. Object-level validation is performed when committing all the changes to the object (generally calling the `EndEdit` method after implementing the `IEditableObject` interface).

- *Domain-level validation*, in which a full object hierarchy is validated as a whole against a set of validation rules, with validation rules spanning across multiple objects. For example, a typical domain-level validation rule may ensure that when an order is placed, there are enough items in stock to fulfill the order. Domain-level validation is usually performed when committing changes back to the server, and often on the server itself.

We'll primarily be focusing on implementing property- and object-level validation here.

Defining Validation Rules

When you expose entities/objects from the server via RIA Services, you define the property- and object-level validation rules on your server-based entities/objects using validation attributes (as described in Chapter 5), and these are then replicated on the corresponding entities created on the client (by the RIA Services code generator).

The validation rules are replicated on the client such that the data can be validated on the client during data entry, and will be run again on the server when the data is submitted.

When exposing entities from RIA Services, you will generally use validation attributes to specify property- and object-level validation rules, but you can also use these attributes in your own client-side classes (such as your ViewModels, when using the MVVM design pattern discussed in Chapter 12). Simply add a using directive to the `System.ComponentModel.DataAnnotations` namespace, and you can use the validation attributes described in Chapter 5 (e.g., `Required`, `StringLength`, `Range`, `RegularExpression`, etc.) in exactly the same manner. For example:

```
[Required]
[StringLength(50)]
[RegularExpression(@"(?<user>[^\s@]+)@(?<host>.+"))]
public string EmailAddress;

[Range(0, 150)]
public int Age;
```

Note that the predefined validation attributes are all property-level rules. To define object-level validation rules, you will need to create a custom validation attribute that inherits from the base `ValidationAttribute` class (in the same way as was demonstrated in Chapter 5). Repeating that same example, here is a custom validation rule for ensuring that the sell finish date on a product is later than the sell start date:

```
public class SellDatesValidationAttribute : ValidationAttribute
{
    protected override ValidationResult IsValid(object value,
                                                ValidationContext validationContext)
    {
        Product product = value as Product;
        return product.SellEndDate > product.SellStartDate ?
            ValidationResult.Success : new ValidationResult(
                "The sell end date must be greater than the sell start date");
    }
}
```

You can then apply the validation rule to your class like so:

```
[SellDatesValidation]
public class Product
```

Of course, you can also create custom property-level validation attributes in exactly the same way (as was also demonstrated in Chapter 5).

■ **Note** Using validation attributes is not the only way of implementing validation rules in Silverlight. They are the means provided by RIA Services; however, you can also manually implement validation rules in the property setters (for property-level validation rules), and in the object's `EndEdit` method for object-level validation rules (assuming you implement the `IEditableObject` interface), and then expose them to the user interface using the means discussed in the next section.

Exposing Validation Errors to the User Interface

There are actually several means of reporting data validation errors from your entities/objects in Silverlight. Validation errors can be exposed by the following:

- Throwing exceptions
- Implementing the `IDataErrorInfo` interface
- Implementing the `INotifyDataErrorInfo` interface

We'll take a look at each of these in turn, primarily focusing on using them for exposing property-level validation errors (I'll also discuss additional issues you need to be aware of when exposing object-level validation errors).

■ **Note** Entities exposed from the server via RIA Services are automatically configured to report both property- and object-level validation errors to the user interface. These entities inherit from the `Entity` class, which automatically implements the `INotifyDataErrorInfo` interface (described shortly), and reports any data validation errors identified by the validation attributes when properties are updated to the bindings via the means provided by that interface. Therefore, the methods described here are primarily included to demonstrate the different methods that Silverlight 4 provides to enable the user interface to be aware of data validation errors, so you can understand how data validation works behind the scenes, and how to implement data validation in your own client-side classes (such as your `ViewModels`, when using the MVVM design pattern discussed in Chapter 12).

Validation by Exception

Prior to Silverlight 4, the only means of notifying the user interface of property-level data validation errors was to throw an exception in the setter of a property. The `ValidationException` exception (found in the `System.ComponentModel.DataAnnotations` namespace) was generally used for this purpose. An example of validating a property value and notifying the user interface of a data validation error using the exception method is demonstrated here:

```
public string Name
{
    get { return name; }
    set
    {
        name = value;

        // Check validation rule
        if (string.IsNullOrEmpty(value))
            throw new ValidationException("You must enter a name for the product");
    }
}
```

If you set the `ValidatesOnExceptions` property on the binding to `True`, then any exception that was thrown when assigning a value to a property via a control will be treated by the binding as a validation error (i.e., when an exception is thrown in the property setter, or when a type converter throws an exception converting the user input before assigning a value to the bound property). As shown earlier, many controls in Silverlight have excellent support for notifying the user of data validation errors, which you can see by setting the `ValidatesOnExceptions` property on the binding to `True` (its default value is `False`) and raising an exception assigning in the bound property's setter. The result will be that the control is restyled appropriately to notify the user that a validation error has been encountered.

■ **Note** You don't need to worry about setting the `ValidatesOnExceptions` property to `True` when the bindings are within a `DataForm` control—they will still be restyled and the validation summary will appear accordingly when there are validation errors on the bound object.

In order to deal with the validation error in code, the `NotifyOnValidationError` property on the binding must have a value of `True` (which is its default value), and the `ValidatesOnExceptions` property on the binding must be explicitly set to `True`. This will cause the `BindingValidationError` event to be raised on the bound control, which you can then handle. A typical binding for this scenario is demonstrated here:

```
<TextBox Text="{Binding Path=Name, Mode=TwoWay, NotifyOnValidationError=True,
    ValidatesOnExceptions=True}"
    BindingValidationError="NameTextBox BindingValidationError" />
```

You can then handle the `BindingValidationError` event on the bound control like so:

```
private void NameTextBox BindingValidationError(object sender,
```

```

                                ValidationErrorEventArgs e)
{
    if (e.Action == ValidationErrorEventAction.Added)
    {
        // Do something with the error
        string errorMessage = e.Error.ErrorContent.ToString();
    }
}

```

■ **Note** The `NotifyOnValidationError` property is used purely to specify whether the `BindingValidationError` event should be raised on the control, and has no bearing on whether the control should display a validation error to the user (as its name might initially suggest).

The IDataErrorInfo Interface

Historically, the process of throwing data validation errors as exceptions has never been popular with most developers, as conceptually data validation errors are not “exceptional” scenarios—and exceptions should be confined to only being used when something unexpected occurs. Additionally, in the past when debugging your application, every time an exception was thrown due to a data validation error, the execution of the application would break and throw you back into Visual Studio. You could still continue running the application, but it was a frustrating development experience (unless you opened the Exceptions window from the Debug menu in Visual Studio and specified that the application should not break when a `ValidationException` exception is raised). This method was also a problem when populating an entity with data, as exceptions could be raised when a value was assigned to a property that may have been invalid at the time (due to a validation rule that compares the values of two properties), but would become valid as further properties were assigned their values.

WPF provided a better way of handling data validation, in which you could implement the `IDataErrorInfo` interface on your entity, exposing validation errors for an entity/object via two properties, `Error` and `Item`, without requiring exceptions to be thrown. In addition, these properties could report *all* the validation errors on an entity/object instead of just that of the property being updated—a big advantage, especially when updating one property invalidates one or more other properties. Silverlight 4 introduced the `IDataErrorInfo` interface from WPF, which we can now use instead of throwing exceptions. The `Error` property exposes a string identifying the validation status of the entity/object as a whole (i.e., a single validation error message). The `Item` property accepts the name of a property as a parameter, and returns any validation error as a string relating to that property. The following code snippet demonstrates implementing the `IDataErrorInfo` interface in a simple class:

```

public class Product : IDataErrorInfo , INotifyPropertyChanged
{
    private string name;
    private Dictionary<string, string> validationErrors =
        new Dictionary<string, string>();

    public string Name
    {
        get { return name; }
    }
}

```



```
set
{
    name = value;

    // Check validation rule
    if (string.IsNullOrEmpty(value))
    {
        validationErrors["Name"] =
            "You must enter a name for the product";
    }
    else
    {
        if (validationErrors.ContainsKey("Name"))
            validationErrors.Remove("Name");
    }

    RaisePropertyChanged("Name");
}
}

public string Error
{
    get
    {
        string error = "";

        if (validationErrors.Count != 0)
            error = "This object is invalid";

        return error;
    }
}

public string this[string columnName]
{
    get
    {
        string error = "";

        if (validationErrors.ContainsKey(columnName))
            error = validationErrors[columnName];

        return error;
    }
}

private void RaisePropertyChanged(string propertyName)
{
    if (PropertyChanged != null)
        PropertyChanged(this, new PropertyChangedEventArgs(propertyName));
}
```

```
    public event PropertyChangedEventHandler PropertyChanged;
}
```

There is no event included in the `IDataErrorInfo` interface to raise when there is a data validation error, so you may be wondering how the user interface is automatically aware of data validation errors on a bound entity/object. When using the exception method of raising data validation errors, the user interface would be immediately notified of a validation error by the “bubbling up” nature of exceptions. Instead, with this new method, the user interface becomes aware of data validation errors by handling the `PropertyChanged` event on the bound entity/object if it implements the `INotifyPropertyChanged` interface (as was demonstrated in the code snippet). When the value of a property is changed, the entity/object implementing this interface should raise the `PropertyChanged` event, and any bindings to this entity/object will handle this event and be notified of the change accordingly. If the entity/object also implements the `IDataErrorInfo` interface, then the bindings will check the `Error/Item` properties for any data validation errors corresponding to their bound property, and handle them appropriately.

In conjunction with the new `IDataErrorInfo` interface, bindings now have a new property called `ValidatesOnDataErrors` (whose default value is `False`), which will treat any errors exposed by the `IDataErrorInfo` interface as data validation errors. You *must* set this to `True` on your bindings (even in the `DataForm` control), or else the controls will not look for validation errors associated with their bound properties (and thus will be unaware of them, and won’t restyle themselves accordingly).

```
<TextBox Text="{Binding Path=Name, Mode=TwoWay, ValidatesOnDataErrors=True}" />
```

If the `NotifyOnValidationError` property on the binding is set to `True` (in addition to `ValidatesOnDataErrors` being set to `True`), then the `BindingValidationError` event will be raised on any controls bound to the property being updated if that property has a validation error associated with it (which is implemented in the same way as demonstrated for validation exceptions).

■ **Note** If the entity does not implement the `INotifyPropertyChanged` interface, then when using this validation method, the user interface will not know about any validation errors relating to properties that have just been updated. Therefore, controls will not display their current validation state, and the `BindingValidationError` event will not be raised on the bound controls.

The `INotifyDataErrorInfo` Interface

However, not only do we now have the `IDataErrorInfo` interface, but we also have a new interface in Silverlight 4 called `INotifyDataErrorInfo` (which WPF doesn’t have). This interface enables data validation to be performed asynchronously (such as on the server), notifying the user interface of any data validation errors by raising an event. The `IDataErrorInfo` interface exposes one method (`GetErrors`, which returns a collection of all the errors related to a given property), one property (`HasErrors`, which returns a Boolean value indicating whether the entity/object has any errors), and one event (`ErrorsChanged`, which is raised when an error is encountered). When using this interface for validation purposes, instead of checking for a data validation error when the `PropertyChanged` event on the bound entity/object is raised, bindings will wait for the `ErrorsChanged` event to be raised on the bound entity/object to check for data validation errors (using the `GetErrors` method) relating to their bound property. In order for the bindings to listen for the `ErrorsChanged` event, they have a property called

`ValidatesOnNotifyDataErrors` (whose default value is `True`), which will treat any errors exposed by the `INotifyDataErrorInfo` interface as data validation errors.

```
<TextBox Text="{Binding Path=Name, Mode=TwoWay, ValidatesOnNotifyDataErrors=True}"/>
```

■ **Note** The bindings treat this method of validation differently from the other methods described. With those methods, for the bound controls to be aware of validation errors, you have to explicitly set the value of the `ValidatesOnExceptions` and `ValidatesOnDataErrors` properties on the associated bindings to `True`. With this method, the default value of the binding's `ValidatesOnNotifyDataErrors` property is already `True`, meaning that there is no need to explicitly assign the value this property for the bound controls to be aware of validation errors.

If the `NotifyOnValidationError` property on the binding is set to `True` (assuming `ValidatesOnNotifyDataErrors` still has its default value of `True`), then the `BindingValidationError` event will be raised on any controls bound to the property being updated if that property has a validation error associated with it (which is implemented in the same way as demonstrated for validation exceptions).

The following code snippet demonstrates implementing the `INotifyDataErrorInfo` interface in a simple class:

```
public class Product : INotifyDataErrorInfo
{
    private string name;
    private Dictionary<string, List<string>> validationErrors =
        new Dictionary<string, List<string>>();

    public string Name
    {
        get { return name; }
        set
        {
            name = value;

            // Check validation rule
            if (string.IsNullOrEmpty(value))
            {
                validationErrors["Name"] = new List<string>();
                validationErrors["Name"].Add(
                    "You must enter a name for the product");
            }
            else
            {
                if (validationErrors.ContainsKey("Name"))
                    validationErrors.Remove("Name");
            }

            RaiseErrorsChanged("Name");
        }
    }
}
```

```

private void RaiseErrorsChanged(string propertyName)
{
    if (ErrorsChanged != null)
        ErrorsChanged(this, new DataErrorsChangedEventArgs(propertyName));
}

public event EventHandler<DataErrorsChangedEventArgs> ErrorsChanged;

public System.Collections.IEnumerable GetErrors(string propertyName)
{
    List<string> errors = null;

    if (validationErrors.ContainsKey(propertyName))
        errors = validationErrors[propertyName];

    return errors;
}

public bool HasErrors
{
    get { return validationErrors.Count != 0; }
}
}

```

Deciding Which Method to Use

As you can see, there are now three different methods for exposing data validation errors from your entities/objects, any of which the validation attributes can take advantage of (as will be demonstrated shortly). I've already discussed the major disadvantages of using the exception-based method of data validation, but you may be wondering which of the other two methods (the *IDataErrorInfo* interface or the *INotifyDataErrorInfo* interface) you should use. In general, you are better off implementing the *INotifyDataErrorInfo* interface, as it can handle asynchronous validation, and you can also store multiple validation errors for each property. Although you could technically use all three methods together, it is best to standardize on a single method for data validation to avoid confusion.

Notifying the User Interface of Object-Level Validation Errors

Similar techniques to implementing property-level validation can also be used for implementing object-level validation (where you want to define a validation rule that encompasses multiple properties on an object). However, instead of executing the validation rules in property setters (as you do for property-level validation), you need to create a method in your class that will execute all the object-level validation rules and report any failures to the user interface. This method is usually best called in the *EndEdit* method in your class (assuming you implement the *IEditableObject* interface).

The *EndEdit* method is automatically called by the *DataForm* and *DataGrid* controls when attempting to commit the changes to the object, and you can use this opportunity to validate the object as a whole. You can execute the rules to validate the object, and then add these failed object validation rules to the list of failed property rules that you are maintaining (and exposing via the means provided when implementing the *IDataErrorInfo* or *INotifyDataErrorInfo* interface).

When returning these object-level validation errors via the means provided by the interfaces, even though the validations are at the object level, you still need to associate the validation errors with properties on the object; otherwise, the errors will not be displayed in the user interface. Controls only display validation errors for properties that they are bound to, and the `ValidationSummary` control (discussed earlier) only displays property- and object-level validation errors for the object assigned to (or inherited by) its `DataContext` property.

Therefore, you should either associate any object-level validation error with all the properties that the rule involves, or simply assign the validation error to one of them instead. Once you've done so, you will also have to notify the user interface of these validation errors. When implementing the `IDataErrorInfo` interface, you will need to raise the `PropertyChanged` event (available by implementing the `INotifyPropertyChanged` interface) for each property involved in the object-level validation rule. When implementing the `INotifyDataErrorInfo` interface, you will need to raise the `ErrorsChanged` event for each property involved in the object-level validation rule.

■ **Note** Neither the `DataForm` nor `DataGrid` controls display validation errors for properties on bound objects that their fields/columns do not bind to, even when exposed via the means provided by the `IDataErrorInfo` or `INotifyDataErrorInfo` interface after the object is validated. However, these controls *do* display validation errors for properties they do not bind to if the corresponding validation rules are implemented as validation attributes.

The Validator Class

Let's take a look at how to validate an entity/object according to its validation attributes. If you create a new class, apply validation attributes to its properties, and then bind it to a `DataForm` control, when you edit the bound fields, they will not be validated until you attempt to commit the changes to the object. However, entities exposed from the server via RIA Services will validate immediately when data is entered in the `DataForm` fields. The reason is that the attributes are simply metadata, and you need to actually execute the validation rule(s) relating to a property in its setter, and notify the user interface yourself if any rules are broken.

As previously noted, entities exposed from the server via RIA Services inherit from the `Entity` class. This class automatically executes any validation rules corresponding to a property when it is updated (i.e., in its setter), and reports any validation errors to the bindings via the `INotifyDataErrorInfo` interface. However, when you decorate your own class's properties with validation attributes, you will need to perform this task yourself—simply decorating the properties with the validation rules is not enough.

To implement the same behavior as the RIA Services entities in your own client-side objects, you need to use the `Validator` class (which you will find in the `System.ComponentModel.DataAnnotations` namespace) to explicitly execute the validation rule attributes decorating a property or an object. The `Validator` class is a static class, with a number of methods that you can use to validate properties and objects decorated with validation attributes.

To implement interactive validation behavior on your own client-side objects, you need to use the methods on this class to validate the new value of a property (in its setter), and to validate the object as a whole in its `EndEdit` method (when implementing the `IEditableObject` interface).

■ **Note** The reason that the bound fields in a `DataForm` control are validated when you attempt to commit the changes is that the `DataForm` control actually ensures the bound entity/object is valid before calling its `EndEdit` method (if it implements the `IEditableObject` interface), using the methods that will be discussed shortly to ensure that the object is valid. It doesn't automatically execute the property-level validation rule attributes, however (their validation rules must be executed within the property setters, as described following). Somewhat inconsistently, however, the `DataGrid` does automatically execute these validation rules.

Validating Properties

To validate the new value for a property (in its setter), you can use either the `ValidateProperty` method or the `TryValidateProperty` method. When you call the `ValidateProperty` method, if the given value for a property fails a validation rule, then it will raise a `ValidationException` exception. When you call the `TryValidateProperty` method, it will simply return a `Boolean` value (indicating whether the new value will be valid) and a collection of the failed rules (i.e., it won't raise an exception). This is the preferred method, as it then allows you to handle the validation errors using the means provided by implementing either the `IDataErrorInfo` or `INotifyDataErrorInfo` interface in your class.

Here's the process to use the property validation methods on the `Validator` object:

1. Create a `ValidationContext` object (whose constructor you pass the object to validate, any services required by the validators, and any other validation state information that you want to pass to them).
2. Assign it the name of the property/member to be validated on the object to validate.
3. Pass the `ValidationContext` object to the appropriate method (`ValidateProperty` or `TryValidateProperty`) on the `Validator` object, along with the value you are validating (plus an empty collection that the method will put the results of the failed validation rules in if you are calling the `TryValidateProperty` method).

■ **Note** As you may have noticed from the example of creating a custom validation attribute earlier in this chapter, the `ValidationContext` object will be passed into each validation attribute on the property, which the validation attributes can then use as they see fit.

An example of this process is demonstrated following:

```
[Required]
public string Name
{
    get { return name; }
    set
```

```

{
    name = value;

    List<ValidationResult> validationResults = new List<ValidationResult>();

    ValidationContext validationContext =
        new ValidationContext(this, null, null);
    validationContext.MemberName = "Name";

    bool isValid = Validator.TryValidateProperty(value, validationContext,
                                                validationResults);

    if (isValid)
    {
        // Remove any validation errors associated with this property
        // (depending on which interface you are implementing)
    }
    else
    {
        // Add the errors in the validationResults variable to your list
        // of errors (depending on which interface you are implementing)
    }

    if (PropertyChanged != null)
        PropertyChanged(this, new PropertyChangedEventArgs("Name"));

    Raise any other required events (e.g., ErrorsChanged)
}
}

```

Validating Objects

You can also use the `Validator` class to validate the object as a whole, based upon its validation attributes using the `ValidateObject` or `TryValidateObject` method (these methods are fairly similar to their property validation counterparts). These methods validate the object using the validation attributes applied to the object itself, and check that any properties with the `Required` attribute have a value. You'll generally validate the object in its `EndEdit` method (when implementing the `IEditableObject` interface).

Note that by default, the `ValidateObject` and `TryValidateObject` methods do not validate the object based upon any of its property-level validation attributes other than the `Required` attribute (i.e., `StringLength`, `Range`, etc.). To include these property-level validations in the object-level validation, you will need to use the overload of the `ValidateObject` and `TryValidateObject` methods that includes the `validateAllProperties` parameter (which you should set to `True`, as demonstrated in the example following).

An example of performing object-level validation that also validates all of the properties on the object is demonstrated following:

```

List<ValidationResult> validationResults = new List<ValidationResult>();
ValidationContext validationContext = new ValidationContext(this, null, null);

```

```
bool isValid = Validator.TryValidateObject(this, validationContext,
                                         validationResults, true);
```

■ **Note** Although these examples have demonstrated validating an object or one of its properties from within the object itself, you can just as easily initiate the validation outside the object (in much the same way as the `DataForm` and `DataGrid` controls automatically do) using exactly the same methods.

Customizing Validation Attribute Error Messages

Worth noting is that the default error messages for validation errors are not always particularly user-friendly when specifying the validation rules using validation attributes. By default, they refer to the property that fails the validation rule by name, which will often be different from the label corresponding to that field in the user interface (especially if the application is localized). If you decorate the property with the `Display` attribute, assigning a friendly name to its `Name` property, then the validation error message will use the value of that property instead of the property name.

Alternatively, you can assign a completely different validation error message by assigning a value to the `ErrorMessage` property of the validation attribute, like so:

```
[Required(ErrorMessage="Please enter a name for the product")]
public string Name;
```

If you are storing your strings in resource files, you can use those instead by setting the `ResourceType` and `ResourceName` properties on the validation attributes instead. For example:

```
[Required(ResourceType=typeof(ProductResources), Name="ProductNameRequiredError")]
public string Name;
```

Submitting Changes to the Server

Throughout this chapter you've learned how to create data entry forms to enable the user to edit data, in addition to adding items to and deleting items from a collection. Eventually, you'll want to send any changes made to this data (usually by the user via the data entry forms) back to the server again. Let's take a look at how to do this using RIA Services.

Change-Tracking

When you retrieve a collection from the server via RIA Services, changes made to that collection (add/edit/delete) will automatically be tracked by the instance of the domain context that was used to retrieve the data (as a `changeset`). Therefore, when you submit the collection back to the server, only the entities that were changed will be sent (reducing the amount of data traffic sent from the client to the server, and ensuring that unchanged entities will not raise concurrency conflicts).

In order to enable change-tracking, you must retain the instance of the domain context object that was used to retrieve the data from the server, and use that same instance to submit the changes back to

the server. The `DomainDataSource` control manages this, but it's something to remember if you are interacting directly with the domain context in code.

■ **Note** You can get the domain context used by a `DomainDataSource` control from its `DomainContext` property.

You can get an `EntityChangeSet` object for a domain context, containing separate collections for the added entities, modified entities, and the removed entities. The following code demonstrates getting the changeset from a domain context named `productDomainContext`:

```
EntityChangeSet changeset = productDomainContext.EntityContainer.GetChanges();
```

Submitting Changes via the `DomainDataSource` Control

Submitting changes to data retrieved using a `DomainDataSource` control is simply a case of calling the `DomainDataSource` control's `SubmitChanges` method. Unfortunately, despite everything demonstrated thus far with regard to the `DomainDataSource` control's ability to be performed declaratively, you need to turn to code in order to call this method.

■ **Note** You could implement a behavior to call this method in a more declarative fashion. Behaviors will be discussed in Chapter 10.

As discussed back in Chapter 5, that there are no insert/update/delete domain operations created on the domain context—instead, you simply call the `SubmitChanges` method on the domain context, which passes the changeset back to the server. RIA Services will then automatically handle calling the appropriate insert/update/delete domain operations on your domain service.

The `SubmittingChanges` event on the `DomainDataSource` control is raised before the changes are submitted (giving you the chance to cancel the operation), and the `SubmittedChanges` event is raised once the action has been performed on the server (where you can find the results).

Submitting Changes via a Domain Context

Submitting the changes to data retrieved directly using a domain context is much the same as doing so on the `DomainDataSource` control. The domain context also has a `SubmitChanges` method, which passes the changeset associated with the domain context back to the server.

However, the domain context doesn't have a `SubmittingChanges` or a `SubmittedChanges` event. Instead, calling the `SubmitChanges` method returns a `SubmitOperation` object, which has a `Completed` event. The following code demonstrates calling the `SubmitChanges` method on a domain context instance (which, as discussed, should have been instantiated earlier to load the data—let's say it was named `productDataContext`), and handles the `Completed` event that will be raised when the operation is complete:

```
SubmitOperation submitOperation = productDataContext.SubmitChanges();
submitOperation.Completed += new EventHandler(ProductSubmitOperation Completed);
```

You can then handle the Completed event and check/act upon the results of the operation:

```
private void ProductSubmitOperation Completed(object sender, EventArgs e)
{
    SubmitOperation submitOperation = sender as SubmitOperation;
    // Use submitOperation object to check results of submit operation
}
```

Handling Errors

If any exceptions are thrown on the server when submitting data, or the client cannot reach the server at all, then these will automatically be thrown as exceptions on the client. You can determine on the client whether any errors occurred when submitting the changes (and prevent the exceptions from actually being thrown) by handling the SubmittedChanged event (if using a DomainDataSource control), or by handling the Completed event of a SubmitOperation object (if submitting via a domain context). You can then insert your own behavior for how errors should be handled. Both use a slightly different means of passing you the results of the operation, so we'll take a look at them separately.

As discussed previously, when you submit changes via the DomainDataSource control, you have a SubmittedChanges event that you can handle. The event handler method has a SubmittedChangesEventArgs object passed in as a parameter that enables you to check the results of the operation. This object has a HasError property whose value will be true when an exception is thrown on the server. You can get the error message that was thrown from the Error property, which is also on this object.

```
bool hasError = e.HasError;
Exception error = e.Error;
```

To prevent the exception being thrown on the client (and consequently bubbling up to be caught by the event handler for the UnhandledException event on the App object, which you'll find in the App.xaml.cs file in your project), you can simply call the MarkErrorAsHandled method on the SubmittedChangesEventArgs object, like so:

```
e.MarkErrorAsHandled();
```

When you submit changes via a domain context, however, the Completed event of the SubmitOperation object does not use the same SubmittedChangesEventArgs object as the DomainDataSource control's SubmittedChanges event did. Instead, it passes in the SubmitOperation object via the sender parameter (as was demonstrated earlier), which you can get the results of the operation from when you cast it to a SubmitOperation. The SubmitOperation object also has a HasError property and an Error property to provide information as to anything that went wrong when submitting the changes, plus it has a MarkErrorAsHandled method that you can call to prevent an exception from being thrown.

```
SubmitOperation submitOperation = sender as SubmitOperation;

if (submitOperation.HasError)
{
```

```
// Do something with the error, such as notify the user
submitOperation.MarkErrorAsHandled();
}
```

The exception is returned simply as an `Exception` type, but you can cast it to a `DomainOperationException` to get more useful information about the exception. Of most interest is the `Status` property that it exposes. This will tell you what type of exception occurred (using the `OperationErrorStatus` enumeration) so you can handle it accordingly. The values in this enumeration of most interest are:

- **ServerError:** An exception was raised on the server, or the application could not reach the server.
- **ValidationFailed:** The data sent to the server has failed the validation rules (which are rerun on the server).
- **Conflicts:** The data being updated/deleted has been changed on the server since it was originally retrieved, causing a concurrency violation). This is discussed in the next section.
- **Unauthorized:** The user does not have permission to perform the operation. (Restricting access to domain operations based upon the user's role is discussed in Chapter 8.)

You can check this value to determine the category of the error and handle it accordingly. The following example demonstrates a structure for handling the errors in the `Submitted` event handler of the `DomainDataSource` control (change `e` to the instance of the `SubmitOperation` if using the domain context method):

```
if (e.HasError)
{
    DomainOperationException error = e.Error as DomainOperationException;

    switch (error.Status)
    {
        case OperationErrorStatus.Conflicts:
            // Handle concurrency violations
            break;
        case OperationErrorStatus.ServerError:
            // Handle server errors
            break;
        case OperationErrorStatus.Unauthorized:
            // Handle unauthorized domain operation access
            break;
        case OperationErrorStatus.ValidationFailed:
            // Handle validation rule failures
            break;
        default:
            // Handle other possible statuses
            break;
    }
}
```

■ **Note** You can test your error handling by simply adding a line of code that throws an exception to your insert/update/delete operations in your domain service.

Handling Concurrency Violations

In Chapter 5, we looked at how to check for concurrency violations when performing update/delete operations. As shown in the previous section, you are notified of concurrency violations found on the server when you handle the Submitted/Completed event (depending on whether you are using a DomainDataSource control or the domain context directly) and check for an exception representing a concurrency conflict in the results. This exception will be a DomainOperationException with a status of Conflicts (as shown previously).

If conflicts have been reported, you can find out which entities failed the concurrency check from the EntitiesInError property of the SubmittedChangesEventArgs/SubmitOperation object (that was passed into the Submitted/Completed event). You can then compare the different versions of each of these entities (the current, original, and store versions) that you can get from the entity's EntityConflict property, and handle the conflict accordingly.

Options you have for handling a concurrency conflict include

- Discarding the user's update and getting them to reenter the data (an easy solution, but not very user-friendly)
- Simply overwriting the values on the server with the user's update in a last-update-wins strategy (potentially resulting in lost data)
- Displaying both the current version (i.e., the version that was sent to the server by the user) and the store version (i.e., the version in the database) to the user and getting them to manually merge the changes
- Attempting to automatically merge the changes programmatically

The names of the properties that failed the concurrency checks are available from the PropertyNames property of the EntityConflict object.

The following example demonstrates a structure for automatically handling the conflicts in the Submitted event handler of the DomainDataSource control (change e to the instance of the SubmitOperation if using the domain context method):

```
if (e.HasError)
{
    DomainOperationException error = e.Error as DomainOperationException;

    if (error.Status == OperationErrorStatus.Conflicts)
    {
        // Loop through the entities with concurrency violations
        foreach (Product product in e.EntitiesInError)
        {
            EntityConflict conflictinfo = product.EntityConflict;
            Product currentProduct = conflictinfo.CurrentEntity as Product;
            Product originalProduct = conflictinfo.OriginalEntity as Product;
            Product storeProduct = conflictinfo.StoreEntity as Product;
```

```
// Handle any conflicts automatically (if you wish)
// You can get the names of the properties whose value has changed
// from the conflictinfo.PropertyNames property

// Force this user's version to overwrite the version on the server
product.EntityConflict.Resolve();
    }
}
```

After resolving the changes, remember to resubmit them back to the server.

■ **Note** Instead of handling the conflicts manually, you may like to take a look at using the generic method Sergey Klementiev created for handling concurrency errors via RIA Services, which he blogged about here: <http://sklementiev.blogspot.com/2010/03/wcf-ria-and-concurrency.html>.

Summary

We've now completed the round-trip of data—with data exposed by the server (covered in Chapter 5), consumed by the Silverlight client (covered in Chapter 6), and modified with the changes submitted back to the server (covered in this chapter). We now have essentially developed an end-to-end Silverlight application. We will build upon this core functionality and add supporting features in the upcoming chapters, including securing our application and styling it to make it more presentable.

CHAPTER 8



Securing Your Application

In any business application, securing data and operations appropriately by restricting their access to approved users should be a significant factor in their design. Not doing so effectively opens the business to threats from snooping competitors, and even sabotage. Therefore, designing effective security into your application is extremely important.

This generally involves validating users' identities, tracking them and their assigned roles, and restricting their access to approved data and operations accordingly.

However, there are additional security-related considerations you should take into account when designing your Silverlight application. For example, if the application is being run outside of the corporate network, then you are at risk of your data being sniffed between the server and the client. Also, exposing your services publicly leaves you at risk of someone attempting to access them directly without going through the Silverlight client. Another concern is that your Silverlight application will be available publicly for download, enabling competitors to decompile it and obtain access to potentially sensitive intellectual property such as algorithms and other assets.

In this chapter, we will take a look at these security-related issues, including how to implement restrictions based the users and their roles, and the various preventative measures you can take to stop your data, applications, and business from being compromised.

Implementing Server-Side Security Using RIA Services

The following are different types of security functions that you will generally need to consider as a part of your application design and implement in the services that expose data and operations from the server:

- Maintaining a list of users, their authentication details, and their roles within the system.
- Validating that users are permitted to access the application (known as *user authentication*). The most common technique for doing this is to require the user to enter a username and password in a login screen (although integration with the user's Windows profile is another method).
- Preventing users from calling specified operations until they are authenticated.
- Preventing users from calling operations that their roles do not permit them to call (including CRUD operations, invoke operations, and custom operations).
- Providing access only to a subset of the data in the database based upon the user's role or ID (e.g., users can view only notes they've entered into the system, not those created by other users).

- Deciding upon what part of a given set of data users are allowed to update and delete based on their IDs or roles. For example, a user may be able to view the notes entered into the system by all the users in the system, but not update or delete notes created by other users unless he or she is a manager (i.e., in the Manager role).

These are the most important concepts to consider when designing a system, and RIA Services (in conjunction with the ASP.NET Membership API) helps provide an easy framework to implement most of them. RIA Services makes it very easy to design security into your system, and if you have used the Silverlight Business Application template as the base for your project, then you will find that user authentication and registration functionality has already been included. Let's take a look at how to handle each of the preceding points on the server side with RIA Services.

The ASP.NET Membership API

As previously stated, if you've used the Silverlight Business Application project template as the base for your project, you will already have the basic infrastructure in place to handle authentication of users (and registration of new users, which can be disabled if not required or wanted). This template uses the ASP.NET Membership API to provide the user authentication and authorization features. You will probably be familiar with ASP.NET Membership if you have previously done any ASP.NET development, and here the same infrastructure is leveraged to provide user authentication and authorization to your Silverlight applications.

ASP.NET Membership's provider model is extremely flexible, with a number of extensibility points that allow you to either use the default provider or plug in your own (with the provider configuration managed through the web project's `web.config` file). This enables you to work with a single interface without worrying about the actual underlying implementation.

The default ASP.NET Membership authentication provider implemented in the Silverlight Business Application project template is the Forms Authentication provider. As in ASP.NET projects, this maintains your authentication status by storing a temporary cookie on the client (containing authentication information) after the user is authenticated, and passing that to the server with each request. The server automatically handles processing this cookie and populating the User identity for the current HTTP context with information about users and their roles.

The other commonly used authentication provider is the Windows Authentication provider. This authenticates users using their Windows accounts and impersonates them on the server. This can be a good alternative to Forms Authentication if your Silverlight application will only be run within the corporate network, saving the user from needing to log into the application. However, if the application will be run outside the corporate network, then Forms Authentication is the best choice.

■ **Note** To use Windows Authentication, you will need to use IIS to debug your application, as the Visual Studio development server does not support this mode. You will need to disable anonymous access on the application in IIS, and enable one of the Windows-based authentication modes (such as Windows Integrated Authentication).

You can specify which authentication provider you wish to use in the web project's `web.config` file, by setting the `mode` attribute of the authentication element to `Forms` or `Windows`:


```
<authentication mode="Forms" />
```

In addition to the authentication providers are membership, role, and profile providers. These providers are used to handle persisting the user, role, and profile data (generally in conjunction with the Forms Authentication provider). The default providers are

- *Membership*: The SQL Membership Provider (SqlMembershipProvider)
- *Role*: The SQL Role Provider (SqlRoleProvider)
- *Profile*: The SQL Profile Provider (SqlProfileProvider)

These providers store users, roles, and profile data in a SQL Server database. These providers and their default options are configured in the `machine.config` file (found in the `C:\Windows\Microsoft.NET\Framework\<.NET Framework Version>\Config` folder), but can be overridden in the `web.config` file in your web project (which you will need to do when you want to specify a database to where the settings will be persisted, as will be demonstrated in the next section).

These default SQL Server providers write to their own database table structure; however, you may be working with an existing database with its own tables storing user/role/profile data. Alternatively, you may be working with a database other than a SQL Server database, or you may not be using a database to store this data at all. In these cases, you will have to create your own custom user/role/profile providers, and plug them into the ASP.NET Membership model by configuring them in the web project's `web.config` file. This won't be covered here in this chapter, but creating a custom membership/role/profile provider is easy—you simply create a class that inherits from the corresponding base provider class, and override its methods and properties to add your own functionality.

This chapter will demonstrate using the Forms Authentication mode with the SQL Membership/Role/Profile providers.

■ **Note** You can find further information on ASP.NET Membership here: <http://msdn.microsoft.com/en-us/library/91f66yxt.aspx>.

Configuring Your Database for Forms Authentication

Assuming your application is still configured to use Forms Authentication (the default mode), then you can run the project created from the Silverlight Business Application project template and register yourself as a new user (all the user interface required to do so is already implemented by the project template).

This all works because it automatically creates a new SQL Server database called `ASPNETDB.MDF` for you in the `App Data` folder of the web project, and creates the required ASP.NET Membership tables. It then attaches this database to SQL Server or SQL Server Express as a user instance. However, you will most likely want to configure it to store the user data in your own database instead.

To do so, you will need to modify the `web.config` file—creating a connection string to the database and pointing the `defaultProvider` property of the `authentication`, `roleManager`, and `profile` sections to the name of that connection.

For the example being used throughout this book, we want to store the membership data in our AdventureWorks2008 database. Therefore, you would need to add a new connection string to your web project's web.config file:

```
<connectionStrings>
  <add name="AdventureWorksDatabase"
        connectionString="Data Source=.\SQL2008;Database=AdventureWorks2008;
                           Integrated Security=true"
        providerName="System.Data.SqlClient" />
</connectionStrings>
```

■ **Note** You cannot use the connection string used by your Entity Framework model, as it is formatted differently than normal database connection strings and cannot be understood by the database provider. This unfortunately means that you need to duplicate your connection strings in your web.config file and maintain them separately.

Then you can add the membership/role/profile provider's configuration to the <system.web> section of your web.config file, with their connectionStringName properties referencing the name given to the connection string:

```
<membership defaultProvider="AWMembershipProvider">
  <providers>
    <clear />
    <add name="AWMembershipProvider"
          connectionStringName="AdventureWorksDatabase"
          applicationName="/AdventureWorks"
          type="System.Web.Security.SqlMembershipProvider,
              System.Web, Version=2.0.0.0, Culture=neutral,
              PublicKeyToken=b03f5f7f11d50a3a"
          enablePasswordRetrieval="false"
          enablePasswordReset="true"
          requiresQuestionAndAnswer="false"
          requiresUniqueEmail="false"
          passwordFormat="Hashed"
          maxInvalidPasswordAttempts="5"
          minRequiredPasswordLength="4"
          minRequiredNonalphanumericCharacters="0"
          passwordAttemptWindow="10"
          passwordStrengthRegularExpression="" />
  </providers>
</membership>

<roleManager enabled="true" defaultProvider="AWRoleProvider">
  <providers>
    <clear />
    <add name="AWRoleProvider"
          connectionStringName="AdventureWorksDatabase"
          applicationName="/AdventureWorks"
```

```

        type="System.Web.Security.SqlRoleProvider,
            System.Web, Version=2.0.0.0, Culture=neutral,
            PublicKeyToken=b03f5f7f11d50a3a" />
    </providers>
</roleManager>

<profile defaultProvider="AwProfileProvider">
  <providers>
    <add name="AwProfileProvider"
        connectionStringName="AdventureWorksDatabase"
        applicationName="/AdventureWorks"
        type="System.Web.Profile.SqlProfileProvider,
            System.Web, Version=2.0.0.0, Culture=neutral,
            PublicKeyToken=b03f5f7f11d50a3a" />
  </providers>
</profile>

```

■ **Note** We are clearing the existing providers in each section using the `<clear />` element (which would have been inherited from the `machine.config` file) before adding our own provider. We give each provider a name, and specify that it is the default provider.

You've now got the providers pointing toward the AdventureWorks2008 database, but they won't work yet, as the AdventureWorks2008 database doesn't have the required ASP.NET Membership tables in its schema, so we'll need to add those and configure our security settings.

Using the command-line (or Run option in the Start menu), run
 C:\Windows\Microsoft.NET\Framework\v4.0.30319\aspnet_regsql.exe.

This will open a wizard that you can point toward your AdventureWorks2008 database, and it will create the required tables, views, and stored procedures. Then, using the ASP.NET Configuration tool for the web application (the rightmost icon in the Solution Explorer window when the web project in your solution is selected), you can set up users and roles for the application. Alternatively, simply run the project and use the registration feature to add yourself as a user.

Authentication

The `AuthenticationService` (located in the `Services` folder) is a standard domain service that enables a user to be logged in and out, and their profile to be retrieved and saved all from the client. It inherits from the `AuthenticationBase` class, which contains a set of default authentication-related functions—including `Login`, `Logout`, `GetUser`, and `UpdateUser`, which make use of the corresponding functionality in the membership provider configured for the application. The `AuthenticationService` class is therefore very simple (because all the functionality is implemented in the base class), as shown in the following code:

```

[EnableClientAccess]
public class AuthenticationService : AuthenticationBase<User> { }

```

Nothing else needs to be added—simply defining this domain service in your web project will allow user authentication to be enabled from the client (including returning the user details and roles to the client) and maintain the user's authentication status by creating a temporary Forms Authentication cookie to be passed between the client and the server with each server request.

■ **Note** This cookie has a default timeout of 30 minutes. That means that if there is no communication with the server for longer than 30 minutes, then the cookie will expire and the user will be logged out. The actual implementation of how the expiry works is a little more complicated, but this is the simple version (you can find the more complicated behavior described here: <http://support.microsoft.com/kb/910439>). You can increase or decrease this value if required (in the `web.config` file), but it's not recommended to make it too big. The longer the lifetime of the cookie, the more chance someone would have to steal it and impersonate the user. The short lifespan of the cookie is essentially a security measure, and should only be changed after careful consideration.

Requiring Authentication

By default, the operations that you've created in your domain service can be called by any anonymous user who can get to your service. Ideally, as a first step, you'll want to limit access to these operations (perhaps some, perhaps all) to only authenticated users of your system.

■ **Note** Remember, limiting access to the service calls on the client doesn't mean that those services are secure. Hackers (or more appropriately, crackers) that want to obtain your sensitive data will attempt to circumvent client limitations by directly connecting to the service on the server to see what they can obtain from it. Therefore, you shouldn't make any data available from the service unless the user is properly authenticated and permitted access to that data. In addition, you shouldn't permit the user to perform any operations (like insert/update/delete) on the data without the appropriate permissions either.

To restrict access to all the operations in a domain service to only authenticated users, decorate the domain service class with the `RequiresAuthentication` attribute:

```
[EnableClientAccess]
[RequiresAuthentication]
public class ProductService :
    LinqToEntitiesDomainService<AdventureWorks2008Entities>
```

If the user is not authenticated and calls one of these operations, the service will raise a `DomainOperationException` stating that access has been denied to the operation, which in turn will be raised on the client.

In some instances you may want to select which operations in your domain service require authentication, leaving others available to anonymous users. This is possible by decorating only the operations that require the user to be authenticated with the `RequiresAuthentication` attribute.

Role-Based Operation Access Restrictions

Perhaps the user is authenticated, but you want to further restrict access to a domain service or an operation based on the user's role (known as authorization). In this case you can simply decorate the domain service class or operation with the `RequiresRole` attribute, and include which roles are permitted to call the operation. The following example shows how to limit access to the operation to only users who are in the Administrators role or the Managers role:

```
[RequiresRole("Administrators", "Managers")]  
public IQueryable<Product> GetProducts()
```

■ **Note** By default, any users registering themselves through the Silverlight Business Application project will be added to the Registered Users role. To create additional roles you can use the ASP.NET Configuration tool for the web application (the rightmost icon in the Solution Explorer window); however, you will need to create the users through this same interface in order to assign them to those roles, or add the ability for roles to be specified when registering a user through the Silverlight project. See the “User Registration” section for more information.

Including the role names as strings in multiple places is not ideal, so it's recommended that you define some constants with the role names and use these instead. Alternatively, you can create your own authorization attribute by inheriting from the `AuthorizationAttribute` class, overriding the `IsAuthorized` method and inserting your own authorization logic, like so:

```
public class GTManagerRoleAttribute : AuthorizationAttribute  
{  
    protected override AuthorizationResult IsAuthorized(  
        System.Security.Principal.IPrincipal principal,  
        AuthorizationContext authorizationContext)  
    {  
        if (principal.IsInRole("Administrators") || principal.IsInRole("Mangers"))  
            return AuthorizationResult.Allowed;  
        else  
            return new AuthorizationResult(  
                "You do not have sufficient rights to perform this operation.");  
    }  
}
```

You can then decorate the operation with this attribute instead of the `RequiresRole` attribute:

```
[GTManagerRole]  
public IQueryable<Product> GetProducts()
```

If you want to restrict access to all the domain operations in a domain service, then you can simply decorate the domain service with the `RequiresRole` attribute (or your own custom attribute) instead.

■ **Note** You may also need to implement per-user permission overrides (where the user can be permitted access to certain functions that their roles do not allow them, but without assigning an additional role with all its permissions)—these permissions can be managed in the user object as properties, with the values stored in the user profile (as demonstrated in the “Exposing Custom User Profile Data” section later in this chapter). You will also need to create your own authorization attribute to handle this scenario.

For insert, update, and delete operations, you might need to reference the instance of the entity being passed into the operation in your authorization attribute. For example, you might want to ensure that the user is permitted to update the given entity. Note that there is an `AuthorizationContext` object being passed into the `IsAuthorized` method that you are overriding. The `Instance` property on that object will have the entity being inserted/updated/deleted assigned to it:

```
Product product = authorizationContext.Instance as Product;
```

If you need to pass additional data into the authorization attributes to use as part of the authorization process, you can override the `Initialize` method of the domain service class, create an instance of the `AuthorizationContext` class, and assign the data to its `Items` property (which is a `Dictionary` that you can assign any number of keyed values to). Assign this to the `AuthorizationContext` property of the domain service, and this object will then be passed to any authorization attributes used in the domain service. For example:

```
public override void Initialize(DomainServiceContext context)
{
    base.Initialize(context);

    Dictionary<string, bool> customUserPermissions = new Dictionary<string, bool>();
    // Load user permissions

    AuthorizationContext = new AuthorizationContext(context);
    AuthorizationContext.Items["CustomUserPermissions"] = customUserPermissions;
}
```

You can then get this object from the `AuthorizationContext` object passed into the `IsAuthorized` method of your custom authorization attributes:

```
Dictionary<string, bool> customUserPermissions =
    authorizationContext.Items["CustomUserPermissions"] as Dictionary<string, bool>;
```

Returning a Subset of Data Based on the User's ID or Role

This is more of an implementation issue than one that can be provided in the RIA Services framework. However, here is an example of how to obtain the user's credentials so you can filter the data being queried accordingly.

The credentials you are likely to require are the user's ID, username, and/or roles. The identity of the user is exposed by the `ServiceContext` object on the `DomainService` base class. You will find a `User` principle object exposed as a property on the `ServiceContext` object. From here, you can get the user's username from the `ServiceContext.User.Identity.Name` property, and determine whether the user is in a specific role using the `ServiceContext.User.IsInRole` method.

```
string userName = ServiceContext.User.Identity.Name;
bool isAdministrator = ServiceContext.User.IsInRole("Administrators");
```

To get additional details about the user (such as the user's ID and roles), you will need to turn to ASP.NET Membership's functions to get those details for the user from the database.

```
// Note that you will need a using statement to System.Web.Security
MembershipUser user = Membership.GetUser(); // User object
Guid userID = (Guid)user.ProviderUserKey; // User ID
string[] userRoles = Roles.GetRolesForUser(); // User roles
```

You can now use these details to filter the data being queried before returning the data to the client.

In addition, you may have a requirement that entities can be passed back to users, but they may not be able to view (or edit) all the fields on that entity (e.g., they may be allowed to view a customer entity, but not permitted to see the credit card number of that customer). This should be handled on the server so that the fields that the user doesn't have access to aren't sent to the client. In this scenario it is best to use POCO (plain-old CLR object) types when returning data, and only populate the fields of each object that the user is permitted access to.

Permitting Update/Delete Operations According to Data

Sometimes you may wish to allow the user to retrieve a collection of entities from the server, but only enable that particular user to update or delete some of them. Like the previous point, this is more of an implementation issue than one that the RIA Services framework can help with. Again, within the operation itself you will need to obtain the list of the user's roles, and based on both the data that is passed into the operation and the user's roles, determine whether the user is permitted to perform that operation. Otherwise, raise an exception to notify the client that the call was unsuccessful.

Exposing Custom User Profile Data

As you may have noticed when we were defining the `AuthenticationService` class earlier in the chapter (in the "Authentication" section), the `AuthenticationBase` class that it inherits from is a generic class, and you are specifying a type of `User`:

```
[EnableClientAccess]
public class AuthenticationService : AuthenticationBase<User> { }
```

The `User` class contains information about the user that you return to the client when a user is authenticated. This class can be found in the `Models` folder. It inherits from `UserBase`, which includes properties exposing the user name, the roles that the user is in, and whether the user has been authenticated. You can add additional properties to this class, and these properties will be returned with the object to the client. For example, you may wish to store additional information about the user (such as their contact details), maintain a set of user preferences on the server, or provide some properties specifying permissions separate from those designated by the user's assigned roles. A corresponding `User` class will be created in the client project with the same properties that you defined on the `User` class in the web project, enabling their values to be read and saved there too. You will find that the Silverlight Business Application project template has already created the `FriendlyName` property on the `User` object:

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

The value of these properties are stored in the profile membership table in the database; and to be able to access them, you will need to define them in the `web.config` file. You will already find the `FriendlyName` entry here, and you can add any additional properties below it:

```
<profile>
  <properties>
    <add name="FriendlyName" />
  </properties>
</profile>
```

You don't have to worry about populating the properties you've created on the `User` object with the data from the profile, as the `AuthenticationBase` class does that automatically using reflection. However, when you add additional properties to the `User` object, be sure to add a corresponding entry to the `web.config` file (with a matching name). Otherwise, an exception will be raised when you attempt to use the authentication service.

User Registration

To add users to the system, you can either use the ASP.NET Configuration tool for the web application, or the registration facility already in the Silverlight project. This facility is enabled via the `UserRegistrationService` (located in the `Services` folder in the web project)—a standard domain service simply exposing an `AddUser` domain operation enabling new users to be registered within the system via the Silverlight client. It's ready to create a user (with the details supplied), assigning them to a default role called `Registered Users` (defined as a constant named `DefaultRole` within the service, enabling you to change it as required), and create the profile for the user (as described in the previous section).

However, in typical business applications you don't allow users to register themselves. Generally, that's the role of another user who has permission to add users to the system. In this case, you want to prevent users from accessing this domain service unless they are authenticated and belong to a given role.

If you don't want to expose the ability to add users to the system publicly, you can simply delete the `UserRegistrationService` class (along with the corresponding functionality in the Silverlight client), preventing users from being created in the system though the application or directly via the domain service.

You can modify the `UserRegistrationService` domain service, adding additional functionality as you see fit. For example, you will generally need to enable the user to select what roles this new user will be assigned to. This will require the ability to return the full list of roles back to the client, and pass the selected roles back to the server when the user is being created (functionality that unfortunately does

not yet exist in the Silverlight Business Application project template). To expose the full list of roles to the client, simply add the following `Invoke` operation to the `UserRegistrationService` domain service:

```
[Invoke]
public string[] GetAllRoles()
{
    return Roles.GetAllRoles();
}
```

You can then add a new property to the `RegistrationData` class (in the `Models` folder) to pass back the selected roles in, and update the `AddUser` domain operation in the `UserRegistrationService` domain service to assign the user to those roles accordingly. Note that the `RegistrationData` class contains all the data about the user that you will use when registering the user, so if you have additional properties that you wish to add (such as data to populate any custom user profile properties that you configured, as per the previous section), then add those properties to this class and update the `AddUser` domain operation in the `UserRegistrationService` domain service to save those property values as required. The registration form is generated by reflecting over the `RegistrationData` class, meaning that it will then automatically display data entry fields for any additional properties that you have added to this class.

■ **Note** If you want to remove the security question and answer properties from the user registration procedure, you can remove them from the `RegistrationData` class and update the `AddUser` domain operation in the `UserRegistrationService` domain service accordingly, but you will also need to update the ASP.NET Membership provider configuration in the `web.config` file in the web project to set its `requiresQuestionAndAnswer` property to `false`. See the “Configuring Your Database for Forms Authentication” section earlier in this chapter for an example.

Avoiding SQL Injection Attacks

Throughout the history of application development, SQL injection has always been a relatively simple technique for hacking into the databases behind applications. This involves crafting a special query entered into a text box and submitting it to the web site. It exploits the fact that many developers simply concatenate the data entered into a form with their SQL database query. Therefore, if you enter the right text into the text box, you can effectively modify the query that is being run against the database!

This exploit can allow the user to gain access to all the records in the database (e.g., people’s names, contact details, and credit card information), and even cause outright damage (such as deleting tables in the database). The latter issue, of course, can be prevented by ensuring that the database user account that the web application is running under has minimal privileges (so that it cannot be used to delete tables).

The methods demonstrated throughout this book are not open to SQL injection (primarily because we’re using the Entity Framework as our data access layer, which manages this issue); however, if you are using a different mechanism to query and persist changes to a database, then you should be aware of this issue and cater for it accordingly. Parameterized database queries are one of the best options for preventing SQL injection, but best practice is to also always validate any user input before using it in a database query or persisting it to the database. Also, never allow a query (such as a complex filter) to be

dynamically created on the client and executed on the server—this makes it even easier for someone to gain complete control over your database than SQL injection does.

Sanitizing Errors Sent to the Client

To crackers, more knowledge makes it easier to find points of weakness that they can target. Therefore, it's always good practice to avoid sending error messages raised on the server directly to the client, as these generally include stack traces and other information detailing the inner workings of your application that crackers can find useful. Instead, any error messages should be sanitized before being sent to the client.

This can be configured in the `customErrors` element of your `web.config` file in the same manner as in a standard ASP.NET application/web site. By default, your ASP.NET application uses the `RemoteOnly` mode (meaning that full exception details will be passed to clients running on the local machine to the server, but will be sanitized when clients are running on remote machines). Setting the `mode` property of the `customErrors` element to `Off` will disable exception sanitizing for all clients (remote or local), and setting it to `On` will enable it for all clients (useful when you want to test the client's experience when exceptions are thrown). For example, add the following element to the `system.web` section of your `web.config` file to sanitize server exceptions before exposing them to the client:

```
<customErrors mode="On" />
```

In order to identify the error such that you can fix the problem, log the error on the server (as described in the “Handling Server Errors” section of Chapter 5).

Implementing Client-Side Security

Now that the server has been configured to authenticate and register users, and restrict access to data and operations accordingly, we should look at implementing the client-side security aspects. In this section we'll take a look at how to authenticate and register users on the client, how to get information about the user (such as their roles and their profile), and take a look at strategies you can use to restrict users from parts of the application based upon their authentication status or assigned roles.

Authenticating and Registering Users

Let's now take a look at what is already in the Silverlight project to support user authentication. When running the Silverlight project, you will find a login hyperlink at the top right of the window, below the About button. This will pop up a Login dialog that you can use to log into the system, as shown in Figure 8-1.

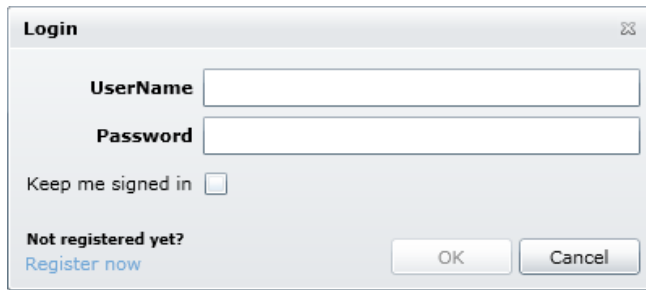
A Windows-style dialog box titled "Login" with a close button (X) in the top right corner. It contains two text input fields: "UserName" and "Password". Below the "Password" field is a checkbox labeled "Keep me signed in". At the bottom left, there is a link "Not registered yet? Register now" where "Register now" is a blue hyperlink. At the bottom right are two buttons: "OK" and "Cancel".

Figure 8-1. The Login dialog from the Silverlight Business Application project template

Of course, the first step you need to take is to actually register as a user, and you can do so by clicking the Register Now hyperlink in the bottom left of the Login dialog. This pops up a registration screen, as shown in Figure 8-2.

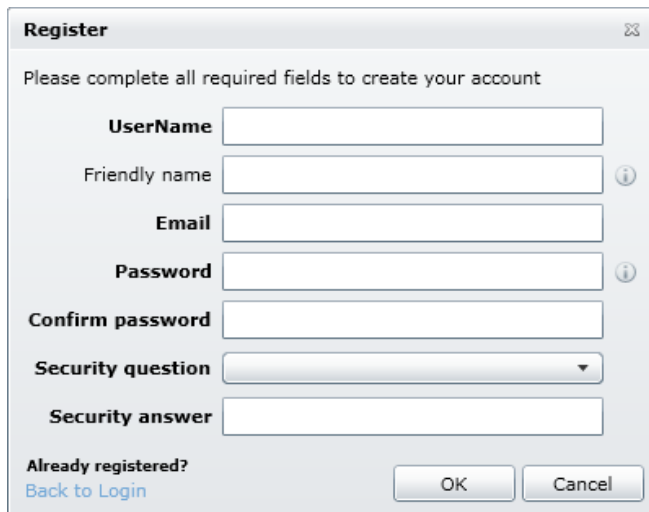
A Windows-style dialog box titled "Register" with a close button (X) in the top right corner. It contains the instruction "Please complete all required fields to create your account". Below this are several input fields: "UserName", "Friendly name" (with an information icon), "Email", "Password" (with an information icon), "Confirm password", "Security question" (a dropdown menu), and "Security answer". At the bottom left, there is a link "Already registered? Back to Login" where "Back to Login" is a blue hyperlink. At the bottom right are two buttons: "OK" and "Cancel".

Figure 8-2. The Register dialog from the Silverlight Business Application project template

■ **Note** By default, the required password strength is very high with the ASP.NET Membership provider. This can be altered via the web.config file using the properties of the ASP.NET Membership provider. See the “Configuring Your Database for Forms Authentication” section earlier in this chapter (particularly the `minRequiredPasswordLength` and `minRequiredNonalphanumericCharacters` properties) for an example.

The Login and Registration screens communicate with the AuthenticationService and UserRegistrationService (respectively) on the server, as discussed earlier. With these services being domain services, a corresponding context will be created in the Silverlight client that can be used to interact with the AuthenticationService (AuthenticationContext) and UserRegistrationService (UserRegistrationContext). However, although the UserRegistrationContext class is instantiated like all other domain contexts, the AuthenticationService is a little different.

Essentially, the AuthenticationContext should be kept alive for the lifetime of the application to maintain the current authentication status—created when it is started, and not destroyed until the application is closed. Therefore, it takes advantage of a feature of Silverlight that extends the Application object to add and maintain these lifetime objects.

Inside the App.xaml file you will find the following XAML:

```
<Application.ApplicationLifetimeObjects>
  <app:WebContext>
    <app:WebContext.Authentication>
      <appsvc:FormsAuthentication/>
      <!--<appsvc:WindowsAuthentication/-->
    </app:WebContext.Authentication>
  </app:WebContext>
</Application.ApplicationLifetimeObjects>
```

Through a somewhat complex web of abstractions that are mostly hidden from you and not required to be understood, this starts by defining the WebContext object (generated by RIA Services) as an application lifetime object (these are object instances maintained throughout the lifetime of the application). You then assign what authentication provider it should use to its Authentication property. Here we are using Forms Authentication, but you can comment out the FormsAuthentication element and uncomment the WindowsAuthentication element in order to use Windows Authentication instead.

■ **Note** The authentication provider you select must match what you have configured in the web.config file in your web project.

You will find a full example in the code-behind the LoginForm view, but the following line demonstrates calling the Login method of your AuthenticationContext object, passing it a username and password:

```
LoginOperation op = WebContext.Current.Authentication.Login(userName, password);
```

When the user has successfully been logged into the system, the LoggedIn event on the AuthenticationContext object will be raised (which you can subscribe to and handle application-wide). There is also a LoggedOut event on the same object that will be raised when the user logs out of the system.

The method of registering a user is a little less straightforward. Being a standard domain context, it works in much the same way as adding new entities to a collection exposed by any domain service, as demonstrated in Chapter 7. You need to populate a new RegistrationData entity with the user details, and then add it to the RegistrationDatas property of the RegistrationContext object. Finally, call the

SubmitChanges method on the RegistrationContext object to send the details to the server, where the user will be added to the system. For example:

```
RegistrationData newUser = new RegistrationData();
newUser.UserName = userName;
newUser.Password = password;
// Etc

UserRegistrationContext context = new UserRegistrationContext();
context.RegistrationDatas.Add(newUser);
SubmitOperation op = context.SubmitChanges();
```

As with the login behavior, you will find a full example of the code required to register a user already implemented in the RegistrationForm view in the project.

■ **Note** The registration form registers the user, and then immediately logs into the system as that user. If you are implementing the scenario described earlier in the chapter where only authenticated users assigned to a role that enables them to create users then you will want to disable this behavior (such that the user is simply registered). To do so, instead of calling the Login method once the user registration is complete, simply close the window instead.

Accessing and Updating User Information

Once the user is logged in, you may wish to customize the user interface to their needs (i.e., target the user experience), read (and save) user preferences, display some information about that user (such as their name), or restrict access to certain functionality based upon their assigned roles. All of this is available to you from the User object that RIA Services exposes on the client through the static Current property on the WebContext class.

To start with, determining whether the current user has logged in and is authenticated simply requires you to check the IsAuthenticated property of the User object:

```
bool isUserAuthenticated = WebContext.Current.User.IsAuthenticated;
```

You can get an array of all the roles that the user is assigned to via the Roles property of the User object:

```
string[] userRoles = (string[])WebContext.Current.User.Roles;
```

Alternatively, you can simply check whether a user is in a given role using the IsInRole method of the User object:

```
bool isAdminRole = WebContext.Current.User.IsInRole("Administrators");
```

■ **Note** Best practice (for maintainability's sake) is to always specify each of your roles as constants, and use those instead of hard-coding the role names when checking whether a user is in a given role.

Any properties that you defined on the `User` object in the web project will automatically have corresponding properties created on the `User` object in the Silverlight project. These are populated with the data from the database automatically when the user is logged in, and this object is returned to the client. Therefore, as soon as the user is successfully logged in you can start accessing their profile settings via these properties. For example, the custom `FriendlyName` property that was created as a part of the Silverlight Business Application project template can be accessed from the client in the following manner:

```
string friendlyName = WebContext.Current.User.FriendlyName;
```

If you are storing user profile data (such as user preferences) that you have changed on the `User` object and you now want to update on the server, simply call the `SaveUser` method on the `AuthenticationContext` domain context:

```
WebContext.Current.User.FriendlyName = "Chris Anderson";  
WebContext.Current.Authentication.SaveUser(true);
```

Implementing Client-Side Restrictions

Although securing access to data and operations should be handled on the server, it can also be useful to implement additional restrictions on the client-side—hiding parts of the application while the user is not logged in, and that the user does not have permission to access when they are. To do so, simply use the methods described in the previous section to determine whether the user is logged in and what roles they are assigned. The strategies that you might choose to implement client-side restrictions will generally be one of the following:

- All functions are visible, but when the user tries to access them, their credentials will be checked and they will be prevented from using the function if not allowed. This doesn't provide the best experience for the user, as using the application becomes somewhat of a hit-and-miss affair, where the user doesn't immediately know what they can and can't do within the application.
- Only functions that the user can access are enabled (with those that they can't being disabled/grayed out). This shows the user all the things that they could do in the application if they had the requisite privileges, but enables them to see that they can't actually use the disabled functionality.
- Only functions that the user can access are visible (those that they can't access are hidden from view). This simplifies the application for the user and helps avoid overwhelming them with options, but they won't know when a function is actually implemented in the application, but just not available to them currently (enabling them to ask for privilege elevation if they require it).

Which means you choose really depends on the nature and design of the application. The first option can be simply implemented using the methods described in the previous section in code before performing the requested function. However, there are a few different means to implement the following two options:

- If you name the controls that should be disabled/hidden depending on the user's privileges, then in your code-behind you can do so when the view is loaded.
- If you are using the MVVM design pattern (described in Chapter 12), you can expose properties representing the permissions from your ViewModel that the controls in the view can bind their `IsEnabled/Visibility` properties to.
- You can write some attached properties that can be added to controls used in a view, controlling whether they are disabled or hidden depending on the user's privileges. A good example of this is in an article by Andrea Boschin on the SilverlightShow.net web site: www.silverlightshow.net/items/A-RoleManager-to-apply-roles-declaratively-to-user-interface.aspx. The RIA Services team has also released a sample project implementing a similar method, which you can download from the WCF RIA Services Code Gallery (in the sample named "Authorization Based Dynamic UI and Navigation") here: <http://code.msdn.microsoft.com/RiaServices>. Kyle McClellan has written instructions for using the library from this sample project here: <http://blogs.msdn.com/b/kylemc/archive/2010/05/03/authorization-sample-101.aspx>.
- You can declaratively bind to properties of the `User` and `AuthenticationContext` objects in XAML. If you open up the code-behind for the `App.xaml` file, you will find that the current `WebContext` object is added to the application's resources when the application starts up. This means you can bind to it just as you would any other resource in XAML. If you want to bind to the `Visibility` property of a control, then you will need to write a value converter to convert a Boolean value to a `Visibility` enumeration (value converters will be covered in Chapter 10), and if you want to control whether the control is disabled or hidden based upon the roles that the user is assigned, then you can write a value converter to do that too. A simple example, however, of declaratively enabling a control only when the user is logged into the system is demonstrated following:

```
<Button Content="Click Me!"
        IsEnabled="{Binding Path=User.IsAuthenticated,
                           Source={StaticResource WebContext}}"/> />
```

Ultimately, the best choice from the preceding options really depends on the structure of your project. If you are using the MVVM design pattern, then the second option tends to give the cleanest and most maintainable solution.

■ **Note** Simply preventing a user from getting to a view by hiding or disabling the button that opens it won't stop them from opening that view. As described in Chapter 4, you can navigate directly to a view simply by entering the URI that opens it in the address bar of the web browser. This means that the user can bypass your security measures based upon what's available in the user interface and go directly to a view (if they know or try guessing the URI). Therefore, you should always check when a user navigates to a view that they have permission to access it, and redirect them to a different view if they don't instead.

Storing Data Locally

It's worth noting that any data stored in isolated storage (which will be discussed in Chapter 14) should not be considered safe. Other Silverlight applications cannot access isolated storage for this application, but data is stored as files on disk and can be read like any other file. The data is usually stored somewhere in the %AppData%\LocalLow\Microsoft\Silverlight\is folder. The paths are not self explanatory, but you could keep looking until you find what you are after. Therefore, it's recommended that you do not store sensitive data, passwords, and so on in isolated storage without encrypting them first.

Encrypting Data Passed Between the Server and the Client

A common “cracking” technique is to sniff data as it travels over the network, capture this traffic, and extract any sensitive data to potentially be used for ulterior motives. You can try a form of this yourself by using a tool such as Fiddler to sniff and capture network traffic on your machine.

By default, RIA Services uses binary encoding when transferring data over the network, but it's still relatively easy to read with a Fiddler add-in, so it should not be considered encrypted traffic.

Ideally, if you are passing sensitive data between the server and the client, you want to encrypt this traffic by communicating over HTTPS using SSL to protect it from this sort of sniffing. This requires a server certificate to be purchased from a certification authority and installed on your IIS server; however, a full discussion on doing so is beyond the scope of this chapter. Alternatively, you can use a self-signed certificate for development and testing purposes. IIS 7.0 on Windows Vista/7 makes this easy to create, as described by Scott Guthrie in this blog post:

<http://weblogs.asp.net/scottgu/archive/2007/04/06/tip-trick-enabling-ssl-on-iis7-using-self-signed-certificates.aspx>.

Assuming you have already installed a server certificate, let's look at what you need to do to get the service operating over HTTPS.

Enforcing that communication must be over HTTPS is specified at the service level, and is simply a case of setting the `RequiresSecureEndpoint` property of the `EnableClientAccess` attribute to `true` for each domain service that must be accessed securely (including the `AuthenticationService`). Any requests to domain services with this property set to `true` over HTTP will be denied access.

It's not necessary that the Silverlight application itself is accessed over HTTPS—you can access it over HTTP and then require server/client communication to be over HTTPS with this method:

```
[EnableClientAccess(RequiresSecureEndpoint=true)]
public class AuthenticationService : AuthenticationBase<User>
```


However, note that in this scenario, by delivering the Silverlight application via HTTP but communicating with the domain services via HTTPS, you are effectively communicating with a different domain than the application's source. This means that you must implement a cross-domain access policy file (that allows cross-scheme access), as will be described in the next section.

■ **Note** When debugging your application using this option, you will need to run it under IIS (instead of the ASP.NET Development Server), as the ASP.NET Development Server (aka Cassini) doesn't support HTTPS communication. You will also need to specify the machine name in the URL instead of localhost when navigating to the application (you can configure this in the project properties of the web project in your solution).

Cross-Domain Access Policies

By default, the Silverlight runtime limits your application to communicating with services that originate from the domain that the Silverlight application was downloaded from (known as the “site of origin”). This restriction was put in place for security purposes in order to avoid denial-of-service (DoS), cross-site forgery, and various other attacks.

However, there are perfectly valid requirements for needing to communicate with servers other than the site of origin. For example, you may want to write an application in which your domain services exist in a different location from where the Silverlight application is downloaded, or that communicates with external services, such as Amazon's web services or the Twitter API. Or you may simply want to download a file from another web site. However, this is prevented by the Silverlight runtime unless that site explicitly permits cross-domain access by providing a cross-domain policy file that Silverlight can download, and permits the application access to that domain.

■ **Note** Silverlight applications running in OOB mode with elevated trust (as described in Chapter 14) do not have this restriction in place—they can communicate with any domain.

Silverlight respects both its own cross-domain policy file (`clientaccesspolicy.xml`) and Adobe Flash's (`crossdomain.xml`). It will first request the Silverlight policy file from the server, and if it's not found, then it will request the Flash policy file. Silverlight will look for these files in the root of the domain. If a valid policy file of either type is returned from the server, then the web request made by the Silverlight application will then be permitted to proceed. However, if no cross-domain policy file is found, an exception (containing a `SecurityException` exception as its inner exception) will be thrown.

■ **Note** Obtaining the cross-domain policy is all done behind the scenes by the Silverlight runtime, when the application attempts to communicate with a domain other than the site of origin. You can monitor this behavior using a web traffic–logging program such as Fiddler to view it, looking for one of these files. Once a valid cross-domain policy has been downloaded for a given domain, the Silverlight runtime will not attempt to download it again for future requests to that domain during the current application session.

Note that a connection to a service running in the same domain but on a different port is still considered a different domain, and thus requires that domain to provide a cross-domain policy file in order to permit the application access. This is particularly pertinent if you are hosting your web service in a different ASP.NET project in your solution than the project serving up your Silverlight application, and you are debugging your project using the ASP.NET Development Server instead of IIS. Because each ASP.NET project is running in its own instance of the ASP.NET Development Server, they are exposed from the localhost domain on a different port from each other. Therefore, the Silverlight runtime sees these as different domains, and requires the domain containing the web service to provide a cross-domain policy file; otherwise, the Silverlight application downloaded from the other project will be refused access to the web service by the Silverlight runtime.

Implementing Cross-Domain Policies for HTTP-Based Communication

This scenario actually provides the easiest means to demonstrate the cross-domain restriction and test a cross-domain policy file. Add a new project using the ASP.NET Empty Web Application project template to your solution, and add a new WCF service to this project using the Silverlight-Enabled WCF Service item template. Reference this WCF service from your Silverlight application and call the `DoWork` method it exposes. When you run the application and call that method, the call will fail with a `CommunicationException` exception. Now add a text file named `clientaccesspolicy.xml` to the root of this project, with the following content:

```
<?xml version="1.0" encoding="utf-8"?>
<access-policy>
  <cross-domain-access>
    <policy>
      <allow-from http-request-headers="*">
        <domain uri="*" />
      </allow-from>
      <grant-to>
        <resource path="/" include-subpaths="true" />
      </grant-to>
    </policy>
  </cross-domain-access>
</access-policy>
```

■ **Note** This policy is implementing totally unrestricted access to your domain by Silverlight clients (of course, standard security restrictions still apply). Best practice is to only open up access to resources you actually want to make available to Silverlight client applications.

Now when you run the application, the service call will be successful. If it fails, try using Ctrl+F5 in the browser to refresh your application, as the policy file may have been cached by the browser.

Implementing Cross-Domain Policies for Socket-Based Communication

Note that when communicating with a server via sockets, a cross-domain policy is required regardless of whether you are attempting to communicate with the site of origin (as a different port is considered a different domain). You can explicitly configure the application to look for a cross-domain policy file hosted on port 80 (the HTTP port) in the root of that domain (i.e., in exactly the same fashion as for standard web requests). However, if that server is not serving HTTP requests, another means is obviously required to obtain a cross-domain policy. The solution involves listening on port 943 and serving up the policy when it's requested by the client. A sample sockets policy file is provided following:

```
<?xml version="1.0" encoding="utf-8" ?>
<access-policy>
  <cross-domain-access>
    <policy>
      <allow-from>
        <domain uri="*" />
      </allow-from>
      <grant-to>
        <socket-resource port="4502-4534" protocol="tcp" />
      </grant-to>
    </policy>
  </cross-domain-access>
</access-policy>
```

Implementing Cross-Scheme Access Policies

Another scenario you should be aware of is one in which you have a cross-scheme communication. An example of this is when you want to deliver the application to the client over HTTP, but have it communicate with the server securely over HTTPS. This too constitutes a cross-domain call, and thus must be explicitly permitted by the server in its cross-domain policy file. Despite the `allow-from` node in the preceding policy file assigning a wildcard to accept callers from any domain, it actually doesn't permit callers using a different scheme to the service. You must explicitly specify the schemes that will be permitted access. Therefore, replace this line from the cross-domain policy file:

```
<domain uri="*" />
```

with these two lines:

```
<domain uri="http://*" />  
<domain uri="https://*" />
```

Restricting Access to Your Application

In some scenarios, you may want only certain users to be able access your application. Silverlight XAP files are easy to decompile, which may be a cause for concern if your application contains, for example, sensitive algorithms or assets. By letting anybody download your application, you are also allowing them to gain access to its contents.

As described in Chapter 2, a XAP file is simply a ZIP file with a different extension. Changing the extension to ZIP (or simply opening the XAP file directly in a compression utility) enables you to get access to all the assemblies and assets in your application. Then, using .NET Reflector (a free tool from Red Gate Software), you can decompile the assemblies back into readable code.

Alternatively, Silverlight Spy (a commercial tool from First Floor Software) allows you to navigate to a URL hosting a Silverlight application, which it will display the contents of (and run). It then allows you to browse to an assembly within the XAP file and decompile it (using .NET Reflector), and view its code.

Therefore, always assume everything within your application is accessible publicly. This includes the address and structure of your services, any encryption keys, any assemblies and algorithms contained therein, and any assets such as graphics and other resources. Anything that is sensitive (generally in the form of intellectual property) should not be used in a Silverlight project. While this is the simplest solution to this problem, it is not always viable. Therefore, you need to look for other alternatives.

Ideally, for best protection, sensitive algorithms should be placed on the server and called via service calls when required. Of course, depending on the nature of the application (being run in disconnected mode, or used by complex simulations, for example), this is not always viable either. Therefore, you might consider using an obfuscator. These programs (generally) don't stop your application from being decompiled, but they do make the results very difficult to read and follow. Their strategy is "security through obscurity." They aren't perfect solutions, but do provide a significant impediment to those trying to decompile your application. There are a number of commercial obfuscation tools available that will obfuscate Silverlight XAP files, and a quick web search will turn up a number of them (be sure to check that they support obfuscating Silverlight XAP files first, however).

■ **Note** To make obfuscation effective, you need to mark as many classes, methods, and properties as `internal` or `private` as possible within your project. The external (i.e., public) interface needs to remain the same so that the obfuscator can't change it. But the internal or private definitions can be safely changed as it can modify all their references accordingly, knowing that nothing externally will be affected.

To protect assets used by your application, you may wish to consider downloading them on demand from the server rather than including them within your XAP file. This means that you can restrict access to them by specifying the rules in a `web.config` file that you place into a folder where restricted assets are located. These rules can be used to restrict the assets in that folder from being downloaded unless the user is authenticated or assigned to a particular role, and you can even control access at the user name level if you wish. This strategy integrates with the ASP.NET Membership security, meaning that all you need to do is to configure the access rules in the `web.config` file (with the

authentication and authorization handled by the ASP.NET Membership providers). You can create the web.config file(s) yourself and manually configure the security settings, but the easiest way to do so is to use the ASP.NET Configuration tool for the web application. Click the Security tab, and then click the “Create access rules” hyperlink in the Access Rules group on this page. You can then use the user interface provided (as shown in Figure 8-3) to create the access rules, without needing to learn the syntax.

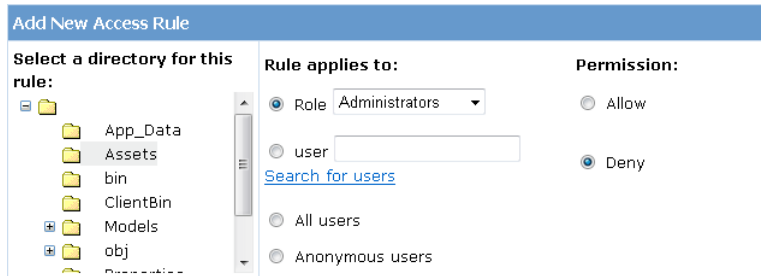


Figure 8-3. Configuring access rules in the ASP.NET Configuration tool

For example, limiting access to a folder in the web project that we create call Assets will create a web.config file in this folder with the following contents:

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <system.web>
    <authorization>
      <deny users="?" />
    </authorization>
  </system.web>
</configuration>
```

You can then download the assets from your Silverlight application once the user is logged in, using the techniques demonstrated in the “HTTP Requests” section of Chapter 5.

This is also a technique that you may wish to use to restrict access to your Silverlight application itself. You can restrict access in the same manner to the ClientBin folder in your web project so that the user can’t download the application unless they are already logged in. This can reduce the importance of protecting your application in the other ways described; however, it does create the issue of how the user will log into the application in the first place. There are a number of strategies that you might choose to implement:

- Because the authentication mechanism in RIA Services is exactly the same as that used in ASP.NET projects, you could create an HTML-based login page that the user can use to log into the system, and then loads the Silverlight application once the user has successfully been authenticated.
- Using some of the techniques described in Chapter 15, you could partition your application into a number of parts and load them on demand—protecting and loading the more sensitive parts only once the user is logged in.

- Implement a “bootstrapper.” Similar to the previous scenario, this will download and run a small Silverlight project—which you could use to provide a simple login screen, and then continue to download the (protected) remainder of the application. The Silverlight Extensions project provides a bootstrapper that you can use to help you, available from <http://slextensions.codeplex.com>.

■ **Note** Dynamically loading parts of your application on demand isn’t possible with applications that are being run in OOB mode.

You may even want to use this method to restrict access to any standard WCF web services that you use in your application. You can restrict access to their SVC files until the user is logged in, providing another degree of security. Note, however, that adding or updating these service references will not work if you have these restrictions in place while developing the application—implement these restrictions on the production server only.

Summary

Security should be designed into your application—not treated as an afterthought. Not all the issues discussed in this chapter will be of a concern to you, but you can pick those that are and cater for them accordingly.

The best way to handle security issues is to try to think like a cracker, attempt to circumvent any security put in place, and look for weaknesses. It can often be worthwhile (depending on the sensitive nature of the application) to get a security specialist to test it for you.

Alternatively, ask another developer to test it from a fresh perspective, without any preconceived knowledge of what security has been implemented in the system. The most important tip is to assume that the client will be compromised, and design security elements into your domain services to always verify that the client has permission to perform every action immediately before doing so (i.e., don’t make assumptions).

CHAPTER 9



Styling Your Application

One of the key features of XAML is that it enables exceptional user interfaces to be designed and implemented in Silverlight. When designing the look and feel of your application however, you don't want to have to customize each control and user interface element individually. This is where the use of style resources comes in—they enable you to maintain a core set of common styles to be used throughout your application.

Expression Blend has been designed to help you in the styling process, but in this chapter, we'll focus on the core fundamentals of styling in Silverlight, particularly how the styling model is structured and works, so that you can then take advantage of it however you wish. This chapter won't discuss how to make your application more attractive, but it will take you through the various aspects of controlling the appearance of controls through XAML and maintaining these as reusable resources.

The Designer/Developer Workflow

The ability for XAML-based controls to be extensively styled and templated enables you to radically customize their appearance and permits the creation of visually pleasing applications. User interfaces can be designed that engage users and draw them into using the application. However, simply using the Silverlight controls out of the box, on the default white canvas, can lead to a rather plain application, often followed by badly implemented attempts to make it more interesting by developers who may not have the skills to do so effectively. Unfortunately, this process results in many unsightly user interfaces. Silverlight projects can, therefore, greatly benefit from having a designer on board to create an attractive user interface.

Including designers as a part of the development process has therefore been a focus for Microsoft. That said, Visual Studio is not an application suited for use by designers, so Microsoft created the Expression Suite targeted toward the needs of designers. From the Expression Suite, the Expression Blend and Expression Design tools are those that will be most useful to graphic designers when working on Silverlight projects.

Expression Blend is targeted toward designing XAML-based user interfaces for Silverlight and WPF. This includes extensive support for laying out controls, styling and templating them, and creating and applying animations to elements of the user interface. With Visual Studio 2010's new Silverlight designer, we now have support for a drag-and-drop design surface that we can view and interact with at design time (a feature not available in Visual Studio 2008 - with hand-coding XAML being the only option). Therefore, developers who worked with XAML in addition to code now have less reliance on Expression Blend than they may have previously. That said, however, Expression Blend still offers many design-related features that Visual Studio does not (without modifying the XAML directly, which can often be complex and inefficient); these include

- Designing and applying animations (including transition animations) to user interface elements. Animations can involve a lot of XAML, which would have to be

hard-coded in Visual Studio. Not only does Expression Blend provide a friendly user interface for creating animations but it also allows you preview the animation at design time without needing to compile and run your application first.

- Control templating (discussed later in this chapter) is vastly easier in Expression Blend (it is a very manual and convoluted task in Visual Studio), as is creating control templates for your own custom controls (discussed in Chapter 11, “Creating Custom Controls”).
- Expression Blend enables you to populate your user interface with sample data at design time, allowing you to preview how the interface will look when running and customize your design accordingly.

These are just some of the areas in which Expression Blend provides guidance and makes those tasks much easier to implement. Therefore, for any nontrivial user interface design, it’s generally recommended (but not essential) that you include Expression Blend as one of your tools.

Just as Visual Studio 2010 has some design-related features, Expression Blend also has some code-related features too. Expression Blend can compile and run projects, and you can also modify code. However, Expression Blend’s code-related features are as minimal as Visual Studio’s design features. There are just enough code-related features in Expression Blend to do basic tasks, but not enough for someone to solely use Expression Blend to create a Silverlight application in an efficient manner.

Microsoft has designed the Visual Studio and Expression Blend tools to enable developers and designers to work alongside each other on the same project in harmony. The two complement each other. Designers can work on the visual aspects of the application, while developers work on the code aspects simultaneously. This approach can be even more effective when using the Model-View-ViewModel (MVVM) design pattern discussed in Chapter 12, as it promotes a clear separation of the presentation of a view and its corresponding logic. This separation makes it much easier to enable both a designer and a developer to work on the same part of the project at the same time, with reduced numbers of collisions between them.

Expression Design is another tool for graphics designers focused toward working with vector graphics. Expression Blend has some support for working with vector graphics, but Expression Design is more targeted in that direction (with Expression Blend targeted toward designing user interfaces). Therefore, you could break up the graphics designer role further to a user interface designer (who would use Expression Blend) and a graphics designer (who would use Expression Design). However, these roles often blend into a single graphics designer role in practice.

Another role that is starting to become more popular in software projects is the user experience designer. A user experience designer typically takes the functional requirements of what the software is to do and creates a design for how the user will interact with the software that implements those requirements in the most efficient and user-friendly manner.

User experience design faces a lot of misconceptions, and many people confuse it with the role of making the application attractive, believing that designing how users interact with the application is the role of the graphics designer. However, these should be treated as two completely separate skill sets: a user experience designer will determine the flow of the application based on the functional requirements, whereas a graphics designer will take this flow and implement it as an attractive user interface.

Another common misconception is that the graphic design or styling of the application should be one of the first tasks undertaken in the project. In fact, user experience design should be the main focus early on, with the styling performed later in the project. Support for the eventual styling should, of course, be included as a part of the user interface development process, but the interaction aspects are far more important to do up front. For this purpose, Microsoft added SketchFlow to Expression Blend. User experience designers can use SketchFlow to map the flow and rough layout of the application in order to generate prototypes

As a summary, Visual Studio is targeted at developers, Expression Blend and Expression Design are targeted at user interface and graphics designers, and SketchFlow is targeted toward user experience designers.

Since this book is primarily targeted for developers, we are focusing primarily on what can be done in Visual Studio. Therefore, aspects such as how to use the Expression tools and how to create animations (which really requires Expression Blend to implement efficiently) will not be covered here.

■ **Note** In theory, the developer, graphics designer, and user experience designer should all be different people with dedicated roles and skill sets. However, in reality, this is not always the case. If your developer role includes design, you may need to become familiar with more than one of these products and switch between them, but promoting a clear separation between these roles generally leads to better software.

Defining Style Resources

The simplest form of styling your user interface is to assign values to properties exposed by controls (using attribute syntax). However, this will very soon lead to your XAML becoming an unmanageable mess, with your XAML files increasing in complexity and becoming unwieldy and hard to read, and any necessary styling changes will often need to be made in numerous places within the application—essentially, you will have spaghetti XAML code.

The answer to this problem is to use style resources. Defining and using style resources has numerous advantages. If we want to change the styling of all our TextBox controls, we only need to set the value in one place, and it will automatically be applied to all the controls that reference that style. This simplifies the XAML by removing the need to apply the same property values on each element repeatedly. It also creates a clearer separation between your control definitions and their appearance.

You could consider style resources to be similar to CSS styles in traditional HTML-based web development. With CSS, you can define a style specifying various property values in your CSS file (or in your HTML page header) and apply that style to HTML elements by assigning the name of the style to their class property. Alternatively, you can define a style that specifies the values that the properties on all controls of a given type should automatically use. You can then override the property values defined in these styles by assigning different property values on the control definitions themselves.

Styling in Silverlight works in a similar fashion. Style resources can be defined in a resource dictionary (or within a XAML file itself) and applied to controls by assigning the style's key to their Style property. Or you can exclude the key from a style resource definition, and all the controls of the given type will automatically use that style (a feature known as *implicit styling*, new in Silverlight 4). Note, however, that unlike CSS, the style resource is specific to the given control type, meaning you cannot create a style that defines a set of property values to be used by both TextBox controls and ComboBox controls for example.

In Chapter 3, “An Introduction to XAML,” we have already discussed the concept of style resources, demonstrating how you could define commonly assigned property values on a control as a style resource and then apply that to the Style property of any controls that use it. However, we will start with a quick overview and move on to discuss some more advanced styling features.

Defining a Style Resource

There are two ways of defining style resources: as explicit and implicit styles. Let's take a look at each of these in turn.

Explicit Styles

Let's say we have multiple `TextBox` controls in our project that will use the same common property values. Instead of assigning the same property values to each control, like this

```
<TextBox Name="FirstNameField" Margin="2" Background="LemonChiffon" Grid.Row="0" />
<TextBox Name="LastNameField" Margin="2" Background="LemonChiffon" Grid.Row="1" />
<TextBox Name="CompanyField" Margin="2" Background="LemonChiffon" Grid.Row="2" />
```

We can, instead, define these properties as a style resource:

```
<Style x:Key="UserFieldsStyle" TargetType="TextBox">
  <Setter Property="Margin" Value="2" />
  <Setter Property="Background" Value="LemonChiffon" />
</Style>
```

When defining our style resource, we are giving it a unique key to reference it by (using the `x:Key` attached property from the XAML namespace) and specifying the type of control it can be applied to (using the `TargetType` property). After that, it's a case of creating a `Setter` element for each property you want to define a value for and providing it the name of the property and its corresponding value. Now, we can reference the style with the `Style` property on our controls, using the `StaticResource` XAML markup extension that points toward the style resource (using the key of the style resource as the reference).

```
<TextBox Name="FirstNameField" Grid.Row="0"
  Style="{StaticResource UserFieldsStyle}" />
<TextBox Name="LastNameField" Grid.Row="1"
  Style="{StaticResource UserFieldsStyle}" />
<TextBox Name="CompanyField" Grid.Row="2"
  Style="{StaticResource UserFieldsStyle}" />
```

■ **Note** You can override the values from the style resource by simply applying alternate property values directly to the control itself (i.e., via inline styles).

This is a process of *explicitly* assigning a style to a control. We gave the style resource a key and explicitly assigned that style to a control.

Implicit Styles

Alternatively, you may want to *implicitly* assign a style resource to all controls within a given scope. This is now possible in Silverlight 4 using the new implicit styles functionality. Simply omit the key when defining a style resource, and all controls of the given type within the scope of that resource will automatically use that style. For example, if you define the previous style resource (without a key defined) at a view level (e.g., in `Page.Resources` or `UserControl.Resources`), all the `TextBox` controls within that view will automatically use that style. If the style resources are defined at the application level, they will be used by controls application-wide (a typical scenario when creating themes, discussed later in this chapter).

Complex values can also be stored in styles. For example, defining a gradient background as a style requires you to use property element syntax when defining the value of the `Background` property, much as you would do if assigning the gradient directly on the control itself:

```
<Style x:Key="UserFieldsStyle" TargetType="Control">
  <Setter Property="Margin" Value="2" />
  <Setter Property="Background">
    <Setter.Value>
      <LinearGradientBrush EndPoint="1,0.5" StartPoint="0,0.5">
        <GradientStop Color="LemonChiffon" Offset="0" />
        <GradientStop Color="#FFFFFFCD" Offset="1" />
      </LinearGradientBrush>
    </Setter.Value>
  </Setter>
</Style>
```

■ **Note** Microsoft recommends that you avoid assigning values to properties in style resources where the property is defining behavior (as opposed to appearance).

Defining Style Resources at Various Locations

Style resources can be defined at various levels of the object hierarchy, and the level at which a style is defined provides its scope. Let's look at the most common locations for defining style resources in your project.

At the View Level

If style resources are defined at the view level (i.e., in `Page.Resources` or `UserControl.Resources` in a XAML file), they will be available to be used by all the controls in that view only, whereas styles defining at the application level can be used by any view in that project.

■ **Note** In addition, you could specify a style resource in the `Resources` property of any element in the control hierarchy, enabling it to be used (only) by the descendents of that element, although this isn't recommended due to potential maintainability issues (specifically because of confusion as to where a style resource definition can be found).

At the Application Level

To define application-wide style resources, the resources will need to be defined in the `Application.Resources` property of the `App.xaml` file.

As a Resource Dictionary

Style resources defined at the application level are often defined in a resource dictionary and merged into the application's resources. This strategy has the following benefits:

- Keeps the style resource definitions from clogging up the `App.xaml` file
- Enables the style resources to be reused among projects
- Enables different style resources dictionaries to be easily swapped in and out
- Enables your application to be themed (discussed later in this chapter)

The best example of this strategy is already implemented in the project you've generated from the Silverlight Business Application project template. All style resources are defined in a resource dictionary called `Styles.xaml` (found in the `Assets` folder). This file is then merged into our application resources (in the `App.xaml` file) like so:

```
<Application.Resources>
  <ResourceDictionary>
    <ResourceDictionary.MergedDictionaries>
      <ResourceDictionary Source="Assets/Styles.xaml"/>
    </ResourceDictionary.MergedDictionaries>
  </ResourceDictionary>
</Application.Resources>
```

■ **Note** Multiple resource dictionaries can be *merged* into a single `Resources` property. Therefore, you can maintain a logical separation between your style resources, avoiding resource dictionaries becoming too large and complex, and enabling multiple users to define styles at the same time. If a style is defined in more than one resource dictionary, the last one to be merged will be used. If you have two implicit styles targeting the same control, the property setters won't be merged into a single style, but only the last style will be used.

Alternatively, if you have only a single resource dictionary to merge, you can simply assign the resource dictionary to the `Application.Resources`:

```
<Application.Resources>
  <ResourceDictionary Source="Assets/Styles.xaml" />
</Application.Resources>
```

You can create a new resource dictionary by adding a new item to your project and selecting the Silverlight Resource Dictionary project template. This will leave you with an empty resource dictionary structure to which you can add your style definitions. For example, here is the `UserFieldsStyle` style resource we created earlier defined in a resource dictionary:

```
<ResourceDictionary
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">

  <Style x:Key="UserFieldsStyle" TargetType="TextBox">
    <Setter Property="Margin" Value="2" />
    <Setter Property="Background" Value="LemonChiffon" />
  </Style>
</ResourceDictionary>
```

As a Resource Dictionary in a Referenced Assembly

Themes are often defined in a resource dictionary, and it can be useful to have this resource dictionary in its own assembly so that you can maintain the theme in a single location and use it in multiple projects.

Add a reference to this assembly in your project. You can then merge the resource dictionary into your application's resources as previously demonstrated. However, the path to the view needs to be of the form `"[/[assembly]];component/[viewpath]"`. For example, if the `ThemeStyles.xaml` view existed in a referenced assembly named `MyTheme.dll`, the path to navigate to it would be `"/MyTheme;component/ThemeStyles.xaml"`.

```
<Application.Resources>
  <ResourceDictionary Source="/MyTheme;component/ThemeStyles.xaml" />
</Application.Resources>
```

In Generic.xaml

Custom controls generally have their styles and control templates defined in a resource dictionary named `Generic.xaml` (which does not need to be merged into any Resource property but must be placed under the Themes folder in the project where the corresponding controls are defined). This will be discussed in Chapter 11, "Creating Custom Controls."

Naming Style Resources

As a part of our design process, we need to look further than just how each control type will be styled, and design and name styles based on their purpose within the application rather than what they consist

of or the type of control they will be applied to. For example, you might have a `TextBlock` style resource that you want to use throughout your application to make the text bold, with a font size of 12 points, and colored navy. How should you name this style resource? Maybe `TextBlockStyle`, or `BoldNavy12ptStyle`, or `PageHeaderStyle` / `PageHeaderTextBlockStyle`?

From both a management and a visual design perspective, `PageHeaderStyle` or `PageHeaderTextBlockStyle` would be best way to name the style resource, because this name makes it easy to see what the purpose of each style is when navigating the style resource definitions. Also, if you are designing a new theme for the application (or plan to support multiple themes), it makes much more sense, as you are linking each style to a purpose rather than linking each style to how it is rendered on screen.

So even if two different definitions of a style resource have exactly the same property values (for example, both 12pt Bold Navy), it is generally still better to define them as separate style resources. This may result in multiple styles containing the same property values but provides better support for future restyling.

Inheriting Style Resources

Only a single style resource can be applied to a control, which may initially seem somewhat limiting when you are designing your style resources. For example, you may wish to define some property values in a base style resource common to most or all controls of a given type, and then you might want to define another style resource with additional property values to be applied to only some controls of that type (without having to define the same property values again on the new style resource that were defined on the base style resource).

While you may not be able to apply both those style resources to a control, you can have your new style resource *inherit* the property value definitions from the base style resource, which will serve the same purpose. This is achieved via the `BasedOn` property on the `Style` object, to which you can assign a reference to another style resource to act as the base style. This will need to be as a resource, hence the need for the `StaticResource` markup extension. For example, say we have this base style resource:

```
<Style x:Key="UserFieldsStyleBase" TargetType="TextBox">
  <Setter Property="Margin" Value="2" />
  <Setter Property="Background" Value="LemonChiffon" />
</Style>
```

When defining a new style resource that should be based on this one, we can reference it using the `BasedOn` property, like so:

```
<Style x:Key="UserFieldsStyle" TargetType="TextBox"
  BasedOn="{StaticResource UserFieldsStyleBase}">
  <Setter Property="TextAlignment" Value="Right" />
</Style>
```

■ **Note** A style resource can only be based on a single style, but that style can be based upon another style, and so on.

Not only can you inherit the property value definitions from a base style resource using this method but you can also override some of those property value definitions. For example, despite the `Background` property having a value of `LemonChiffon` in the base style resource, we are overriding that value in this style resource to assign a value of `White` instead:

```
<Style x:Key="UserFieldsStyle" TargetType="TextBox"
      BasedOn="{StaticResource UserFieldsStyleBase}">
  <Setter Property="Background" Value="White" />
</Style>
```

If you have a style resource that inherits a property value, but you want to use the default value in this resource instead, you can simply null its value like so:

```
<Setter Property="Background" Value="{x:Null}" />
```

■ **Note** The target control type from which you inherit a style resource must match that of the new style resource. For example, you cannot have a style resource targeting the `TextBox` control inherit from one targeting the `ComboBox` control. You can, however, have a style resource that inherits from one targeting one of its base classes. For example, a style resource targeting the `TextBox` control can inherit from a style resource targeting the `Control` class.

Styling Constraints

There are a few issues that you should be aware of when defining style resources:

- Only a single style resource can be assigned to a control, and any further property values that need to be applied must be done inline on the control definition itself.
- Style resources cannot be applied to controls of different types. Each style resource has a single control type that it applies to, meaning that you cannot have a `TextBox` and a `ComboBox` control reference the same style resource. You can assign the target type to be a class common to both those controls (such as `Control`), but you will often find that the properties you wish to define values for will not exist on that base class, and therefore, the project will not compile. Note that attempting this with implicit style resources will not work. The type of controls that will automatically use implicit styles must match the given target type of the style resource; otherwise, the style will not be applied.
- Once you have assigned a style resource to a control, any implicit style that the control may have otherwise used will no longer be used by this control.

Control Templates

You have now seen how we can define a style resource and apply it to controls, but what if we completely want to change the appearance of the control, including the appearance or function of

elements not exposed by the control as properties? In Silverlight, this is actually possible by applying an alternate *control template* to the default one defined by the control. The structure of Silverlight custom controls encourages a separation between presentation and behavior. The behavior of the control should be defined in the code, and its appearance should be defined in the control template. Between the two is a contract defining what the code needs access to from the presentation layer. With this clear separation of concerns, you can completely modify the appearance of a control without affecting its behavior, as long as you adhere to the control's behavior/presentation contract. This is an extremely powerful feature of XAML and really demonstrates its flexibility when defining user interfaces.

■ **Note** A *style* enables you to simply assign property values to a control, while a *control template* enables you to directly modify the core XAML of the control itself.

Default Control Templates

Default control templates (and styles) are defined in the `Generic.xaml` file in the Themes folder of the project or assembly containing the control. Each control template has its own resource dictionary within this file. When you want to apply your own alternate custom control template to a control, the easiest way to start is by making a copy of the default control template of that control and working from there. The easiest way to extract the default control template is using Expression Blend. It will extract the control template for that control from the `Generic.xaml` file in the control's assembly and place it (based on your selection) in the resources of the user control or the application as a new resource. You can then change this control template resource as you see fit—move elements around, add additional control elements, remove elements, and restyle elements—as long as you keep the template parts and states referenced by the control itself.

Unfortunately, without using Expression Blend, the only way to get the existing control template is to follow these steps:

1. Open the `Generic.xaml` file of the assembly that contains the control you want to modify. If you don't have the source code for the control, you will need to open the assembly using Reflector and extract the `Generic.xaml` file from there.
2. Find the style resource / control template pertaining to the control, and copy it into your project as a new style resource / control template.
3. Assign that new style resource / control template to your control.

As you can see, this is a somewhat messy and convoluted process, but Expression Blend helps to simplify and streamline this process.

Templating a Control

To make a copy of a control's default template using Expression Blend (so that you can edit it), right-click the control (in design view), and select **Edit Control Parts (Template) ► Edit a Copy**. This will perform the previously mentioned process automatically and leave the file ready for you to make the required modifications (which you can do visually via the Blend design view or by modifying the XAML manually).

Note that despite the name of the function, this process actually creates a style resource for the control, not a control template resource (though the style resource contains the control template), and this style resource will be referenced by the `Style` property on the control (a style can set property values, and `Template` is a property on the control). If you already have a style assigned to this control, a control template resource would be created instead and assigned to the control's `Template` property (as you might have otherwise expected as per the name of the function).

You will find that creating a control template resource inserts a lot of XAML markup into your XAML file (the amount will depend on the complexity of the control, and some controls like the `DataGrid` can be *very* complex). There is no way to minimize the amount of XAML generated without simplifying the visual structure of the control itself. For example, if you only wanted to make a slight adjustment to an existing element in the template, it's an all-or-nothing approach. Before customizing a control template, though, it is worth ensuring there isn't a style or template property on the control that will allow you to style or customize just a part of the control without resorting to customizing the entire template.

For the sake of completion, you don't actually have to create styles and control templates as resources (though it makes little sense not to, because if the style or template is a resource, you can easily apply it to other controls within the same scope). They can also be defined inline to a control definition using property element syntax, and thus will apply to that control instance only. For example:

```
<TextBox>
    <TextBox.Template>
        <!-- Control template definition can go here -->
    </TextBox.Template>
</TextBox>
```

Control Template Structure

Let's now take a look at exactly what a control template is and how it is structured. Often you will hear this structure referred to as the *parts and states model*.

At its simplest, a control template simply defines the XAML that makes up the static presentation of the template that never changes, for example:

```
<ControlTemplate TargetType="Button">
    <Grid>
        <!-- etc -->
    </Grid>
</ControlTemplate>
```

■ **Note** In Chapter 11, “Creating Custom Controls,” we'll go through the structure of control templates in more detail as we look at how to build custom controls.

Visual States

It's rare that a control will have a single static appearance, as almost all controls will change their appearance in some way (such as when it has focus or is disabled). In that case, we would define a base look for the control upon which changes to the control's appearance would be made when appropriate,

but we also include the elements that would be visible in the alternative states even if not visible in the base state (with their visibility set to collapsed or opacity set to 0).

For example, a button looks a particular way when it is placed on a form (its default state). Then when it has the focus, is disabled, pressed, hovered over, and so on, its appearance will be updated accordingly. This is where Visual States come in—each visual state defined describes how the base look of the control will be altered when that state is active to achieve the appearance of the control for that state. This essentially involves altering the elements that make up the base look for the control (e.g., changing the color, visibility, or opacity of an element) to visually indicate that the control is in that state. In other words, we define each state in our control template and what changes need to be made to the base look to achieve the desired appearance indicating that state. These changes are actually defined as animations, as will be discussed in the next section (“State Transitions”).

What triggers a state change on a control? Essentially, the control behavior (i.e., the code for the control) determines this based on the events it receives and/or the value assigned to one (or more) of its properties. For example, when the user clicks a button and holds the mouse button down (in response to the `MouseDown` event, which would mean the button is in the `Pressed` state), then the button needs to change its visual state to indicate as such. Alternatively, the button may be disabled (`IsEnabled="false"`), in which case the button needs to indicate that as its state (the `Disabled` state). The control needs to be able to determine what state it should be in response to each situation, but then it delegates to the Visual State Manager to update the appearance of the control accordingly.

Note that no changes should be made to the visual appearance of the control from the code; all presentation details on a per-state basis should be defined within the control template only.

You may also want to have a constant animation for the control while it is in a particular state (such as a button pulsing when the mouse is over it). This is known as *in-state animation*. This is achieved using a storyboard containing various animations that change the property values of elements within the control over a given duration (and will generally be set to continually repeat while the control is in that state).

A simple example of animating a property on an element within the control is demonstrated in the following example. This animation is used to make the button appear to be pressed changing the `Opacity` property of the `PressedBorder` element to 1 (essentially making that element visible).

```
<VisualState x:Name="Pressed">
  <Storyboard>
    <DoubleAnimationUsingKeyFrames
      Storyboard.TargetProperty="(UIElement.Opacity)"
      Storyboard.TargetName="PressedBorder">
      <EasingDoubleKeyFrame KeyTime="0" Value="1"/>
    </DoubleAnimationUsingKeyFrames>
  </Storyboard>
</VisualState>
```

■ **Note** Each state must exist within a state group, as will be described shortly.

A full discussion of implementing animations is beyond the scope of this book, although you will find further examples in Chapter 11, “Creating Custom Controls.”

State Transitions

A control moves from state to state via a *state transition*. State transitions can be represented in the control template using the following:

- An explicitly defined animation defined within a *storyboard* (in the same way as in-state animations) over a given duration, for example:

```
<VisualTransition GeneratedDuration="0:0:2" From="Inactive" To="Active">
  <Storyboard>
    <DoubleAnimation From="0" To="1" d:IsOptimized="True"
      Storyboard.TargetProperty="(UIElement.Opacity)"
      Storyboard.TargetName="LayoutRoot" />
  </Storyboard>
</VisualTransition>
```

- An automatic transition that smoothly animates the elements in the control between their configuration in each state over a given duration, in a process managed by the Visual State Manager, for example:

```
<VisualTransition GeneratedDuration="0:0:2" From="Inactive" To="Active" />
```

■ **Note** State transitions should (as a general rule) have a very short duration (no more than half a second) so as to not slow the user's interaction with the application.

Visual State Groups

We need to take into account one more complexity, and that's when a control can be in more than one state at the one time. For example, one state a button is in may be the Focused state (indicating the button has input focus), but it may also be in the Pressed state (where the button has been clicked and the mouse button hasn't yet been released). So how do we represent both states visually at the same time? Well, this is possible by having the states exist in different visual state groups. The use of visual state groups means that we can group related states together. Although only one in each group can be active at the one time, multiple states can be active across all the defined groups, and the control's appearance can indicate each of these together. So as per the previous example, the base look is updated to indicate the button is pressed, and the focus rectangle will be drawn to indicate the button has focus.

```
<VisualStateManager.VisualStateGroups>
  <VisualStateGroup x:Name="CommonStates">
    <!-- Define common states here -->
  </VisualStateGroup>
  <VisualStateGroup x:Name="Focused">
    <!-- Define focused states here -->
  </VisualStateGroup>
  <VisualStateGroup x:Name="Pressed">
    <!-- Define pressed states here -->
  </VisualStateGroup>
</VisualStateManager.VisualStateGroups>
```

Template Parts

The final concept to understand is that of template parts. A control may need to reference certain visual elements that belong to it in the code for that control (i.e., the behavior management code). Because of the strict separation between the behavior and the presentation and the ability for an alternative control template to be applied to the control, these elements need to be defined as required elements in any control template that is applied. These elements are known as *template parts*.

Theming

Theming (sometimes also known as *skinning*) is essentially the process of either styling or retemplating control types to customize their appearance (overriding their default styles or control templates). Themes are generally designed as a way of maintaining the look of the controls in the application in a single location in your project (ensuring consistency across your application and simplifying maintenance). However, they are also particularly useful when creating packaged software that should make use of corporate colors and display the customer's branding.

Based on the defined theme, the styles and control templates within that theme are automatically applied to their corresponding control instances (usually using implicit styles) wherever they may appear throughout the application. For example, you might have a number of themes that completely change the appearance of buttons. You might have a resource dictionary for each theme and define a different button control template for in each, and all the buttons in your application will automatically use the control template from the selected theme file rather than their default one.

Creating a Custom Theme

You can create your own theme file by creating a resource dictionary, placing all of your style resources in it, and merging it with the application's resources (as detailed earlier in the chapter). You can also define this resource file in a separate assembly, which you can then use in other projects (also described earlier).

You can define a new control template in the style resource to completely change the look of the control. Assign it to the control's `Template` property defined in the style, like so:

```
<Style TargetType="Button">
  <Setter Property="Template">
    <Setter.Value>
      <ControlTemplate TargetType="Button">
        <!-- Control template goes here -->
      </ControlTemplate>
    </Setter.Value>
  </Setter>
</Style>
```

If you make this style resource an implicit style (as demonstrated), all controls of the specified target type for the style will automatically have the control template applied (thus resulting in a theme-like behavior).

Silverlight Toolkit Themes

The Silverlight Toolkit includes an array of predefined themes that you may like to use in your project that retemplate the standard Silverlight controls (and the Silverlight Toolkit controls) to provide a new distinct look to your application. You can use these as they are or modify them to suit your own tastes. More information can be found on the Silverlight Toolkit's CodePlex site at <http://silverlight.codeplex.com>.

Silverlight Navigation/Business Application Themed Project Templates

Microsoft has started to release project templates that both restructure the visual layout of the application and retheme all the core Silverlight and Silverlight Toolkit controls (in other words, define new control templates for all these controls).

You can download and install these from the Visual Studio Online Gallery, via Visual Studio's Extension Manager. From the Online Gallery, navigate to the Templates ► Silverlight category, and you will find them there. At the time of this writing, only the themes for the Silverlight Navigation Application project types are available for installing as project templates, but you can download Silverlight Business Application projects implementing these themes at www.microsoft.com/downloads/details.aspx?FamilyID=e9da0eb8-f31b-4490-85b8-92c2f807df9e&displaylang=en.

You will find a project template for each theme you install in the Silverlight category of Visual Studio's New Project dialog, which you can then use when creating a project.

New themes will be released over time but, at the time of this writing, include the following:

- Accent Color
- Cosmopolitan
- Windows 7

Icons and Images

Effective use of icons and images throughout your application can really enhance its visual appeal. Designing quality icons is really the job of a graphics designer, but if you don't have the budget for a graphics designer, there is still the viable (and relatively cheap) option of purchasing a stock icon library. Most libraries come as raster files (such as .png files), though some companies are starting to include or sell separately the original vector images.

■ **Note** While .png files work perfectly well in Silverlight, in some instances, you may prefer to use vector (i.e., XAML-based) images instead. Vector images are scalable without losing quality, can be modified easily, and are often smaller than the equivalent raster image (depending on the complexity of the image). However, vector images are also more CPU intensive, so you need to choose the appropriate format for your particular requirements. As a general rule, for small images (such as icons) that don't need to be scaled raster images are a better choice. For larger images (especially those that will change size), vector images may better suit.

Note that vector icons often come as Adobe Photoshop (.psd), Adobe Illustrator (.ai), or Scalable Vector Graphics (.svg) files and need to be converted to XAML (possibly losing some definition along the way) to be used within a Silverlight application. There is a plug-in available for Adobe Illustrator to export to XAML (if you have Adobe Illustrator), or you can open these files from within Expression Blend (using the File ► Import Adobe Illustrator File menu option) or Expression Design (using File ► Open). Opening .ai files in Expression Design is a little more successful but often results in black shapes where it couldn't render the corresponding element correctly (most likely due to using rendering features not available in Silverlight). Ideally, you want the icons and images already converted to XAML files when you purchase them to save you these problems.

If you do buy icons or images formatted as XAML, ensure that they are formatted for Silverlight and not WPF (the file formats differ slightly, so if they are formatted for WPF, they will not be usable in Silverlight without modification). That said, if the Expression Design file type is also included, you can use that to export a new XAML file targeting Silverlight instead.

Animation

Once you've got the look of your application right, you can apply some animation effects to give it a bit of additional flair. Animating certain aspects of your user interface (such as page transitions) can add a certain pizzazz to your application and add to its "wow" factor.

However, you should use animations sparingly, so you do not distract users from their primary task when using your application. Animation can impress the user the first couple of times but can quickly become an annoyance. In business applications in particular, you must be aware of this as users will often spend considerable time in their working day with the application, and poorly implemented animations get old quickly. Transition animations (if you decide to use them) should be fast and not slow the user's interaction with the application.

Animations use storyboards to define the times at which various parts of the animation start and stop. Silverlight then handles the process of getting from one frame to another between the animation's start and stop time.

Creating animations is a huge topic and beyond the scope of this book. If you are creating animations, Expression Blend is an essential tool to help you in this task, with a lot of functionality geared toward designing animations.

Pixel Shaders / Effects

Pixel shaders enable you to add effects to Silverlight controls (by modifying the pixels of their rendered output) before displaying them on the screen. Typical effects include shadows, blurs, reflections, and color alteration, but the options are almost endless.

There are two pixel shaders included in the Silverlight 4 runtime:

- DropShadowEffect
- BlurEffect

Applying one of these effects is as easy as assigning the pixel shader to the Effect property of a control. The following example demonstrates assigning a drop shadow to a control (with the result shown in Figure 9-1):

```
<Button Content="Shadow" Width="75" Height="23">
  <Button.Effect>
```

```

        <DropShadowEffect Opacity="0.5" />
    </Button.Effect>
</Button>

```



Figure 9-1. A button with a drop shadow effect

You can create your own pixel shaders using the High-Level Shader Language (HLSL), which you then need to compile into a .ps file (using the fxc command-line tool included with the DirectX SDK). However, a quick web search will probably turn up a pixel shader that provides the effect that you are after; many are available for download.

■ **Note** Pixel shader effects are rendered in software. Generally, they can be rather CPU intensive to render, so you should use them sparingly to avoid negatively affecting the performance of your application.

Miscellaneous Styling Tips

Here are a few miscellaneous tips that you may find useful when designing your style resources.

Defining Constants in XAML

In instances where you want to maintain a set of constant values that can be used by multiple controls or style resources but maintained in a single location, you can simply define those values as resources along with a key you can use to reference them. For example, here we are defining a color constant that we can use:

```
<SolidColorBrush x:Key="BackgroundColorResource" Color="LemonChiffon" />
```

You can then assign this resource directly to a control (using the StaticResource markup extension):

```
<TextBox Background="{StaticResource BackgroundColorResource}" />
```

Or you can use it in a style resource definition:

```

<Style x:Key="UserFieldsStyleBase" TargetType="TextBox">
    <Setter Property="Margin" Value="2" />
    <Setter Property="Background"
        Value="{StaticResource BackgroundColorResource}" />
</Style>

```

This tip can be particularly useful when defining theme files, where you may define a core color scheme using constants that you can maintain in one place and simply change their values between theme files.

Restoring a Control's Default Style

If you have an implicit style being applied to a control that you do not wish to have any styling applied, you can stop the style being applied by setting the `Style` property of the control to null, for example:

```
<TextBox Style="{x:Null}" />
```

This control will retain its default style instead.

Summary

In this chapter, you learned the fundamentals of styling and templating Silverlight controls. Note that styles shouldn't be randomly defined as required but designed with a view of how they fit into the overall look of your user interface. When styling an application, it is best to take a wholistic view of the process and actually design how the styles will be applied across the application, rather than taking a piecemeal approach (assuming the user experience design process has taken place by this stage, providing the framework for our visual structure and the flow of the user interaction). In Chapter 11, "Creating Custom Controls," we'll look at actually building controls that can be styled and template using this model.

CHAPTER 10



Advanced XAML and Data Binding

Throughout this book, you’ve learned how to work with and write XAML, and you’ve also learned how to bind to data. By structuring your application well and using the available binding mechanisms effectively, you can reduce the amount of code required in the code-behind (which can be difficult to test) and have the XAML take control of displaying the appropriate data and letting the user interact with it. However, at times, you will probably come up against difficult scenarios when trying to achieve this outcome, whose solution may not be immediately obvious. In this chapter, we will take a deep dive into the more advanced features of XAML and data binding in Silverlight and provide you with the tips, tricks, and techniques to harness the full power of XAML and the data binding engine.

Advanced XAML

We covered the basics of XAML back in Chapter 3, “An Introduction to XAML,” and expanded on most of those concepts through the following chapters. Let’s now look at a few handy advanced features of XAML that you will find very useful when building your user interfaces.

Comments in XAML

You can use the standard XML tags (which are incidentally the same as HTML’s) for inserting comments into your XAML. Begin the comment with `<!--`, and end it with `-->`, for example:

```
<!-- A XAML comment -->
```

At times, you might also want to comment out some XAML (instead of deleting it), for example:

```
<!-- <TextBlock Text="{Binding Path=Name}" /> -->
```

Note, however, that you can’t nest comments. For example, attempting to comment out the following XAML (which already contains a comment)

```
<Grid>
    <!-- <TextBlock Text="{Binding Path=Name}" /> -->
</Grid>
```

will result in invalid XAML, as the end comment of the comment currently in the XAML will cause the outer comment to end too early. Therefore, you will need to remove (or comment out) any comments currently in the XAML first. MoXAML Power Toys (discussed later in this chapter) has a solution for this however.

■ **Note** As a shortcut, use the key chord Ctrl+K+C in Visual Studio to comment out the selected XAML, and use Ctrl+K+U to uncomment it again.

Defining Constants

Back in Chapter 9, “Styling Your Application,” we briefly discussed defining constants in XAML, such as brush resources. In Chapter 3, “An Introduction to XAML,” I also mentioned that strings could potentially be defined as resources, but we didn’t investigate the topic further at the time. Let’s look at how we can define strings and other types (such as integers and doubles) as constants in XAML now. This technique can be particularly useful for maintaining common values in a theme file (for example), and supporting localization in your application by maintaining a set of strings in resource dictionaries (where the appropriate resource dictionary can be merged in with your application’s resources).

■ **Note** Only a limited number of types from the `System` namespace can actually be declared in XAML. For example, you cannot define a `DateTime` using this method.

The first step is to declare a namespace prefix to the `System` namespace in the root element of your XAML file where the constant is to be defined, like so:

```
xmlns:sys="clr-namespace:System;assembly=mscorlib"
```

You can then declare the constants (as resources) using the types in that namespace:

```
<sys:String x:Key="OKText">OK</sys:String>
```

■ **Note** You can declare these constants as you would any other resources, to the `Resources` property of any element in the object hierarchy. The constant can then be used by any control below that element in the object hierarchy. Alternatively, you can define the constants in a resource dictionary and merge it into the object hierarchy where required.

You can then use the `StaticResource` markup extension (as you would with any resource) to make use of the constant:

```
<Button Content="{StaticResource OKText}" />
```

■ **Note** When assigning the constant to a content property (such as in the preceding example), you must use attribute syntax rather than content element syntax (as discussed in Chapter 3, “An Introduction to XAML”).

Let’s put it all together in a simple example:

```
<UserControl x:Class="Chapter12Sample.Views.ConstantsExample"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:sys="clr-namespace:System;assembly=mscorlib">
    <UserControl.Resources>
        <sys:String x:Key="OKText">OK</sys:String>
    </UserControl.Resources>
    <Grid x:Name="LayoutRoot">
        <Button Content="{StaticResource OKText}" />
    </Grid>
</UserControl>
```

Using OR to Combine Enumerated Values in XAML

Sometimes, you find that you need to use OR to join together a number of values from an enumeration and assign the result to a property in XAML. The `CommandButtonsVisibility` property on the `DataForm` control is a perfect example of this. As was demonstrated in Chapter 7, “Building Data Entry Forms,” this property determines what command buttons should be displayed in the form by using OR to bring together the corresponding values from the `DataFormCommandButtonsVisibility` enumeration. Whereas you might do the following in code

```
ProductsDataForm.CommandButtonsVisibility = DataFormCommandButtonsVisibility.Add |
    DataFormCommandButtonsVisibility.Delete;
```

you can do the same in XAML by simply concatenating the value names together, separated by commas:

```
CommandButtonsVisibility="Add,Delete"
```

■ **Note** This only works on control properties that accept a combination of enumeration values. Most control properties will tend to only accept a single value from a set of enumerated values.

Triggers, Actions, and Behaviors

Ideally, we want as little code as necessary to be put in the code-behind in a view. Designers especially don’t want to have to write and edit code to perform simple tasks. Instead, they want to define as much of the behavior of the user interface as possible by simple drag, drop, and set property operations in the

Expression Blend designer (with that behavior defined declaratively in XAML). However, to implement simple actions, you will often find that you need to turn to the code-behind.

What's more, most of these behaviors are fairly generic, often following a common pattern and implemented regularly. Take this scenario, for example: Let's say you have a Save button, and when the button is clicked, you want to call the `SubmitChanges` method on a `DomainDataSource` control. As discussed back in Chapter 7, "Building Data Entry Forms," despite being able to retrieve and display data declaratively via the `DomainDataSource` control, when it comes to submitting those changes, you need to turn to the code-behind to call that method. You need to handle the `Click` event of the button, and in the event handler in the code-behind, you call the `SubmitChanges` method on that `DomainDataSource` control. As you can imagine, for projects that use the `DomainDataSource` control to request data from and submit it to the server, implementing this type of behavior is a regular task. However, there *is* an alternative to performing the steps listed earlier, by using a combination of a trigger and an action instead—features provided by the Expression Blend Interactivity library.

Using Predefined Triggers and Actions

A trigger waits for something to happen (usually by listening for a control event to be raised), and when that event is raised, it invokes one or more actions (which do something in response). Unlike WPF, Silverlight doesn't (for the most part) have a concept of triggers and actions. The `EventTrigger` is the only trigger available, and it will only respond to the `Loaded` event of a control and start an animation, meaning that it has limited use (and any use of it is generally discouraged). Because of this lack of triggers and actions in Silverlight (and due to the inability to extend WPF's triggers and actions), the Expression Blend team created its own triggers and actions implementation (in addition to another concept called behaviors, discussed shortly).

Although designed for use with Expression Blend, you can also use any of the predefined triggers, actions, and behaviors based on this implementation in Visual Studio (although there will be no designer support like there is in Expression Blend). In addition, you can write your own triggers, actions, and behaviors.

Adding the Required Assembly References

To use (and create) triggers, actions, and behaviors, you will need to start by adding a reference to the `System.Windows.Interactivity.dll` assembly to your project. Note that this isn't a part of the Silverlight SDK but is instead installed as a part of Expression Blend.

■ **Note** If you don't have Expression Blend, you can get this assembly by downloading the Expression Blend SDK for free from the Microsoft web site at

www.microsoft.com/downloads/details.aspx?displaylang=en&FamilyID=75e13d71-7c53-4382-9592-6c07c6a00207.

Depending on whether you have installed it as part of Expression Blend or its SDK, you should find it under the `Silverlight\v4.0\Libraries` path of the corresponding product.

Understanding how Triggers and Actions Interact

Let's consider the scenario previously described (clicking a button calls the `SubmitChanges` method on a given `DomainDataSource` control) and how we can implement it using triggers and actions. Start by breaking this scenario down:

- The *trigger* will be the button being clicked, so to capture this, we'll need to use a trigger that can listen for an event raised by a control.
- In response to the trigger, the *action* will call the `SubmitChanges` method on a `DomainDataSource` control, so we'll need to use an action that can call a method on a given control.

Note that both the trigger and the action that we've extracted from this scenario are very generic. You can keep it at this very generic level if you wish, which has the advantage that you can simply use the predefined generic trigger and action that fill these requirements (no need to write your own). However, being so generic has the disadvantage that each time you want to implement the scenario, you will need to configure both the trigger and the action to perform their tasks (you will need to assign the event to listen for on the trigger, and you will need to assign the method to call on the target control for the action). Thus you may wish to create a custom trigger and action to provide a more targeted solution to the problem and reduce the configuration burden on the developer/designer.

■ **Note** You can find a range of generic triggers, actions, and behaviors in the `Microsoft.Expression.Interactions.dll` assembly. This (like the `System.Windows.Interactivity.dll` assembly) is installed with Expression Blend (and the Expression Blend SDK), so you can add a reference to it in your project to use the triggers/actions/behaviors that it provides. For example, in this assembly, you will find triggers such as `KeyTrigger`, `DataTrigger`, `PropertyChangedTrigger`, and `TimerTrigger` (you will actually find `EventTrigger` in `System.Windows.Interactivity.dll`). In the `Microsoft.Expression.Interactions.dll` assembly, You will also find actions such as `HyperlinkAction`, `ChangePropertyAction`, and `CallMethodAction`. There are also plenty of sources of triggers, actions, and behaviors available on the Web. The best places to start finding predefined triggers/actions/behaviors would be <http://expressionblend.codeplex.com> and <http://gallery.expression.microsoft.com>.

Let's start by looking at how to implement the generic solution first and then move onto writing and implementing a custom trigger and action to create a more targeted solution.

Configuring Triggers and Actions using Expression Blend

The easiest way to configure triggers and actions by far is via Expression Blend. Triggers and actions (and behaviors) were specifically designed for use by designers so that they didn't need to write and edit code, and thus Expression Blend has extensive support for them. In Expression Blend, you can simply drag an action from the asset library (you'll find them in the Behaviors category, as shown in Figure 10-1) and drop it on a control.

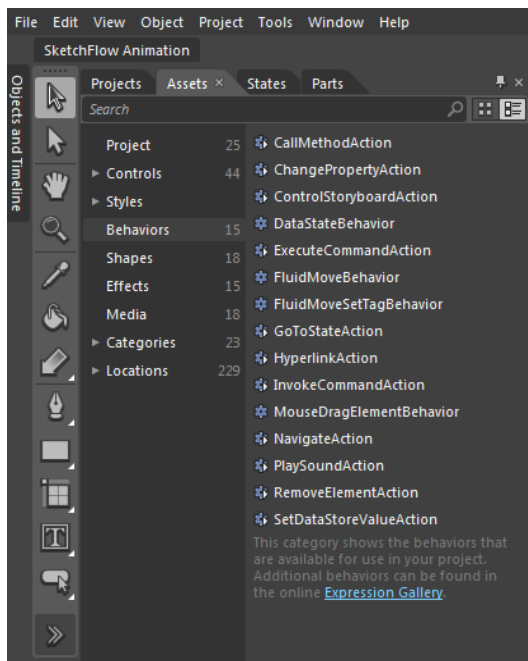


Figure 10-1. Behavior assets in Expression Blend

Blend will automatically create a trigger and assign the action to it. You can then configure the trigger and action via the Properties window. Visual Studio unfortunately doesn't have this level of support built in; instead, you must manually define the trigger and action in XAML by hand (although you can configure the trigger and the action once you've created them via the Properties window).

If you plan to leverage triggers and actions extensively in your application, using the Expression Blend designer (instead of Visual Studio's) will make the whole process much easier and faster.

■ **Note** Coverage of Expression Blend is beyond the scope of this book. However, Kirupa Chinnathambi has a great blog post on using behaviors in Expression Blend at <http://blog.kirupa.com/?p=351>. You assign triggers and actions to controls in Expression Blend in much the same way as he demonstrates with behaviors. In addition, an article on the SilverlightShow.net web site by Andrej Tozon also walks you through using behaviors with Expression Blend; see www.silverlightshow.net/items/Exploring-Silverlight-behaviors-look-Ma-no-code.aspx.

Implementing Triggers and Actions by Hand in XAML

Start by adding references to the `Microsoft.Expression.Interactions.dll` and `System.Windows.Interactivity.dll` assemblies to your project (as described earlier) if you haven't

already done this. The next step to implementing a trigger or action is to define the interactivity and interactions namespace prefixes in your XAML file:

```
xmlns:i="http://schemas.microsoft.com/expression/2010/interactivity"
xmlns:ei="http://schemas.microsoft.com/expression/2010/interactions"
```

The first is for the core Expression Blend interactivity functions (from which you'll primarily use the `EventTrigger`), and the second is for the additional predefined interactions from the `Microsoft.Expression.Interactions.dll` assembly (e.g., `PropertyChangedTrigger`, `ChangePropertyAction`).

Now, find the Save button in the XAML, define the `Interaction.Triggers` attached property on the button, and add an `EventTrigger` trigger to it, specifying that it should listen for the `Click` event on the control it is attached to:

```
<Button Content="Save" Height="23" Width="75">
  <i:Interaction.Triggers>
    <i:EventTrigger EventName="Click">
      <!--Assign action(s) here-->
    </i:EventTrigger>
  </i:Interaction.Triggers>
</Button>
```

■ **Note** The control that triggers will listen to events on defaults to the control on which you define the `Interaction.Triggers` attached property. You can listen for events raised on a different control instead if you wish by binding the `SourceObject` property of the trigger to that control (using `ElementName` binding, as discussed later in this chapter).

Now that you've configured your trigger, you need to assign one or more actions to it (in place of the "Assign action(s) here" comment) for it to invoke. Here, we are calling the `SubmitChanges` method on a given `DomainDataSource` control.

```
<ei:CallMethodAction TargetObject="{Binding ElementName=productsDDS}"
  MethodName="SubmitChanges"/>
```

■ **Note** The `TargetObject` property of the action is used to specify upon which you want the action to act. If the action does not have this property set, the action will act upon the control that it is attached to.

Now, run your application. You will find that pressing the button will call the `SubmitChanges` method on the `DomainDataSource` control—all implemented declaratively, with no code necessary.

As you can see, this is a useful feature for both designers and developers alike. However, it is very generic and thus, accordingly, requires additional configuration (setting the event to listen for on the

trigger and the method to call on the action) to what may be required by a custom and more targeted solution. Let's now try to reduce the amount of configuration required by creating a custom trigger and action.

Creating a Custom Trigger

Let's start by creating a custom trigger. When writing custom triggers (and this applies to actions and behaviors too), you are encapsulating a piece of logic in a reusable interaction component that you can attach to a control. Although you are writing these components in code, (in essence) you are writing this code in a reusable componentized fashion to save you from writing code when it comes to actually applying the logic to elements in your user interface (where you can then apply and configure this logic declaratively in XAML).

Let's create a trigger that will invoke the associated action(s) when the `Click` event is raised on the `Button` control it is attached to. Add a new class to your project called `ButtonClickTrigger`. This class should inherit from either `TriggerBase` or `TriggerBase<T>` (from the `System.Windows.Interactivity` namespace). By having it inherit from `TriggerBase<T>`, you are able to constrain the trigger to only applying to a given control type. For this example, it will inherit from `TriggerBase<Button>`.

The next step is to override the `OnAttached` and `OnDetaching` methods and handle any events on the attached control that will be used to trigger the corresponding action(s) (including removing those event handlers when detaching). In our trigger, we want to handle the `Click` event of the button, so we handle that event on the control exposed by the `AssociatedObject` property on the base class.

The final step is to invoke the actions that have been associated with the trigger, by calling the `InvokeActions` method on the base class. This takes a parameter that allows an object to be passed to the actions, but we can assign it to null because we have nothing to pass.

The code for the trigger follows; as you can see, it is very simple:

```
public class ButtonClickTrigger : TriggerBase<Button>
{
    protected override void OnAttached()
    {
        base.OnAttached();

        AssociatedObject.Click += new RoutedEventHandler(AssociatedObject_Click);
    }

    protected override void OnDetaching()
    {
        base.OnDetaching();

        AssociatedObject.Click -= new RoutedEventHandler(AssociatedObject_Click);
    }

    private void AssociatedObject_Click(object sender, RoutedEventArgs e)
    {
        InvokeActions(null);
    }
}
```


You can now use this in place of the more generic `EventTrigger` trigger, without the need for configuring the event name. Note that, of course, you will have to assign a namespace prefix to its namespace in your XAML file in order to be able to use it:

```
xmlns:b="clr-namespace:AdventureWorksLOB.Behaviors"
<Button Content="Save" Height="23" Width="75">
  <i:Interaction.Triggers>
    <b:ButtonClickTrigger>
      <!--Assign action(s) here-->
    </b:ButtonClickTrigger >
  </i:Interaction.Triggers>
</Button>
```

■ **Note** For more complex triggers, you can add properties to the class to enable them to be configured when they are used and use those values in your logic. This also applies to actions and behaviors. You can implement them as standard CLR properties, but before you are able to bind to them, they will need to be dependency properties. Implementing dependency properties is discussed in Chapter 11, “Creating Custom Controls.”

Creating a Custom Action

Let’s create a custom action that will be performed in response to a trigger. This action will call the `SubmitChanges` event on the target `DomainDataSource` control. Add a new class to your project called `SubmitChangesAction`. This class should inherit from `TriggerAction`, `TargetedTriggerAction`, or their generic counter parts (from the `System.Windows.Interactivity` namespace).

■ **Note** The difference between `TriggerAction` and `TargetedTriggerAction` is that `TriggerAction` acts on the control to which the action is assigned, while the `TargetedTriggerAction` acts on a different control (as specified by the designer/developer).

For this example, the control we are attaching the action to is a button, but we actually need to act on a `DomainDataSource` control. Therefore, we’ll inherit from `TargetedTriggerAction<DomainDataSource>`.

■ **Note** Something to be aware of is that the generic parameter `<T>` for the `TriggerAction<T>` action is used to specify the type of control that it can *be attached to* (and will designate the type of the `AssociatedObject` property on the base class), while the generic parameter `<T>` for the `TargetedTriggerAction<T>` action is used instead to specify the type of control that it can *target* (and will designate the type of the `Target` property on the base class). If you need to constrain what type of control that a `TargetedTriggerAction` action is attached to, you can decorate the class with the `TypeConstraint` attribute, passing this attribute the type of control it can be attached to as a parameter.

The next step is to override the `Invoke` method. You can now implement your action's logic in this method. Note that this method has a single parameter, which will pass into the method whatever object was passed to the parameter of the trigger's `InvokeActions` method. The following code is for an action that calls the `SubmitChanges` method on the target control when it is invoked:

```
public class SubmitChangesAction : TargetedTriggerAction<DomainDataSource>
{
    protected override void Invoke(object parameter)
    {
        Target.SubmitChanges();
    }
}
```

■ **Note** There are additional methods from the base class that you can override if you require. All the trigger base classes have an `OnAttached` and `OnDetaching` method that you can override (as is available when creating your own triggers), and if you are inheriting from `TargetedTriggerAction` (or its generic counterpart), you can also override its `OnTargetChanged` method.

Assigning it as an action to be invoked by our `ButtonClickTrigger` trigger, our final trigger/action declaration to implement the scenario in XAML is

```
<Button Content="Save" Width="75">
    <i:Interaction.Triggers>
        <b:ButtonClickTrigger>
            <b:SubmitChangesAction TargetObject="{Binding ElementName=productsDDS}" />
        </b:ButtonClickTrigger>
    </i:Interaction.Triggers>
</Button>
```

To make this trigger/action pair even easier to use in Expression Blend, you can specify a default trigger that the action will be assigned to when it is dropped onto a control. Our `SubmitChangesAction` action has been designed for use in conjunction with a `ButtonClickTrigger` trigger, so ideally, when the action is dropped onto a control, a `ButtonClickTrigger` trigger should be created automatically for the `SubmitChangesAction` action to be assigned to. We can tell Expression Blend which trigger the action should use by default by decorating the action class with the `DefaultTrigger` attribute, like so:

```
[DefaultTrigger(typeof(Button), typeof(ButtonClickTrigger))]
```

The first parameter specifies the type of control that the action is attached to, and the second parameter specifies the type of trigger to use when attached to a control of that type. For example, in the preceding example, we are specifying that when the action is attached to a `Button` control, it should use the `ButtonClickTrigger` trigger by default. You can apply this attribute multiple times to the class if you wish, enabling you to specify a different default trigger for each type of control the action is attached to. You can also pass additional values to be assigned to the trigger by passing additional parameters to the `DefaultTrigger` attribute.

■ **Note** Generally, if a trigger and an action are tightly coupled, it's often better to use a behavior instead.

This is a very simple action that we've created, but you can create much more complex actions depending on the needs of your user interface. However, they are discrete atomic blocks of interaction logic only; they do something and then exit (you can think of them as being much like a method in code). Actions cannot wait for another event to be raised, nor can they preserve their state between trigger invocations. If you need to provide a more persistent interaction, implementing behaviors may be a more suitable alternative.

Creating and Using Behaviors

Triggers and actions often come under the banner of behaviors, so when you read about behaviors, often you will find that the author is actually referring to triggers and actions rather than behaviors (which can make the concept somewhat confusing). Even Expression Blend includes actions in the Behaviors category in the Asset Library.

A *behavior* is actually a different concept than triggers and actions (although they do share some similarities). Whereas triggers and actions are defined separately (and can be mixed and matched), behaviors combine these two concepts into a single component, tightly coupling the trigger and the action. Behaviors are attached and detached from a control in the same way as triggers are (and thus are alive for the lifetime of the control they are attached to). However, unlike a trigger, which expects to have one or more associated actions to perform the interaction logic, the behavior handles all the interaction logic internally.

At a conceptual level, you can think of a behavior as being designed to add additional functionality to the control that it is attached to. This may be a simple trigger/action combination or may implement a more involved interaction such as a drag-and-drop type operation.

Behaviors inherit from either the `Behavior` class or the generic `Behavior<T>` class and are found in the `System.Windows.Interactivity` namespace.

One of the big advantages of combining triggers and actions into a single component is that it enables a preservation of state between the “trigger” events. A typical scenario where you might want to maintain state between events is implementing a drag-and-drop operation. To implement this type of operation, you need to handle multiple events (including the `MouseDown`, `MouseMove`, and `MouseUp` events), and maintain the state of the drag-and-drop operation between them.

Behaviors typically define additional behavior and logic for the control that they are attached to as opposed to interacting with other controls (unlike the custom action we created that interacts with a `DomainDataSource` control when a button is clicked).

For example, a typical behavior to add to a text box might be one that automatically selects all the text when the text box gets the focus. This behavior entirely revolves around the text box it is attached to only, without interacting with any other controls.

However, there's nothing stopping you from adding the ability to interact with other controls in a behavior; there's simply no built-in provision for it (i.e., there's no `TargetedBehavior` base class provided like there is a `TargetedTriggerAction` base class).

Let's revisit our previous example of calling the `SubmitChanges` method on a `DomainDataSource` control when a button is clicked, but this time, we will implement it as a behavior:

```
public class SubmitChangesBehavior : Behavior<Button>
{
```

```

[CustomPropertyValueEditor(CustomPropertyValueEditor.ElementBinding)]
public DomainDataSource Target
{
    get { return (DomainDataSource)GetValue(TargetProperty); }
    set { SetValue(TargetProperty, value); }
}

public static readonly DependencyProperty TargetProperty =
    DependencyProperty.Register("Target", typeof(DomainDataSource),
        typeof(SubmitChangesBehavior), null);

protected override void OnAttached()
{
    base.OnAttached();

    AssociatedObject.Click += new RoutedEventHandler(AssociatedObject_Click);
}

protected override void OnDetaching()
{
    base.OnDetaching();

    AssociatedObject.Click -= new RoutedEventHandler(AssociatedObject_Click);
}

private void AssociatedObject_Click(object sender, RoutedEventArgs e)
{
    Target.SubmitChanges();
}
}

```

■ **Note** Comparing this code to that of the custom trigger and action that we created, you'll notice that the behavior essentially just brings the trigger's and action's code together into a single component. Because there is no target concept in a behavior, we needed to implement a property that the designer/developer could assign the `DomainDataSource` control to act upon. So the designer/developer could bind this property to a `DomainDataSource` control, we needed to make it a dependency property, which will be explained further in Chapter 11, "Creating Custom Controls."

You can now implement this behavior in XAML like so:

```

<Button Content="Save" Width="75">
    <i:Interaction.Behaviors>
        <b:SubmitChangesBehavior Target="{Binding ElementName=productsDDS}" />
    </i:Interaction.Behaviors>
</Button>

```

Which Should I Use?

The big question you probably want an answer to is when should you use a behavior over a trigger/action combination? Ultimately, there are no fixed rules, and in some cases, there is a bit of overlap where either means is appropriate (as was demonstrated with our button click/submit changes example).

If the trigger and action are intrinsically tied together and you want to minimize the amount of configuration necessary when using them, then a behavior is probably the ideal candidate to implement the interaction logic. If you have complex interaction logic where state needs to be preserved between event handlers (such as a drag and drop operation), then a behavior is essentially your only choice.

However, if you want to provide a more flexible use story and have interaction logic that could be triggered by different types of controls (and/or in different types of ways), then implementing a separate trigger and action may be a better plan of action.

The possibilities are endless with triggers, actions, and behaviors, and going back through the examples provided throughout this book until now, you could replace almost all of the code-behind that we used with a trigger/action combination or a behavior. Now that you understand these concepts, every time you consider adding code to the code-behind, first consider whether encapsulating the logic in a trigger/action pair or a behavior might be a better alternative.

Triggers, actions, and behaviors are a very powerful tool at your disposal, promoting both code reuse and interaction logic modularization/encapsulation. They are testable, and (arguably) the biggest advantage is that they reduce/remove the need for designers to write interaction logic code.

Blendability

To be able to effectively design your application's user interface, designers often like to know the shape of the data that they are binding to (enabling them to take full advantage of the Expression Blend or Visual Studio XAML designer without resorting to editing the XAML directly). This allows them to see what fields are available in the data, and bind to them as required.

It can also be beneficial to be able to populate a view with data at design time to view the resulting output of its design as the user would see it (instead of having to constantly compile and run the application to do so).

These features make the process of designing user interfaces more interactive, and are known as *blendability*. As you may have guessed from the name, this concept primarily relates to user interface design in Expression Blend, but we can also harness these design time features in the Visual Studio XAML designer via the same underlying mechanism (although with less support than provided by Expression Blend).

The Design-Time Data Mechanisms

As you may recall from Chapter 3, "An Introduction to XAML," the design time data mechanisms can be found in the Expression Blend 2008 namespace (<http://schemas.microsoft.com/expression/blend/2008>), which is automatically declared in your XAML files with a prefix of *d*. In this namespace, you will find the following data-related attached properties:

- `DataContext`
- `DesignData`
- `DesignSource`

You will also find the `DesignData` and `DesignInstance` markup extensions. We'll be using these to provide the blendability features in the XAML designer.

Defining the Shape of the Data

Let's start by defining the type of data that we will bind to, so that the data binding expression builder (discussed later in this chapter) knows the structure of the data that we are binding to. Back in Chapter 6, "Implementing Summary Lists," you may have noticed that when you drag an entity (returned from the server via RIA Services) from the Data Sources window and drop it onto the design surface, the `DomainDataSource` control that's created for you has a design time data property already applied:

```
<riaControls:DomainDataSource AutoLoad="True"
    d:DesignData="{d:DesignInstance my1:ProductSummary,
        CreateList=true}"
    Height="0" Width="0"
    LoadedData="productSummaryDDS_LoadedData"
    Name="productSummaryDDS"
    QueryName="GetProductSummaryListQuery">
    <riaControls:DomainDataSource.DomainContext>
        <my:ProductsContext />
    </riaControls:DomainDataSource.DomainContext>
</riaControls:DomainDataSource>
```

The rather confusingly named `d:DesignData` design time attached property (it has nothing to do with the `d:DesignData` markup extension, discussed shortly) is used to define the type of data that is served by a data source control (via its `Data` property). Hence, this property can only be applied to data sources (such as the `DomainDataSource` control). The benefit of applying this design-time property to the `DomainDataSource` control is that you will now be able to select the properties to bind to in the data binding expression builder (on the specified entity) when binding to data that it returns (as you'll see in Figure 10-5 later in this chapter).

To configure this property, we assign it a `d:DesignInstance` markup extension. This markup extension is primarily used to return an object of a given type (or a collection of objects of that type). The default property on this markup extension is the `Type` property. You'll note from our example that it specifies that the `DomainDataSource` control will be returning `ProductSummary` entities. You will also note that the `CreateList` property of this markup extension is set to true. This is used to indicate whether the `d:DesignInstance` markup extension should return a *collection* of the given object type or a single instance of the object type.

By default, the `d:DesignInstance` markup extension reflects over the given object type and creates and returns a corresponding substitute design time type (or an empty collection of that type if the `CreateList` property is set to true), which the XAML designer can use where required. However, it also has an `IsDesignTimeCreatable` property, which, when set to true, will actually instantiate and return an object of the given type (which is only of use when the `CreateList` property is set to false). Once you assign values to that object's properties in its constructor, these values will appear in the XAML designer—essentially populating it with data at design time.

You can also assign the `d:DesignInstance` markup extension to the `d:DataContext` design time attached property. Whereas the `d:DesignData` property could only be applied to data source controls (which serve up data), the `d:DataContext` property is designed to be applied to controls that will bind to data via their `DataContext` property. Like data assigned to the `DataContext` property of a control, the value assigned to the `d:DesignContext` property will be inherited down the object hierarchy. For example, setting the `d:DataContext` property on a `Grid` (as demonstrated by the `TextBlock` control in the

following example) will enable the data binding expression builder to display the available properties that the controls within the Grid can be bound to.

```
<Grid d:DataContext="{d:DesignInstance Type=my1:ProductSummary}">
    <TextBlock Text="{Binding Path=Name}" />
</Grid>
```

Using Sample Data

We've made the designer much friendlier for creating bindings, so let's now look at populating it with some sample data at design time. Although we can populate a single instance of an object with data in its constructor and set the `IsDesignTimeCreatable` property on the `d:DesignInstance` markup extension to true to populate the XAML designer with some data (as was previously demonstrated), it is hardly an ideal means for doing so. The design-time data has to be hard-coded in the class (not a good idea), and this method is only suitable when binding to a single object (rather than a collection of objects). There is another option however, and that's using the `d:DesignData` markup extension (in place of the `d:DesignInstance` markup extension).

The `d:DesignData` markup extension has only a single property, named `Source` (which is, therefore, its default property). The design-time data needs to be defined in a separate XAML file, and you simply point the `Source` property to a XAML file that defines the design-time data, and that data will then be displayed in the XAML designer. The easiest way to create this XAML file and populate it with data is from within Expression Blend; there, you can import some sample data from XML, create random data conforming to the structure of a given class, or simply create your own structure before populating it with random data as required. Explaining how to create sample data files is beyond the scope of this book, but you can find more information on doing so in Expression Blend in the Expression Blend help or manually in Visual Studio at this MSDN page: [http://msdn.microsoft.com/en-us/library/ff602279\(VS.95\).aspx](http://msdn.microsoft.com/en-us/library/ff602279(VS.95).aspx). The sample data XAML files should have a `Build Action` of either `DesignData` (where a substitute design-time type will be generated and used in place of the objects that the sample data is populating), or `DesignDataWithDesignTimeCreatableTypes` (where the actual objects the sample data is populating are instantiated and used at design time). You can then use the `d:DesignData` markup extension to point to the sample data XAML file and assign it to the `d:DataContext` attached property as required:

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

Creating Consolidated Namespaces

Back in Chapter 3, “An Introduction to XAML,” we briefly discussed the concept of consolidated namespaces, where multiple namespaces could be combined into a single namespace declaration. This enables you to use the one namespace prefix covering multiple namespaces and/or assemblies, instead of declaring separate prefixes for each of these individually. You can recognize a consolidated namespace, as it has a URI namespace instead of a CLR namespace. The default and the XAML namespaces that you find declared in all XAML files are examples of consolidated namespaces, as is the SDK namespace:

```
xmlns:sdk="http://schemas.microsoft.com/winfx/2006/xaml/presentation/sdk"
```

These consolidated namespaces are predefined for you, but it is entirely possible for you to create your own if you wish using the `XmlnsDefinition` attribute (assuming that there is no overlap between

the class names in each separate namespace/assembly that you wish to include as a part of the consolidation).

There are a number of reasons why you might like to do so. One is to reduce the number of namespace prefix declarations you need to add to each XAML file. Another is to ensure consistency of the prefixes defined for each namespace across XAML files—different developers on a project using different prefixes will make maintaining the application a painful chore. An additional advantage is that the developers don't need to know or worry about which namespace a particular class or control is in to use it.

Let's take a look at creating a consolidated namespace to consolidate a number of namespaces in your Silverlight project. In the `AssemblyInfo.cs` file in your project (which you can find under the `Properties` folder), start by adding a `using` directive to the `System.Windows.Markup` namespace. Now, you need to come up with a URI to refer to the consolidated namespace you are about to create. A rough standard for doing so is something along these lines:

```
"http://schemas.yourbusinessurl.com/XAML"
```

Now, simply add attributes to your assembly that each associate a CLR namespace with this consolidated namespace URI:

```
[assembly: XmlnsDefinition("http://schemas.yourbusinessurl.com/XAML",
    "AdventureWorksLOB.Purchasing")]
[assembly: XmlnsDefinition("http://schemas.yourbusinessurl.com/XAML",
    "AdventureWorksLOB.ViewModels.Production")]
[assembly: XmlnsDefinition("http://schemas.yourbusinessurl.com/XAML",
    "AdventureWorksLOB.ViewModels.Sales")]
```

Next, you can declare a namespace prefix in a XAML file to this consolidated namespace URI, and then all the classes or controls across the combined CLR namespaces will be accessible via that prefix:

```
xmlns:vm="http://schemas.yourbusinessurl.com/XAML"
```

If you wish, you can define a default prefix for that consolidated namespace using the `XmlnsPrefix` attribute:

```
[assembly: XmlnsPrefix("http://schemas.yourbusinessurl.com/XAML", "vm")]
```

This prefix will be used by Visual Studio and Expression Blend to define the corresponding namespace prefix in the XAML file (if necessary) when you drag a control (whose namespace is included in a consolidated namespace) from the toolbox onto the designer surface.

■ **Note** In the next chapter, we discuss creating custom controls, and you may also find this technique useful when creating custom control libraries in order to consolidate all the controls from different namespaces within that library assembly into a single URI namespace (which you define within that library assembly).

MoXAML Power Toys

MoXAML Power Toys is a handy open source Visual Studio add-in by Pete O’Hanlon, providing a number of XAML and code helper functions for both Silverlight and WPF developers, which you can download at <http://peteohanlon.wordpress.com/moxaml-power-toys/>.

Some of the most useful features include the following:

- It includes a commenting helper. If you try to comment out XAML in the normal fashion previously described and that XAML already contains comments, you will end up with invalid XAML. This function handles this scenario by first modifying the existing comment markers within the selected XAML, resulting in valid XAML instead.
- A scrubber feature that cleans and formats the XAML in a file according to given options, such as converting tabs to spaces, putting attributes on their own line, and so on.
- You are able to create a user control from the selected XAML using the Extract User Control feature.
- As described in Chapter 3, “An Introduction to XAML,” by default all member variables created for controls are scoped as public. The `AddFieldModifier` function in MoXAML Power Toys goes through a XAML file and sets the `FieldModifier` property of each control with a name to “private”.
- As discussed in Chapter 7, “Building Data Entry Forms,” for bindings to be aware when the bound property’s value has changed, the bound object needs to implement the `INotifyPropertyChanged` interface and raise its `PropertyChanged` event when each property’s value is changed. This requires a backing member variable and additional code—automatically implemented properties do not implement this behavior. To simplify the task of creating properties that implement this behavior, the Notify Properties feature in MoXAML Power Toys enables you to right-click an automatically implemented property and select the Notify Property menu item from the MoXAML Power Toys submenu (in the context menu), and it will turn the property into a fully implemented property, with a backing member variable, and raise the `PropertyChanged` event.
- Creating dependency properties (discussed properly in Chapter 11, “Creating Custom Controls”) can be rather complex task. The Create Dependency Property feature pops up a window that will help you create a dependency property with the options that you specify.

You can get help for how to use these functions from the MoXAML Power Toys web site.

Advanced Data Binding

So far in this book, you’ve learned how to bind your user interface controls to objects such that the user can view and edit data exposed by these objects. Instead of data being pushed into these controls, the extensive data binding support provided by XAML enables them to pull the data in instead (in other words they are essentially controlling the process of consuming the data). You’ve learned that

- The object assigned to the `DataContext` property of a control is inherited down the object hierarchy.
- There are three different binding modes: `OneTime`, `OneWay`, and `TwoWay` (discussed in Chapter 3, “An Introduction to XAML”). You need to set the mode to `TwoWay` in order to be able to update the bound object property via a control in the user interface.
- You can notify the bindings when bound property values on an object have changed (using the `INotifyPropertyChanged` interface, discussed in Chapter 7, “Building Data Entry Forms”).
- You can notify the bindings when bound property values (or the bound object itself) are invalid (by throwing exceptions, implementing the `IDataErrorInfo` interface, or by implementing the `INotifyDataErrorInfo` interface, as discussed in Chapter 7, “Building Data Entry Forms”).
- The `ObservableCollection` generic type can be used to maintain items in your collections, which will notify the bound user interface controls when items have been added or removed from the collection so they can update themselves accordingly (as discussed in Chapter 6, “Implementing Summary Lists”). Alternatively, you can implement the `INotifyCollectionChanged` interface in your own collection classes to enable them to exhibit the same behavior.
- You can create a “view” of a collection by wrapping it in a collection view, enabling the data in the collection to be manipulated (e.g., filtering, sorting, grouping, and paging) without actually modifying the underlying collection. Collection views also provide a current record pointer (that tracks the current item in the collection), enabling multiple controls bound to the same collection view to be kept synchronized. They also provide the ability to move this pointer and navigate through the items in the collection.

In this section, we’ll discuss some of the more advanced data binding features supported by Silverlight.

Assigning the Source of a Binding

When we first discussed data binding back in Chapter 3, “An Introduction to XAML,” you learned that a binding has both a source and a target, with the binding source obtained from the bound control’s `DataContext` property. The object assigned to the `DataContext` property is inherited down through the object hierarchy, so an object assigned to the `DataContext` property of a Grid control for example would automatically be available to all the controls contained within that Grid via their own `DataContext` property.

However, what if you wanted a property on a control to bind to something *other* than the control’s `DataContext` property (such as a resource, or a property on another control)? Let’s take a look at the different means available to achieve this in Silverlight.

Using the Binding's Source Property

If you set the `Source` property of the binding, it will use the given object as the binding source instead of the object assigned to the `DataContext` property of the control. If you're assigning this property in XAML, you will most likely be binding it to a resource (using the `StaticResource` markup extension).

For example, let's say you have a `Product` object that is defined as a resource (with `productResource` as its key) somewhere up the object hierarchy (defining and instantiating a class as a resource for binding to will be discussed later in this chapter). The standard way of binding to this resource is by assigning it to the `DataContext` property of the target control using the `StaticResource` markup extension. For example, binding the `Name` property of the `Product` object to the `Text` property of a `TextBox` control, you would use the following XAML:

```
<TextBox DataContext="{StaticResource productResource}" Text="{Binding Name}" />
```

Using the `Source` property, the following binding expression will provide the same results as the previous one:

```
<TextBox Text="{Binding Name, Source={StaticResource productResource}}" />
```

The `Source` property provides you with an advantage in the situation where you want to bind a control's property to a given source, without altering the inherited data context for its child controls (where that data context has been set somewhere further up the hierarchy) or altering the binding source for its other properties.

■ **Note** Alternatively, if you are creating the data binding in code (which will be discussed later in this chapter), you can assign any object to this property to act as the source of the binding.

ElementName Binding

In Chapters 6 and 7, we were binding the `ItemsSource` property of various controls to the `Data` property of a `DomainDataSource` control but didn't go into depth on how this was achieved. The `DomainDataSource` control was retrieving data from the server and exposing that data publicly via its `Data` property. We then consumed that data by binding the `ItemsSource` property of our `ListBox/DataGrid/DataForm` control to the `DomainDataSource` control's `Data` property. To bind a control's property to the property of another (named) control in the view like this, we needed to use a special kind of binding called `ElementName` binding. Using the `ElementName` property on a binding, you could provide it the name of a control (within the same name scope) that would act as the source for the binding (instead of the object assigned to the control's `DataContext` property). Let's take a look at some examples of employing `ElementName` binding to bind the properties of two controls together in XAML.

A very simple example is to link the `Text` property of two `TextBox` controls. When you modify the text in one text box, the second text box's text will update automatically:

```
<StackPanel>
    <TextBox Name="FirstTextBox" />
    <TextBox Name="SecondTextBox" Text="{Binding Text, ElementName=FirstTextBox}" />
</StackPanel>
```

■ **Note** If you change the mode of the second text box's binding to `TwoWay`, modifying the second text box's text will also update the first text box. That said, in this direction the first text box will only be updated once the second text box loses focus.

Similarly, we can display the current value of a Slider control in a text block by binding the Slider control's `Value` property to the TextBlock control's `Text` property:

```
<TextBlock Text="{Binding ElementName=sourceSlider, Path=Value}" />
```

The following XAML demonstrates taking the `ItemsSource` property of a `ListBox` control (acting as the target), and binding it to the `Data` property of a `DomainDataSource` control:

```
<ListBox ItemsSource="{Binding ElementName=productSummaryDDS, Path=Data}" />
```

This example demonstrates binding the `IsBusy` property of the `BusyIndicator` control to the `IsBusy` property of a `DomainDataSource` control (which will display the `BusyIndicator` control when the `IsBusy` property of the `DomainDataSource` control is true):

```
<controlsToolkit:BusyIndicator
    IsBusy="{Binding ElementName=productSummaryDDS, Path=IsBusy}" />
```

In this final example, we are binding the `Target` property of the `Label` control to a text box. Note that by not specifying a path for the binding, we are binding the `Target` property to the text box itself, not a property on that text box:

```
<sdk:Label Content="Name:" Target="{Binding ElementName=ProductNameTextBox}" />
```

RelativeSource Binding

You may have noticed that there is a `RelativeSource` property on the `Binding` markup extension. This enables you to bind to a source relative to the target. To obtain a reference to the source to assign to this property, you need to use the `RelativeSource` markup extension `RelativeSource` markup extension. Somewhat limited in comparison to the corresponding markup extension in WPF (which can search for an ancestor of a particular type), the `RelativeSource` markup extension only supports two modes: `Self` and `TemplatedParent`.

The `Self` mode returns the target control and is useful for binding together two properties on the same control. This is generally most useful when you have attached properties on a control that you want to bind to properties on the control itself. For example, perhaps you want to display a tooltip when the user hovers over a text box, displaying the full contents of that text box in the tooltip (where perhaps it doesn't all fit in the confines of the text box). Here, we'll use the `ToolTipService.ToolTip` attached property (which you may remember being demonstrated back in Chapter 3, "An Introduction to XAML"), and bind it to the `Text` property of the text box using the `Self` mode of the `RelativeSource` markup extension:

```
<TextBox Text="{Binding Path=CurrentItem}"
          ToolTipService.ToolTip="{Binding Text,
          RelativeSource={RelativeSource Self}}" />
```

The `TemplatedParent` mode is only applicable when the control is contained within a control template or a data template. It returns the templated item and enables the target property to be bound to a property on that item. When used inside a data template (such as the data template for a `ListBox` item), the `TemplatedParent` mode will return the content presenter for that templated item. Note that it doesn't actually return the `ListBoxItem`; the data template is actually being applied to the content presenter of the item, so it's the content presenter that will be returned (unlike when using a control template to template the `ListBoxItem`, where the `TemplatedParent` mode will return the `ListBoxItem` itself).

For example, the following binding expression in a data template will get the actual height of the content presenter:

```
"{Binding RelativeSource={RelativeSource TemplatedParent}, Path=ActualHeight}"
```

Another scenario where this binding may be useful in a data template is when you want to get the data context of the templated item. The data context is still inherited down the hierarchy in data templates, but if you assign a different data context on a control in the data template (such as a `Grid` for example), you could use this method to obtain and bind to the original data context for the item again.

■ **Note** Used in a control template, the binding expression `"{Binding RelativeSource={RelativeSource TemplatedParent}}"` is the equivalent of the `"{TemplateBinding}"` markup extension. Unlike the `TemplateBinding` markup extension however (which only supports a one way binding), the `RelativeSource` markup extension can be used as a two way binding, which can be very useful in some scenarios when creating control templates for custom controls. This will be discussed further in Chapter 11, "Creating Custom Controls."

Binding a Control Property to its Data Context

You can bind a property of a control to the binding source assigned to its `DataContext` property (either assigned directly to the property on the control or inherited down the object hierarchy), by simply setting its value to `"{Binding}"`. This is a common scenario when binding a number of controls to the same inherited data context (e.g., the data context of a `Grid` control is bound to a collection, and multiple controls within that `Grid` inherit that data context and use it as their binding source). To bind the `ItemsSource` property of a `ListBox` control in the `Grid` to the collection, you can set its binding expression to `"{Binding}"`.

■ **Note** The `"{Binding}"` binding expression is equivalent to `{Binding Path=.,}`, which is also the same as `{Binding Path=}`.

Binding to a Property in the View's Code-Behind

You've seen how you can bind to a property on bound objects, resources, and other controls in the view, but what if the source of the property you want to bind to is actually the view's code-behind class? There's two ways you can go about this. The first is to bind the `DataContext` property of the root element in your XAML file to itself using the `RelativeSource` binding, like so:

```
DataContext="{Binding RelativeSource={RelativeSource Self}}"
```

This will get a reference to the code-behind class for the XAML file and bind the `DataContext` property to it, which will then be inherited down the object hierarchy. Any controls within that object hierarchy will therefore automatically use the code-behind class as their binding source. For example, if you have a property named `UserName` in the code-behind, displaying the value of that property in a `TextBlock` control within the view would therefore simply require the following XAML:

```
<TextBlock Text="{Binding UserName}" />
```

However, this method does have the pitfall of requiring that no controls further up in the object hierarchy have been assigned a different data context (overriding our `RelativeSource` binding with a different binding). If this is the case, there is another method you can use instead. Start by giving the root element in your XAML file a name (via its `Name` property). In this example we have named it `ControlRoot`. Now, you can simply use `ElementName` binding to assign that element as its binding source:

```
<TextBlock Text="{Binding UserName, ElementName=ControlRoot}" />
```

■ **Note** This technique is particularly useful when creating user controls that expose properties to the host. The properties are defined in the code-behind; the host sets their values; and the XAML can consume those values directly via these bindings (without requiring the intervention of the code-behind apart from getting or setting the property values). However, custom controls have a slightly different means of handling this type of scenario, as will be discussed in the next chapter.

Binding to a Resource

Throughout this book so far, most of the data bindings that we have implemented have obtained their source data either from a `DomainDataSource` control (via `ElementName` binding, using a pull-type mechanism) or by assigning the source data to either the `DataContext` property or the `ItemsSource` property of a control from the code-behind (using a push-type mechanism). As has been discussed along the way (with the pros and cons of each discussed back in Chapter 6, “Implementing Summary Lists”), the ability for controls to request data using a pull-type mechanism so that they can populate themselves is generally the better approach. However, by using a `DomainDataSource` control to enable the pulling of data, we are effectively tightly coupling our user interface and data access logic, which isn't ideal.

In Chapter 12, “The Model-View-ViewModel (MVVM) Design Pattern,” we'll look at implementing a design pattern that enables a pull type approach to data binding (with the XAML in charge of

instantiating the source object and binding to it), but without the tight coupling provided by the `DomainDataSource` control.

However, let's take the first few steps into implementing a part of that design pattern, by exposing data from a custom data class (which is known in the MVVM design pattern as a `ViewModel`), defining and instantiating it as a resource in a XAML file, and then binding the controls to its properties.

■ **Note** We are using aspects of the `ViewModel` concept from the MVVM pattern for this example, but you don't need to be using the full MVVM design pattern in order to take advantage of the techniques demonstrated here. There are many other reasons and uses why you would want to define and instantiate objects as resources and bind to them.

Creating a Class to Bind To

Start by creating the class that we will use as our binding source (generally you will have a `ViewModels` folder in your project for these classes), and name it `ProductViewModel`. Add some properties to it that we will bind to in the XAML (eg. `Name`, `ProductNumber`, etc):

```
public class ProductViewModel
{
    public string Name { get; set; }
    public string ProductNumber { get; set; }
}
```

For the purposes of experimentation, simply assign some hard-coded data to the properties on the class in the class's constructor. Alternatively, you can set the property values in the XAML when defining and instantiating it as a resource.

■ **Note** The class must have a default constructor (i.e., a constructor without any parameters). If you do not define any constructors, a default constructor will be created for you, but otherwise, you will need to ensure that there is one added to the class.

Instantiating the Class as a Resource in XAML

Now, declare a namespace prefix in your XAML file for the namespace where the class can be found:

```
xmlns:vm="clr-namespace:AdventureWorksLOB.ViewModels"
```

The next step is to define and instantiate the class as a resource:

```
<vm:ProductViewModel x:Key="productResource" />
```

You can also assign values to the properties on the class at this stage if required:

```
<vm:ProductViewModel x:Key="productResource" Name="Bike" ProductNumber="B001" />
```

■ **Note** Only a few data types are supported to be assigned natively in XAML (such as strings, Booleans, ints, and doubles). Other more complex types (such as Decimals and DateTimes) require type converters to be applied to their properties in order to handle the conversion. Creating and using type converters will be discussed in Chapter 11, “Creating Custom Controls.”

Binding to the Resource

Now, you can bind to this object resource (using the `StaticResource` markup extension to reference it) however you wish:

```
<TextBox Text="{Binding Name, Source={StaticResource productResource}}" />
```

■ **Note** Any data that you’ve hard-coded in the class or assigned via the resource’s properties will appear at design time in the bound controls.

Automatically Creating and Binding the Resource Using the Data Sources Window

Now that you’ve seen the manual way of defining the resource and creating controls that bind to it, there is a somewhat easier method to have these done automatically. Back in Chapter 6 - Implementing Summary Lists, you saw how to drag an entity (returned from the server via RIA Services) from the Data Sources window and onto the design surface, with that action automatically creating a `DomainDataSource` control to retrieve the data from the server and controls to display and edit the entity.

Although you won’t find the class that you created listed in the Data Sources window, you can add it (via the Add New Data Source button) and then drag and drop it onto the design surface. A resource will be created (although it will be a `CollectionViewSource` wrapping a collection of the given objects, slightly different from what was demonstrated earlier), and controls will be created to bind to it. This method expects the source of the `CollectionViewSource` to be set in the code-behind, and you will find that by dragging the class from the Data Sources window onto the design surface automatically adds some (commented out) code to the code-behind demonstrating how to do so.

Defining Resources in the Code-Behind (For Binding To)

Back in Chapter 8, “Securing Your Application,” we discussed how the `WebContext` object (generated and used by RIA Services) was added to the application’s resources when the application starts up. This resource will be available application-wide, enabling all the views in the application to bind to it.

If you open up the `App.xaml.cs` file and find the `Application_Startup` event handler method, you will find this line (well, at least in Silverlight projects that were created using the Silverlight Business Application project template that is):

```
this.Resources.Add("WebContext", WebContext.Current);
```

As you can see, it simply specified the element to which the resource was to be added (i.e., the `Application` object), provided it with a key (`WebContext`), and passed it the object to be added as a resource (`WebContext.Current`). You can then bind to this in XAML as you would any other resource:

```
<Button Content="Click Me!"
        IsEnabled="{Binding Path=User.IsAuthenticated,
                           Source={StaticResource WebContext}}" />
```

■ **Note** This is binding via nested properties (`User.IsAuthenticated`), which will be discussed further in the next section.

This actually demonstrates a useful technique that you may like to use yourself. Instead of instantiating objects as resources in XAML, you can alternatively instantiate them and add them as resources from the code-behind (applying any additional configuration to the object as required).

Binding to Nested Properties

Generally, when creating a binding, you’ll be binding to a single property on the source object. For example, the `DataContext` property of a `TextBox` control might be assigned a `Person` object, and you might bind to the `FirstName` property on that object (with the `FirstName` property returning a string):

```
<TextBox DataContext="{StaticResource personResource}" Text="{Binding FirstName}" />
```

However, what if the `Person` object doesn’t have a `FirstName` property but instead has a `Name` property that returns a `PersonName` object, and it’s this object that has the `FirstName` property (so you’re effectively dealing with nested properties). How do you handle that sort of binding? This is possible by using the same style of dot notation you use in C# code to traverse down the object hierarchy:

```
<TextBox DataContext="{StaticResource personResource}"
        Text="{Binding Name.FirstName}" />
```

You can traverse down multiple levels of the object hierarchy for as far as you need.

Binding to Indexed Properties

In addition to binding to nested properties, it's also possible to bind to indexed properties (with both integer and string indexes). The index selection syntax is much the same as it is in C#, simply append the index to the name of the property returning the collection surrounded by square brackets. For example, if a **Person** object had an **Addresses** property that returned a collection of **Address** objects, you could bind to the first **Address** object using the integer index syntax:

```
<TextBox DataContext="{StaticResource personResource}"
    Text="{Binding Address[0].AddressLine1}" />
```

If the collection has string indexes (such as a **Dictionary<string, string>**), you can refer to the item using the string index syntax, like so:

```
<TextBox DataContext="{StaticResource personResource}"
    Text="{Binding Address[HOME].AddressLine1}" />
```

where **HOME** is the string index of the **Address** object to return.

If the given index is not found, the binding will silently fail (although an error will be written to the Output window in Visual Studio, as will be discussed later in this chapter when discussing debugging data bindings).

Binding to Collection Views

As you've already seen, collection views are used to wrap a collection, enabling the data in the collection to be manipulated (e.g., filtered, sorted, grouped, and paged) without actually modifying the underlying collection. When you bind to a collection, some controls that manipulate the data for display internally wrap that collection in a collection view themselves (such as the **DataGrid**, which enables the data to be sorted). You can, however, pass the control a collection view instead in order to share the state of the data in that control externally (such as the sorting and selected item).

For example, the **DomainDataSource** control returns data from its **Data** property wrapped in a **PagedEntityCollectionView** view, and data from its **DataView** property wrapped in a **DomainDataSourceView** view. You can also wrap a collection in a collection view yourself if you wish (either using a **PagedCollectionView** or obtaining one via the **CollectionViewSource** view proxy).

Back in Chapter 6, "Implementing Summary Lists," we discussed how to manipulate the data in a collection using a collection view to control how that data was displayed in a control bound to it. We also briefly discussed in that chapter how collection views provide a current record pointer (that tracks the current item in the collection) and exposes a number of methods and properties that you can use to move this pointer and navigate through the items in the collection.

The primary benefit of this feature is to enable multiple controls that are bound to the same collection view to be kept synchronized, and it's this ability that we'll be focusing on here.

Let's now take a deeper look at this synchronization behavior, and some not so obvious implementation details for binding to collection views.

Binding to Objects in a PagedCollectionView

In this example, we'll bind to the objects contained within a **PagedCollectionView** collection view.

■ **Note** For the discussion here, the `PagedEntityCollectionView` collection view and the `DomainDataSourceView` collection view have the same binding and synchronization behavior as the `PagedCollectionView` collection view. Therefore, we will focus on the behavior of the `PagedCollectionView` collection view here, but the content applies equally to those other types of collection views.

For this example, we will have a collection of `Person` objects, which are to be displayed in a `ListBox` control. We also have a `TextBox` control and want to show the name of the selected person in the list box in it (with the text changing in the text box whenever a new item is selected in the list box). Figure 10-2 demonstrates the final output.

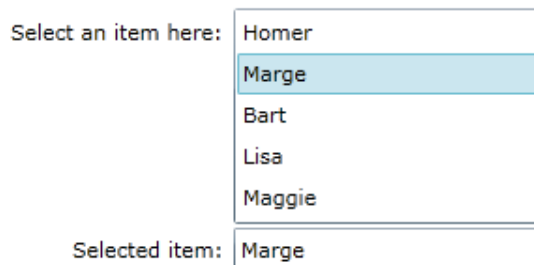


Figure 10-2. The output from this example (with minor embellishments)

■ **Note** Admittedly, for this particular example, you could just as easily (if not more easily) use `ElementName` binding (binding the `Text` property of the `TextBox` control to the `SelectedItem` property of the `ListBox` control), but it's useful to look at how this scenario can be implemented using a collection view that they both bind to (with the collection view tracking the current item). One of the advantages of using a collection view for this task (over `ElementName` binding) is that it enables you to track and manage the current item from a bound class (such as a `ViewModel`, when using the MVVM design pattern), without actually needing to know anything about the control in the bound view that is binding to it. For example, say you have a `ViewModel` class that exposes a collection view containing a number of items, which a `ListBox` in the view binds to. The `ViewModel` class can find out which item is selected in that `ListBox` from the collection view itself, without knowing what type of control is binding to it, or having to directly reference that bound control.

Creating the Object Collection

The first step is to create a class to represent a person (which we'll name `Person` and create in the `Models` folder in the project). For the purposes of this example, we'll simply have a single `Name` property on the object:

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

■ **Note** Of course, you could use a collection of strings for a simple example like this instead, but binding to a collection of objects is the common scenario that we want to demonstrate.

Now, create a class that populates and returns a collection of **Person** objects that we can bind to, wrapped in a **PagedCollectionView**. The following **SampleData** class creates a collection of strings (populated with some names of a popular animated family), wraps the collection in a **PagedCollectionView**, and exposes this view publicly:

```
public class SampleData
{
    public PagedCollectionView PeopleView { get; set; }

    public SampleData()
    {
        List<Person> people = new List<Person>();
        people.Add(new Person() { Name = "Homer" });
        people.Add(new Person() { Name = "Marge" });
        people.Add(new Person() { Name = "Bart" });
        people.Add(new Person() { Name = "Lisa" });
        people.Add(new Person() { Name = "Maggie" });
        PeopleView = new PagedCollectionView(people);
    }
}
```

It's a good idea to compile your application at this stage, because you will be able to see the data that you've added to the collection in the constructor of the class in the XAML designer.

Instantiating the Class as a Resource and Binding to It

Go to the XAML designer, and define the class as a resource (you will also have to declare the **models** namespace prefix):

```
<models:SampleData x:Key="sampleDataResource" />
```

Now, add a **ListBox** control and a **TextBox** control to a **StackPanel**. Bind the **StackPanel**'s **DataContext** property to the **PeopleView** property on the resource you created, and then bind the **ListBox** control's **ItemsSource** to its **DataContext** property, and bind the **TextBox** control's **Text** property to the **Name** property on the **Person** object:

```
<StackPanel DataContext="{Binding Path=PeopleView,
    Source={StaticResource sampleDataResource}}">
    <ListBox ItemsSource="{Binding}" DisplayMemberPath="Name"

```

```
        Height="120" Width="120" />  
        <TextBox Text="{Binding Name}" Width="120" />  
    </StackPanel>
```

■ **Note** You may (understandably) find the TextBox control's binding in the preceding example a little confusing. Although the data context being inherited is a collection view, we are binding directly to a property on the view's current item object (which returns a `Person` object). So what is going on here? In this type of scenario, the data binding engine recognizes that the binding source is a collection view and assumes that you don't actually want to bind to the collection view itself. It, therefore, drills down to the view's `CurrentItem` property and binds to the given property on that instead. If you wish, you can use "`{Binding CurrentItem.Name}`" as the binding expression instead (with exactly the same results), but explicitly drilling down to the `CurrentItem` property on the collection view isn't actually necessary (unless you have a property on the object you want to bind to that has the same name as a property on the collection view object, such as `Count`).

You'll note that the designer populates the controls automatically with the data you defined in the class, with the `ListBox` control displaying the strings and the `TextBox` control displaying the selected string. If you now run your project, you will find that the text in the text box changes accordingly as you select different items in the list box.

How Does This Example Work?

The reason this example works is that, when the `ListBox` control is bound to a collection view (and you will see this behavior with other controls that bind to a collection of items, such as the `ComboBox`, `DataGrid`, and `DataForm`), it automatically (by default) synchronizes the `CurrentItem` property of the collection view with its selected item. When you select a different item in one of those controls, they update the `CurrentItem` property of the collection view to the newly selected item, such that any controls bound to that property will have their binding source changed accordingly.

Although most of the time, this is the behavior that you will want, sometimes you may wish to have more control over the process. You will find that some of these controls (such as the `ListBox` control) have an `IsSynchronizedWithCurrentItem` property, whose default value is `null`. You can't actually set this property to `true` (it will throw an exception if you try), as it depends on whether or not the binding source is a collection view for whether synchronization with the current item is possible. If the binding source is a collection view, current item synchronization will automatically be turned on, although you can explicitly turn it *off* if you wish by setting the property to `false`. If you do so, you can control what item is the current item in the collection view manually (in code), using the record navigation methods (such as `MoveCurrentToNext` and `MoveCurrentToPosition`).

■ **Note** As an experiment, try simply binding both controls directly to the collection (i.e., without it being wrapped in a collection view), and observe the result. You'll find that the items still appear in the `ListBox` control, but the `TextBox` control is now trying to bind to a `Name` property on the collection itself (rather than an object in the collection), which doesn't exist, and thus that binding will fail.

Binding to Objects in a CollectionViewSource

In the last example, we implemented a collection view using a `PagedCollectionView` collection view, but this time we will use a `CollectionViewSource` view proxy instead. You may remember (from Chapter 6 - Implementing Summary Lists) that unlike the `PagedCollectionView` (which is a collection view), the `CollectionViewSource` isn't actually a view as such, but a view proxy. You assign a collection to its `Source` property, and it provides a corresponding collection view from its `View` property (in the form of a `ListCollectionView`). Let's return a `CollectionViewSource` view proxy from our `SampleData` class now instead of a `PagedCollectionView` collection view:

```
public class SampleData
{
    public CollectionViewSource PeopleViewSource { get; set; }

    public SampleData()
    {
        List<Person> people = new List<Person>();
        people.Add(new Person() { Name = "Homer" });
        people.Add(new Person() { Name = "Marge" });
        people.Add(new Person() { Name = "Bart" });
        people.Add(new Person() { Name = "Lisa" });
        people.Add(new Person() { Name = "Maggie" });

        PeopleViewSource = new CollectionViewSource();
        PeopleViewSource.Source = people;
    }
}
```

You can still bind to the objects exposed by this collection view proxy in exactly the same way as we did with the `PagedCollectionView` collection view:

```
<StackPanel DataContext="{Binding Path=PeopleViewSource,
    Source={StaticResource sampleDataResource}}">
    <ListBox ItemsSource="{Binding}" DisplayMemberPath="Name"
        Height="120" Width="120" />
    <TextBox Text="{Binding Name}" Width="120" />
</StackPanel>
```

However, once again, you probably will find this binding somewhat confusing (as it was when binding to the `PagedCollectionView` collection view), but this time for a different reason. Unlike the `PagedCollectionView`, which is itself a collection view, the `CollectionViewSource` is not a collection view - it is a collection view proxy. To get to the view itself, we should have to go via its `View` property (for the `ListBox` binding), and to get to the current item we should have to access its `View.CurrentItem` property! However, we are doing neither in the XAML above!

So once again, you may be wondering what's going on here, but you will find that that answer is pretty much the same. This time, the binding engine recognizes that it is binding to a `CollectionViewSource`, and assumes that it's not the `CollectionViewSource` itself that you want to bind to, so it automatically drills down to its `View` property to get to the collection view to bind to (for the `ListBox` binding), and automatically drills down to its `View.CurrentItem` property to get to the object to bind to (for the `TextBox` binding). The `CollectionViewSource` essentially becomes transparent when binding to it, such that you are binding to the collection view that it exposes, rather than the `CollectionViewSource` itself.

■ **Note** You can turn off this View property drilling behavior for the `CollectionViewSource` by setting the `BindsDirectlyToSource` property on the property bindings to true. You may wish to do this if (for example) you wanted to bind to the `Source` property of the `CollectionViewSource` instead of its `View` property, in which case you will need to explicitly define the full path for the binding from the `CollectionViewSource` level.

Binding to a collection view, and particularly binding to a `CollectionViewSource` can be a confusing concept, but once you understand it, it makes the binding process simpler and more succinct.

■ **Note** One of the benefits of binding to a `CollectionViewSource` is that the collection view doesn't need to be defined in code. Instead, you can define a `CollectionViewSource` as a resource in XAML if you wish, like so:

```
<models:SampleData x:Key="sampleDataResource" />
<CollectionViewSource x:Key="peopleViewSource"
    Source="{Binding PeopleCollection, Source={StaticResource sampleDataResource}}" />
```

where `PeopleCollection` is a property on the `SampleData` class listed earlier in place of the `PeopleViewSource` property, returning the collection as a `List<Person>` (instead of a `CollectionViewSource`).

Enhancing Data Binding

The `Binding` class has a number of properties that you can use to further configure a data binding rather than simply connecting the value of the target property directly to the source property. These properties are the keys to enabling the power and flexibility of Silverlight's data binding capabilities, so let's take a look at them now.

String Formatting

Back in Chapter 7, "Building Data Entry Forms," we covered the ability for formatting a bound property's value using the `StringFormat` property of the binding before it is passed to the target control to be consumed. As a recap, this can be a very useful feature when binding to a nonstring property (such as a `DateTime`, or a `decimal`). You can use the `StringFormat` property on the binding to set the format (using the same types of formatting strings as you would use if you were to format the value using its `ToString` method in code (using either standard or custom formats). For example, the following binding uses a standard formatting string to display the bound property's value in the text box formatted as currency:

```
<TextBox Text="{Binding TotalCost, Mode=TwoWay, StringFormat=C}" />
```

Or you can format a date as a long date (e.g., Sunday, May 16, 2010) using a standard formatting string:

```
<TextBox Text="{Binding StartDate, Mode=TwoWay, StringFormat=D }>
```

format a date as a custom format (e.g., 2010-05-16) using a custom formatting string:

```
<TextBox Text="{Binding StartDate, Mode=TwoWay, StringFormat=yyyy-MM-dd }>
```

You can also format a decimal, single, double, or float to a given number of decimal places (e.g., 3.14) using a custom formatting string:

```
<TextBox Text="{Binding PI, Mode=TwoWay, StringFormat=0.00 }>
```

If you want to include commas or other special characters in your string format, enclose the formatting string within apostrophes (not quotes, as they are used to delimit the whole property assignment):

```
<TextBox Text="{Binding StartDate, Mode=TwoWay, StringFormat='MMM dd, yyyy' }>
```

Alternatively, you can simply escape them by inserting a backslash (\) prior to the special character:

```
<TextBox Text="{Binding StartDate, Mode=TwoWay, StringFormat=MMM dd\, yyyy }>
```

■ **Note** The available standard formatting strings and custom format specifiers that can be used for formatting strings are detailed in the following MSDN topic (and its related topics): <http://msdn.microsoft.com/en-us/library/26etazsy.aspx>. Another source with examples can be found in the Visual Studio 2010 data binding expression builder (discussed later in this chapter).

Alternative Value When a Source Property Value is Null

If the value of a property being bound to is null, you can specify an alternate value to use instead using the `TargetNullValue` property of the binding. This is a useful feature when you are binding to a nullable type and want to provide a default value when one isn't provided, for example:

```
<TextBlock Text="{Binding TotalAmount, TargetNullValue=0 }>
```

Fallback Values

If the source of the binding is null (i.e., the value of both the `DataContext` property of the control and the `Source` property of the binding are null), or the property being bound to does not exist on the object assigned as the source of the binding, you can specify an alternative value to use instead using the `FallbackValue` property of the binding. For example, this binding will display “Oops!” if the binding source is null:


```
<TextBox Text="{Binding Name, Mode=TwoWay, FallbackValue=Oops! }>
```

UpdateSourceTrigger

Back in Chapter 7, “Building Data Entry Forms,” we discussed implementing the `IDataErrorInfo` interface to add transaction type behavior to our classes (begin, accept, or cancel changes). However, you may not have the luxury of implementing this interface in the classes that you bind to, so rather than rolling back unwanted changes (the behavior that the `IDataErrorInfo` interface enables), you could alternatively prevent changes being made to the object until you explicitly commit them (such as when the user clicks the OK button). This behavior can be enabled using the `UpdateSourceTrigger` property of the bindings. This property has an enumeration type with two possible values: `Default` and `Explicit`. `Default` (obviously) is the default value, where the source property of the binding is updated automatically by the binding when it has been changed via the user interface (usually once the control loses the focus). However, assigning `Explicit` as the value of this property means that in order for the source property of the binding to be updated, you must manually initiate the update.

Let’s look at a simple example of enabling the user to update the name of a product via a text box:

```
<TextBox Name="NameTextBox" Text="{Binding Name, Mode=TwoWay}" />
```

By default, as soon as this text box loses the focus, the `Name` property on the bound object will be updated with the text entered in the text box accordingly. However, you can prevent this behavior by setting the `UpdateSourceTrigger` property of the binding to `Explicit`:

```
<TextBox Name="NameTextBox"
  Text="{Binding Name, Mode=TwoWay, UpdateSourceTrigger=Explicit}" />
```

Now, when the text box loses the focus, the bound value will *not* be updated. Say you have an OK button in the view, and you want to commit the changes made to the field back to the bound object when the button is clicked. To do so, you can handle the button’s `Click` event in the code-behind and get the binding expression for the text box’s `Text` property (using the `GetBindingExpression` method on the text box, which returns a `BindingExpression` object). You can then explicitly force the bound property of the binding to be updated using the `UpdateSource` method on the `BindingExpression` object, like so:

```
BindingExpression expr = NameTextBox.GetBindingExpression(TextBox.TextProperty);
expr.UpdateSource();
```

Value Converters

We’ve seen how you can format the bound value (using the `StringFormat` property of a binding) before passing it onto the target control, but what if you want to *convert* the value to a completely different value (given some predefined rules) instead? A perfect example is attempting to bind a Boolean value to the `Visibility` property of a control. This is not possible, because the `Visibility` property only accepts values from the `Visibility` enumeration (`Visible` or `Collapsed`). You could modify the property on the object that you are binding to in order to return a value from the `Visibility` enumeration instead, but this is not always possible (or ideal). There is, however, a way we can transform the Boolean value to a `Visibility` enumeration as a part of the binding process, and that’s using a *value converter*.

A value converter is a class that implements the `IValueConverter` interface (from the `System.Windows.Data` namespace), which has two methods:

- The **Convert** method is passed (as a parameter) the value of the property on the source object of the binding, and you can implement the required logic to convert it to the value to be assigned to the target property (returned as the result of the method).
- The **ConvertBack** method is used in two-way bindings to convert the target value (modified in the user interface, and passed into the method as a parameter) back to a value to be assigned to the bound property.

Creating a Simple Value Converter

Let's take a look at implementing a simple value converter to convert a Boolean value to a **Visibility** enumeration:

```
public class BoolToVisibilityValueConverter : IValueConverter
{
    public object Convert(object value, Type targetType, object parameter,
                        System.Globalization.CultureInfo culture)
    {
        return (bool)value ? Visibility.Visible : Visibility.Collapsed;
    }

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

Note that we didn't bother implementing any logic for the **ConvertBack** method, as bindings to the **Visibility** property of a control will only ever need to be one-time or one-way bindings (it doesn't expose user input). Therefore, this method will never be used and hence isn't worth implementing. However, for other types of conversions where you will want to convert the user's input back to something else, you will actually need to implement the logic to do so in this method.

■ **Note** If you used the Silverlight Business Application project template as the basis for your application, you will find that there are a few value converters already in your project (in the **Helpers** folder) that you can take a look at, such as the **NotOperatorValueConverter** value converter that converts true values to false, and false values to true.

Using a Value Converter

Once you've created your value converter, you need to instantiate it in the resources somewhere in your object hierarchy. This will probably be in the resources of a page / user control, but you could also assign it to the resources of the **Application** object (in the **App.xaml** file) in order to use the value converter application-wide. First, declare a namespace prefix to the namespace where your value converter can be found, for example:

```
xmlns:vc="clr-namespace:AdventureWorksLOB.ValueConverters"
```

Next, instantiate the value converter in the resources of a control in the object hierarchy:

```
<vc:BoolToVisibilityValueConverter x:Key="VisibilityConverter" />
```

Now, you can use this value converter by assigning it to the `Converter` property of the data binding (using the `StaticResource` markup extension to obtain a reference to it). The following example demonstrates hiding a Save button if the value of the `IsDirty` property of a bound object being edited (the object being inherited down the object hierarchy to the save button's `DataContext` property) is true:

```
<Button Content="Save" Height="23" Width="75"
  Visibility="{Binding IsDirty, Converter={StaticResource VisibilityConverter}}"/>
```

Passing a Parameter to the Value Converter

Sometimes you find that your need to pass additional information to the value converter in order to perform its task. You can pass the value converter an additional value (defined in XAML) using the `ConverterParameter` property of the binding.

Prior to Silverlight 4 (when support for the `StringFormat` property of the binding was added), value converters were commonly used to format the bound property value. The `ConverterParameter` property enabled generic formatting value converters to be created, with the actual string format to use passed to the converter using this property. Of course, this scenario is no longer necessary, but there are still plenty of other scenarios where this feature can be used.

A different scenario is one mentioned when discussing binding to `RadioButton` controls in Chapter 7, “Building Data Entry Forms,” where you want to bind each of the `RadioButton` controls to a single property on the bound object exposing an enumeration type, with each `RadioButton` control representing a different enumeration value. As was discussed in that chapter, this is not a scenario that you can implement using standard data binding, but using a value converter in the binding and passing it a parameter makes this possible. The following value converter demonstrates creating a generic enumeration-to-Boolean value converter that compares the passed in value with the parameter passed in, returning true if they match, false if they don't:

```
public class EnumEqualityValueConverter : IValueConverter
{
    public object Convert(object value, Type targetType, object parameter,
        System.Globalization.CultureInfo culture)
    {
        return value.ToString() == parameter.ToString();
    }

    public object ConvertBack(object value, Type targetType, object parameter,
        System.Globalization.CultureInfo culture)
    {
        return Enum.Parse(targetType, parameter.ToString(), true);
    }
}
```

■ **Note** In this scenario, we have a `TwoWay` binding (the user can select an alternative age bracket via the radio buttons), so we need to be able to convert the selected radio button back to the enumeration value that it represents via the value converter. Therefore, you'll note that we needed to actually implement the `ConvertBack` method for this converter.

Now, to use this value converter (assuming you have already declared a namespace prefix and instantiated it as a resource), simply assign the value converter to the `Converter` property of the `RadioButton` control's bindings and assign the enumeration value that each `RadioButton` control represents to the `ConverterParameter` property of the binding (which will be compared to the bound property's value by the value converter and will return true if they match):

```
<RadioButton Content="&lt; 20" IsChecked="{Binding AgeBracket, Mode=TwoWay,
    Converter={StaticResource EnumConverter}, ConverterParameter=BelowTwenty}" />
<RadioButton Content="20 to 29" IsChecked="{Binding AgeBracket, Mode=TwoWay,
    Converter={StaticResource EnumConverter}, ConverterParameter=Twenties}" />
<RadioButton Content="30 to 39" IsChecked="{Binding AgeBracket, Mode=TwoWay,
    Converter={StaticResource EnumConverter}, ConverterParameter=Thirties}" />
<RadioButton Content="40 Plus" IsChecked="{Binding AgeBracket, Mode=TwoWay,
    Converter={StaticResource EnumConverter}, ConverterParameter=FortyPlus}" />
```

Passing a Culture to a Value Converter

There is also a `CultureInfo` object passed into the value converter methods as a parameter. This can be used when displaying data that needs to be formatted according to a given culture.

By default, the language-culture passed into the value converter is obtained from the `Language` property assigned to the control (using the standard language-culture strings, such as `en-US`, denoting English language with US culture). This value is inherited down the object hierarchy, so if you assign a culture to this property on the root node in your XAML file, all the controls in that XAML file will also inherit that value. If a culture is not explicitly assigned via this property, then it will default to US English (`en-US`). You can explicitly pass a specific language-culture to a value converter using the `ConverterCulture` property on the binding. Any language-culture that you specify here will be passed into the value converter via the culture parameter.

Passing an Entire Object to a Value Converter

If you want to use multiple properties of the source object in the converter to produce a single output (such as combining the first name and surname properties into a single string for display), this is possible using a value converter too. Bind directly to the object itself in the binding (instead of a property on the object), and the entire object will be passed into the `Convert` method of the value converter via the value parameter. You can then cast it to the object type, get the first name and surname property values, combine them into a single string, and return that string as the result of the method, as demonstrated in the following code:

```
public class FullNameValueConverter : IValueConverter
{
    public object Convert(object value, Type targetType, object parameter,
```

```

        System.Globalization.CultureInfo culture)
    {
        Person person = value as Person;
        return person.FirstName + " " + person.Surname;
    }

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

```

■ **Note** For this type of scenario, you would usually be better adding a `FullName` property to your `ViewModel` (when using the Model-View-ViewModel (MVVM) design pattern, as discussed in Chapter 12) and binding to that instead. However, if you are not implementing the MVVM pattern, then this provides an alternative to having to modify the objects you are binding to in order to suit the requirements of the user interface.

Data Binding in Code

At times, you may find that you need to bind the property of a control in the code-behind. This is possible using the `GetBindingExpression` and `SetBinding` methods on the control.

To create a new binding and assign it to a control property, simply instantiate a `Binding` object (found in the `System.Windows.Data` namespace), assign the binding configuration to its properties as required, and apply the binding to the target control property using the control's `SetBinding` method. The `SetBinding` method requires you to pass it the `Binding` object, and the dependency property identifier associated with the property to bind to on the target control (remembering that you can only assign data bindings to dependency properties).

For example, let's bind the `Name` property of a `Product` entity to the `Text` property of a `TextBox` control named `NameTextBox`:

```

Binding binding = new Binding("Name");
binding.Mode = BindingMode.TwoWay;
NameTextBox.SetBinding(TextBox.TextProperty, binding);

```

This is a simply example of creating a binding in code, but of course, you can assign values to all the other binding properties that we have previously discussed, such as `StringFormat`, `TargetNullValue`, `FallbackValue`, and `Source`.

You can obtain the binding assigned to a control's property using the `GetBindingExpression` method on the control (which we discussed earlier).

```

BindingExpression expr = NameTextBox.GetBindingExpression(TextBox.TextProperty);

```

As its name suggests, this method does not actually return a `Binding` object but returns a `BindingExpression` object instead. The `BindingExpression` class has two properties: `ParentBinding` and `DataItem` (in addition to the `UpdateSource` method discussed earlier). The `ParentBinding` property

provides the **Binding** object assigned to the control property, and the **DataSource** property provides the object acting as the source for the binding.

Getting and Setting Attached Property Values in Code

Although it's a somewhat different topic to data binding, getting and setting the value of an attached property on a control is similar to getting and setting bindings. Use the **GetValue** method of a control (passing it the dependency property corresponding to the attached property) to get the given attached property's value for that control. For example, to get which grid row a text box control named **NameTextBox** is in (hence, get the value of the **Grid.Row** attached property on the text box):

```
int row = (int)NameTextBox.GetValue(Grid.RowProperty);
```

You can then assign a different value to that attached property if you wish, using the **SetValue** method of the control, and passing it the corresponding dependency property and the new value to assign to it:

```
NameTextBox.SetValue(Grid.RowProperty, row);
```

Additional Tips

Let's take a look at some additional miscellaneous data binding tips that you may find useful when binding to data.

Visual Studio 2010 Data Binding Expression Builder

As mentioned back in Chapter 3, "An Introduction to XAML," Visual Studio 2010 provides a data binding expression builder (from the Properties window), which can be of help when learning how to write data binding expressions. This data binding expression builder will lead you through the process of creating a data binding expression.

Select the target control in the designer, find the target property that you want to bind to in the Properties window, and click the Advanced Properties icon next to its name. Select the Apply Data Binding menu item from the popup menu (as shown in Figure 10-3).

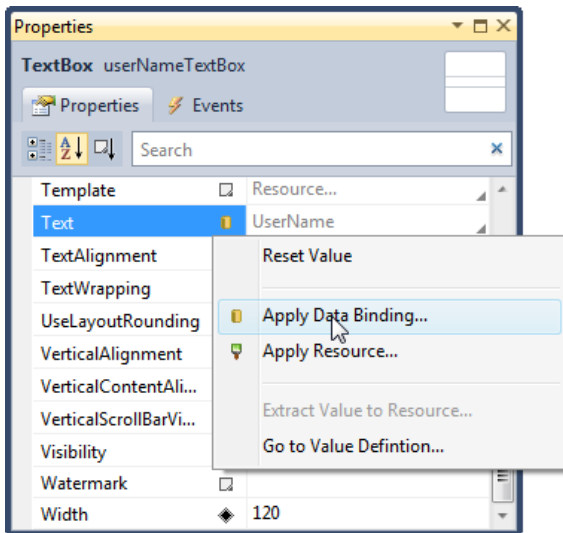


Figure 10-3. Opening the data binding expression builder

This will open the data binding expression builder as a pop-up window (as shown in Figure 10-4). Note that the builder has an accordion style layout, where you click each header to open the corresponding pane and select the options. The first step is to select a source for the binding.

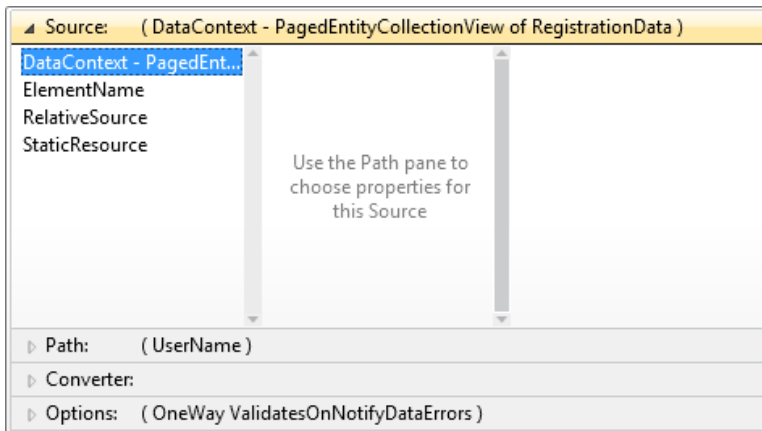


Figure 10-4. Selecting a data binding source

The next step is to select a path for the binding (as shown in Figure 10-5).

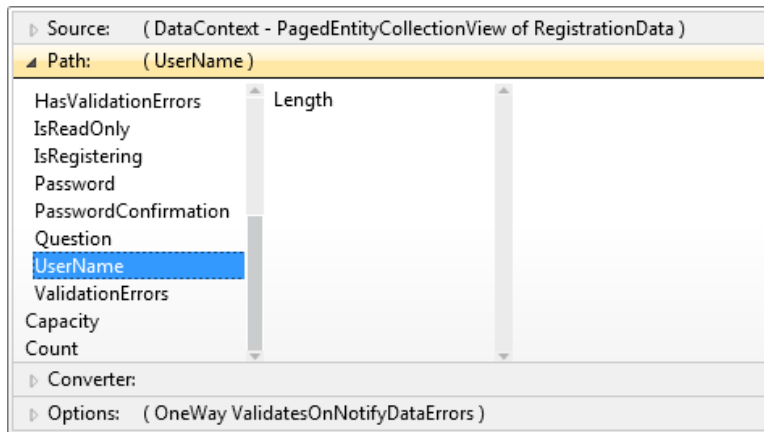


Figure 10-5. Selecting a path to bind to on the source

■ **Note** The properties of the binding source aren't displayed in this pane unless the builder actually knows what type of object is being bound to. As you can see from Figures 10-4 and 10-5, it knows that the binding source is a `PagedEntityCollectionView` that wraps `RegistrationData` objects. It knows this because the source of the data being inherited by the data context of the control being bound is the `Data` property of a `DomainDataSource` control, which uses the design-time data properties (discussed earlier in the chapter) to define the type of data that the `DomainDataSource` control will return. Alternatively, if you are binding to a class that is defined and instantiated in the XAML file as a resource, the builder will also be able to determine the properties available on the binding source to display here. If you are binding to another control in the view, again the builder will know what properties are available to bind to. However, in scenarios where it isn't able to determine what type of object the binding source is (which would be the case when you are assigning the data to bind to in the code-behind but haven't used the design-time data properties to specify the type of the data that will be bound to), then you won't be able to use the builder to select the source property to bind to unless you've provided design-time data definitions for the source data that will be bound to. The only other alternative is to manually assign the data binding expression to the target property in the XAML.

If you wish, you can select a value converter to use in the binding, as shown in Figure 10-6.

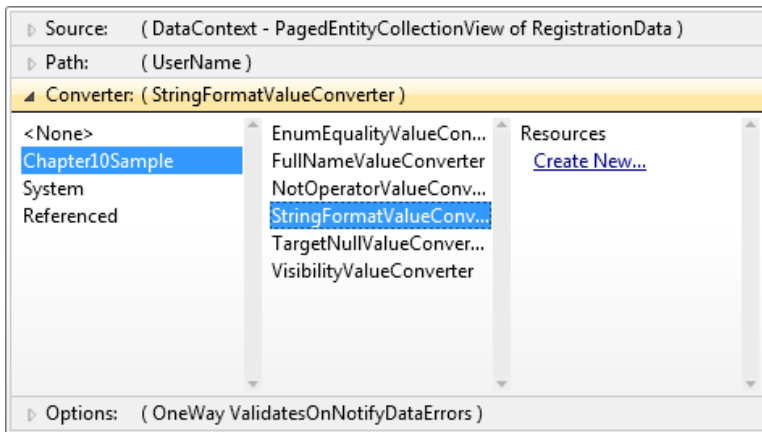


Figure 10-6. Selecting a value converter for the binding

■ **Note** Once the converter has been created as a resource (which you can do by clicking the Create New button in the Resources list), a parameter field will appear in the list where you can specify a parameter for the converter if you wish.

The final step is to select any additional options that should be applied to the binding, as shown in Figure 10-7.

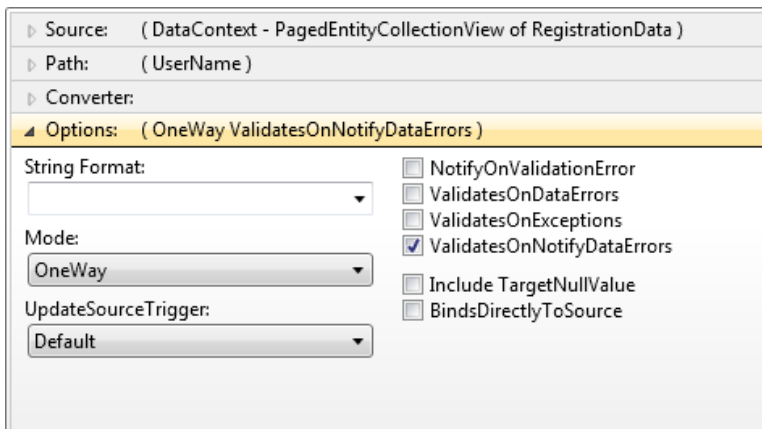


Figure 10-7. Selecting additional (miscellaneous) binding options

After selecting the options, simply clicking somewhere outside the data binding expression builder pop-up will close it and apply the generated binding string to the control property.

Debugging Data Binding Issues

It is very easy to make a mistake (such as misspelling a property name in the path of the binding) when creating a binding or break a working binding when you refactor a class that acts as a binding source (such as renaming a property). At other times, bindings simply don't work in the way you would expect.

These types of data binding errors can be particularly hard to track down at times (and it can be hard to even recognize that they exist), as when bindings fail, they fail silently without throwing an exception in the application. There are, however, some techniques you can use to help you identify and track down data binding errors, which we'll take a look at here.

The first method is to simply take a look at the Output window in Visual Studio (select View ~TRA Output from the main menu if you can't find it). Binding errors are logged to the Output window when the application is being debugged, with a corresponding error message, the binding source object and property, and the binding target control and property. For example, by setting the binding path to the `Nam` (a misspelling) property instead of `Name`, will result in the following error being logged to the Output window:

```
System.Windows.Data Error: BindingExpression path error: 'Nam' property not found on
'AdventureWorksLOB.Models.Product' 'AdventureWorksLOB.Models.Product' (HashCode=29083993).
BindingExpression: Path='Nam' DataItem='AdventureWorksLOB.Models.Product'
(HashCode=29083993); target element is 'System.Windows.Controls.TextBox'
(Name='NameTextBox'); target property is 'Text' (type 'System.String')..
```

This provides you with a fair amount of information about the error, which, in most cases, will help you track it down.

Unfortunately, in nondebugging scenarios, you cannot use this technique. An option is to assign a value to the `FallbackValue` property of the binding. Although it doesn't tell you *why* the binding failed, it can help prove that the binding *did* fail:

```
<TextBox Text="{Binding Name, Mode=TwoWay, FallbackValue=Binding failed!}" />
```

You could alternatively use the helper created by Karl Shifflett (creator of the XAML Power Toys, detailed back in Chapter 7, “Building Data Entry Forms”) which he has called Glimpse. One of its features is displaying bindings that have no source. You can get more details about Glimpse and download it at <http://karlshifflett.wordpress.com/2009/06/08/glimpse-for-silverlight-viewing-exceptions-and-binding-errors/>.

Another trick you might like to try is to add a value converter to your binding and put a breakpoint in its `Convert` and `ConvertBack` methods. This will enable you to hook into the binding process and perhaps help you identify the issue that you are having with that binding.

Even if your bindings are hooked up correctly, they may fail for many other reasons. Here are just a few troubleshooting tips you may wish to check:

- If the bound control property isn't being updated accordingly when the underlying source property value is updated, ensure that the binding mode isn't set to `OneTime`. Next, check that the property setter is raising the `PropertyChanged`

event (implemented by the `INotifyPropertyChanged` interface), and check that the property name being passed to that event actually matches that property's name.

- If the source property isn't being updated when its value is changed via the user interface (such as entering a different value into a text box), start by ensuring that you've set the binding mode to `TwoWay`. Also, check that the `UpdateSourceTrigger` property of the binding isn't set to `Explicit`, and if it is, ensure that the `UpdateSource` method of the binding is called in the code-behind.

Binding Using Property Element Syntax

You can actually specify bindings using property element syntax. The following example demonstrates binding to a property named `Name` using property element syntax:

```
<TextBox>
  <TextBox.Text>
    <Binding Path="Name" />
  </TextBox.Text>
</TextBox>
```

MultiBinding

At times, you want to bind a control property to multiple properties on the binding source object, taking each of those property values and merging them into a single value to be consumed by the target. This is known in WPF as a *MultiBinding*, but this feature is not available in Silverlight. Earlier in this chapter, we discussed combining the values of multiple properties together while data binding using a value converter, which is a way to get around this omission. That said, this method isn't that different to using a `MultiBinding` in WPF, as it requires the use of a value converter too. `MultiBinding` does have the advantage of being a slightly more generic solution (although you could make the value converter solution a little more generic by passing the names of the source properties to use in the converter parameter). Colin Eberhardt has created a means for implementing `MultiBindings` in Silverlight, which you can obtain at www.scottlogic.co.uk/blog/colin/2010/05/silverlight-multibinding-solution-for-silverlight-4/.

■ **Note** If you are using the Model-View-ViewModel (MVVM) design pattern (discussed in Chapter 12), then `MultiBinding` isn't particularly necessary, because you can simply add new properties (which expose the combined property values) to your `ViewModel` that the value converter requires.

Summary

In this chapter, you've seen a number of advanced tips and techniques that you can use when writing XAML and binding to data. Some of these you will use regularly, and others somewhat rarely, but either way, having them in your toolkit and at your disposal will greatly add to your ability to solve difficult problems in implementing your Silverlight business applications.

CHAPTER 11



Creating Custom Controls

When designing user interfaces in any technology, you ideally want to encapsulate the look and behavior of commonly used user interface elements into a reusable component that you can use elsewhere in your application or even in other applications. Creating custom controls also reduces the amount of code required in the code-behind, simplifies the XAML in your views, and helps maintain a consistency in your user interface.

We already have a wide range of controls available to use from the core Silverlight control set, the Silverlight Toolkit, and open source controls on CodePlex. Plus, there are many more in the form of third-party controls (from companies such as Telerik, ComponentOne, and Infragistics). Occasionally however, you will find that you have specific requirements that no existing control currently provides. It's at this point that you should consider creating your own control for that purpose.

There are two ways to do create a reusable user interface component in Silverlight, as a user control or as a custom control. User controls generally combine multiple existing controls into a single component, while you will usually start from scratch with custom controls and provide the look and behavior yourself. In this chapter, we will look at creating and using both of these types of components. Some implementation details (such as exposing properties and events) are much the same between the two types of components, but then, other details and capabilities differ greatly (particularly when it comes to styling and templating). However, let's start by simply extending an existing control to add additional behavior to it, before continuing on to focusing on how to create and use user controls and custom controls.

Adding Functionality to an Existing Control

If you simply want to add behavior to an existing control, then you can inherit from that control (rather than creating a whole new control from scratch). For example, let's say you want the text in a TextBox control to automatically be selected when the control gains the focus, enabling the user to easy overwrite the existing text (instead of the default behavior where the cursor is simply placed at the end of the existing text).

There are two options to achieve this behavior:

- The first option is to create an action (as described back in Chapter 10, "Advanced XAML and Data Binding"), which is triggered when the control gets the focus and selects the text as required (or alternatively you could create a behavior instead). You can then apply that to each control that should implement that behavior. However, this requires the designer/developer to remember to apply that behavior to each control and adds to the complexity of the XAML in your views.

- If the behavior is a common requirement for the control in your project, another option is to inherit from that control, adding the behavior to it, and using the inherited control in its place.

Using our example of a `TextBox` that automatically selects all the text when it receives the focus, we can create a new class called `CustomTextBox`, inherit from `TextBox`, and add the behavior:

```
public class CustomTextBox : TextBox
{
    protected override void OnGotFocus(RoutedEventArgs e)
    {
        base.OnGotFocus(e);
        this.SelectAll();
    }
}
```

By default, the new control (`CustomTextBox`) will continue to use the default control template that the control it inherits from uses (`TextBox`). So for our `CustomTextBox` control, it will continue to use the `TextBox`'s default control template. If you want to use a different default control template, simply assign the control's type to the `DefaultStyleKey` property and define a corresponding style resource in the `generic.xaml` file.

```
DefaultStyleKey = typeof(CustomTextBox);
```

■ **Note** There are a number of primitive controls that don't appear in the Toolbox, but you may wish to inherit from them. These primitive controls form the base of many other controls. For example, the `Selector` primitive control forms the base for the `ListBox` and the `ComboBox` controls. You will find these primitive controls in the `System.Windows.Controls.Primitives` namespace, with controls including `ButtonBase`, `Popup`, `RangeBase`, `RepeatButton`, `ScrollBar`, `Selector`, `Thumb`, and `ToggleButton`.

Creating User Controls

User controls enable you to build a user interface component that leverages one or more existing controls, combining them together to form a single unit. Creating a user control is as simple as adding a new Silverlight User Control item to your project and adding controls to it using the XAML designer (designing it as you would a view). In fact, as you discovered back in Chapter 4, "The Navigation Framework," the `MainPage` class itself is a user control. It then appears in the Toolbox, where you can drag and drop it into your views as required. You can also define properties and events in its code behind, which can be used by the views that consume the user control.

Creating a Simple User Control

Let's create a simple user control that includes a label and a text box together as a single component. Add a new item to your project using the Silverlight User Control item template, named `FormField.xaml`. Lay it out with a `Label` control and a `TextBox` control in a `Grid`, like so:

```
<Grid x:Name="LayoutRoot" Background="White">
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="120" />
        <ColumnDefinition Width="180" />
    </Grid.ColumnDefinitions>
    <sdk:Label Name="dataLabel" Padding="0,0,5,0"
        HorizontalAlignment="Right" HorizontalContentAlignment="Left"
        Target="{Binding ElementName=dataField}" />
    <TextBox Name="dataField" HorizontalAlignment="Left" VerticalAlignment="Center"
        Grid.Column="1" Width="170" />
</Grid>
```

■ **Note** You will need to declare the `sdk` namespace prefix (for the `http://schemas.microsoft.com/winfx/2006/xaml/presentation/sdk` namespace) in the root element of your XAML file.

This and setting the size of the user control (Width = 300, Height = 30) results in the designer output shown in Figure 11-1.



Figure 11-1. The `FormField` control in the XAML designer

If you compile your project now and open the `MainPage.xaml` file in the designer, you will find that your `FormField` user control appears in the Toolbox. You can now drag it from the Toolbox and onto the design surface of the `MainPage` view, just as you would any other control.

■ **Note** The MoXAML Power Toys described back in Chapter 10, “Advanced XAML and Data Binding,” provides a feature where you can select some XAML (from the XAML view in the designer) and extract it to a user control, which can help you refactor elements of your views into a separate reusable user control. If you are working in Expression Blend, you can select the elements in the designer, right-click, and select `Make Control` from the context menu.

Exposing Properties

For simple user controls that may simply display a static layout (such as some text) or consume and display some data via their `DataContext` property, you may not need to provide any sort of additional interactivity with the view that consumes this control (by exposing properties, methods, and events). This sort of scenario will only apply to the simplest of controls, so let's look at exposing some properties from our `FormField` user control.

When you dropped the `FormField` control onto the design surface of your view, there was no way to get/set the text in the label or text box, but we can expose some properties from the user control to enable this now.

There are two ways to expose a property from a user control:

- As a standard property
- As a dependency property

Both of these are implemented differently. In most cases, you should implement each property as a dependency property (so you can assign a binding or other markup expression to it), but let's look at the simpler task of implementing standard properties first.

Standard Properties

In Chapter 7, “Building Data Entry Forms,” we discussed the need to implement full properties with a backing store in your entity/object classes (i.e., not the automatically implemented properties), so that we could raise the `PropertyChanged` event when the property values were updated (as expected when implementing the `INotifyPropertyChanged` interface). As you will recall, implementing the `INotifyPropertyChanged` interface enables control property bindings to be aware that the bound property's value has changed and to update the property on the control accordingly. It's recommended that your user control implements the `INotifyPropertyChanged` interface, and any standard property you create on your user control raises the `PropertyChanged` event in its setter.

Let's expose a `Label` property (to get and set the text displayed in the `Label` control) and a `Value` property (to get and set the text in the `TextBox` control). Implement the `INotifyPropertyChanged` interface in the user control's code-behind (as detailed in Chapter 7), and add the following code:

```
private string label;
private string value;

public string Label
{
    get { return label; }
    set
    {
        label = value;
        OnPropertyChanged("Label");
    }
}

public string Value
{
    get { return value; }
    set
```



```

    {
        this.value = value;
        OnPropertyChanged("Value");
    }
}

public event PropertyChangedEventHandler PropertyChanged;

protected void OnPropertyChanged(string propertyName)
{
    if (PropertyChanged != null)
        PropertyChanged(this, new PropertyChangedEventArgs(propertyName));
}

```

If you compile your project and return to the MainPage.xaml file in the designer, you will find that these two properties appear in the XAML IntelliSense and in the Properties window, enabling you to get and set their values. Of course, they don't actually do anything as yet, because they actually need to be dependency properties to bind control properties to them. Therefore, let's take a look now at how to implement dependency properties.

Dependency Properties

When WPF was being created, the team decided that standard CLR properties didn't fully meet the requirements for the new presentation technology they were developing. The standard properties didn't automatically notify any associated bindings when their value was changed (you needed to implement the `INotifyPropertyChanged` interface to do so), and more importantly, they needed properties that could self-compute or resolve their value based on a number of external sources (provided by an expression or value precedence criteria, such as a data binding expression). To achieve these requirements, the WPF team came up with the concept of *dependency properties*, which have also been adapted to Silverlight (albeit in a limited capacity).

■ **Note** They are named *dependency* properties because the property's value *depends* on external sources, such as data binding or resources.

Why Do We Need Dependency Properties?

Dependency properties are possibly one of the most complex aspects of Silverlight to get your head around, but they are also a fundamental concept that you should understand, even if you aren't planning on creating your own custom controls.

Implementing standard properties on your controls may be acceptable for some properties, but as soon as you need to assign a markup expression to one, you will find that doing so throws an exception when you run your application. You may recall from Chapter 3, "An Introduction to XAML," that any property you assign a markup extension (such as a data binding expression) *must* be a dependency property. Markup expressions are evaluated at runtime, so the property needs to recognize that it has been assigned a markup expression and evaluate it. Assigning a binding markup extension (for example) to a standard property will result in the XAML engine attempting to assign the binding object to the

property, which will usually result in an exception as it tries (and fails) to convert a binding object to the property's type (such as a string, integer, or Boolean). This is one of the more common scenarios where dependency properties are required.

Registering a Dependency Property

To declare a dependency property on a class, the first thing you need to ensure is that the class inherits from the `DependencyObject` class. The features of dependency properties that were previously listed are actually provided by this class (storing and calculating values), and thus your control needs to inherit from it to enable them. As it is, the classes you will inherit from for your control (such as `UserControl`, `Control`, and `ContentControl`) have already inherited from the `DependencyObject` class (usually a few generations back), and you should not need to inherit from the `DependencyObject` class directly.

The next step is to register your dependency property with the dependency system, using the `DependencyProperty.Register` method, as demonstrated here:

```
public static readonly DependencyProperty ValueProperty =
    DependencyProperty.Register("Value", typeof(string), typeof(FormField), null);
```

As you can see, this looks nothing like a standard CLR property definition. This syntax seems (and is) rather complicated and somewhat scary when you first look at it, so let's break it down into its component parts so we can make some more sense of it. In this example, we are registering a dependency property named `Value` on the class. Let's look at the first part:

```
public static readonly DependencyProperty ValueProperty;
```

As you can see, we are declaring a field (known as a *dependency property identifier*) of type `DependencyProperty`, which we are naming `ValueProperty`. This field will be used to reference the property when getting and setting its value. The convention when declaring dependency properties is to suffix the dependency property identifier's name with "Property". So for a dependency property named `Value`, you should have a corresponding dependency property identifier named `ValueProperty`. Note that this field is marked `public`, `static`, and `readonly`, which is required for dependency property declarations.

Let's now take a look at what is being assigned to the `ValueProperty` dependency property identifier:

```
DependencyProperty.Register("Value", typeof(string), typeof(FormField), null);
```

The static `Register` method on the `DependencyProperty` class registers the dependency property with the dependency system, returning a `DependencyProperty` object that we then assign to `ValueProperty`. To this method we are passing (in order):

- *The name of the dependency property:* "Value"
- *The dependency property's type:* `typeof(string)`
- *The type of the class hosting the dependency property (i.e., the name of the user control):* `typeof(FormField)`
- *Property metadata, in the form of a `PropertyMetadata` object:* In our example, we are simply passing that parameter a value of `null` (as property metadata will be discussed further shortly).

Getting and Setting a Dependency Property Value

You may be wondering how this property actually works, since it's declared as a static readonly field. It *doesn't* actually store the value of the property; the `DependencyObject` object stores and calculates it instead (as previously indicated). When you register the dependency property with the dependency system, you pass it the method the type of class to host it (the third parameter, `typeof(FormField)`). The dependency system registers the dependency property with the dependency object, which maintains the value of each dependency property (in a `Dictionary`).

To actually get or set the value of the dependency property, you need to use the `GetValue` or `SetValue` method provided by the `DependencyObject` class, passing it the dependency property identifier to get or set the corresponding value for.

For example, to get the value of the `ValueProperty` dependency property, you would use the following code:

```
string value = GetValue(ValueProperty).ToString();
```

■ **Note** The `GetValue` method returns the value as an object, and you therefore need to cast it to the required type.

And to set the value of the `ValueProperty` dependency property, you would use this code (where `value` is a variable containing the value to assign to the dependency property):

```
SetValue(ValueProperty, value);
```

■ **Note** The `GetValue` and `SetValue` methods inherited from the `DependencyObject` class are public methods, and thus can be called from both within and outside of the class.

Creating a Standard CLR Property Wrapper

Whether you want to get or set the value of a dependency property from inside or outside the confines of the class, calling the `GetValue` and `SetValue` methods on the class is hardly the friendliest means of accessing the property. They're certainly not as friendly as working with standard CLR properties. That's why, when you see a dependency property declared on a class, you will generally see a corresponding (and somewhat friendlier) standard CLR property wrapping these methods and simplifying their access, like so:

```
public string Value
{
    get { return (string)GetValue(ValueProperty); }
    set { SetValue(ValueProperty, value); }
}
```

■ **Note** The XAML parser expects these CLR properties for dependency properties that you want to assign values to in XAML, even if it doesn't always use them (as will be explained shortly). You should use the same name for the standard CLR property as the name that you registered the dependency property with.

Therefore, to assign a value to the dependency property, you can use

```
valueField.Value = newValue;
```

instead of

```
valueField.SetValue(FormField.ValueProperty, newValue);
```

And to obtain the value of a dependency property, you can use

```
currentValue = valueField.Value;
```

instead of

```
currentValue = valueField.GetValue(FormField.ValueProperty).ToString();
```

■ **Note** There is no need to raise the `PropertyChanged` event from the `INotifyPropertyChanged` interface in your CLR property wrapper when the value of the dependency property is updated. As previously stated, one of the benefits of dependency properties is that they automatically support value change notifications, so there is no need to raise this event when the property is maintained as a dependency property.

There is a *very* important point to note when creating standard CLR property wrappers for your dependency properties. You should never add any logic to the getters or setters of these properties, as there is no guarantee that this logic will be executed. For example, when you assign a markup extension to a dependency property, the CLR property wrapper getter and setter are not called. Instead, the runtime interacts directly with the dependency property (via the `GetValue/SetValue` methods on the class) without the involvement of the standard CLR property wrapper. If you need to perform some logic when the value of the property is modified, you should provide a method to call when the property's value is modified in the dependency property's metadata. You can then add your logic to that method, as described in the next section.

Dependency Property Metadata

In our earlier example, we kept things simple when demonstrating how to register a dependency property and simply passed null to the `DependencyProperty.Register` method's parameter that you could use to configure the property's metadata. In Silverlight, the property metadata you can configure includes the following:

- A default value for the property
- A callback method that will be called when the value of the dependency property is modified

These are assigned to a `PropertyMetadata` object, which is then passed to the `DependencyProperty.Register` method. The constructor of the `PropertyMetadata` class has a number of overloads, enabling you to either or both these values. For the purpose of these examples, we'll just look at the overloads where each metadata item is passed alone to the constructor.

As a general rule, you shouldn't set the default value of a dependency property in the control's constructor. Instead, you can provide a default value for a dependency property by assigning it in the property's metadata. The code below demonstrates setting a default value of "default" to the property via the `PropertyMetadata`'s constructor (as a parameter):

```
public static readonly DependencyProperty ValueProperty =
    DependencyProperty.Register("MyProperty", typeof(string), typeof(FormField),
        new PropertyMetadata("default"));
```

■ **Note** When creating custom controls (discussed later in this chapter), another alternative means of setting the default value of the dependency property is to assign it a value in the style for the control in the `generic.xaml` file.

To be notified when the value of the dependency property has changed, pass a callback method to the `PropertyMetadata`'s constructor, like so:

```
public static readonly DependencyProperty ValueProperty =
    DependencyProperty.Register("Value", typeof(string), typeof(FormField),
        new PropertyMetadata(ValuePropertyChanged));
```

and define a corresponding method to be called:

```
private static void ValuePropertyChanged(DependencyObject d,
                                         DependencyPropertyChangedEventArgs e)
{
    // Insert value changed logic here
}
```

As you can see, two parameters are passed into this method:

- `d`: The instance of the object for whom the property's value has changed
- `e`: A `DependencyPropertyChangedEventArgs` object containing additional information about the value modification

The `DependencyPropertyChangedEventArgs` object has three properties, `NewValue`, `OldValue`, and `Property`. The `NewValue` and `OldValue` properties are fairly self-explanatory, and the `Property` property passes the dependency property identifier whose value has changed.

■ **Note** You may have noticed that this method is `static`. Therefore, rather than executing the value changed logic in this method (where you have to qualify any member, property, or method access with the object instance passed into the method, it may be worthwhile to call a nonstatic method on the object instance from this method that implements the logic instead.

If you want to obtain the default value of a dependency property, you can get it from the metadata using the `GetMetadata` method on the dependency property identifier and passing it the type of the control hosting it, for example:

```
PropertyMetadata metadata = ValueProperty.GetMetadata(typeof(FormField));
string defaultValue = metadata.DefaultValue.ToString();
```

Value Precedence

As discussed earlier, the core nature of a dependency property is that its value depends on a number of external sources, and these sources may each have a different value. Therefore, the dependency property needs to determine its value by placing an order of importance on each source, and using the one with the highest importance that also has a value. This order of importance is known as value precedence and is ordered as follows (from highest importance to lowest):

Animation: If an animation provides the dependency property with a value, it takes precedence over all sources.

Local value: A specific value has been assigned to a dependency property.

Template: A control template has been applied to the control and assigns the dependency property a value.

Style: A style has been applied to the control and assigns the dependency property a value.

Default value: If none of the preceding sources provide the dependency property with a value, its default value will prevail.

Simplifying Creating Dependency Properties

As you can see from this discussion about dependency properties, a reasonable amount of code is required to create a dependency property, and initially, attempting to create them can be quite daunting. There is help at hand however, in the form of property snippets. Visual Studio has the `propdp` snippet that you can use, although it was designed for use with WPF. The only change you need to make is to change `UIPropertyMetadata` to `PropertyMetadata`. If you're not happy doing so, you can modify the snippet yourself, or you can download and use the Silverlight-specific snippets created by Robby Ingebrechtsen at <http://blog.nerdplusart.com/archives/silverlight-code-snippets>. You can use the `sldp` snippet or the `sldpc` snippet from his snippet library in place of the `propdp` snippet. A number of additional Silverlight-specific snippets are included in the library that you may also find useful (such as `slvc` for creating a value converter and `slevent` for creating an event).

Obtaining the Local Value

As per the value precedence order, a local value assigned to a dependency property has a high level of importance in the determination of the dependency property's value. You can use the `ReadLocalValue` method to determine whether the dependency property has a local value assigned. As opposed to the `GetValue` method (which gets the value of the dependency property based on all its sources and the value precedence), the `ReadLocalValue` method returns a value only when a local value has been assigned. If no local value is assigned, it will return a value of `DependencyProperty.UnsetValue`:

```
object localValue = ReadLocalValue(ValueProperty);

if (localValue == DependencyProperty.UnsetValue)
{
    // No local value has been assigned to this dependency property
    // and it gets its value from other sources
}
else
{
    // This dependency property has a local value
}
```

Resetting a Dependency Property's Value

At times, you don't want to assign the dependency property a new value but simply want to clear the specific value that you have assigned to it (known as a *local value*). This enables it to return to resolving its value from its other sources (template, style, or default) as per the value precedence order. You can use the `ClearValue` method (provided by the `DependencyObject` class) to clear its value, passing it the dependency property identifier of the dependency property to clear, like so:

```
ClearValue(ValueProperty);
```

Dependency Properties in Summary

Dependency properties can be a difficult topic to grasp initially, so here are some of the most important points you need to remember about them:

- Dependency properties are just like other standard CLR properties that you declare on your classes, but they have additional features that extend their utility in the Silverlight environment (or more specifically cater to XAML's demands).
- Dependency properties are needed to support some of Silverlight's advanced features, such as data binding and animation.
- The most important features they support include evaluating markup extension expressions at runtime and automatically notifying bound controls when their value changes.
- Although dependency properties are declared as static properties, their values are maintained per object instance, just like standard CLR properties. When you call the `GetValue` or `SetValue` methods, you are calling them on an object instance; therefore, the dependency system knows which object instance the value relates to and can get or set the corresponding value accordingly.

- Dependency properties and their values are managed by the dependency system. Therefore, the developer does not need to create a member variable to maintain the property value for the class. Instead, the dependency system maintains these values in a hash table, and you ask it to get or set a property value using the `GetValue` or `SetValue` methods.
- Even though it isn't required, dependency properties are generally wrapped in standard CLR properties (which call the `GetValue` and `SetValue` methods) to more simply get or set their values in code.
- These standard CLR property wrappers should not have any logic in their getters and setters.

Consuming the Property Values in Your Control

Now that you've created your properties as either standard CLR properties or (preferably) dependency properties, it's time to actually wire them up so that the label displays the string assigned to the `Label` property, and the text box displays the value assigned to the `Value` property.

Throughout this book we've discussed the push and pull mechanisms for populating views with data. Using the push-based model, the code supporting the views needs to know about the view itself and its contents and have control of the view. By implementing a pull-based model (taking advantage of XAML's powerful binding features), you enable the code to take a back seat, simply acting as a provider to the view. The view can then take charge of how it's populated, providing a cleaner model where the code-behind simply serves the needs of the view, without actually needing to know anything about the view itself. This concept also applies to creating controls and is especially applicable to creating custom controls (as you'll discover later in this chapter, where there is a wider separation between the XAML and the code than for user controls).

■ **Note** Although your initial urge will probably be to push the data into the control from the code-behind, implementing the pull model will enable a cleaner separation of presentation from behavior and improve the testability of the control.

Using our `FormField` control as an example, we have exposed two properties publicly in the code-behind, `Label` and `Value` (preferably as dependency properties). This enables the view hosting the control to set the label of the control, and get or set its value (i.e., the text in the text box). We can then pull the data into the `Label` and `TextBox` controls in our user control by simply binding their `Content` and `Text` properties to the corresponding properties in the code-behind, using the techniques described in Chapter 10, "Advanced XAML and Data Binding" (see the section titled "Binding to a Property in the View's Code-Behind").

■ **Note** The way you bind to the properties in the code for a user control is quite different than the way you do so with custom controls, as you will find later in this chapter.

Unfortunately (and rather frustratingly), there are design-time issues when configuring the user control's `DataContext` property to bind to the user control object itself (such that we can bind the properties of the controls within the user control to the properties exposed by the user control object). In the previous chapter (Chapter 10, “Advanced XAML and Data Binding”), we discussed two techniques for doing this, and these will work at runtime but won't work at design time.

For example, say you have a property that you have created in the code-behind of the user control called `Value`, and you want to bind the `Text` property of a `TextBox` control (within the user control) to it. To be able to bind to the user control itself, you would set the `DataContext` property of the user control to `"{Binding RelativeSource={RelativeSource Self}}"` (using one of the techniques from Chapter 10). You can then set the `TextBox`'s `Text` property's binding to `"{Binding Path=Value, Mode=TwoWay}"` to bind to the `Value` property defined in the code-behind. This will work perfectly at runtime, with the text box displaying whatever value is assigned to the user control's `Value` property as expected.

However, at design time, nothing will be displayed in the text box in the user control. The `DataContext` property's binding returns null at design time, so the text box has no source to bind to (resulting in the text box's binding failing). In some circumstances, this may not be an issue, because design time support is not considered a priority.

If design time support is required however, there are some ways to get around the issue. One solution is to, rather than having the controls pull the data from the user control's properties, push the data into the controls from the code-behind instead, for example:

```
dataField.Text = "Value";
```

Alternatively, you can create the control properties bindings in the code-behind, assigning the current user control instance to their `Source` property, for example:

```
dataField.SetBinding(TextBox.TextProperty,
    new Binding("Value") { Source = this,
                           Mode = BindingMode.TwoWay });
```

Neither of these solutions is particularly elegant, but they do work. However, once again, they require the code-behind to have knowledge of the controls defined in the XAML and their bindings. However, if you want design-time support with user controls, you will unfortunately need to make that trade-off.

Type Converters

When you assign the value of a property in XAML, whatever value you give it, you are simply providing that value as a string. When you run your application, before it can assign the specified value to the property, the XAML parser needs to convert the value from the string that you provided to the property's type.

For a predefined subset of property types (string, integer, Boolean, and so on), this conversion is handled automatically by the XAML parser, and you will be immediately able to set the values of those properties in XAML when you consume the user control in a view. However, only a very limited number of types are actually supported natively by the XAML parser and assigning a value to a property whose type isn't supported will result in an exception similar to the following:

```
Failed to create a 'System.Decimal' from the text '42'.
```

If you need to expose a property from your control as a type not supported by the XAML parser (such as `Decimal` or `DateTime`), and you want to enable that value to be assigned in XAML, you will need

to implement a type converter. A *type converter* (as its name suggests) is used to provide the logic for converting one type to another.

To create a type converter, you will need to create a new class that inherits from `TypeConverter` (from the `System.ComponentModel` namespace) and override its methods (`CanConvertFrom`, `ConvertFrom`, `CanConvertTo`, `ConvertTo`), providing them with the appropriate logic.

■ **Note** In the previous chapter, we discussed value converters. Value converters can be used as part of the data binding process to completely change the source value to that to be used by the destination according to a set of rules determined by the source value. Type converters are quite similar in nature to value converters, but their role is (generally) only to translate the string values defined in the XAML file to the type of the control/object properties they are being assigned to, with this logic defined by the source and destination types rather than the values themselves.

Let's take a look at the implementation of a simple type converter for the `Decimal` type:

```
public class DecimalTypeConverter : TypeConverter
{
    public override bool CanConvertFrom(ITypeDescriptorContext context,
                                        Type sourceType)
    {
        return sourceType == typeof(string);
    }

    public override bool CanConvertTo(ITypeDescriptorContext context,
                                      Type destinationType)
    {
        return destinationType == typeof(string);
    }

    public override object ConvertFrom(ITypeDescriptorContext context,
                                       System.Globalization.CultureInfo culture,
                                       object value)
    {
        return Convert.ToDecimal(value);
    }

    public override object ConvertTo(ITypeDescriptorContext context,
                                     System.Globalization.CultureInfo culture,
                                     object value, Type destinationType)
    {
        return value.ToString();
    }
}
```

As you can see, very little logic is actually required, with each method only requiring a single line of code to implement. The `CanConvertFrom` and `CanConvertTo` methods determine whether this converter can actually convert between the type being represented (a decimal) and a given type (note that they don't actually check whether the value itself can be converted). This type will always be a string (when solving the problem of converting the value assigned in XAML), so we only need to support a source/destination type of string. Then in the `ConvertFrom` and `ConvertTo` methods, you can perform the actual conversion of the given value to and from a string.

■ **Note** You can find a `DateTimeTypeConverter` and a `TimeTypeConverter` in the source code for the Silverlight Toolkit. Rather than creating a type converter for each type, another option is provided by Anthony Jones, who has an implementation of a generic type converter that you may wish to use, which you can obtain at <http://geekswithblogs.net/codingbloke/archive/2010/05/14/silverlight-convertible-typeconverter.aspx>.

To use our type converter, you need to decorate the property in your user control with the `TypeConverter` attribute, passing it your converter as a parameter, for example:

```
[TypeConverter(typeof(DecimalTypeConverter))]  
public decimal TotalCost { get; set; }
```

■ **Note** XAML has built-in support for enumeration types, so you don't need to implement a type converter for each of those types.

Implementing the `ISupportInitialize` Interface

If you have logic that is executed when the value of a property is changed, often you won't want this logic to actually be implemented on your user control when initializing its property values until all the initial values have been assigned. This is particularly important when you have properties that are dependent on one another or whose values need to be set in order, or related properties whose values need to be applied as a batch.

This is where the `ISupportInitialize` interface (from the `System.ComponentModel` namespace) can help. It includes two methods, `BeginInit` and `EndInit`, which are called before and after (respectively) the initial values are assigned to the properties from the XAML. This enables you to set a flag (using a member variable) when the `BeginInit` method is called, and if that flag is set, you skip any logic in your property setters. You can then reset the flag in the `EndInit` method and execute any logic at that point as required.

■ **Note** Markup extensions assigned to dependency properties won't have been evaluated even after the `EndInit` method is called, so you will not be able to use these values in the logic that you want to execute immediately after this method call.

Exposing Methods

You expose a method from your user control in exactly the same way as you would expose a method publicly from a class, for example:

```
public void DoSomething()
{
    // Do something here
}
```

Exposing Events

Back in Chapter 3, “An Introduction to XAML,” you learned about the concepts of direct events and routed events in Silverlight. Direct events only raise the event on their source, while routed events bubble up the event along the object hierarchy until the root visual is reached or a control marks the event as handled.

■ **Note** There is no concept of tunneled events in Silverlight as there is in WPF.

Defining a direct event is done in exactly the same way as you would in any standard C# application. Simply choose an existing delegate to use (such as the standard or generic `EventHandler` delegate) or create your own, and define your event. The following example demonstrates creating an event named `ValueChanged`, which simply makes use of the standard `EventHandler` delegate:

```
public event EventHandler ValueChanged;
```

It's recommended, when implementing events in your custom controls, that you raise your events from a protected virtual method, enabling any controls that inherit from your control to suppress that event (and handle it differently). The convention is to name the method with the same name as your event but prefixed with “On,” for example:

```
protected virtual void OnValueChanged()
{
    if (ValueChanged != null)
        ValueChanged(this, new EventArgs());
}
```

When you want to raise the corresponding event, simply call this method.

■ **Note** Remember to always check for any listeners to your event before attempting to raise it (by checking if it's null, as demonstrated). Otherwise, an exception will be raised if you attempt to raise it but no-one is listening for it.

Unfortunately, you cannot create your own routed events in Silverlight. If you want to provide a routed event on your control, you may wish to take a look at the custom implementation provided in the project found here at <http://sl3routedevents.codeplex.com/>.

Determining If in Design-Time or Runtime Mode

One thing that you may not have realized is that when you are viewing the control in the Visual Studio or Expression Blend designer, the control (including its code-behind) is actually being executed from the latest compiled version of your project. This is why you need to compile your project after creating the control in order for it to appear in the Toolbox so you can use it. However, sometimes you need to implement a behavior in your control that should be executed at run-time, but not at design-time (such as communicating with a web service to retrieve data for display, for example).

To determine whether your control is being executed in a designer, you can simply check the value of the `IsInDesignTool` property of the `DesignerProperties` class (found in the `System.ComponentModel` namespace). When true, the control is executing in the Visual Studio or Expression Blend designer, and when false, it is executing at runtime within your application.

```
if (DesignerProperties.IsInDesignTool)
    MessageBox.Show("Is in design-time mode");
else
    MessageBox.Show("Is in run-time mode");
```

Constraining the User Control's Size

If you want your user control to have a fixed size, handle its `SizeChanged` event, and set its width and height as required in the handler, for example:

```
private void UserControl SizeChanged(object sender, SizeChangedEventArgs e)
{
    this.Width = 300;
    this.Height = 30;
}
```

Creating Custom Controls

An alternative way of creating a reusable control is by creating a custom control. Custom controls are structured quite differently to user controls, requiring a much stricter separation of their look (defined in the XAML), and their behavior (defined in code). This strict separation between the control's look and behavior enables the controls to be retemplated (have their look completely redefined) when they are used. To enable a custom control to be retemplated, you need to provide a formal contract between them using the parts and states model (discussed back in Chapter 9, "Styling Your Application").

When you should create a custom control instead of a user control is a difficult line to define. One of the advantages of a custom control is that they are templatable (unlike user controls). User controls are typically used simply to bring a number of controls together into a single reusable component, and thus are not intended to be templated in the same way as custom controls are. Being less structured than custom controls, user controls are easier to create.

Essentially, if you are simply encapsulating some common functionality that is more or less specific to a single project, combining that functionality into a user control is the most appropriate strategy. However, if you want to create a very generic control that could be used in any project then a custom control is your best choice.

■ **Note** As a general rule, third-party controls are always custom controls.

Let's take a look at creating a control that will display an animation while something is happening in the background, named `WaitIndicator` (which you can use as an alternative to the `BusyIndicator` control discussed back in Chapter 6, "Implementing Summary Lists"). A still from the animation is displayed in Figure 11-2.



Figure 11-2. The `WaitIndicator` control

■ **Note** The Silverlight Toolkit is a great resource for helping you understand how to write Silverlight custom controls. It can even provide a great starting point when creating your own controls.

Creating the Custom Control

Generally, when you create custom controls, you will want to maintain them in a separate project from your main project as a control library (making it easier to reuse the controls).

Start by creating a new Silverlight Class Library project named `MyControlLibrary` in your solution. Delete the default `Class1.cs` file so that the project is empty. Now, add a new item to your project using the Silverlight Templated Control item template (as opposed to the Silverlight User Control we used earlier to create a user control), and name it `WaitIndicator`.

This will create a `WaitIndicator.cs` file, and it will also automatically create a `Themes` folder, containing a resource dictionary file named `generic.xaml` (as you can see in Figure 10-3).

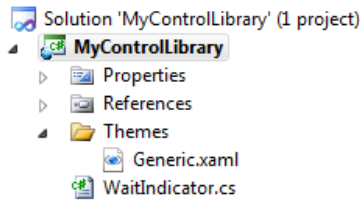


Figure 11-3. The solution structure for the class library

■ **Note** There is no XAML file corresponding to the custom control's code file like there is with user controls. Instead, the default control template (i.e., the XAML defining its default look) is defined in the `generic.xaml` file.

The Control Structure

Let's take a closer look at the files that were created for us. First, we have the `WaitIndicator.cs` file, which defines the control and its associated behavior:

```
public class WaitIndicator : Control
{
    public WaitIndicator()
    {
        this.DefaultStyleKey = typeof(WaitIndicator);
    }
}
```

Note that the `WaitIndicator` class inherits from `Control` (instead of `UserControl` as the user control did). In its constructor, it sets the `DefaultStyleKey` property in order to link the control's behavior (defined in this class) to its look. How this works will be discussed shortly.

Whereas user controls define their look in a `.xaml` file (which holds the presentation template for just that control), custom controls define their look in the `generic.xaml` resource dictionary file (in the project's `Themes` folder) as a control template. This `generic.xaml` file holds the control templates for all the custom controls in that assembly, each as a separate style resource.

The Silverlight Templated Control item template created this `generic.xaml` file for us, and defined an empty control template for our control:

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

    <Style TargetType="local:WaitIndicator">
        <Setter Property="Template">
            <Setter.Value>
                <ControlTemplate TargetType="local:WaitIndicator">
                    <Border Background="{TemplateBinding Background}">
```

```

        BorderBrush="{TemplateBinding BorderBrush}"
        BorderThickness="{TemplateBinding BorderThickness}">
    </Border>
</ControlTemplate>
</Setter.Value>
</Setter>
</Style>
</ResourceDictionary>

```

You can then define the look of your control in the control template defined in this style. By default, the control template simply incorporates a border around your control.

■ **Note** An important aspect to recognize and understand about the structure of control templates when designing custom controls is that the behavior (i.e., code) *is* the control itself. The control template defines the visual characteristics of the control, but it can be replaced and switched, with alternative templates applied interchangeably as required. The structure of custom controls promotes a strict separation between its look and its behavior, with a contract on the control itself defining the interface that a control template must adhere to. The result of this structure is that the behavior will have no knowledge of the contents of the control template (apart from what it expects to be available, as defined in its contract). The control templates, however, *will* know about the control, its contract, and the properties that it exposes. It's this structure that enables a custom control to be retemplated.

When you use a control in your project, it needs to determine what template to use. If one has not been explicitly specified when it is used, it will need to use its default template (if one has been defined). Custom controls know to always look in the `generic.xaml` file under the `Themes` folder for their default control template, and find the correct control template by matching the type applied to the `DefaultStyleKey` property (which should be set in the control's constructor, as previously demonstrated) to the type assigned to a style resource's `TargetType` property.

Defining the Control's Default Template

Back in Chapter 9, "Styling Your Application," we discussed how you can retemplate a custom control to give it a completely new look, and you learned about the parts and states model for controls in Silverlight with the `VisualStateManager`. You learned that the structure of a control template includes:

- States
- State transitions
- State groups
- Template parts

You use that same breakdown when creating your control template, except you won't have an existing template to work from as you did then (since we are now creating the template from scratch).

Therefore, we need to break up the visual aspects of our control in order to define each of these in its control template.

■ **Note** By structuring your control's default template properly and keeping a strict separation between its look and behavior, you will make it possible for your control to be completely retemplated when it is being used (as was demonstrated in Chapter 9, “Styling Your Application”).

Creating/Editing the Control Template in Expression Blend

The easiest way to create a control template is in Expression Blend. Visual Studio doesn't enable you to modify (or even view) control templates defined in the generic.xaml file in the designer, instead only displaying the message shown in Figure 11-4.



Figure 11-4. The message shown in the XAML designer when opening a ResourceDictionary

Therefore, unless you are happy writing all the XAML by hand you are better off working in Blend instead. We won't be discussing how to use Expression Blend to create control templates in any depth here, but let's take a brief look at the basics of getting started doing so.

When you open the generic.xaml file in Expression Blend, you will get a message stating that the file cannot be edited in design view. However, if you open the Resources tab and expand generic.xaml entry, you will see a list of the style resources in the file, and you can click the Edit Resource button next to one in order to view the control template defined in it. To be able to modify this control template, right-click the control in the design view, and select Edit Template ► Edit Current from the context menu. You will now be able to define the states, state groups, and state transitions in the States tab, and define the state animations and state transition animations in the Objects and Timeline tab.

Creating the Base State

The best place to start when creating a control template is to simply define the base state for our control, and work from there. This state is technically not a state at all but defines all the visual elements of the control that will be used by each of the states. You should add this to the control template that was created for you in the generic.xaml file, as demonstrated in the following XAML:

```
<ControlTemplate TargetType="local:WaitIndicator">
  <Border Background="{TemplateBinding Background}"
          BorderBrush="{TemplateBinding BorderBrush}"
          BorderThickness="{TemplateBinding BorderThickness}">
    <Canvas x:Name="LayoutRoot" Opacity="0">
      <Ellipse x:Name="Ellipse1" Fill="#1E777777">
```

```

        Canvas.Left="0" Canvas.Top="11" Height="8" Width="8"/>
<Ellipse x:Name="Ellipse2" Fill="#1E777777"
        Canvas.Left="3" Canvas.Top="3" Height="8" Width="8"/>
<Ellipse x:Name="Ellipse3" Fill="#1E777777"
        Canvas.Left="11" Canvas.Top="0" Height="8" Width="8"/>
<Ellipse x:Name="Ellipse4" Fill="#2E777777"
        Canvas.Left="19" Canvas.Top="3" Height="8" Width="8"/>
<Ellipse x:Name="Ellipse5" Fill="#3E777777"
        Canvas.Left="22" Canvas.Top="11" Height="8" Width="8"/>
<Ellipse x:Name="Ellipse6" Fill="#6D777777"
        Canvas.Left="19" Canvas.Top="19" Height="8" Width="8"/>
<Ellipse x:Name="Ellipse7" Fill="#9C777777"
        Canvas.Left="11" Canvas.Top="22" Height="8" Width="8"/>
<Ellipse x:Name="Ellipse8" Fill="#CC777777"
        Canvas.Left="3" Canvas.Top="19" Height="8" Width="8"/>
    </Canvas>
</Border>
</ControlTemplate>

```

This XAML defines eight ellipses arranged in a circle (as shown in Figure 11-2), each with a base color of #777777 (a gray color) but with varying degrees of alpha channel applied in the fill to lighten this color. It's the animation of this Fill property (specifically the alpha value) that will make the wait indicator “spin.”

Ideally (and as a standard practice), the XAML you define here should define the layout and look of the control in its initial (otherwise, Normal) state. Any elements/controls that should not be visible in this initial state should have their Visibility property set to Collapsed, or their Opacity property set to 0 (which makes it invisible, but it will still consume its given area). Note that the Canvas in this XAML has its Opacity property set to 0 accordingly, as the control is to be invisible by default.

Identifying States and Organizing Them Into State Groups

As you learned in Chapter 9, “Styling Your Application,” each state defines an alternative appearance of your control that visually represents the current condition that it's in, and in particular, the changes required to the base visual appearance of the control (defined as an animation) to indicate that state.

For the simplest controls, you may need to only support a single state (in which case this can be your base state, and you won't need to define any state groups or states in its default control template). However, in most cases your custom control will need to support multiple states, and at times be in more than one state simultaneously.

Therefore, the next step when creating your control is to identify what states your control needs to support, and whether the control can exist in multiple states simultaneously. Let's use the CheckBox control as an example. A check box has the following states:

- Normal
- MouseOver
- Pressed
- Disabled
- Checked

- Unchecked
- Indeterminate
- Focused
- Unfocused
- Valid
- Invalid

As you may be able to tell, some of these states are mutually exclusive (such as the checked and unchecked states), whereas others (such as the checked state and the focused state) could be simultaneously (but independently) be applied to the control.

If you find there needs to be support for the control to be in more than one state simultaneously, you will need to group the states into sets where all the states in each set are mutually exclusive and give each grouping a name. The CheckBox's states are grouped as follows:

CommonStates: Normal, MouseOver, Pressed, and Disabled

CheckStates: Checked, Unchecked, and Indeterminate

FocusStates: Focused and Unfocused

ValidationStates: Valid and Invalid

■ **Note** In the actual implementation of the CheckBox control, the Invalid state is actually split into InvalidFocused and InvalidUnfocused. However, we'll disregard this for the purposes of simplicity in this discussion.

The control will always exist in only one of these states from each group. However, it can only exist in one state from each of these groups simultaneously. Hence it can only be in the Focused *or* the Unfocused state (never both at the same time), but it can be simultaneously in the Normal, Checked, Focused, and Valid states (for example). This is why we have visual state groups.

Each state group can only contain mutually exclusive states. Before your control transitions to a new state, the visual state manager will first automatically transition the control away from any existing state that it is in within that same state group.

■ **Note** Even if you've determined that all the control's states are mutually exclusive, and the control does not need to exist in multiple states simultaneously, you still need to define one state group that will contain the various states you define. In other words, all states must exist within a state group.

Our WaitIndicator control is actually very simple when it comes to its states. Our control will simply display an animation that can be turned on and off, so with no user interaction or input involved, there

is no need for focus states, mouse over states, or validation states. This means that the control will need two primary states for use at runtime, which we'll call

- Inactive (invisible and not animating)
- Active (visible and animated)

We'll also include an additional state named `Static`, which we'll only use when the control is being displayed in the designer (so that the control is visible but not animated, to avoid distracting the designer).

Since each of these states is mutually exclusive, we only need a single state group, which we will call `CommonStates` (convention when defining custom control templates is to have the core control states in a state group with this name). We can define this state group and its related states in the control template like so:

```
<ControlTemplate TargetType="local:WaitIndicator">
  <Border Background="{TemplateBinding Background}"
          BorderBrush="{TemplateBinding BorderBrush}"
          BorderThickness="{TemplateBinding BorderThickness}">
    <VisualStateManager.VisualStateGroups>
      <VisualStateGroup x:Name="CommonStates">
        <VisualState x:Name="Inactive" />
        <VisualState x:Name="Static" />
        <VisualState x:Name="Active" />
      </VisualStateGroup>
    </VisualStateManager.VisualStateGroups>
    <!--XAML for the default/base state of this control goes here-->
    <!--as defined earlier. Removed for brevity purposes-->
  </Border>
</ControlTemplate>
```

Implementing the States

We've now defined these states, but they don't actually do anything yet. Each state now needs to define how the base state will be transformed to visually indicate that the custom control is in that state. This is transformation is handled via an animation that is defined as a part of each state.

As a general rule, you should define an initial state in each state group, which will be empty (i.e., no modifications to the base state). This will provide a starting point for the control that you can then return to. The initial state for our `WaitIndicator` control is the `Inactive` state, so the base state will be configured as per the requirements for the `Inactive` state (recall that we set the `Opacity` property of the `LayoutRoot` to 0 when defining the base state earlier, making the contents of the control invisible), and we'll therefore make no changes to the base state in the definition for the `Inactive` state.

The next state we need to implement is the `Static` state. This will make the ellipses in the wait indicator visible, although we won't animate them in this state. In the base state, we had set the `Opacity` property of the `LayoutRoot` to 0, so to make it visible, we need to animate this property to change its value from 0 to 1 over a duration of 0 seconds (resulting in it immediately becoming visible when we transition to that state). Therefore, we need to expand its state definition in the control template from

```
<VisualState x:Name="Static" />
```

to

```

<VisualState x:Name="Static">
    <Storyboard>
        <DoubleAnimation Duration="0" To="1"
                        Storyboard.TargetProperty="(UIElement.Opacity)"
                        Storyboard.TargetName="LayoutRoot" />
    </Storyboard>
</VisualState>

```

As you can see, we use an animation to change the value of the `Opacity` property of the element named `LayoutRoot` to 1 (via a combination of the `Storyboard.TargetName`, `Storyboard.TargetProperty`, and the `To` property of a `DoubleAnimation` object). The `Duration` property for this animation is set to 0 (seconds), so the value of the `LayoutRoot`'s `Opacity` property will change from 0 (its base value) to 1 (as per the animation) immediately. Extending this duration from 0 seconds will result in the `LayoutRoot` object fading into view (as the value of its `Opacity` property will change linearly from 0 to 1 over the given duration). You could also specify an easing function such that the value changes nonlinearly if you wish.

■ **Note** All animations are defined within a `Storyboard` (as you can see from the XAML). Creating these animations (and the corresponding XAML) is a relatively simple and quick process using Expression Blend.

The final state that we need to implement for our `WaitIndicator` control is the `Active` state. Like the `Static` state, we need to make the ellipses visible, but we also need to implement a repeating in-state animation.

While this in-state animation is running, it decrements the alpha value of the ellipse's `Fill` property every 0.15 seconds and repeating itself every 1.2 seconds. Each ellipse is 0.15 seconds out of phase, which gives the visual illusion of rotation. There are nine values for each ellipse's alpha value, with the first and the last alpha values being the same.

The XAML defining this animation is quite lengthy, so it won't be included here in its entirety, but here is a snippet of the animation for two of the ellipses (with its color being animated). Note how the `RepeatBehavior` property of the `ColorAnimationUsingKeyFrames` object is set to `Forever`, meaning that once complete it will restart continually until the control transitions away from the `Active` state:

```

<ColorAnimationUsingKeyFrames
    Storyboard.TargetName="Ellipse1"
    Storyboard.TargetProperty="(Fill).(SolidBrush.Color)"
    BeginTime="0" RepeatBehavior="Forever">

    <LinearColorKeyFrame Value="#CC777777" KeyTime="00:00:00" />
    <LinearColorKeyFrame Value="#9C777777" KeyTime="00:00:00.15" />
    <LinearColorKeyFrame Value="#6D777777" KeyTime="00:00:00.3" />
    <LinearColorKeyFrame Value="#3E777777" KeyTime="00:00:00.45" />
    <LinearColorKeyFrame Value="#2E777777" KeyTime="00:00:00.60" />
    <LinearColorKeyFrame Value="#1E777777" KeyTime="00:00:00.75" />
    <LinearColorKeyFrame Value="#1E777777" KeyTime="00:00:00.90" />
    <LinearColorKeyFrame Value="#1E777777" KeyTime="00:00:01.05" />
    <LinearColorKeyFrame Value="#CC777777" KeyTime="00:00:01.20" />
</ColorAnimationUsingKeyFrames>

```

```

<ColorAnimationUsingKeyFrames
    Storyboard.TargetName="Ellipse2"
    Storyboard.TargetProperty="(Fill).(SolidBrush.Color)"
    BeginTime="0" RepeatBehavior="Forever">

    <LinearColorKeyFrame Value="#1E777777" KeyTime="00:00:00" />
    <LinearColorKeyFrame Value="#CC777777" KeyTime="00:00:00.15" />
    <LinearColorKeyFrame Value="#9C777777" KeyTime="00:00:00.3" />
    <LinearColorKeyFrame Value="#6D777777" KeyTime="00:00:00.45" />
    <LinearColorKeyFrame Value="#3E777777" KeyTime="00:00:00.60" />
    <LinearColorKeyFrame Value="#2E777777" KeyTime="00:00:00.75" />
    <LinearColorKeyFrame Value="#1E777777" KeyTime="00:00:00.90" />
    <LinearColorKeyFrame Value="#1E777777" KeyTime="00:00:01.05" />
    <LinearColorKeyFrame Value="#1E777777" KeyTime="00:00:01.20" />
</ColorAnimationUsingKeyFrames>

```

You can download the full code for this control from the book's web site.

■ **Note** When you run the control, it won't actually be in one of the states you have defined. Instead, it will be in its base state. Ideally, you want it to be in one of your defined states, so immediately after applying the control template to the control (in the `OnApplyTemplate` method, described shortly), you should tell the `VisualStateManager` to go to the initial state within each state group.

Adding State Transition Animations

When you transition from one state to another, you may wish to ease the visual impact of jumping between the states by using a transition animation. Using the `VisualTransition` object, you can specify an animation for how the control should transition between two given states, from any state to a given state or from a given state to any other state.

There are two ways that you can implement a transition. The first is to let the `VisualStateManager` determine how it should transition between the two states. You simply specify the duration for the transition and the states you want it to go to and from, and it will work out how to transition between those states smoothly over the given duration (such as changing colors, opacity, and control positions). In the following example, a `VisualTransition` is defined, specifying that the transition between the `Inactive` and `Active` states should be smoothly animated between over the duration of two seconds.

```

<VisualStateGroup x:Name="CommonStates">
    <VisualStateGroup.Transitions>
        <VisualTransition GeneratedDuration="0:0:2" From="Inactive" To="Active" />
    </VisualStateGroup.Transitions>
    <!--Visual State definitions removed for brevity-->
</VisualStateGroup>

```

■ **Note** You can omit assigning a state to the `From` property a value if you want the transition to apply whenever to are transitioning to the state assigned to the `To` property (regardless of what state the control was previously in). Alternatively, you can omit assigning a state to the `To` property a value if you want the transition to apply whenever to are transitioning from the state assigned to the `From` property (regardless of what state the control is changing to).

Alternatively, we can explicitly define our own animation by adding a `Storyboard` to the `VisualTransition` with the animation. For example, this transition defines an animation that will fade in the `LayoutRoot` by changing the value of its `Opacity` property from 0 to 1 over a period of two seconds when transitioning from the `Inactive` to the `Active` state:

```
<VisualTransition GeneratedDuration="0:0:2" From="Inactive" To="Active">
  <Storyboard>
    <DoubleAnimation From="0" To="1"
                      Storyboard.TargetProperty="(UIElement.Opacity)"
                      Storyboard.TargetName="LayoutRoot" />
  </Storyboard>
</VisualTransition>
```

A state transition may seem like the best place to define the changes that should be made to the base state to define what the control looks like when it transitions to a given state, but this is not the case. For example, when our `WaitIndicator` control transitions to the `Static` or `Active` states, the `LayoutRoot` element needs to become visible (i.e., the value of its `Opacity` property needs to be set to 1). However, a transition is *not* the place to define the animation to do this, for a number of reasons. Transition animations can be skipped when moving from one state to another, so there's no guarantee that the animation will be executed. More importantly however is the fact that any changes made to the control's element's properties are only applied for the duration of the transition. If for example you define an animation as per the previous example where we make the `LayoutRoot` element visible, as soon as the transition animation is complete the control will enter the destination state, and the `VisualStateManager` will apply that state's changes to the base state - not to the changes that were made during the transition. Therefore, unless you set the `LayoutRoot` element to be visible within the state itself, it will return to being invisible as soon as it enters that state. Therefore, the rule is that all changes to the base state required by a state should be defined in the state itself.

Our `WaitIndicator` control has no need to transition between states, so we won't define any state transitions for it.

Defining Template Parts

You may recall from Chapter 9, "Styling Your Application," that template parts are used to define elements within your control template that you need to reference and interact with from code. Because of the strict separation between the files defining the look and the behavior of the control, the code (i.e., behavior) knows nothing about the control template. Each control that's given a name in the XAML for a user control has a corresponding member variable created in the code-behind (or a code-generated partial class extending the code-behind to be more exact) that the code-behind can use to reference and interact with those controls. However, this is not possible, since there is no coupling between the look

and the behavior of custom controls. Instead, a custom control needs to define a contract specifying what elements it needs to reference from the code, and hence expects to find in the control template.

Determining what template parts are required in the control template is really dependent on the needs of the code for the custom control, and thus don't need to be identified until you start defining the control's behavior. Once it identifies the template parts that it requires in its contract, however, you will need to conform to it by ensuring that there are corresponding controls defined in the control template and given the expected names.

When a template part is defined in the control's contract, it essentially states that "an element of this type with this name must be present in the control template." We'll look at defining this contract shortly, but for the time being, we simply need to ensure that we conform to that contract (if it exists as yet) and make the required elements available accordingly.

For example, let's say the contract defines a template part, expecting an element of type `Ellipse` with a name of `Ellipse1` (how it does so will be discussed shortly). Therefore, we simply need to ensure that we have an `Ellipse` with a name of `Ellipse1` in our base state:

```
<Ellipse x:Name="Ellipse1" Fill="#1E777777" Canvas.Left="0" Canvas.Top="11"
        Height="8" Width="8"/>
```

In our `WaitIndicator` control, as we do not actually need to interact with any of the elements in the control template from the code, we therefore have no need to define template parts.

Binding to Properties in the Code

Earlier in this chapter, we discussed consuming properties defined on a user control in its XAML by binding to them. Binding to the control's properties in the control template for a custom control, however, requires you to bind to them using a different method. Rather than using the standard bindings, you will need to create a one-way binding by using the `TemplateBinding` markup extension, or a two-way binding by using the `RelativeSource` markup extension. Let's look at both of these methods now.

One-Way Binding with the `TemplateBinding` Markup Extension

You can create a one-way binding in your control template that binds to the control object using the `TemplateBinding` markup extension. This binding will automatically find the templated control and bind to the specified property on it. For example, if your custom control defines a property named `HeaderText`, you can bind the `Text` property of a `TextBlock` control to it like so:

```
<TextBlock Text="{TemplateBinding HeaderText}" />
```

■ **Note** This is where the pull-based model discussed earlier in relation to user controls comes in again, enabling the code to take a back seat, simply acting as a provider to the view. Although as mentioned earlier, the code/behavior has no knowledge of the control template (apart from what it defines in its contract), you should note that the control template *does* know about the behavior—hence this scenario is ideal for implementing the pull-based model. To implement a push-based model, you would have to define the `TextBlock` control as a template part so that you could refer to it in the control's code; therefore, the pull-based model is a much better way.

Two-Way Binding Using the RelativeSource Markup Extension

The `TemplateBinding` markup extension is a `OneWay` binding only, and it has no `Mode` property to alter this like other binding types. Therefore, if you need the control that you are binding to in order to update the property that it is bound to (which requires a `TwoWay` binding), you will not be able to use the `TemplateBinding` markup extension for this purpose.

For example, say the `TextBlock` control from the previous example was actually a `TextBox` (where the user could modify the bound value). The `TemplateBinding` markup extension would be of no use here, as it would not enable the bound value to be updated according to the user's input.

To enable the bound property to be updated, you can use a combination of the `Binding` markup extension and the `RelativeSource` markup extension (as detailed in Chapter 10, "Advanced XAML and Data Binding") to bind to the property instead. The `Binding` markup extension will enable you to set up a `TwoWay` binding, and the `RelativeSource` markup extension will enable you to bind to the templated parent.

This binding is equivalent to the `TemplateBinding` example we used earlier, except this enables the binding to be `TwoWay`:

```
<TextBox Text="{Binding HeaderText, Mode=TwoWay,
                    RelativeSource={RelativeSource TemplatedParent}}" />
```

Splitting Your Generic.xaml File Into Smaller Pieces

When you start adding additional controls to your control library, you will start to find that the `generic.xaml` file starts to become unwieldy and hard to navigate. Each control template can become quite large, compounding the problem even further. You may wish to consider defining your control templates in separate resource dictionary files (one control template per file), and merge those into the `generic.xaml` file using the techniques described back in Chapter 3, "An Introduction to XAML." Note, however, that you can't use a relative path when setting the `Source` property of the `ResourceDictionary` entry that points to the XAML resource file to merge in the `generic.xaml` file. Instead, you must use the full path to the XAML resource file, including the assembly name. For example, to merge the contents of the `WaitIndicator.xaml` file that's under the `Themes` folder of the `MyControlLibrary` assembly, you would use the following XAML:

```
<ResourceDictionary.MergedDictionaries>
  <ResourceDictionary
    Source="/MyControlLibrary;component/Themes/WaitIndicator.xaml" />
</ResourceDictionary.MergedDictionaries>
```

This will make your `generic.xaml` file much more manageable and make it easier to modify the control templates for your custom controls when necessary.

Defining the Control's Behavior

In the control's behavior, you will define and expose properties (generally as dependency properties), methods, and events, just as you did with the user control. Where it will differ, however, is when you actually need to interact with elements defined in the control's template. Because of the strict separation between the control's look and its behavior, and because the control knows nothing about the template that has been applied to it, this presents a problem.

This is where the control needs to identify the parts and states that it needs to reference and interact with in the control template, in the form of a contract. When the control is used, the Silverlight runtime will apply the corresponding control template to it (which will either be explicitly specified when the control is used or, otherwise, the default control template), and notify the control when it has done so (by calling the `OnApplyTemplate` method on the control). It's up to the control in this method to get a reference to all the parts it requires (as defined by the contract) and store these references in member variables, which it can use to refer to them when required.

Let's take a deeper look at implementing the contract and behavior for a custom control.

Defining the Contract

If you want to interact with anything in the control template from the control's code, you should define a contract (by decorating the control's class with attributes), specifying the parts and states that the control expects to exist in the control template.

■ **Note** It's not essential to define this contract on the control, but it is recommended as it formalizes the requirements of the control from its template and provides benefits when retemplating the control in Expression Blend.

To define a state, you decorate the class with the `TemplateVisualState` attribute, setting its `Name` and `GroupName` properties (using named parameters) according to the corresponding state that it expects to be defined in the control template. For example, this attribute defines that the control expects a visual state named `Inactive` to be exist in the control template, under the `CommonStates` state group:

```
[TemplateVisualState(Name = "Inactive", GroupName = "CommonStates")]
```

When you need to refer to a named element within the control template, you should define a template part in the control's contract denoting that requirement, using the `TemplatePart` attribute. Using named parameters, you specify the name that you expect an element in the control template to have and the type of control it should be. For example, this attribute defines that the control expects a template part named `Ellipse1` to exist in the control template, of type `Ellipse`:

```
[TemplatePart(Name = "Ellipse1", Type = typeof(Ellipse))]
```

Ideally, you should define as few template parts as possible, because each template part places additional restrictions and constraints on the freedom of the control template designer and adds coupling points between the control and its template. Therefore, only define a template part if you really need access to it from the code, and see if you can expose a property that the element can bind to in order to implement the requirement instead. For example, instead of specifying a `TextBlock` as a template part so that you can set its `Text` property from the control's code, expose the text to be display via a property (with a scope of `internal`) on the control that the `Text` property of the `TextBlock` can bind to.

You may have noticed that when defining the visual states and the template parts on the control, these were defined as strings. *Magic string* values are never a good idea (especially when they need to be used in multiple locations within the code), so it's generally better to define them as constants on the

class. The name of each state, state group, and template part should be defined as a constant within the control's class, and you can then use them instead of the magic strings. For example, `StateInactive`, `StateGroupCommon`, and `PartEllipse1` in the following example are all string constants that are defined in the control's class and used in place of the magic strings from the previous examples:

```
[TemplateVisualState(Name = StateInactive, GroupName = StateGroupCommon)]  
[TemplatePart(Name = PartEllipse1, Type = typeof(Ellipse))]
```

Connecting the Code and the Default Control Template

When a control is initialized, the Silverlight runtime will automatically determine what template it should use (either its default template from the `generic.xaml` file or the template provided by the consuming view) and apply it to the control. Once it's done that, it will call the `OnApplyTemplate` method in the control's code. This is defined in the base `Control` class, and you will need to override it to be notified that the template has been applied.

```
public override void OnApplyTemplate()  
{  
    base.OnApplyTemplate();  
}
```

Once the template has been applied, you should get a reference to all the elements required by the control (as defined as template parts in the contract) from that control template and store these references in member variables that the control can use when it needs to interact with them. To get a reference to these elements, use the `GetTemplateChild` method (which is defined in the base `Control` class). Pass the method the name of the element that you want to get a reference to from the control template, and it will return the instance of that element (or null if it's not found).

```
Ellipse1 = GetTemplateChild("Ellipse1") as Ellipse;
```

■ **Note** The variable `Ellipse1` is a member variable that we've previously defined in the control's class of type `Ellipse`.

You should get a reference to each element in the control template that's defined as a template part like this and assign them to member variables in the `OnApplyTemplate` method of the control's class. You can do this for each template part explicitly, or you could use reflection to do it automatically for you. Assuming you have defined all your required template parts in the control's contract and have a corresponding member variable of the correct type and with a name matching that of the template part, you could use the following code to automatically get a reference to each element in the control template that is defined as a template part and assign them to a corresponding member variable in the control's class with the same name:

```
public override void OnApplyTemplate()  
{  
    base.OnApplyTemplate();
```

```
// Get all the attributes on this class that are of type TemplatePartAttribute
object[] templateParts =
    this.GetType().GetCustomAttributes(typeof(TemplatePartAttribute), true);

// Loop through each of these attributes, get the member variable with the same
// name as the template part, get the template part, and assign it to the
// member variable
foreach (TemplatePartAttribute attribute in templateParts)
{
    FieldInfo field = this.GetType().GetField(attribute.Name,
        BindingFlags.Instance | BindingFlags.NonPublic | BindingFlags.Public);

    field.SetValue(this, GetTemplateChild(attribute.Name));
}
}
```

Once you've done this, you can interact with each of the elements via the corresponding member variable that you've assigned its reference to throughout the control's code, for example:

```
Ellipse1.Fill = new SolidColorBrush(Colors.LightGray);
```

If a template part defined in the contract doesn't actually exist in the control template, the `GetTemplateChild` method will return null. You will need to decide how you want to handle this scenario. You can choose to continue without this part or throw an exception. If you decide to continue without the template part, you will need to check if it's null each time before you try to use it in the code.

■ **Note** Be sure not to try interacting with any of the elements defined as template parts before the control template has been applied and references to the elements have been obtained. For example, let's say you want to set a property on an element defined as a template part when the value of a property on the control itself is changed. If the control's property is assigned a value in the XAML when it's being used, that property will actually be assigned the value *before* the `OnApplyTemplate` method is called. Therefore, you won't have a reference to the element as yet, which will cause problems. This is one of the reasons why you should use template parts as sparsely as possible, and look at binding the properties of the corresponding element to properties exposed by the control instead (where possible).

Handling Events for Elements Defined in the Control Template (in the Code)

If you need to handle events raised by elements in your control's template, you will need to first ensure that those elements are defined as template parts and that you get a reference to them in the control's `OnApplyTemplate` method. Once you've gained a reference to the element in this method, add the required event handlers that you need:

```
Ellipse1.MouseEnter += new MouseEventHandler(Ellipse1_MouseEnter);
```

You can then respond accordingly to the events in the event handlers that you've defined (which may involve accordingly raising an event on the control itself to notifying the consuming view of that event).

Transitioning Between States

To transition from one state to another in your control, you can use the `GoToState` method of the `VisualStateManager` class, passing it the control to change the state for, the name of the state to transition to, and whether or not to display any transition animation defined between those states. The `VisualStateManager` will then handle the transition to that state accordingly. The following example demonstrates transitioning to the state named `Active` on the current control, with transition animations turned on:

```
VisualStateManager.GoToState(this, "Active", true);
```

■ **Note** If the state that you are attempting to transition to does not exist in the control template, this method will fail silently. It does, however, return a Boolean value that specifies whether or not it failed.

Implementing the `WaitIndicator`'s Behavior

The behavior for our `WaitIndicator` control is actually very simple. We have no need to define template parts in the control's contract, although we do have three states that we need to define:

```
[TemplateVisualState(Name = StateInactive, GroupName = StateGroupCommon)]
[TemplateVisualState(Name = StateActive, GroupName = StateGroupCommon)]
[TemplateVisualState(Name = StateStatic, GroupName = StateGroupCommon)]
```

So that we didn't need to use magic strings in the contract's attributes, we defined the state names and state group name as constants:

```
private const string StateActive = "Active";
private const string StateInactive = "Inactive";
private const string StateStatic = "Static";
private const string StateGroupCommon = "CommonStates";
```

We then override the `OnApplyTemplate` method. We have no template parts that we need to obtain a reference to, so this is quite minimal in its implementation. We do, however, get it to transition from its base state at this point to the state that it should currently be in (by calling the `SetVisualState` method, that we are about to define).

```
public override void OnApplyTemplate()
{
    base.OnApplyTemplate();
    SetVisualState();
}
```

We'll now create a method named `SetVisualState` that defines the logic determining what state the control should be in and goes to that state accordingly:

```
private void SetVisualState()
{
    if (DesignerProperties.IsInDesignTool)
        VisualStateManager.GoToState(this, StateStatic, true);
    else
        VisualStateManager.GoToState(this, IsBusy ? StateActive : StateInactive,
                                      true);
}
```

Finally, we define a dependency property that will be used to control what state the control should be in (busy [animated] or not busy [invisible]). This is a simple Boolean property, and when its value is changed, it will call the `SetVisualState` method that will update its current state accordingly:

```
public static readonly DependencyProperty IsBusyProperty =
    DependencyProperty.Register("IsBusy", typeof(bool), typeof(WaitIndicator),
        new PropertyMetadata(false, IsBusyPropertyChanged));

public bool IsBusy
{
    get { return (bool)GetValue(IsBusyProperty); }
    set { SetValue(IsBusyProperty, value); }
}

private static void IsBusyPropertyChanged(DependencyObject d,
                                          DependencyPropertyChangedEventArgs e)
{
    ((WaitIndicator)d).SetVisualState();
}
```

Testing the Control

Now that the control is complete, you can test it. Start by compiling the `MyControlLibrary` project, as the control will actually be executing when displayed in the designer.

Assuming the `MyControlLibrary` project and the Silverlight test project are in the same solution, you will find that when you open a view in your Silverlight test project that the control will automatically appear in your Toolbox (as shown in Figure 11-5).

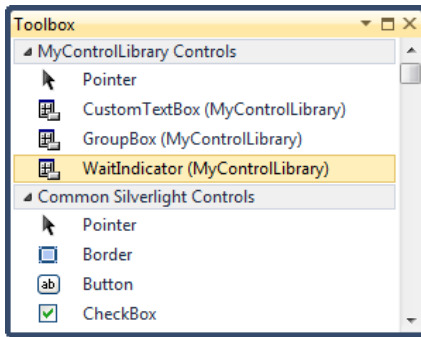


Figure 11-5. The *WaitIndicator* control in the Toolbox

■ **Note** If the *MyControlLibrary* project and the Silverlight test projects are not in the same solution, right-click the Toolbox, select *Choose Items* from the context menu, navigate to the compiled *MyControlLibrary* assembly, and add the control(s) from that assembly to the Toolbox.

Now, you can simply drag the control onto the design surface. Using the Properties window, set its *IsBusy* property to *True*. When you run your project, the control will be visible and animate.

Content Controls

Sometimes, you want to create a control that can contain some custom content. The *Button* control is a good example, enabling you to customize what is displayed within the button (enabling you to add an image, etc.). The *Button* control can only contain a single control as its content but other content controls such as the *Grid* control can contain multiple controls as its content. Let's take a look at how you can create a custom content control that can contain either a single control as its content or multiple controls.

Containing a Single Control as Content

The *Button* control is a *content control*, and you can make your own control a content control by inheriting from *ContentControl* (instead of *Control*). This will expose a *Content* property on your control, which another control can be assigned to when the control is being consumed in a view (using content element syntax, as described in Chapter 3, “An Introduction to XAML”).

To display a control assigned to the control's *Content* property, you need to include a *ContentPresenter* in your control template, specifying where that control should be displayed within the bigger picture of your control template. Think of a *ContentPresenter* as being a placeholder for the content of the control (provided by the consumer). Let's create a *GroupBox* control to demonstrate this concept. This control will be a content control that has a header and display some arbitrary content below it (as provided by the consuming view). Figure 11-6 demonstrates the final result.

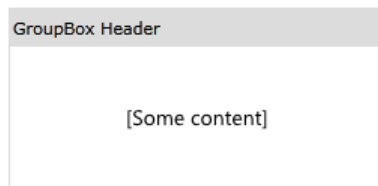


Figure 11-6. The GroupBox control

As you can see, we only want the content to be displayed below the header. Therefore, we position the ContentPresenter in our control template in the location where the content should be displayed. In the control template's XAML, which follows, you can see that we are positioning the ContentPresenter in the second row of the Grid (with the header taking the first row):

```
<Style TargetType="local:GroupBox">
  <Setter Property="BorderThickness" Value="1" />
  <Setter Property="BorderBrush" Value="Gainsboro" />
  <Setter Property="Template">
    <Setter.Value>
      <ControlTemplate TargetType="local:GroupBox">
        <Border BorderBrush="{TemplateBinding BorderBrush}"
          BorderThickness="{TemplateBinding BorderThickness}">
          <Grid Background="{TemplateBinding Background}">
            <Grid.RowDefinitions>
              <RowDefinition Height="25" />
              <RowDefinition />
            </Grid.RowDefinitions>

            <Rectangle Fill="{TemplateBinding BorderBrush}" />
            <TextBlock Text="{TemplateBinding HeaderText}"
              Foreground="{TemplateBinding Foreground}"
              VerticalAlignment="Center" Margin="2" />

            <<ContentPresenter Grid.Row="1" Margin="2" />>
          </Grid>
        </Border>
      </ControlTemplate>
    </Setter.Value>
  </Setter>
</Style>
```

The ContentPresenter's default control template automatically binds its Content property to the control's Content property and its ContentTemplate property to the control's ContentTemplate property, and it will handle displaying whatever control is assigned to the control's Content property within itself.

The corresponding code for this control is minimal, simply setting the DefaultStyleKey property for the control and defining a HeaderText dependency property (which you will see was bound to using the TemplateBinding markup extension in the control's template):

```
public class GroupBox : ContentControl
{
```



```

public GroupBox()
{
    DefaultStyleKey = typeof(GroupBox);
}

public static readonly DependencyProperty HeaderTextProperty =
    DependencyProperty.Register("HeaderText", typeof(string),
        typeof(GroupBox), null);

public string HeaderText
{
    get { return (string)GetValue(HeaderTextProperty); }
    set { SetValue(HeaderTextProperty, value); }
}
}

```

You can now assign a control as content to this control using content element syntax, like so:

```

<my:GroupBox Height="120" Width="250" HeaderText="GroupBox Header">
    <TextBlock Text="[Some content]"
        VerticalAlignment="Center" HorizontalAlignment="Center" />
</my:GroupBox>

```

■ **Note** You may recall from Chapter 3, “An Introduction to XAML,” that in order to be able to assign a value to a property on a control using content element syntax, that property must be defined as the content property for the control. To set a property as the content property, you need to decorate the control’s class with the `ContentProperty` attribute, passing it the name of the property to automatically be assigned the control’s content (i.e., the XAML between the control’s beginning and end tags). This enables you to avoid using property element syntax (which requires you to explicitly specify which property you are assigning the value to) and simplifies the XAML when the control is being consumed in a view. When you inherit from `ContentControl`, the `ContentControl` class already has the `ContentProperty` attribute applied, specifying that the Content property is to be automatically assigned the control’s content. Therefore, you do not need to worry about decorating your control’s class with this property (unless you wish to specify a different property to be automatically assigned the control’s content).

Containing Multiple Controls as Content

One of the limitations of our current `GroupBox` control is that it allows you to directly assign only a single control as its content. The Content property is of type `Object`, and attempting to assign multiple controls to the Content property in XAML will result in an error stating, “The property ‘Content’ is set more than once.” If you want to be able to assign multiple controls using content element syntax on a custom control, you will need to specify a different property as the control’s content property, and that property must be a collection type.

■ **Note** There are a number of examples of controls that accept more than one control, such as the Grid, Canvas, and StackPanel controls. These controls inherit from Panel (instead of ContentControl, as this control does), whose content property is its Children property. We'll create a similar property with the same name on our GroupBox custom control and set it to be our content property for the control.

Start by defining a property on your control that will maintain a collection of the content controls. For this purpose, we'll create a property named Children, of type ObservableCollection<UIElement>.

```
public static readonly DependencyProperty ChildrenProperty =
    DependencyProperty.Register("Children", typeof(ObservableCollection<UIElement>),
        typeof(GroupBox), new PropertyMetadata(new ObservableCollection<UIElement>()));

public ObservableCollection<UIElement> Children
{
    get { return (ObservableCollection<UIElement>)GetValue(ChildrenProperty); }
    set { SetValue(ChildrenProperty, value); }
}
```

We now need to define the new Children property as our content property, by decorating the control's class with the ContentProperty, like so:

```
[ContentProperty("Children")]
public class GroupBox : ContentControl
```

You will now be able to assign multiple controls to the Children property using content element syntax, like so:

```
<my:GroupBox HeaderText="GroupBox Header" Height="90" Width="200">
    <TextBlock Text="Child 1" />
    <TextBlock Text="Child 2" />
    <TextBlock Text="Child 3" />
    <TextBlock Text="Child 4" />
</my:GroupBox>
```

Note that none of the TextBlock controls will actually be displayed in the GroupBox's content area as yet. All we are doing is adding these controls to the collection exposed by the Children property. The ContentPresenter in the control template is still bound to the control's Content property, so we'll need to change this to display the controls added to the Children collection.

How you will arrange these controls within the content area of the GroupBox control when you do display them is the next question. For the purpose of this example, let's divide the content area of the control into a two-by-two grid and display the first four controls assigned to the Children collection, one in each cell. There are a number of ways you could implement this, and we won't cover any of these ways in depth, but the basic concept is as follows. Each cell in the grid will contain a ContentPresenter. The Children property on the control is of type ObservableCollection, meaning that we can handle the CollectionChanged event that it exposes, which will be raised when an item is added or removed from this collection. In this event handler, you can then assign each item in the collection to a ContentPresenter in the control template. This will result in the output shown in Figure 11-7.

GroupBox Header	
Child 1	Child 2
Child 3	Child 4

Figure 11-7. A content control with multiple controls as content

Attached Properties

Back in Chapter 3, “An Introduction to XAML,” we discussed the concept of attached properties. In summary, these are properties that are registered on a control, but are then able to be used by *any* control. You can define your own attached properties, either with or without a backing dependency property.

Let’s say you want to define an attached property named `ContentArea` on your control (which you could define on the `GroupBox` control, enabling the controls assigned to its `Children` property to specify which cell in the two-by-two grid they should appear in). To define `ContentArea` as an attached property (of type `integer`) requires two static methods to be defined on the control, one named `GetContentArea` and the other named `SetContentArea`, each with the following signatures:

```
public static int GetContentArea(DependencyObject obj)
{
}

public static void SetContentArea(DependencyObject obj, int value)
{
}
```

By following this convention (of both the method names and their corresponding signatures), the Silverlight runtime will recognize these two methods combined as defining an attached property of type `integer`, named `ContentArea`. For example, you can now assign the `ContentArea` attached property a value on another control, like so:

```
<TextBlock Text="Child 1" my:GroupBox.ContentArea="1" />
```

Whenever a control assigns a value to the `ContentArea` attached property on another control, the `SetContentArea` method will be called. A reference to the control that assigned the value will be passed into the method as the first parameter, and the value that has been assigned to the attached property will be passed in as the second parameter. You then need to store this value along with the corresponding control that it was defined on. When you need to get the value for the attached property as assigned on a given control, you can then call the `GetContentArea` method, passing a reference to the given control into the method as its parameter.

■ **Note** If you want to limit the type of controls that the attached property can be assigned a value on, change the type of the first parameter as per the example above from `DependencyObject` to the type of the control that it should be limited to.

One of the issues of defining an attached property like this (without a backing dependency property) is that you'll need to maintain your own repository (within the custom control that defines the attached property), which stores the value assigned to this property for each control that does so. Another issue is that you can't assign the attached property a binding. Therefore, you will usually see attached properties defined with a corresponding dependency property that will store that value.

When registering the backing dependency property, instead of calling the normal `Register` method to register the dependency property, call the `RegisterAttached` method instead if it is an attached property, for example:

```
public static readonly DependencyProperty ContentAreaProperty =
    DependencyProperty.RegisterAttached("ContentArea", typeof(int),
        typeof(GroupBox), null);
```

You will then be able to use this as the store for the values, like so:

```
public static int GetContentArea(DependencyObject obj)
{
    return (int)obj.GetValue(ContentAreaProperty);
}

public static void SetContentArea(DependencyObject obj, int value)
{
    obj.SetValue(ContentAreaProperty, value);
}
```

■ **Note** The `propdp` snippet was discussed earlier in this chapter as helping make defining dependency properties quick and easy. Visual Studio also has a built-in snippet to help in creating attached properties too, named `propa`. Like the `propdp` snippet, it too was designed for WPF and will require the reference to `UIPropertyMetadata` to be changed to `PropertyMetadata` when using it in Silverlight projects.

Summary

In this chapter, you learned about each of the methods you can create reusable controls for use in your application. Which method you should use really depends on what you are trying to achieve:

- Inherit from an existing control if you simply want to add your own additional behavior to it.

- User controls are best for encapsulating some common functionality that is more or less specific to a single project.
- Custom controls are best when you want to create a more generic control that could be used in other projects and be able to retemplate them when they are used.
- Content controls enable your control to act as a container for other controls.

CHAPTER 12



The Model-View-ViewModel (MVVM) Design Pattern

Few topics surrounding Silverlight are so widely discussed and debated as the Model-View-ViewModel design pattern (most commonly referred to by its acronym MVVM). This pattern is used by many Silverlight and WPF developers to form the structure of their applications. The blogosphere is overflowing with this topic, with many varied opinions on how it should be implemented and with little in the way of a consensus. There's so much chatter that it even still inspires passionate debate on just what the core principles of MVVM actually are.

Unfortunately, because of the lack of a standard and the amount of noise and controversy surrounding the pattern, you can face complete information overload while researching the pattern, with too many conflicting opinions to deal with and a lack of practical examples. This tends to lead to a general sense of confusion and often leads to developers giving up in frustration and returning to old code-behind practices. Adding to this is the fact that the project templates in Visual Studio are not geared toward using MVVM, the tooling (particularly Visual Studio, with Expression Blend having better support for wiring Views to ViewModels) does not direct the developer in this direction, and the Silverlight runtime itself hasn't had the features required to implement all the concepts of MVVM (such as support for commands, although this has improved in Silverlight 4).

Regardless of all the controversy and lack of standardization of the pattern, there is a general agreement in the development community that regardless of how you implement the pattern, doing so is good practice. Developers who implement this pattern will invariably say that their code is much better for doing so.

Up until now (possibly controversially), we have avoided discussion or implementation of the MVVM design pattern in the examples in previous chapters in order not to overload you with too many competing new topics at the one time. Doing so would be too much to take in, and redirect your focus from the concepts that were actually trying to be imparted. That said, most of the concepts used by MVVM have now been discussed as stand-alone topics, making this the best time to put these pieces together in terms of a design pattern for structuring your project in a more robust fashion.

■ **Note** Implementing MVVM requires the understanding of too many concepts in Silverlight to have used it from the start of this book. The path that we have tried to take is to provide an understanding of the core concepts of Silverlight/XAML, demonstrate how to get a working program going, and now, we are finally ready to look at a way of structuring your Silverlight applications in a better way. Be warned, however, like learning Silverlight itself, learning and implementing MVVM is a challenge and provides a steep learning curve, requiring you to think differently to how you would when using the traditional code-behind Model. But again, once you’ve worked through that, implementing the pattern will feel completely natural, and you will find it hard to go back to the old code-behind Model again.

In this chapter, we’ll take a look at the core concepts of the MVVM design pattern and practical aspects of their implementation, in order to give you a good grounding of the pattern upon which you can grow. We’ll try and avoid the more controversial aspects of the pattern and too much theorizing. We’ll take a simple approach, rather than going into a full-blown MVVM implementation discussed in the concept of a framework (as the framework you choose to use should be determined by the nature of your project—if indeed you choose to use a framework at all). That said, this chapter won’t include all the answers to questions you may have, but it will provide you with a good understanding of MVVM’s core principles.

The Purpose of Implementing MVVM

The primary purpose of implementing MVVM (although even this point is still being debated) is to maintain a clear separation of concerns between the View’s look and its behavior, decoupling them and using data bindings and commands or behaviors as the glue that connects them.

If this sounds like a familiar concept, it should. We discussed almost exactly the same concepts back in Chapter 11, “Creating Custom Controls,” in relation to how the structure of a custom control is designed to maintain a clear separation of its look (i.e., control template) from its behavior (in its class). If you skipped over that chapter (not being concerned with creating custom controls), you should go back and read it now (particularly the sections detailing separation of concerns, template binding, dependency properties, and discussion of the push versus pull of data into the user interface), as it provides a lot of prerequisite knowledge on which this chapter will build. It also provides a good point to start practicing implementing the separation of concerns principles (in the context of designing user controls or custom controls), which you can then apply to your Views (albeit with some differences) using the MVVM design pattern.

Although the implementation will be somewhat different, the MVVM design pattern does share a lot in common with the way that you design custom controls. As described shortly, you have a View (as defined back in Chapter 4, “The Navigation Framework”), and you also have a ViewModel (which defines the logic and behavior of the View and exposes data to it), taking this code out and away from the View’s code-behind. This behavior, logic, and data is then presented to the View in much the same way as the behavior, logic, and data for a custom control is presented to its template for consumption, leveraging the powerful data binding features enabled by XAML. Some perceive that MVVM’s use in a project should result in zero code-behind in a View, although in reality, this outcome is rarely practical.

Why You Should Implement MVVM

One of the big issues with having all (or much of) the logic and behavior of a View written in its code-behind is that it's simply not testable. Code-behind contains an intermingling of the View's look and its behavior, with each requiring (and having) implicit knowledge of one another. This bidirectional tight coupling makes testing the View's behavior incredibly difficult and messy.

■ **Note** Implementing unit testing and performing test-driven development in Silverlight is not covered in this book, but it is an important technique that you should investigate and preferably implement in your application development process.

Even if you do not write unit tests against your code (although you should strongly consider doing so) and don't consider testability a concern, implementing MVVM in your application provides numerous other benefits:

- It allows you to create blendable projects, enabling you to easily populate your Views with data at design time, so the designer can see what the user interface looks like populated with data without needing to compile and run the project.
- It enhances the designer/developer workflow, reducing the friction when both parties work simultaneously on the look and the behavior and logic of the View without interfering with one another.
- It provides a step toward structuring your project to support the implementation of inversion of control principle and dependency injection.
- It enables you to potentially reuse the View logic across different View implementations.
- The loose coupling enables you to potentially apply different Views to the same ViewModel. For example, if you wanted to create a Windows Phone 7 edition of your application (which would require a different View designed to suit the Windows Phone 7 controls and form factor), you could reuse your ViewModels in that project.

■ **Note** If you decide not to write unit tests against your project, having implemented MVVM in your project allows you to retrofit your project with unit tests at a later time without requiring too much (if any) restructuring.

As you can see, there are many potential benefits gained from implementing the MVVM design pattern. However, one of the reasons why MVVM is possibly not more widely used is the difficulty in attempting to grasp the finer details of its implementation (even if all the theory makes sense). With a lack of consensus in how to implement the pattern and the fact that developers aren't immediately being

pushed in that direction by the Silverlight tooling and templates, getting up to speed with implementing MVVM can be somewhat of a struggle. Only the dedicated developers prepared for the challenge in order to achieve better code structure tend to currently be prepared to face the hurdles it initially faces them with. Sticking points still exist (displaying dialogs from a ViewModel is a prominent one), but elegant solutions to these problems are gradually being identified and documented. That said, most of the issues revolve around the developers themselves, often attempting to take too pure an approach in order to achieve zero code-behind, which leads to fighting an uphill (and often unnecessary) battle.

Ultimately, it must be said that like learning Silverlight itself, once you get over the initial hurdles and the mental leap, you will never look back; MVVM will just flow as a natural way of designing software.

MVVM Theory

The MVVM design pattern is fundamentally about dividing up your application into layers (providing a clean separation and boundary between them) and providing the connections between the components in those layers. When all is said and done, that's what it comes down to.

Although Views don't typically lend themselves to having a clean separation between their look and their behavior like custom controls do (where the separation is enforced), the core reasoning behind maintaining the separation between look and behavior is the same, and many of the same structural principles still apply.

Let's take a look at the core principles of the MVVM design pattern, and how you can put these principles in place in your applications.

The Layers

The MVVM design pattern consists of three core layers, with the names of the components found in each layer used to form the pattern's name:

- Model
- View
- View Model

You could think of the pattern as being quite similar to the traditional three-tier pattern (consisting of a presentation tier, business logic tier, and data tier). Although they are quite different patterns and serve different needs, we could compare the layers in the MVVM design pattern to the tiers in the three-tier pattern:

- The View layer corresponds to the presentation tier.
- The Model layer corresponds to the data tier.
- The ViewModel layer corresponds to the business logic tier.

However, all of the layers forming the MVVM design pattern actually exist within the presentation tier of the three-tier pattern (with your Silverlight application acting as the entire tier), with each layer focused on serving the needs of the user interface. Figure 12-1 demonstrates how the layers within the MVVM design pattern relate to each other (with the arrows representing a "knows about" or "communicates with" interaction).

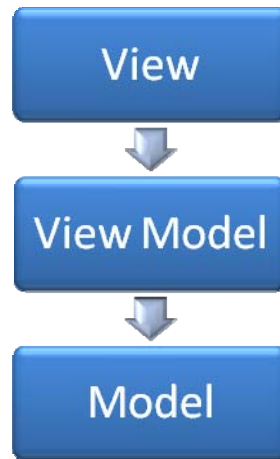


Figure 12-1. *The relationships between the View, ViewModel, and Model layers*

As Figure 12-1 shows, outside of their own layers, the Views should only ever know about ViewModels, and the ViewModels should only ever know about the Models. However, in practice, this is not always followed and is not necessarily a concept that you should enforce. Whilst the layers should (debatably) never know anything about the layers above them, some advocate allowing Models to be passed directly to a View (from the ViewModel).

■ **Note** As will be discussed later, the ViewModel-first approach is another ongoing debate that violates the ideal scenario of each layer not knowing anything about the layer above. In a ViewModel-first approach, the ViewModels will be responsible for instantiating the Views, and will hence have knowledge of the View layer.

Let's now take a closer look at the purpose of each of these layers.

Views

You are already well versed in the purpose of Views by now. A View consists of both a XAML file and a code-behind class. Throughout this book, we have made extensive use of the navigation framework, and in this context, both the XAML file and the class inherit from Page (as discussed back in Chapter 4, “The Navigation Framework”). However, this is of no consequence and has no impact on the implementation of the pattern. You may choose use Prism, or perhaps no framework at all. For the purpose of the examples in this chapter however, we will assume you are using the navigation framework for structuring and composing your user interface.

Theoretically, when using MVVM, the View's code-behind class should be considered essentially irrelevant, with little to no code written in that class. Instead, any logic and behavior for the View should be placed in the ViewModel class (where possible), leaving the code-behind class empty, a goal often referred to as “zero code-behind”.

■ **Note** Zero code-behind is not a requirement of the MVVM pattern, and often, it is an unrealistic goal. Some tasks are simply better performed in the code-behind and attempting to do otherwise will lead to increasing the complexity of your code and decreasing its maintainability, which all in all is the ultimate goal and primary focus of implementing the MVVM pattern. Focus on minimizing the use of code-behind instead of removing it completely.

The View will pull data from the ViewModel, using the extensive data-binding capabilities provided by Silverlight/XAML. Any events that the View wants to raise (to be handled by a method defining some behavior or logic in the ViewModel) can either be published as a command (that the ViewModel can subscribe to), or an action or behavior can be used to call a method on the object assigned to the View's data context (which will be the ViewModel).

Models

A Model is an object, usually containing data (obtained from the server) for use by the View. When you are making use of RIA Services to communicate with the server, your Model objects will be the entities (objects) returned by a domain service.

View Models

Although the View and the Model both leverage existing components that you are already familiar with, the ViewModel is an entirely new piece to the puzzle. View Models are used to maintain the state of a View. In this, they have a dual purpose—to expose data to the View and to handle the View's logic and behavior, essentially serving the needs of the View.

■ **Note** For the purposes of this explanation (to keep the scenario simple), we will be following the case where each View has a single corresponding ViewModel (i.e., a one-to-one relationship). However, Views and ViewModels can exist in multiple configurations: a ViewModel may have multiple Views, or Model objects may be wrapped in ViewModels before being exposed to the View. These configurations will be discussed further shortly, but for now, we will focus on the scenario where each View has a corresponding ViewModel.

Although its name may indicate that the purpose of a ViewModel is to provide a View of the Model (abstracting the Model for consumption by the View), it's really the other way around. The ViewModel is used to maintain the state of the View and, therefore, should be considered *a Model of the View*. That state will be maintained within the ViewModel (exposed as properties), with operations (exposed as methods) that can be invoked by a command or an action or behavior from the View (usually in response to a control event). The View will then bind to this ViewModel to consume the data that it exposes and invoke operations on it (generally in order to trigger a change in the View's state, represented by the ViewModel).

By exposing both the data (via properties) and the logic or behavior (via methods) publicly from the `ViewModel` class (enabling their consumption by the `View`), the `ViewModel` also becomes inherently testable. Ideally, therefore, all the behavior and logic for a `View` (that can't be handled in the `View`'s XAML) should be placed in its corresponding `ViewModel`, leaving minimal (if any) code in the `View`'s code-behind.

View / View Model Configurations

Throughout this chapter, we'll be demonstrating a scenario where we have a one-to-one relationship between the `Views` and the `ViewModels`, with each `ViewModel` representing its corresponding `View` as a whole. This is an ideal scenario for learning the pattern (and reasonably common in practice), however it isn't a fixed requirement of the pattern. Let's take a look at some additional configurations you may come across or use yourself.

One View Model Serving Multiple Views

For example, you may have one `ViewModel` that serves multiple `Views`; a wizard is a good example of where this might be required. However, this would require a slight alteration to our diagram, as when implementing this configuration (generally), the `ViewModel` will be responsible for creating the `View` and will therefore know about the `View`. This factors into the `ViewModel`-first debate, which will be discussed later in this chapter.

One View Interacting with Multiple View Models

In turn, a single `View` may interact with multiple `ViewModels`. Instead of having the `ViewModel` exposing `Model` objects directly to the `View`, some developers choose to wrap their `Model` objects in `ViewModels`, abstracting the `Model` from the `View`. Instead of the `View` binding directly to the data (i.e., the `Models`), it will bind to a `ViewModel` instead.

For example, a `View` may populate a list box with a collection of `ProductViewModel` objects, with each wrapping a `Product Model` object. These `ViewModels` could essentially be considered a view of the `Model`. Additional properties, calculations, logic, etc, could be added to these `ViewModel` objects (serving the needs of the `View`), saving the `Model` from being polluted with `View`-related requirements.

■ **Note** Another potential benefit of this practice is that it can take away the need to implement value converters in your bindings to convert the property values of the `Model` to suit the needs of the `View`, and these conversions can be performed within the `ViewModel` and exposed as properties that the `View` can directly bind to and consume. For example, rather than implementing a value converter to convert a `Boolean` value exposed by a property on the `ViewModel` to a `Visibility` value, you could expose the `Visibility` value directly from the `ViewModel`. That said, doing this tends to make your `ViewModel` a little too coupled to the needs of the `View`, which some developers consider to be a bad practice. Therefore, if you choose to do this, do so sparingly.

The big issue with the practice of wrapping all your Model objects in ViewModels is the amount of overhead and additional code required to wrap each of the Model objects in a ViewModel, often resulting in properties on the ViewModel that purely exist to relay property values from the Model objects. Most of the time the outcome of this practice is simply an unnecessary duplication of code. Therefore, whether you undertake this practice will generally be on a case-by-case basis.

■ **Note** The XAML Power Toys add-in (discussed back in Chapter 7, “Building Data Entry Forms”) has a Create ViewModel For Class option that you can use to take a lot of the manual work out of the task of wrapping Model objects in ViewModels.

MVVM in Practice

The examples throughout this section will implement a very simple scenario to demonstrate the MVVM design pattern and how the layers communicate with one another. For the purpose of demonstration, we’ll look at implementing each aspect of the MVVM design pattern in a simple yet common example. This example will implement a login screen that allows the user to enter a user name and password, authenticate credentials, and navigate to another View if the authentication was successful or display an error if it failed.

■ **Note** For this scenario, we’ll be assuming a one-to-one relationship between the View and the ViewModel to keep the implementation of this example simple.

Creating the Layers

To implement our simple scenario, let’s start by creating the objects in the layers independently, before bringing them together.

Creating the View

By now, you will be familiar with creating new Views using the navigation framework. We’ll create a simple View for this example consisting of two text boxes (with corresponding labels) for the user to enter a user name and password into and a button that will initiate the user authentication check on the server, as per the following XAML:

```
<navigation:Page x:Class="Chapter12Sample.Views.LoginView"
    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:navigation="clr-namespace:System.Windows.Controls; ➤
```

```

        assembly=System.Windows.Controls.Navigation"
xmlns:sdk="http://schemas.microsoft.com/winfx/2006/xaml/presentation/sdk"
mc:Ignorable="d" Title="Login">

<Grid x:Name="LayoutRoot" Background="White" Width="300" Height="110">
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="90" />
    <ColumnDefinition Width="*" />
  </Grid.ColumnDefinitions>
  <Grid.RowDefinitions>
    <RowDefinition Height="35" />
    <RowDefinition Height="35" />
    <RowDefinition Height="*" />
  </Grid.RowDefinitions>

  <sdk:Label Content="User Name:" Margin="5,0"
    HorizontalContentAlignment="Right" />
  <sdk:Label Content="Password:" Margin="5,0"
    HorizontalContentAlignment="Right" Grid.Row="1" />

  <TextBox Grid.Column="1" Name="UserNameTextBox" Height="25" />
  <TextBox Grid.Column="1" Grid.Row="1" Name="PasswordTextBox" Height="25" />

  <Button Content="Log In" Grid.Column="1" Grid.Row="2" Height="30" Width="90"
    HorizontalAlignment="Right" VerticalAlignment="Top"
    Name="LoginButton" Margin="0,5,0,0" />
</Grid>
</navigation:Page>

```

This gives you the View layout shown in Figure 12-2.



The image shows a simple login interface. It consists of two text input fields. The first field is preceded by the label 'User Name:'. The second field is preceded by the label 'Password:'. Below these two fields is a button labeled 'Log In'.

Figure 12-2. The login View

Creating the Model

For the purposes of this example, where we are primarily concerned with the interaction of the View and the ViewModel, we'll be disregarding the Model from the equation to keep things simple. However, this would (in practice) be a user object such as the User entity returned from RIA Services.

Creating the View Model

While implementing the previous two layers (containing the Views and Models) will be familiar to you, the ViewModel is a new piece that you will need to implement in the puzzle that is MVVM. As mentioned previously, ViewModels expose data to the View as properties and operations as methods.

Usually, your ViewModels will inherit from a base class (e.g., `ViewModelBase`) that provides common functionality required by all the ViewModels in your project. At a minimum, this base class will implement the `INotifyPropertyChanged` and the `INotifyDataErrorInfo` or `IDataErrorInfo` interfaces (as was discussed back in Chapter 7, “Building Data Entry Forms”). If you make use of a MVVM framework, this base class will generally be provided as a key part of the framework. The sample code accompanying this chapter contains a simple base ViewModel that you could alternatively use, as per Figure 12-3.

■ **Note** You may choose to have your `ViewModelBase` class inherit from `DependencyObject`, so that you can implement dependency properties if you wish. However, dependency properties are rarely required in ViewModels, and thus inheriting from `DependencyObject` is not really necessary.

What other properties will be created on your ViewModel will depend on the strategy you have chosen to take for exposing data from Model objects to your View. Each property from the Model that should be available to the View might have a corresponding property on the ViewModel (which the View can bind to), or you may simply return the Model object directly to the View for its consumption. In this simple example, we don't need to consume data from Model objects in our View, but we'll look further at this scenario of consuming data using the MVVM design pattern later in this chapter when we discuss its use in conjunction with RIA Services.

The class diagram of the ViewModel that we will use for this example is shown in Figure 12-3.

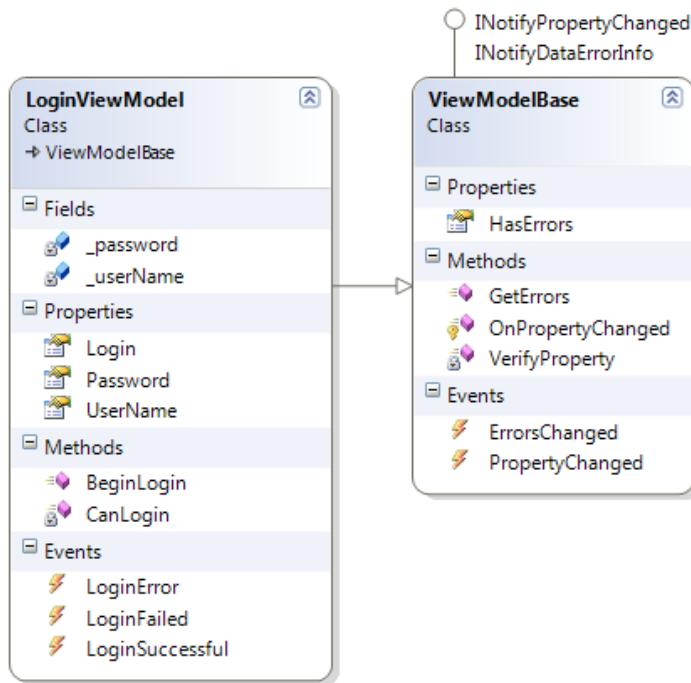


Figure 12-3. The *LoginViewModel* class and the *ViewModelBase* class

■ **Note** All properties exposed by the ViewModel should raise the `PropertyChanged` event (from implementing the `INotifyPropertyChanged` interface) when their value changes to notify the View so that it can update itself accordingly. You can apply the same data annotations as you use in your entity classes to provide validation for your ViewModel properties, but note that you will have to use the `Validator` class (as described in Chapter 7, “Building Data Entry Forms”) to validate the properties in their setters.

Connecting the Layers

We now have three independent layers, with a reasonably clean boundary defined between each, but we need to connect them together. As previously discussed, the View will know about the ViewModel, and the ViewModel will know about the Model. Essentially, this means that the View will have a reference to the ViewModel, and the ViewModel will have a reference to each of the Models it makes use of. Let’s take a look at how each of these layers references each other (where permitted by the pattern).

Connecting the Models to the View Models

There's nothing special about the way that the Model objects are connected to the ViewModel; often, the ViewModel will hand off the tasks to a service agent that handles retrieving/instantiating them and then returns them to the ViewModel. It can then retain a reference to these objects for future use.

However (assuming you are using RIA Services to obtain the data from the server), you will remember from Chapter 6, "Implementing Summary Lists," that there are two methods you can use to communicate with the domain services on the server. You can use the DomainDataSource control, or you can use the data context object directly. The DomainDataSource control is designed for use in Views, and thus would completely bypass the ViewModel if you were to use it, a violation of the MVVM design pattern. Hence, when implementing MVVM, you should communicate with a domain service from the ViewModel using its corresponding data context directly (which acts as a service agent).

Connecting the View Model to the View

The ViewModel is connected to the View by way of having an instance of it assigned to the DataContext property of the View, enabling properties of the controls in the View to bind directly to properties on the ViewModel.

However, how this ViewModel instance is instantiated can vary greatly and is one of the more hotly debated implementation details of the pattern. We'll discuss these methods further later in this chapter, but for now, we will be taking the approach in which the View is responsible for instantiating the ViewModel.

You generally instantiate the ViewModel either within the View's XAML or its code-behind. To be able to do so in XAML, the XAML parser requires the ViewModel to have a parameterless constructor. If this is not possible (e.g., the View needs to pass the ViewModel the ID of an entity to load), you will have to instantiate it from the code-behind instead.

For example, instantiating a ViewModel in XAML and assigning it to the DataContext property of the View can be done as follows:

```
<navigation:Page x:Class="Chapter12Sample.Views.LoginView"
    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:navigation="clr-namespace:System.Windows.Controls;
        assembly=System.Windows.Controls.Navigation"
    xmlns:vm="clr-namespace:Chapter12Sample.ViewModels"
    mc:Ignorable="d" Title="Login">
    <navigation:Page.DataContext>
        <vm:LoginViewModel />
    </navigation:Page.DataContext>

    <Grid x:Name="LayoutRoot">
        <!-- Removed for the purpose of brevity -->
    </Grid>
</navigation:Page>
```

Alternatively, to do the same in the constructor of the class in the code-behind would be implemented like so:

```
this.DataContext = new LoginViewModel();
```

View and ViewModel Interactions

How the View and the ViewModel interact is the most difficult piece to understand of the MVVM puzzle, and most of the perceived complexity of the MVVM design pattern stems from this aspect. These can be broken down into three core interactions, which we will look at individually, as they are each handled and implemented differently:

- Pulling data into the View from the ViewModel
- Enabling the View to invoke an operation (method) on the ViewModel
- Enabling the View to respond to an event in the ViewModel

Pulling Data into the View from the ViewModel

By now, you should be fairly familiar with the power of Silverlight's data-binding capabilities and how to take advantage of them in your applications. The ViewModel maintains the View's state (which is exposed as properties), and bindings (defined within the View) provide a two-way connection bridging the boundary between the View and the ViewModel. This enables the View to consume the data from the ViewModel and update the ViewModel again with any changes made to the data in the View (via the two-way data binding connection demonstrated in Figure 12-4).



Figure 12-4. The interaction between the View and the ViewModel using bindings

The Text property of both text boxes in our View will be bound to properties on the ViewModel (using two-way binding). The Text property of the UserNameTextBox text box in the View will be bound to the UserName property on the ViewModel, and the Text property of the PasswordTextBox text box in the View will be bound to the Password property on the ViewModel, like so:

```
<TextBox Grid.Column="1" Name="UserNameTextBox" Height="25"
    Text="{Binding UserName, Mode=TwoWay}" />
<TextBox Grid.Column="1" Grid.Row="1" Name="PasswordTextBox" Height="25"
    Text="{Binding Password, Mode=TwoWay}" />
```

■ **Note** If you bind the control properties to the ViewModel's properties, the bindings also listen for the `PropertyChanged` and the `ErrorsChanged` events raised by the ViewModel (if the `INotifyPropertyChanged` and `INotifyDataErrorInfo` interfaces are implemented) indicating that a property's value has changed or that the collection of validation errors has changed. Hence, this is an additional, hidden interaction between the layers.

One issue that you may have is when you need to bind a control (such as a `ListBox` or a `DataGrid`) in the View to a collection exposed from the ViewModel (as a property) but don't have that collection available at the time that the ViewModel is instantiated. For example, say you need to bind a `ListBox` control in the View to a collection of objects exposed by the ViewModel. However, the ViewModel doesn't have that collection available when it is instantiated, as it has to go off and retrieve the collection from the server. How do you handle this problem?

The answer is to have the ViewModel expose the collection as a property of type `ObservableCollection`. When the ViewModel is instantiated, an empty `ObservableCollection` should be instantiated and made available to the View for the `ListBox` control to bind to. Then when the collection has been retrieved from the server, its contents should be added to the empty `ObservableCollection` collection (instead of replacing it), which (as we discussed back in Chapter 6, "Implementing Summary Lists") the View is listening to for changes and updates accordingly as the data becomes available.

■ **Note** There will also often be an additional property exposed from the ViewModel named `IsBusy`. This is a Boolean value indicating whether or not the ViewModel is waiting for a response from the server (when retrieving or persisting Models (also known as entities), which a control such as the `BusyIndicator` control in your View can then bind to (as described in Chapter 6, "Implementing Summary Lists"). You can then bind this `IsBusy` property to the `IsBusy` property on a `BusyIndicator` control (using one-way binding), which will enable the View to display the indicator whilst the ViewModel is busy communicating with the server, informing the user that the application is waiting for data to be returned. However, to keep this scenario simple, we'll omit this property and binding for the time being.

Enabling the View to Invoke an Operation (Method) on the View Model

Operations (containing logic and behavior) will be exposed from your ViewModel as public methods. For example, in our login screen scenario, we have a `BeginLogin` method in the ViewModel and a Login button in the View. The `BeginLogin` method needs to be invoked in response to the user clicking the Login button in the View (hence the method should be invoked in response to the `Click` event of the button being raised).

There are three ways you can invoke an operation on a ViewModel in Silverlight:

- In the code-behind
- Using an action or behavior that will be triggered by an event being raised on a control
- Using commands (new to Silverlight 4)

Invoking the operation in the code-behind is the traditional solution to this problem, but it doesn't typically conform to the MVVM way of interacting between the View and the ViewModel. Therefore, we'll look instead at how to implement the other two methods now (as pictured in Figure 12-5).



Figure 12-5. The interactions of the View invoking an operation on a ViewModel

Actions/Behaviors

We discussed triggers, actions, and behaviors back in Chapter 10, “Advanced XAML and Data Binding,” these aren’t native functionality available in Silverlight but are available using the Expression Blend Interactivity library. You can leverage the `CallMethodAction` action from this library to call a method on the ViewModel, which provides a solution for invoking operations on the ViewModel from the View. In other words, when something happens in the user interface, call a method in the ViewModel.

For example, to call the `Login` method on the ViewModel in response to the `Click` event of the button, you would add the following XAML (in bold) to the button:

```
<Button Name="LoginButton" Content="Log In" Height="30" Width="90">
  <i:Interaction.Triggers>
    <i:EventTrigger EventName="Click">
      <ei:CallMethodAction TargetObject="{Binding}"
        MethodName="Login"/>
    </i:EventTrigger>
  </i:Interaction.Triggers>
</Button>
```

■ **Note** As was detailed back in Chapter 10, “Advanced XAML and Data Binding,” although this seems like a lot of XAML, Expression Blend makes configuring actions and behaviors easy, assigning them to a control using a point-and-click approach and configuring them using the Properties window. Unfortunately, Visual Studio doesn’t have the same ability built in.

Commands

Commands are a new topic that we haven’t yet discussed thus far in this book. In essence, commands provide a means for encapsulating an operation and its parameters away from the ViewModel. The logic for the operation is moved outside the ViewModel and into its own class (but acting upon the

ViewModel), decoupling the logic from the ViewModel state and enabling the logic to be reused. For example, you may have a login operation that can be invoked by two different ViewModels. Commands allow you to encapsulate this login logic and reuse it.

You may also like to think of commands as providing a means to bind to operations (as opposed to data). The command is exposed from the ViewModel as a property, enabling controls to bind to it and to invoke it in response to a given event.

■ **Note** Commands aren't strictly a part of the MVVM design pattern (being a design pattern of their own) but are commonly used to support the MVVM design pattern.

Creating a Command

For example, our Login method could be encapsulated in a command class and invoked when the Login button is clicked. To create a command class, it must implement the ICommand interface, as shown in Figure 12-6.

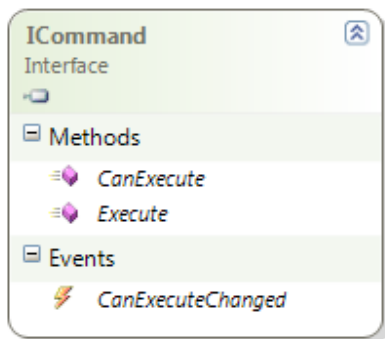


Figure 12-6. The ICommand interface

The CanExecute property specifies whether or not the command can be executed. When the command is bound to a button, the button will be enabled or disabled accordingly. In order for the button's enabled status to update when the property updates, the CanExecuteChanged event on the command should be raised. Any logic to be executed when the command is executed will go in the Execute method.

■ **Note** It's a common problem to forget to raise the CanExecuteChanged event after changing the value of the CanExecute property. If you find that your button is disabled despite having updated the value of the CanExecute property, this is most likely the source of the issue.

To create a simple command, create a new class named `LoginCommand` and implement the `ICommand` interface.

```
public class LoginCommand : ICommand
{
    public bool CanExecute(object parameter)
    {
        return true; // Alternatively, provide some logic here
    }

    public event EventHandler CanExecuteChanged;

    public void Execute(object parameter)
    {
        // Your command's logic goes here
    }
}
```

The next step is to expose this command from your `ViewModel` as a property (to make it available to the View). To do so, add the following property to your `ViewModel`:

```
private ICommand loginCommand = new LoginCommand(); // Member variable

public ICommand Login
{
    get { return loginCommand; }
}
```

■ **Note** Often when implementing a command in this manner, you will pass an instance of your `ViewModel` or an object to its constructor for the command to perform the operation upon. Alternatively, you might pass the `ViewModel` or the object to the command as a parameter, as will be discussed shortly.

Invoking a Command

Although the `ICommand` interface was available in Silverlight 3, there was no support provided for invoking commands by any controls until Silverlight 4 (i.e., there were no `Command` and `CommandParameter` properties on any controls). Therefore, to invoke a command in response to an event raised by a control in Silverlight 3, you had to use the `InvokeCommandAction` action from the Expression Blend Interactivity library.

Silverlight 4 has added limited support for invoking commands, but only those implemented on the `ButtonBase` class, which can now invoke a given command when its `Click` event is raised. `ButtonBase` isn't designed to be used directly, but it acts as the base class for the `Button`, `HyperlinkButton`, `ToggleButton`, and `RepeatButton` controls, all of which inherit this behavior.

To invoke a command in response to a button click, you bind the control's `Command` (and optionally `CommandParameter`) properties to the property on the `ViewModel` that exposes the command. In our login screen scenario, this means we can now bind the `Command` property of the `Login` button in the View to the `Login` property on the `ViewModel`, which will invoke the command when the button is clicked:

```
<Button Command="{Binding Login}" Content="Log In" />
```

■ **Note** Unfortunately no other controls have support for commands in Silverlight, but you can still use the `InvokeCommandAction` action to invoke a command in response to any event they raise (such as in response to an item in a `ListBox` control being selected).

The DelegateCommand

If you are more interested in the ability to bind to operations than moving the logic for an operation outside the `ViewModel`, you may wish to consider implementing a `DelegateCommand`. This command is designed to enable you to keep your operation methods within your `ViewModel` but still be able to expose them as command properties and bind to them without needing to create a class for each command. This results in making the implementation of commands much simpler.

An example of implementing a command on your `ViewModel` as a `DelegateCommand` (which is being used to call a method named `BeginLogin` on the `ViewModel`) follows:

```
// Member variable
private ICommand loginCommand = DelegateCommand(BeginLogin, CanLogin);

public ICommand Login
{
    get { return loginCommand; }
}

public void BeginLogin(object param)
{
    // Operation logic goes here
}

private bool CanLogin(object param)
{
    if (param != null && param.GetType() == typeof(bool))
        return (bool)param;
    else
        return true;
}
```

As you can see, rather than creating a separate command, we are simply instantiating an instance of the `DelegateCommand` class and passing it the method on the `ViewModel` to call when the command is invoked and a method with the logic specifying whether the command is allowed to be invoked.

Unlike WPF, Silverlight 4 doesn't have any `ICommand` implementations (like the `DelegateCommand` command) built in, but you can implement your own. You can find the code for an implementation of the `DelegateCommand` command in the sample code accompanying this chapter and use that in your project. You will also find an implementation of the `DelegateCommand` command in Prism (see the "Frameworks" section later in this chapter).

■ **Note** Another popular alternative to the `DelegateCommand` is the `RelayCommand`. An implementation of the `RelayCommand` command can be obtained from the MVVM Light Toolkit (one of the popular MVVM frameworks for Silverlight, which you will find listed in the “Frameworks section” later in this chapter).

Passing a Parameter to a Command

You can pass a parameter through to the `Execute` method of the command if you wish. This will be assigned to the `CommandParameter` property of the button, and its value will be passed into both the `CanExecute` method and the `Execute` method when the button is clicked.

For example, you can bind this property to a property on another control (such as the `SelectedItem` property on a `ListBox` control), and the value of that control’s property will be passed to the `Execute` method of the command as its parameter. Alternatively, you can bind it to an object exposed by your `ViewModel` (or the `ViewModel` itself) for it to perform the operation on.

The following example demonstrates passing the `ViewModel` to the command as a parameter (assuming that the `DataContext` property of the `Button` control has been assigned the `ViewModel` instance):

```
<Button Command="{Binding Login}" CommandParameter="{Binding}" Content="Log In" />
```

Actions/Behaviors vs. Commands

Whether you use actions/behaviors or commands to invoke methods on a `ViewModel` really comes down to personal taste. Commands enable you to encapsulate logic and decouple it from your `ViewModel` so that it can be reused, but they do require additional code to implement. The `CallMethodAction` action, however, is much easier to implement but requires a number of additional lines of XAML to be added to your View. That said, this is not generally an issue if you use Expression Blend to wire your View and `ViewModel` up. Therefore, the `CallMethodAction` action is often the preferred means of invoking a method on a `ViewModel`.

Enabling the View to Respond to an Event in the View Model

Looking at the other side of the coin now, how does the `ViewModel` notify the View that something has happened (to which it may wish to respond)? Exposing properties that elements in the View can bind to will suit some scenarios, but what if you need the View to respond to a notification from the `ViewModel` (such as when it raises an event)?

This is particularly important in Silverlight, where all calls to the server are asynchronous. In our login screen scenario, we ask the `ViewModel` to authenticate the user (by calling a method or invoking a command on the `ViewModel`) but don’t know the result when the method returns. Instead, an event handler is called on the `ViewModel` when the response from the server has been received, and we *then* need the View to respond accordingly. If the user was successfully authenticated, the View should navigate away from the login screen to the next View (such as a dashboard screen). If the authentication failed, the screen should display a message to the user notifying them of the failure.

We can implement this behavior by raising events (one for when the authentication is successful and another for when it fails) from our `ViewModel` and implement triggers in the View that listen for these events and respond accordingly (as per Figure 12-7).



Figure 12-7. The interaction of the View handling notifications from the ViewModel

In our example ViewModel, we have defined three events:

- LoginError
- LoginFailed
- LoginSuccessful

Let's look at implementing the LoginSuccessful event. In your ViewModel, you can raise this event in the standard way:

```
if (LoginSuccessful != null)
    LoginSuccessful(this, new EventArgs());
```

Now, in your View, you can use the EventTrigger trigger and listen for the LoginSuccessful event to be raised on the ViewModel. It can then perform an action in response (which for this scenario will require a custom action named NavigateAction to navigate to a given View), as demonstrated here:

```
<i:Interaction.Triggers>
  <i:EventTrigger SourceObject="{Binding Mode=OneWay}"
    EventName="LoginSuccessful">
    <mvvm:NavigateAction Url="/Dashboard" />
  </i:EventTrigger>
</i:Interaction.Triggers>
```

Layer Interaction Summary

Now that we've discussed the core concepts of the MVVM pattern, let's put them together in a single diagram (shown in Figure 12-8) so that you can get a complete overall perspective on the pattern.

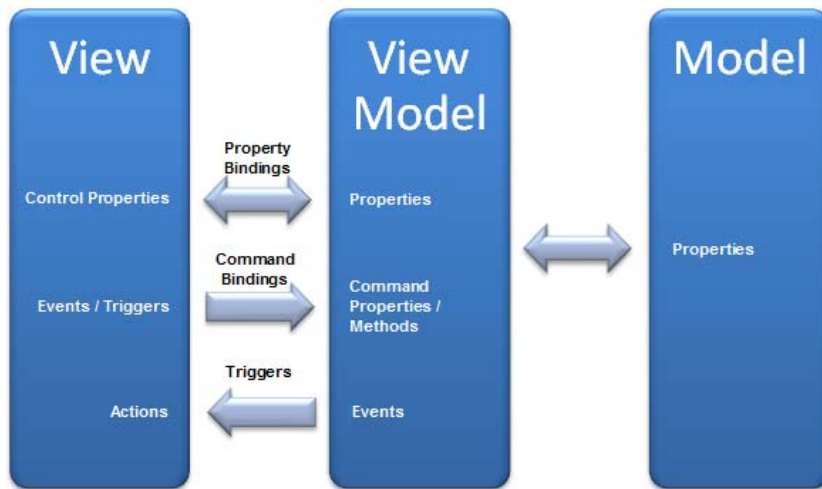


Figure 12-8. An overview of the interactions between the MVVM layers

Hotly Debated Implementation Details

As has been indicated throughout this chapter, there are still many implementation details for the pattern being heavily debated and yet to be standardized. It's worth pointing out these issues and their corresponding pros and cons. You will face many decisions in how you implement this pattern in regard to these issues, but with no clear conclusion on them currently, the decision will really come down to what suits your project and development process best.

Should Model Objects be Exposed Directly to the View?

As discussed earlier in this chapter, some developers like to keep a clean separation between the View and the Model layers, wrapping the Model objects in ViewModels before exposing them to the View. From a pure design perspective, ideally the View should know nothing about the Model objects and interact only with the ViewModel objects (keeping the View and the Model layers completely decoupled).

However, from a practical perspective, wrapping your Model objects in ViewModels adds additional overhead, increases the amount of code and the complexity, and reduces maintainability. Ultimately, it's generally best (depending on the scenario) to simply expose your Model objects (or collections of Model objects) directly to the View via a property on the ViewModel.

Should View Models Copy Data to and from the Models?

If you have chosen to not expose your Model objects directly to the View (exposing the data via a ViewModel instead), you will need to decide between these options:

- You will simply create property setters and getters in your ViewModel that expose the corresponding property values from the Model (with the ViewModel maintaining the original Model objects).
- Alternatively, you should copy all the data from a Model object into the ViewModel and rehydrate (or update) the Model object with the data from the ViewModel when you want to persist it back to the server/store again. This method adds additional overhead, which you may not wish to deal with.

What Should Come First, the View or the View Model?

This question has two parts and can be asked in regard to both the design and to the instantiation of the View and ViewModel. Whether you design the View or the ViewModel first will really depend on your development practices.

However, how the View and the ViewModel are instantiated is a more important question. We've demonstrated throughout this chapter that the View creates the ViewModel, and hence, the View maintains ownership over the ViewModel, with the ViewModel simply serving the View's needs (but without actually knowing anything about it).

ViewModel First

Some schools of thought prefer that the ViewModel should come first and handle instantiating the Views (and assign a reference to itself to the View's DataContext property). This is may suit scenarios where you have multiple Views being served by a single ViewModel.

View First

Alternatively, the View may handle instantiating the ViewModel or it may pass this job off to a ViewModel locator, getting it to return/instantiate the correct ViewModel and pass it back for the View to bind to. This ViewModel locator pattern is used by the MVVM Light Toolkit.

Completely Decoupled

Yet another school of thought prefers that both the View and the ViewModel be completely decoupled from each other and instantiated independently. This is generally the method preferred by those using inversion of control principles.

Should I Use a MVVM Framework?

A number of frameworks are available that can provide additional support for structuring your application using the MVVM design pattern (as discussed later in this chapter). However, some developers believe that you are better off rolling your own and that it's not necessary or worthwhile to implement a major framework. To properly grasp the principles of MVVM and identify if and where a framework can help you (without becoming too overwhelmed by all their features), it's probably best that you start off (in a small test project) without using a framework but investigate the available options when implementing a major project. Many of the popular frameworks go beyond simply supporting the MVVM design pattern and add a lot of helpful infrastructure to your project, so they are well worth looking into.

Should I Implement Dependency Properties or Standard Properties?

Whether you implement dependency properties or standard properties in your ViewModel ultimately comes down to personal preference. Because you are binding to the properties (which act as the *source* of the data), they don't have a need to be dependency properties (as opposed to developing custom controls where their properties can be bound to an object, hence acting as the *consumer* of the data).

Therefore, as a general rule, standard properties that raise the `PropertyChanged` event (from the `INotifyPropertyChanged` interface) are the best choice for implementing properties in ViewModels.

MVVM and RIA Services

In a typical business application, the ViewModels in your project will commonly be required to interact with the server via RIA Services. In this scenario, your entities exposed from the server will act as your Model objects in the MVVM pattern. A ViewModel will generally request data from the server, expose it to the View (where it can be Viewed and updated) and persist any changes to the data back to the server again.

Let's look at a fairly common scenario of requesting the details of a product from the server, enabling it to be updated by the user, and persisting these changes back to the server again—all using the MVVM design pattern. This is a scenario we looked at back in Chapter 7, “Building Data Entry Forms,” but we only looked at using the `DomainDataSource` control that enabled the View to communicate directly with the server and using a data context in the code-behind. We'll now look at the structurally more robust and testable means of handling the server communication in a ViewModel rather than in the View, using the MVVM design pattern.

Implementing the Model

As previously mentioned, the Model objects will be the entities exposed from the server via RIA Services. For this example, we'll use a simple `Product` entity containing three fields, as shown in Figure 12-9.

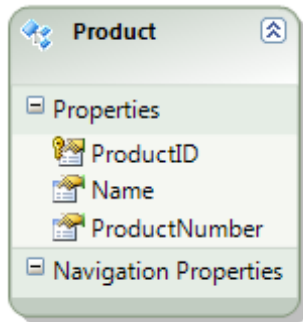


Figure 12-9. The *Product* entity / Model object

This entity will be exposed directly to the View, which will bind to it.

Implementing the View Model

Although you can technically instantiate a `DomainDataSource` control in your `ViewModel` and interact with the server via it, communicating directly with the domain context object is the usual approach. You already know the techniques of how to do this, as they were discussed in detail in Chapter 6, “Implementing Summary Lists,” and Chapter 7, “Building Data Entry Forms.”

In summary, you create an instance of a domain context, which you (generally) maintain for the lifetime of the `ViewModel`.

For this example, we’ll create a `ProductDetailsViewModel` class that inherits from our `ViewModelBase` class and handles requesting the data from the server, exposing it to the View (where it can be viewed and updated) and persisting any changes to the data back to the server again. The class diagram for our `ViewModel` is shown in Figure 12-10.

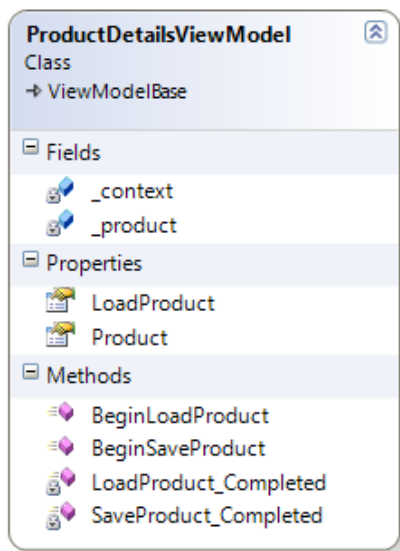


Figure 12-10. *The `ProductDetailsViewModel` class*

The most important aspects of this `ViewModel` follow:

- The `Product` property exposes a `Product` entity to the View.
- The `LoadProduct` command will call the `BeginLoadProduct` method, which will request the product data from the server.
- The `BeginSaveProduct` method will persist any changes to the `Product` entity back to the server.

■ **Note** You may be wondering why we are using a `LoadProduct` command to invoke the `BeginLoadProduct` method from the View but not providing a command to save the product (instead requiring the View to call the `BeginSaveProduct` method directly). This is primarily done to enable the demonstration of both techniques of calling methods on a `ViewModel`. Having a `SaveProduct` command is perfectly acceptable, rather than calling the `BeginSaveProduct` method directly. However, the `BeginLoadProduct` requires a parameter to be passed to it (the ID of the product to load), which using a command makes easier (the `CallMethodAction` doesn't have support for passing the method a parameter, whereas a parameter can be passed to a command).

The complete code for this `ViewModel` class follows:

```
using System;
using System.Linq;
using System.ServiceModel.DomainServices.Client;
using System.Windows.Input;
using Chapter12Sample.Web;
using Chapter12Sample.Web.Services;
using SimpleMVVM;

namespace Chapter12Sample.ViewModels
{
    public class ProductDetailsViewModel : ViewModelBase
    {
        #region Member Variables
        private ProductContext context = new ProductContext();
        private Product product = null;
        #endregion

        #region Public Properties
        public Product Product
        {
            get { return product; }
            set
            {
                product = value;
                OnPropertyChanged("Product");
            }
        }
        #endregion

        #region Commands
        public ICommand LoadProduct
        {
            get
            {
                return new DelegateCommand(BeginLoadProduct, (o) => true);
            }
        }
        #endregion
    }
}
```

```

#region Server Calls (and Completed event handlers)
public void BeginLoadProduct(object param)
{
    int productID = Convert.ToInt32(param);

    var op = context.Load( context.GetProductQuery(productID), true);
    op.Completed += LoadProduct Completed;
}

private void LoadProduct Completed(object sender, EventArgs e)
{
    LoadOperation op = sender as LoadOperation;

    if (!op.HasError && op.Entities.Count() != 0)
    {
        this.Product = op.Entities.First() as Product;
    }
    else
    {
        // Handle this error appropriately - omitted for brevity
        this.Product = null;
    }
}

public void BeginSaveProduct()
{
    if ( product != null)
    {
        var op = context.SubmitChanges();
        op.Completed += new EventHandler(SaveProduct Completed);
    }
}

private void SaveProduct Completed(object sender, EventArgs e)
{
    SubmitOperation op = sender as SubmitOperation;

    if (!op.HasError)
    {
        // You may wish to notify the View that the save is complete
        // by raising an event here, and responding to it in the View
    }
    else
    {
        // Handle this error appropriately - omitted for brevity
    }
}
}
#endregion
}
}

```

■ **Note** This ViewModel has been stripped to its core functionality (excluding error handling, maintaining an `IsBusy` property, and so on), so that you can focus on the most important aspects of the implementation. In a real project, however, this additional functionality would be expected.

Implementing the View

The final step is to create our View and wire it up to the ViewModel. Figure 12-11 shows the final View in action.

Product ID:

Product Name:

Product Number:

Figure 12-11. The View to edit the details of a product

This View loads the product with the ID specified in the Product ID text box when the Load button is clicked. The details will be displayed in the Product Name/Number text boxes (which are bound to the Product entity returned from the server), which the user can then edit. Clicking the Save button will save these changes back to the server.

The following XAML provides the complete XAML for this View and is wired up to our ViewModel:

```
<navigation:Page x:Class="Chapter12Sample.Views.ProductDetailsView"
    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:navigation="clr-namespace:System.Windows.Controls;
        assembly=System.Windows.Controls.Navigation"
    xmlns:vm="clr-namespace:Chapter12Sample.ViewModels"
    xmlns:i="http://schemas.microsoft.com/expression/2010/interactivity"
    xmlns:ei="http://schemas.microsoft.com/expression/2010/interactions"
    xmlns:sdk="http://schemas.microsoft.com/winfx/2006/xaml/presentation/sdk"
    mc:Ignorable="d" d:DesignWidth="640" d:DesignHeight="480"
    Title="Product Details">
<navigation:Page.DataContext>
    <vm:ProductDetailsViewModel />
</navigation:Page.DataContext>

<navigation:Page.Resources>
    <Style TargetType="TextBox">
        <Setter Property="Margin" Value="2" />
    </Style>
```

```

</navigation:Page.Resources>

<Grid x:Name="LayoutRoot" Margin="30">
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="110" />
    <ColumnDefinition Width="140" />
    <ColumnDefinition Width="*" />
  </Grid.ColumnDefinitions>
  <Grid.RowDefinitions>
    <RowDefinition Height="29" />
    <RowDefinition Height="5" />
    <RowDefinition Height="29" />
    <RowDefinition Height="29" />
    <RowDefinition Height="*" />
  </Grid.RowDefinitions>

  <sdk:Label Content="Product ID:" />
  <TextBox Name="ProductIDTextBox" Grid.Column="1" Text="1" />
  <Button Name="LoadButton" Content="Load" Height="23"
    HorizontalAlignment="Left" Width="75"
    Command="{Binding LoadProduct}"
    CommandParameter="{Binding ElementName=ProductIDTextBox, Path=Text}"
    Grid.Column="2" Margin="2" />

  <Line Grid.Row="1" Grid.ColumnSpan="3"
    X2="330" Y1="2.5" Y2="2.5" Stroke="Black" />

  <sdk:Label Content="Product Name:" Grid.Row="2" />
  <sdk:Label Content="Product Number:" Grid.Row="3" />

  <TextBox Text="{Binding Path=Product.Name, Mode=TwoWay}"
    Grid.Row="2" Grid.Column="1" />
  <TextBox Text="{Binding Path=Product.ProductNumber, Mode=TwoWay}"
    Grid.Row="3" Grid.Column="1" />

  <Button Name="SaveButton" Content="Save" Height="23" Width="75"
    HorizontalAlignment="Right" VerticalAlignment="Top"
    Grid.Row="4" Grid.Column="1" Margin="2">
    <i:Interaction.Triggers>
      <i:EventTrigger EventName="Click">
        <ei:CallMethodAction TargetObject="{Binding Mode=OneWay}"
          MethodName="BeginSaveProduct"/>
      </i:EventTrigger>
    </i:Interaction.Triggers>
  </Button>
</Grid>
</navigation:Page>

```

■ **Note** In a real project, you would also display a busy indicator for when the application is communicating with the server, display any errors, and so on. However, for the purposes of this example, we will keep the XAML as simple as possible.

As you can see, we have a very elegant and testable result from using the MVVM design pattern, without actually requiring any code-behind.

Frameworks

Although the core principles of the MVVM design pattern are reasonably straightforward (once you get your head around the concepts), in practice, maintaining a clean separation between the View and its corresponding ViewModel can be a difficult task at times. However, a number of frameworks have developed solutions for the most common hurdles that you will face (displaying dialogs, navigation, View and ViewModel composition, etc.) and are well worth looking into. Here are some of the most popular frameworks that provide support for the MVVM design pattern (and often a host of additional other features) that you may wish to consider:

- *MVVM Light Toolkit*: <http://mvvmlight.codeplex.com>
- *Prism*: <http://compositewpf.codeplex.com>
- *Caliburn*: <http://caliburn.codeplex.com>
- *Silverlight.FX*: <http://projects.nikhilk.net/SilverlightFX>
- *nRoute*: <http://nroute.codeplex.com>

Summary

The MVVM design pattern is one that all Silverlight developers should learn and understand, even if they don't use it in their projects. There are competing design patterns that you may wish to look into (such as Model-View-Presenter, also known as MVP), but MVVM does lend itself particularly well to applications developed in Silverlight.

However, this a pattern is still in a state of flux, with no fixed, agreed upon, or uniform standard. Therefore, don't become too concerned with whether you are doing it the right way—*there isn't one*. Rather than jumping in at the deep end, try to ease into MVVM instead, preferably on a small project to start with.

You will get overwhelmed if you try to read everything written about the pattern and understand it completely before actually putting the pattern into practice. You can easily fall into analysis paralysis by doing so; don't overthink it. The best approach is to take what's written here, start implementing the core concepts in a small test project, and expand on them as the project grows. It's often best to do this before jumping headfirst into using a complex framework.

Don't be too pedantic and go overboard in trying to follow the pattern in its purest form, because often, this is an unachievable goal, and attempts to do so will potentially result in messier and more complicated code than you would have when not following the pattern.

Do what's right for your project at the time, and don't get caught up in the zero code-behind misconceptions. If a task takes more work than it should, requires messy workarounds, or requires you to fight the pattern to implement simple application requirements, you're taking the wrong approach. You will hit walls along the way, but do persist—it's worth the effort.

CHAPTER 13



Printing and Reporting

Prior to Silverlight 4, much was made of the fact that Silverlight didn't have any native printing abilities, and this was often used as an excuse to back the common belief that Silverlight was not ready for business applications (because printing/reporting is a vital requirement in most business applications). Alternative solutions to work around this omission were implemented that could provide reporting and printing support in Silverlight applications in the meantime, but Silverlight 4 introduced the long-awaited printing functionality developers were after. However, some of those alternative solutions tend to still be the most viable option for implementing reporting.

In this chapter, we will discuss several means of integrating printing capabilities into your Silverlight application (and look at Silverlight's native printing functionality), but we will focus on (arguably the most viable solution of) generating a PDF on the server that's displayed within the application using its native viewer (which provides the printing functionality).

The Importance of Implementing Reporting

Businesses need to be able to store their data, but they also want to obtain information from the system to track and monitor various aspects of the business such as performance, cash flow, forecasts, sales vs. targets, and so on. An ideal means of achieving this is through implementing reporting functionality in the application.

A report generally consists of data retrieved from one or more databases inserted into a predefined template—often with some sort of intermediate processing to turn the data into useful information. Reports can range from the extremely simple to the extremely complex, but both follow the same workflow from an application perspective. At runtime, data is the following:

- Extracted from the database
- Passed through to the report generator/writer, which processes the data and formats the results in a human-readable form suitable for printing

The complexity of processing the data, populating the report, and formatting the results is handled by the reporting engine using the rules defined in the template.

■ **Note** Users won't always want to have to produce reports for information that they regularly monitor but don't print—this functionality is probably better implemented within an application dashboard. Therefore, reports are most appropriately used when printing is required (which it is in most line-of-business applications). This means that whatever your solution for displaying reports in Silverlight, it must also include the ability to print them.

Examples of simple reports include invoices and data printouts, which require minimal processing and generally just insert data from a database directly into the report. More complex reporting may involve mining the data in the database to determine patterns or relationships (correlations) between various data points or to create predictive models upon which management can base their ongoing tactics and help determine their future strategies. Some reports may take some time to produce when a lot of processing is required, and if generating reports places a high load on the web server, then this process can be offloaded to a separate dedicated server (such as SQL Server Reporting Services). So in summary, reporting helps enable business intelligence and knowledge management from within your line-of-business application.

Potential Solutions for Implementing Reporting

There are a number of ways to implement reporting (and consequently printing) in Silverlight applications, each with its pros and cons. Let's take a look at each of these possibilities.

Generating a PDF

As a general rule, you are most likely to be used to generating a report on other projects using a reporting tool, such as the local report designer in Visual Studio or Crystal Reports, and then displaying the report in the provided viewers. Unfortunately, neither of these tools has a viewer for Silverlight, but an alternative solution is to output the reports they generate to PDF, stream that through to the client, and display the PDF:

- In a new window
- In an IFrame overlaid on the application (when running inside the browser)
- In a WebBrowser control to display the PDF (when running outside the browser)

When the PDF is displayed in the window, it will be displayed in the Adobe Reader plug-in. This provides the ability to print the PDF, enabling you to both display and print the report in the application (provided that the user has the Adobe Reader plug-in installed).

■ **Note** This was the preferred solution when printing was not available in Silverlight, and it tends to still provide the best solution for printing complex reports.

Generating HTML

You can use the same techniques for displaying plain HTML in a Silverlight application as described for displaying a PDF, which can then be printed by the browser. If you have your reports generating as HTML, this provides another potential solution for printing reports.

Generating Office Documents

Once again, you can use the same techniques for displaying Word and Excel documents in a Silverlight application as described for displaying a PDF and HTML. These will be loaded in their corresponding viewers within the IFrame/WebBrowser control (if the user has those applications installed), which they can then print. If you have your reports generating as Word or Excel documents, this provides another potential solution for printing reports.

■ **Note** One of the advantages of generating Word and Excel documents (rather than PDFs) is that it gives the user the opportunity to modify the generated report before printing it. Most users tend to have Word installed, and when using the Open XML standard (DOCX) format introduced in Microsoft Word 2007 (with the Office Compatibility Pack downloadable from Microsoft to open these files in earlier versions of Word), it can be easy to populate a Word document template with data on the server (without requiring messy automation). In some cases, it may be better to move to a reporting engine that can generate DOC or DOCX as their output (such as scaling up to SQL Server Reporting Services).

Using Silverlight 4 Printing Functionality

Silverlight 4 finally provided the most requested feature for Silverlight—printing. This allows you to print an object hierarchy (such as what’s displayed on a screen). That said, it’s a somewhat primitive method of implementing printing, requiring you to manually handle the entire printing process (including page breaks, and so on). It also sends the output to the printer as a bitmap (rather than using a vector-based output), resulting in low print quality and large files being sent to the printer. Both of these issues will also be reflected if you print to a PDF. Therefore, the printing functionality in Silverlight 4 is (arguably) best suited to printing screens (instead of reports).

Third-Party Report Viewers

A number of third-party reporting solutions have viewers for Silverlight. Let’s take a look at some of them.

Perpetuum Software’s Report Sharp-Shooter for Silverlight

Perpetuum Software (www.perpetuumsoft.com) has a Silverlight viewer control named Silverlight Viewer for Reporting Services, which displays reports generated by SQL Server Reporting Services directly

within Silverlight and enables you to print them. If you use SQL Server Reporting Services to generate reports, then this may be a viable option (Perpetuum also has its own report designer/engine with a Silverlight Viewer control called Report Sharp-Shooter for Silverlight that you may want to use).

At the time of writing, a single license for Perpetuum's Silverlight Viewer for Reporting Services is \$500.

Telerik's Telerik Reporting

Telerik (www.telerik.com) has a Silverlight report viewer for its Telerik Reporting, which enables you to view reports generated by its reporting engine in your Silverlight application. At the time of writing, however, these reports could not be printed without exporting them to PDF first.

At the time of writing, a single license for Telerik Reporting starts at \$399.

DevExpress's XtraReports Suite

DevExpress (www.devexpress.com) has a Silverlight report viewer for their XtraReports Suite, which enables you to view and print reports generated by its reporting engine in your Silverlight application.

At the time of writing, a single license for DevExpress's XtraReports Suite is \$499.99 with full source code and \$349.99 without full source code.

First Floor Software's Document Toolkit + Aspose Converter

First Floor Software (www.firstfloorsoftware.com) has a Document Toolkit that enables XML Paper Specification (XPS) documents to be displayed in a Silverlight application. XPS is a XAML-based document format from Microsoft, which makes it perfect for displaying documents such as reports within a Silverlight application. Therefore, if you can output your reports to XPS, you can display them in your application using the Document Toolkit.

To create an XPS file, you can simply print a document to the Microsoft XPS Document Writer. XPS documents are zip files, and you can inspect their contents by changing their extension to `.zip` and expanding them. You'll note that the core document contents are stored as XAML, so you could potentially harness this to generate reports on the server using a template XPS document, populating it on the fly when a user requests a report, and serving it up to the Silverlight client for display. This is a very crude solution, but it would work. For example, you could parse these files (such as using regular expressions to replace predefined tokens in the document with actual values), zip up the document (with an `.xps` extension), and serve this to the Silverlight client for display.

A somewhat better solution is to automatically generate the XPS document, which you can do using Aspose.Words for .NET (another commercial product, available here: www.aspose.com). You can find an example of implementing this in the Document Toolkit Extensions project on CodePlex: <http://documenttoolkit.codeplex.com>.

■ **Note** Some sample code is available in a number of places on the Internet that demonstrates displaying XPS documents in Silverlight (with source code) but is incomplete and done purely as proof of concepts. You may want to look at these too, but the Document Toolkit is currently the only full-fledged XPS viewer available for Silverlight without creating your own. At the time of this writing, there is some attempts at enabling it to display PDFs in addition to XPS documents.

At the time of this writing, a single license for First Floor Software's Document Toolkit is €299. A single license for Aspose.Words for .NET starts at €899.

Office Integration Using COM

In Chapter 14, we will discuss running the application outside the browser with elevated trust. This scenario gives you access to COM, through which you can interact with the various Office applications installed on the user's machine (Word, Excel, and so on).

You can use this technique to enable your Silverlight application to generate a report directly in Word or Excel, which the user can then print.

Choosing a Reporting Strategy

We've looked at the potential solutions for implementing reporting in Silverlight, and it will depend on the nature of your project as to which is the most appropriate. In the remainder of this chapter, we will look at the two most common scenarios:

- Using Silverlight's printing features to print a screen (which it is most suited for)
- Generating a PDF report on the server using a reporting tool/engine and displaying it on the client (where it can be printed using the Adobe Reader plugin's printing capabilities)

Printing Functionality in Silverlight

As mentioned earlier in this chapter, Silverlight 4's printing functionality is best suited to printing screens as opposed to reports. Therefore, if the user has a list of items or a data entry form screen they want to print, using Silverlight 4's built-in printing functionality is perfectly suited for this purpose.

Silverlight's printing functionality revolves around the `PrintDocument` class. This class provides a very primitive means of sending content to the printer, but it's quite easy to use for simple tasks. It works using an event-based model, where you do any work required in response to its events. This class has three events to which you can respond:

- **BeginPrint:** You can take this opportunity to initialize the content you want to print.
- **PrintPage:** You use this event to send content for a page to the printer. This event is fired for each page that is printed.
- **EndPrint:** The printing is complete or an error has occurred.

The only event you need to handle is the **PrintPage** event, although it's useful to handle the **EndPrint** event so that you can determine whether an error has occurred.

To start a print job, create an instance of the **PrintDocument** class, add event handlers for the events you want to handle on this object, and then call the **Print** method to start printing (to which you pass a name for the document). Note that at this stage you haven't specified what content is to be printed; you actually do this in response to the **PrintPage** event.

```
PrintDocument pd = new PrintDocument();
pd.BeginPrint += Document_BeginPrint;
pd.PrintPage += Document_PrintPage;
pd.EndPrint += Document_EndPrint;
pd.Print("Silverlight 4 Printing Demo");
```

When you call the **Print** method, the standard Windows print dialog will be displayed. Assuming the user goes ahead with the printing, the **BeginPrint** event of the **PrintDocument** object will be raised, after which the **PrintPage** event will be raised. This is where you can now send some content to the printer.

The event handler method is passed a **PrintPageEventArgs** object, which has four properties:

- **HasMorePages**
- **PageMargins**
- **PageVisual**
- **PrintableArea**

```
private void Document_PrintPage(object sender, PrintPageEventArgs e)
{
}
```

You assign an object hierarchy to the **PageVisual** property, representing the content to send to the printer for that page. This can be a control from your view (or your view itself), enabling you to very easily print parts of the screen. For example, if you have a **DataForm** control named **ProductDataForm** that you want to print, your event handler for the **PrintPage** event will be as follows:

```
private void Document_PrintPage(object sender, PrintPageEventArgs e)
{
    e.PageVisual = ProductDataForm;
    e.HasMorePages = false;
}
```

Note that you are setting the **HasMorePages** property of the **PrintPageEventArgs** object to false. To print additional pages, you need to set this property to true. The event will be raised again, and you can

send the next page's content to the printer (and so on). When you have reached the last page to be printed, set this property to false. The `EndPrint` event will then be raised, and the print job will complete.

For simple printing tasks such as printing a screen, the `PrintDocument` class provides a nice, easy means of doing so. However, when it comes to printing reports and other more complex printing jobs, it can become quite unwieldy. Therefore, the next solution (generating and displaying PDF reports) may be better suited to those needs.

■ **Note** Pete Brown and David Poll have a simple report writer project on CodePlex that you may want to look into if you want to generate reports using Silverlight's built-in print functionality. This project includes handling page headers and footers, page numbering, pagination, and the report footer, among other features. You can get more information and download it from <http://silverlightreporting.codeplex.com>.

Generating and Displaying a PDF Report

Let's now look at how you can use a common reporting tool to generate the reports as PDFs and display them within the application. We will use a reporting tool to generate the PDF and then display the report in the application (by overlaying an `IFrame` on the application when running inside the browser and by using the `WebBrowser` control when running outside the browser).

Generating a Report on the Server

The first step is to actually generate the PDF reports on the server and make them available to the Silverlight client. Let's look at how you can go about implementing this.

Choosing a Reporting Tool/Engine

When implementing reports, you could take the path of manually creating the reports yourself (a time-consuming and mostly unnecessary meta-problem that diverts focus away from actually solving the business problem at hand), or you can use one of the many reporting tools available on the market to manage the reporting component of your system.

There are three major components to a reporting tool:

- The report designer
- The reporting engine
- The report viewer

The report designer enables you to create the structure of the report, its formatting, and the rules (filter/sorting/grouping logic, database queries, calculations, and so on)—effectively it's used to create a report template. The reporting engine then uses these templates to create and populate the reports when they are requested. When reports are served up to the client, the client needs some means of displaying the report to the user. This is where the report viewer component comes in; the reporting engine may generate the reports in a proprietary format in which a custom viewer is required, or it may

generate the report in an open or common format (such as PDF, DOC, XLS, and so on) in which that format may have its own viewer.

When choosing a reporting tool, you need to ensure all three components are compatible with the technologies used in your system. Since the report designer is used only to create the templates, it's not really dependent on the system; however, since each report designer works in different way, you will need to ensure it has the ability to cater for the complexity of your reports. You need to have programmatic access to the reporting engine so you can ask it to generate a report (using the parameters and data you pass to it), capture this output, and stream it back to the client (unless the engine can expose the reports on the Web that your application can directly access). Finally, the reporting tool needs to contain a report viewer control that can display the report in your Silverlight application.

There are reporting solutions for Windows Forms and ASP.NET provided with each copy of Visual Studio (Crystal Reports for Visual Studio in the Professional or higher editions, and the Visual Studio Report Designer aka Report Viewer in the Standard or higher editions and in the Visual Web Developer 2010 Express Edition after installing an add-in). While the report designer and reporting engine components of each are suitable for our system, neither of these generates reports in a format that can be viewed or printed within a Silverlight application, and they don't have a Silverlight-enabled report viewer control. This is a problem since you need to display the report within your application to the user and then give them the ability to send it to their printer. However, the reports can be generated as PDF files, which using the method described shortly can be displayed in your Silverlight application and printed.

In most cases, you are by far best off using a commercial reporting tool; otherwise, reports become difficult to maintain and modify, and maintaining the reporting tool tends to become a never-ending project on its own. Some reporting tools (such as Telerik Reporting and DevExpress's XtraReports) come with source code, so there is little reason to create your own tool for fear of vendor lock-in.

■ **Note** You may have to try various reporting tools to find one with the flexibility you require (reporting tools are often a source of frustration for developers), but ultimately in the vast majority of cases you are better off using an off-the-shelf reporting tool. Many a project developer has created a black hole in the cost and time budget because of attempting to write their own reporting tool and has suffered greatly for this decision. There are many reporting tool packages out there, and although none of them is perfect, one will tend to satisfy most client requirements at a reasonable cost.

We will focus on generating the reports using the Visual Studio Report Designer along with the Local Report Engine here, because it's a reasonably capable reporting tool (it uses the same template format as used by SQL Server Reporting Services), and it's a reporting tool that all Silverlight developers will have available, since it's provided with Standard and higher editions of Visual Studio. Many third-party reporting tools are available—most of which can export reports to PDF and be used in the same manner described in this chapter. If you have a preferred reporting tool, then it should be fairly easy to substitute the one used here with your own.

■ **Note** Visual Studio 2010 Standard edition and higher have the report designer built in, but this is omitted from the Visual Web Developer 2010 Express edition. At the time of writing, Microsoft has not released an add-in to provide the Visual Web Developer 2010 Express edition with a report builder (as it did for the 2008 version), so this will be an issue if you are using this for developing your Silverlight applications. You could always download the Visual Web Developer 2008 Express edition and download the add-in from www.microsoft.com/downloads/details.aspx?FamilyID=b67b9445-c206-4ff7-8716-a8129370fa1d to design your reports. Alternatively, you can try using the open source *fyiReporting* designer, which you can get from www.fyireporting.com.

Unfortunately, the reporting tool we are using doesn't have a particular name that we can use to refer to it as a whole. It's particularly hard to find help on it on the Web because of this (www.gotreportviewer.com is probably the best place to start if you want more information). It's often referred to as Report Viewer, which apart from not being particularly unique is also a bit confusing since the tool is not just used for viewing reports. You may want to try searching for RDLC (the file extension of the report templates). Throughout this chapter we will refer to the report designer as the Visual Studio Report Designer, the reporting engine as the Local Report Engine, and the report viewer control as the Report Viewer.

Choosing an Output Format

The Local Report Engine can render to a number of different formats, including the following:

- HTML (when hosted within the ASP.NET Report Viewer control)
- A Word document
- An Excel document
- An image
- A PDF document

Each of these can be displayed within a browser (HTML and images natively, Excel and PDF documents by browser plug-ins).

Only the image output can be displayed directly within a Silverlight application, but this is not a particularly suitable option because of the large file size of the images and because the quality of the printouts would be low when attempting to print them (which couldn't be done directly within Silverlight anyway). Therefore, we need to look at other means of rendering the report outside the Silverlight application, using the capabilities of the browser instead.

Of the four remaining options, we can exclude rendering to Excel because it would only generally be appropriate to do so when generating matrix- or table-based reports.

Although we can now (using the 2008 or higher version) print directly from ASP.NET's Report Viewer control, it does require the installation of an ActiveX control, which will work only in Internet Explorer running on Windows (not Macs) and will not work within Firefox. You want your Silverlight application to run cross-browser and cross-platform, so this could be considered an unnecessary

restriction to put on your application in most scenarios, and printing is a vital feature for implementing reporting.

This leaves PDF or Word documents as the only viable options, and both provide the functionality we require. They both do the following:

- Display on the screen exactly as they appear on the printed page
- Can be printed directly from within the browser where they are being viewed (assuming the user has the Adobe Reader plug-in installed, which is not an unrealistic expectation)
- Can be saved to the user's hard disk (or a network drive) for archiving purposes
- Can be easily e-mailed to colleagues
- Can be published to content management/collaboration/document management systems such as Microsoft SharePoint

For the purpose of this chapter, you will be generating your reports as PDFs.

Serving Reports to the Client

To serve reports to your Silverlight application, you need to create a means to enable the application to request a report from the server, generate the report (using the Local Report Engine and rendered as a PDF), and stream this back to the application for display.

■ **Note** This section follows a pattern created by the author, so don't worry if you haven't heard terms such as *report renderer* before—they are essentially just elements of this pattern.

Solution Overview

To handle report requests on the server, we'll create a *HTTP handler* in our web project. The Silverlight application will make a web request to the HTTP handler (which is a simple HTTP request, via a URL). The HTTP handler will then do the following:

- Ensure the user is authenticated
- Find the corresponding report renderer (discussed shortly) for the report requested
- Let it handle the logic for generating the report

The report renderer will return the report to the HTTP handler, which will then stream it back to the client as a PDF.

You'll also create *report renderers*. Each report will have a corresponding report renderer that handles the following:

- The logic behind generating that report (such as specifying the location of the report template)

- Ensuring the user has permission to generate the report based upon the roles to which they are assigned
- Obtaining the data from the database to populate the report with

■ **Note** You will need to add a reference to `Microsoft.Reporting.WebForms.dll` to your web project to enable reports to be generated using the Local Report Engine.

Creating the HTTP Handler

Let's start by taking a look at the HTTP handler that will serve up the reports. To make creating the handler easier, the sample code accompanying this chapter has a class named `BaseReportHandler` that your HTTP handler can inherit from, providing much of the base functionality required by your handler. Figure 13-1 shows the class diagram for this class.

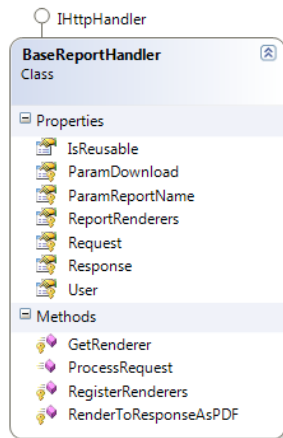


Figure 13-1. The `BaseReportHandler` class diagram

The `BaseReportHandler` class has been designed to permit reusability of the HTTP handler code that handles report requests in a generic fashion. This class handles the following:

- Ensuring the user is authenticated
- Finding the renderer for the report they have requested (soon to be discussed) and checking whether the user is authorized to view the report
- Requesting that the renderer handle the report generation
- Streaming it back to the client as a PDF file

All that needs to be done to create your own HTTP handler is to inherit from `BaseReportHandler` and override the `RegisterRenderers` method. The purpose of this method is to populate the `ReportRenderers`

dictionary in the base class (which defines all the supported renderers), providing each with a unique name that will be used to reference the renderer and the renderer object's type. An example is as follows:

```
protected override void RegisterRenderers(Dictionary<string, Type> reportRenderers)
{
    reportRenderers.Add("ProductDetails", typeof(ProductDetailsReportRenderer));
    reportRenderers.Add("ProductCatalog", typeof(ProductCatalogReportRenderer));
}
```

When a report is requested, the base class will then find the entry matching the `ReportName` query parameter from the HTTP request and instantiate the associated report renderer from the specified type.

■ **Note** Alternatively, you could put each of the renderers in a separate assembly, reference these assemblies in the `web.config` file, and load them dynamically for even more generic code (that you could then add reports to without needing to recompile the code). However, for the purposes of this example, we will keep the implementation as simple as possible.

Creating a Report Renderer

Moving onto report renderers, a report renderer is a pattern (created by the author) used to simplify the generation of reports. Often there are a few processes in generating a report (such as populating the report with data, populating subreports with data, and so on) and properties to set (such as the report path, name, authorization rights, orientation, size, and so on) that are specific to that report. Rather than attempting to create an overly complex generic solution, responsibility is offloaded for generating a report to its associated report renderer class. This class will handle all the creation and population of the report and return it to the HTTP handler as a `LocalReport` object ready for streaming back to the client.

In the sample project accompanying this chapter is a base class to help manage this workflow named `BaseReportRenderer`, whose class diagram is shown in Figure 13-2.

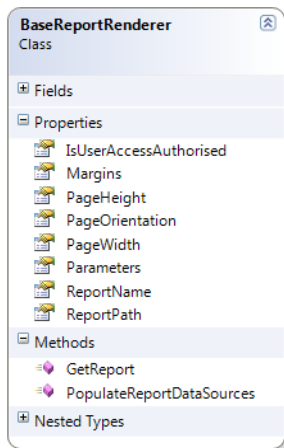


Figure 13-2. The BaseReportRenderer class diagram

Each report should have a renderer class that inherits from **BaseReportRenderer** and overrides the appropriate methods and properties. The following code demonstrates generating a report containing the details of a product:

```
public class ProductDetailsReportRenderer : BaseReportRenderer
{
    public override string ReportPath
    {
        get { return @"Reports\Products\ProductDetailsReport.rdlc"; }
    }

    public override string ReportName
    {
        get { return "Product Details"; }
    }

    public override bool IsUserAccessAuthorised
    {
        get
        {
            // Sample role check - user must be a manager
            return HttpContext.Current.User.IsInRole("Managers");
        }
    }

    protected int ParamProductID
    {
        get
        {
            int paramValue = 0;
            const string paramName = "ProductID";
```

```

        if (Parameters[paramName] != null)
            int.TryParse(Parameters[paramName], out paramValue);

        return paramValue;
    }
}

public override void PopulateReportDataSources(ReportDataSourceCollection
                                              dataSources)
{
    List<ProductDetailsReportData> productDetails =
        ProductDetailsReportData.GetProductDetails (ParamProductID);
    dataSources.Add(new ReportDataSource("ProductDetails", productDetails));
}
}

```

The workflow implemented in the `GetReport` function in the base class is as follows:

- Create a `LocalReport` instance.
- Set the location to the report template.
- Populate the report data sources.
- Return this `LocalReport` instance back to the HTTP handler.

If this is the required workflow, you can just override the `ReportPath` property and the `PopulateReportDataSources` function in the report renderer (the minimum requirements for any report to be rendered). For more complex scenarios, you can override the `GetReport` function to have full control over this workflow.

Returning Data to Populate the Report

The `PopulateReportDataSources` function in the report renderer is where most of the work in the renderer is done. This function needs to obtain the required data for the report from the database and pass it to the `LocalReport` instance. The Local Report Engine can take in either `DataTables` or collections of objects as a data source. Ideally, we'd like to go through our entity framework model to populate the report, and for our sample implementation, we'll populate the report with collections of objects.

■ **Note** For a collection to appear as a project data source in the report builder, a class in the project needs to have a static method that returns a collection (such as a `List<T>` or array) of objects. You'll note that you can't simply select entities from the Entity Framework model when building a report because of this requirement for a static method. In any case, reports often require data combined from multiple entities, so you should create your own data source class to populate the report with, much like the presentation model classes that you pass back to the Silverlight application to populate summary lists.

To enable this, we've created a simple class containing only properties to hold the data and a static `Get` method to run the query on our Entity Framework model that returns a populated collection of itself. This class may be shared between multiple reports if they use the same data source and query, or it may be specific to just one report. The `PopulateReportDataSources` function in the report renderer can then call the static `Get` method and assign the results to the appropriate data source on the report. The following code demonstrates (an abbreviated version of) the `ProductDetailsReportData` class used in the previous example by the report renderer:

```
public class ProductDetailsReportData
{
    public int ProductID { get; set; }
    public string Name { get; set; }
    public string ProductNumber { get; set; }
    // Etc - additional properties removed for brevity

    public static List<ProductDetailsReportData> GetProductDetails(int productID)
    {
        using (AdventureWorksEntities context = new AdventureWorksEntities())
        {
            var reportData = from p in context.Product
                            where p.ProductID == productID
                            select new ProductDetailsReportData()
                            {
                                ProductID = p.ProductID,
                                Name = p.Name,
                                ProductNumber = p.ProductNumber,
                                // Etc - removed for brevity
                            };

            return reportData.ToList();
        }
    }
}
```

We need to consider that the query that returns the data to populate the report may need to take some parameters to include in its `where` clause. These should be the parameters of the static `Get` function to be set by the report renderer, but some of these parameters will not be static values that can be hard-coded into the renderer. Instead, these values may differ between requests (such as a report that prints out the details of a product based upon the product ID of the product selected to print them for). In the base report renderer class, there is a `Parameters` property (a `NameValueCollection`) that the base HTTP handler will assign all the HTTP request query parameters to. The report renderer can then find the required parameters and their values from this to pass through to the static `Get` function of the data class. The previous two code samples demonstrated this.

Securing Access to the Report

To limit the users who are permitted access to a report, you can override the `IsUserAccessAuthorised` property in the report renderer and determine based upon the user's role whether they should be permitted access to the report (as was demonstrated in the `ProductDetailsReportRenderer` code). You may also want to limit what data the user is permitted to view by their user ID or role (such as only

allowing a salesperson to view a list of their own sales). This should be implemented in the `PopulateReportDataSources` function as part of the `where` clause in your LINQ to Entities query.

■ **Note** It's important to implement security on your reports because they may contain the most sensitive business data that you would not want anyone to obtain access to. Another function security can provide is to filter the data in a report to contain only the data that a specific user has the rights to view, while using a common report template (such as permitting a salesperson to view their own sales figures but no one else's).

Deploying the Report Viewer/Engine

When deploying the application to the server, you will need to install the runtime for the report engine (which isn't installed as part of the .NET Framework). If you don't have the Microsoft Report Viewer Redistributable 2010, you can download it from www.microsoft.com/downloads/details.aspx?FamilyID=a941c6b2-64dd-4d03-9ca7-4017a0d164fd. Alternatively, you can simply copy the referenced reporting DLLs to your `Bin` directory on the server.

Displaying the Report on the Client

Unfortunately, you cannot view a PDF document from within a Silverlight application (although the Document Toolkit from First Floor Software did make a partial proof of concept attempt at doing so). However, you can take advantage of the browser's ability to host the Adobe Reader plug-in (which will host your PDF report), which is what we'll be looking at now.

Creating the HtmlViewer Control

You now need to display the PDF that you generated on the server in your Silverlight application. Doing so requires a little bit of work. When you navigate to a PDF file in your web browser, it will be displayed within the browser using the Adobe Reader plug-in (assuming you have it installed). Ideally, we'd like to implement the same sort of behavior in our application, and we'll look at doing that now.

Solution Overview

Silverlight 4 introduced a `WebBrowser` control that you could display the PDF within; however, when you run your application inside the browser, you will get the output shown in Figure 13-3.

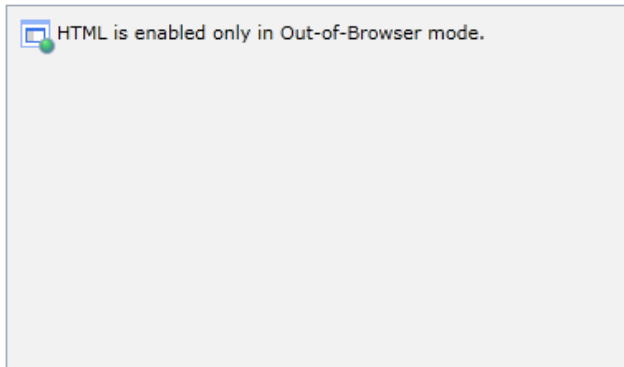


Figure 13-3. The WebBrowser control when used in an application running inside the browser

For security purposes, the WebBrowser control will work only when the application is running outside the browser (as will be discussed in Chapter 14), which is fine if your application is designed to run only outside the browser. You can use this control in your application and navigate to the URL for your report, and it will be displayed within the control. However, if your application also needs to be able to run inside the browser, another solution is required.

Simply opening the report in another browser window is a possible solution but is not ideal because it separates the display of the report from the application.

Ultimately, the best solution (although a little complex to implement) is to manipulate the underlying HTML page (using the HTML Bridge) and overlay an IFrame onto your application. You can then display the PDF in this IFrame in order to achieve the effect you want. This won't work when the application is running outside the browser (because the HTML Bridge is not available in this mode), but you can use the WebBrowser control in this mode instead.

To support displaying an IFrame within the application, you will wrap this functionality into a reusable user control. It will handle creating and destroying the IFrame when running inside the browser and use the WebBrowser control when running outside the browser. This way, wherever support is required in this or other applications for displaying HTML content within the application itself, you can easily drop it in.

The HTML Bridge

Essentially the HTML Bridge is the means to interact with the underlying browser and document object model (DOM) from a Silverlight application. You may also see this referenced as HTML DOM Interoperability. Throughout this chapter you will be using the HTML Bridge to provide various features such as opening a pop-up window, creating elements in the DOM (such as an IFrame), inspecting the DOM (checking properties of the Silverlight plug-in), and handling JavaScript events.

To access the HTML Bridge, add a reference to the `System.Windows.Browser` namespace, and then you can access the DOM of the underlying HTML document that is hosting the plug-in via the static `HtmlPage` class.

■ **Note** The HTML Bridge is not available when the application is running outside the browser, because it's not (technically) being hosted by an HTML page. It is, however, being hosted by a browser control (Internet Explorer on Windows and Safari on Mac), which then hosts your Silverlight application. If you're interested in finding out more, try using Microsoft Spy++ to inspect the window and its structure.

Steps to Overlaying an IFrame on Your Application

Now that you know a bit about the HTML Bridge, let's look at some of the issues that you will face in displaying an IFrame over the top of your Silverlight plug-in.

Setting Windowless Mode

The primary issue that you face is that by default the Silverlight plug-in is in "windowed" mode, where the plug-in handles rendering the Silverlight content in its own window, on top of any HTML content in the underlying HTML document. Therefore, even though you are able to create the IFrame in the underlying HTML document, it won't be visible because the Silverlight plug-in occupies the entire browser area, rendering on top of any HTML content in the DOM of the page (regardless of the z-index of the plug-in or the IFrame). Therefore, you need to ensure windowed mode is turned off so that the browser is in charge of the rendering instead of the Silverlight plug-in, and then your Silverlight content will be rendered alongside the underlying HTML content.

■ **Note** There are performance issues in placing the browser in charge of the rendering, so the decision to use windowless mode needs to be carefully considered. Silverlight is optimized for rendering complex animations and video, and there will be a significant performance loss in these two areas when using windowless mode. Considering that we are building a line-of-business application here, the loss in rendering performance is unlikely to affect us in most cases, so the benefits in this case should outweigh the costs.

To enable windowless mode, you need to open the HTML page hosting the Silverlight plug-in, find the object tag, and add a `param` node to its children that specifies a value for the `Windowless` property—setting its value to `true` as follows:

```
<param name="Windowless" value="True" />
```

Creating the IFrame

The next step is to create the IFrame in the underlying HTML page. The IFrame is contained within a DIV element so that you can set its position properly. For this DIV, you'll set its z-index to 99 (so it displays on top of other controls, assuming their z-index is less than 99) and set its positioning mode to absolute. This means you can set its position by its coordinates in the browser window. You can then append this DIV as a child of the FORM tag in the DOM and append the IFrame element as a child of the DIV tag.

```

divElement = HtmlPage.Document.CreateElement("div");
divElement.Id = "HtmlFrameDiv";
divElement.SetStyleAttribute("position", "absolute");
divElement.SetStyleAttribute("z-index", "99");

iframeElement = HtmlPage.Document.CreateElement("iframe");
iframeElement.Id = "HtmlFrame";
iframeElement.SetAttribute("scrolling", "no");
iframeElement.SetAttribute("frameborder", "0");

HtmlElement formElement =
    HtmlPage.Document.GetElementsByTagName("form")[0] as HtmlElement;

formElement.AppendChild(divElement);
divElement.AppendChild(iframeElement);

```

Positioning and Sizing the IFrame

Now that you have created the required elements in the DOM, you need to position and size them. As previously mentioned, this involves setting the coordinates and size of the DIV, and you will also set the height and width of the IFrame to support multiple browsers (in some browsers the IFrame fills the area of the DIV, and in others it doesn't).

Because the content being displayed is not being rendered by Silverlight, you need to take resizing of the IFrame into account yourself. If the control is resized (such as if the browser window was resized or a grid splitter was implemented that enabled the user to resize or move the location of the control), you need to handle the `LayoutChanged` event of the control and adjust the size and location of the IFrame accordingly.

■ **Note** We're deliberately not handling the `SizeChanged` event for this purpose—the reason being that if the control location was simply moved (rather than resized), the `SizeChanged` event is not raised. Therefore, although the `LayoutChanged` event is raised even when the control hasn't been moved or resized (and can be raised many times for many reasons because it is raised when anything occurs in the visual tree), it's the only event that will handle all situations—even if it isn't ideal (it is best to never handle this event if an alternative is available, because of the numerous times and purposes that it can be raised).

Additional Features and Functionality

The `HtmlViewer` control is already created and available for download as part of the sample code accompanying this chapter. However, let's take a look at some of the more important aspects of how it was built.

Ensuring That the Windowless Property Is Set

For the control to work properly, the developer must have turned on the `Windowless` property for the Silverlight plug-in (as previously discussed). Since we are creating a reusable control, it's worth checking

to make sure the developer has turned on this `Windowless` property and raise an exception if not in order to save confusion as to why the `IFrame` isn't being displayed.

With the HTML Bridge, you can navigate through the DOM of the page and ensure this property is turned on. First you need to get a reference to the plug-in element (which is rendered as an object tag). Rather than searching for the object tag in the page, you can simply get a reference to it using `HtmlPage.Plugin`. Then you can enumerate through its children looking for param tags that have a name attribute of `Windowless`. If the tag isn't found or its value is set to false, then you can throw an exception.

```
HtmlElement silverlightObject = HtmlPage.Plugin;
bool isWindowless = false;

foreach (HtmlElement param in silverlightObject.Children)
{
    if (param.TagName == "param")
    {
        string name = param.GetAttribute("name");
        string value = param.GetAttribute("value");

        if (name == "Windowless")
        {
            isWindowless = Convert.ToBoolean(value);
            break;
        }
    }
}

if (!isWindowless)
    throw new Exception("The Silverlight plugin needs to be in windowless mode!");
```

Positioning and Sizing the IFrame

This involves working out the location of the control with reference to the application root visual element (in other words, the whole area occupied by the Silverlight plug-in) by creating a transform object to transform the coordinates of the two and relating (0, 0) on the control to (0, 0) on the application root visual element. This gives you the coordinates of the control with reference to the application root visual element but not necessarily with reference to the browser window itself, if the Silverlight control is not located at (0, 0) to the browser window. Therefore, for completeness, you should calculate the offset of the Silverlight plug-in in the underlying page (moving up the DOM and adding offsets of each parent element) and add that to your calculations.

Because the Silverlight application is filling the entire browser window, it will be located at (0, 0), making this calculation unnecessary, so you will make this an assumption and forgo these additional calculations. Therefore, you now can use these coordinates to set the `left` and `top` properties of the `DIV`, and the `width` and `height` properties are simply the `width` and `height` of the control.

```
if (iframeElement != null && this.ActualHeight != 0)
{
    GeneralTransform gt = this.TransformToVisual(Application.Current.RootVisual);
    Point pos = gt.Transform(new Point(0, 0));
    divElement.SetStyleAttribute("left", pos.X.ToString() + "px");
    divElement.SetStyleAttribute("top", pos.Y.ToString() + "px");
    divElement.SetStyleAttribute("width", this.ActualWidth.ToString() + "px");
```



```

divElement.SetStyleAttribute("height", this.ActualHeight.ToString() + "px");

iframeElement.SetStyleAttribute("width", this.ActualWidth.ToString() + "px");
iframeElement.SetStyleAttribute("height", this.ActualHeight.ToString() + "px");
}

```

Determining When Content Is Loaded

A useful feature would be to monitor when the content has loaded (handling the JavaScript `onLoad` event raised by the `IFrame` via the HTML Bridge), in turn raising an event on this control to notify the application as such. This could potentially be useful in showing an animation and/or indicating that the application is communicating with the server.

Capturing this DOM event in Silverlight is actually quite easy—when creating the `IFrame`, you can use the `AttachEvent` method on the HTML element, specifying the JavaScript event to capture and the `.NET` method to handle the event. However, you strike a problem in loading a PDF in the `IFrame`, because for some reason the `onLoad` event is never raised (presumably it is because a separate plug-in takes over loading the page), although it is raised correctly for HTML pages.

Therefore, because the reports will be PDF files, you won't be able to use this feature in your application to indicate that server communication is in progress (though the functionality has still been included in this control for reuse purposes when displaying HTML pages). It is preferable to show something to the user rather than a blank area while the report loads, such as a "Please wait—loading..." message in the control area. If you set some HTML locally to display immediately in the `IFrame` before navigating to the report URL on the server, then this will continue to display until the report has been retrieved from the server (and the Adobe Reader plug-in had loaded itself and the report). Figure 13-4 demonstrates a simple example of doing so.

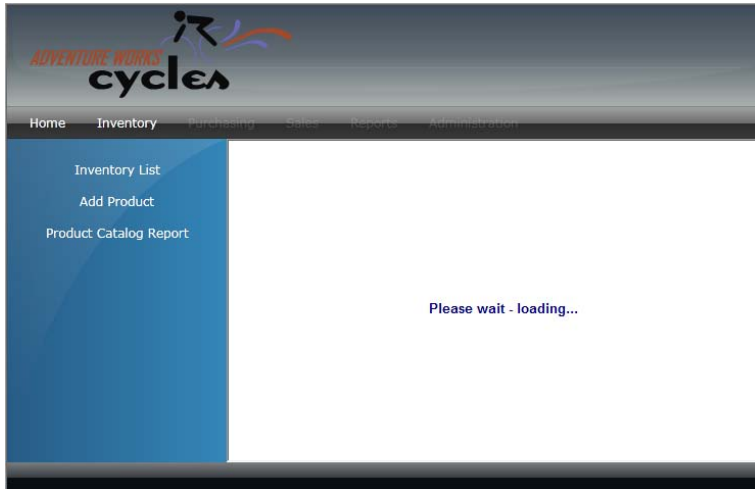


Figure 13-4. The "Please wait" message

If you attempt to build up the page to display using the DOM object available via the HTML Bridge, you will find that it is displayed differently in IE compared to how it is in Firefox (it will be laid out differently). IE will display the content in quirks mode, and there is no way to set the DOCTYPE for the loading page via the DOM. Therefore, you need to use an alternative means of creating and loading this

page locally. Another method is to get a reference to the empty document created in the `IFrame` automatically when it is created and use it instead to set the content to display. You can then open it for direct HTML content writing, write the entire HTML document to be displayed (including the DOCTYPE, html, head, and body tags), and then close it. This document would then be displayed in the `IFrame`. The open, write, and close functions are not available via the HTML Bridge but can be called using the `Invoke` method on the document object.

```
// Add some HTML to the iframe before requesting the document
HtmlWindow contentWindow = iframeElement.GetProperty("contentWindow") as HtmlWindow;
HtmlDocument document = contentWindow.GetProperty("document") as HtmlDocument;

// Write the html to display to the document
document.Invoke("open", null);
document.Invoke("write", "<HTML to display goes here>");
document.Invoke("close", null);
```

■ **Note** There appears to be a bug in Internet Explorer for which there doesn't seem to be a workaround. Even after the report has loaded, the progress bar in the status bar continues to show. This appears to be because of a combination of writing directly to the document in the `IFrame` and then navigating to a different page—if you comment one or the other out, the progress bar is hidden correctly, but using both together leaves it visible. This problem does not appear to occur in Firefox.

Destroying the IFrame

The final issue you need to find a solution for is how to know when to remove the `IFrame` from the underlying DOM. Silverlight user controls don't have a `Closing` or `Closed` event, the `Dispose` event when implementing `IDisposable` isn't automatically called, and the destructor/finalize being called is unreliable because it relies on the idiosyncrasies of the garbage collector as to when the control is cleaned up.

You'll still implement the `IDisposable` interface and use the `Dispose` method to remove the `IFrame` from the DOM, but you need to know when to call this method. When it's created, the control will walk its way up the visual tree and see whether it can find a `Page` object that is hosting it. If so, then the application is using the navigation framework, and the `Page` object exposes a `NavigationService` object whose `Navigating` event you can hook into to determine when the page is being navigated away from. When this event is raised, you can call the `Dispose` method to remove the `IFrame`.

If the application isn't using the navigation framework, then the `HtmlViewer` control won't know when the `IFrame` should be removed, and the application hosting it will need to call its `Dispose` method at the appropriate time.

Using the WebBrowser Control When Running Outside the Browser

As previously mentioned, the HTML Bridge isn't available when running outside the browser, so you can't create an `IFrame` in this scenario. However, Silverlight 4 introduced the `WebBrowser` control that you can use instead in out-of-browser scenarios. The `HtmlViewer` control determines whether the application is running inside or outside of the browser and will create an `IFrame` or a `WebBrowser` control accordingly, providing a seamless solution to this problem.

Using the HtmlViewer Control

Now that you have the HtmlViewer control, displaying reports is easy. The first task is to ensure that your Silverlight application object is set to run in windowless mode (using the method described in the previous section of adding the following parameter to the HTML page hosting the Silverlight plug-in); otherwise, when you navigate to the view hosting, the control it will raise an exception.

```
<param name="Windowless" value="True" />
```

Then it's as easy as opening the XAML file for the view that will host it, declaring the namespace prefix mapped to the control namespace, and then declaring the control usage like so:

```
<layout:HtmlViewer Name="htmlViewer" Height="Auto" Width="Auto"  
    Url="http://localhost/Report.ashx?ReportName=ProductDetails&ProductID=1" />
```

replacing the `Url` property with the URL to the HTTP handler on the server. For the sample application, you will be binding this `Url` to a view model, or setting it in the code-behind, which will point to the HTTP handler (`Report.ashx`) that you created earlier to serve up the reports, although you could specify any link with content that can be displayed within a browser.

For the link, you need to point back to the `Report.ashx` HTTP handler on the same server that's hosting the Silverlight application, passing it some parameters (such as the report name, filter criteria, and so on). This procedure requires actually finding out where the Silverlight application was downloaded from (`Application.Current.Host.Source`) and building the URL accordingly. To help with this, you will find a class called `ReportRequest` in the sample code accompanying this chapter that contains a helper function for building up this URL when requesting reports, which also takes in key/value pairs as parameters and includes them as part of the URL's query string.

■ **Note** The `ReportRequest` also contains another helper function for requesting reports. Instead of using the `HtmlViewer` control to display the report “within” the application, you might want to use the `OpenReportInNewWindow` method, which will open the report in a new browser window. This is a simple process that uses the HTML Bridge—the `HtmlPage.PopupWindow` function will pop up a new window with the specified options (such as the size of the window, whether the menu bar or scroll bars should be shown, and so on) and will navigate to the specified URL.

Figure 13-5 demonstrates the final result of displaying a “Product Catalog” report in the `HtmlViewer` control. Note that this report is part of the SQL Server Reporting Services examples on CodePlex (www.codeplex.com/MSFTRSPProdSamples).

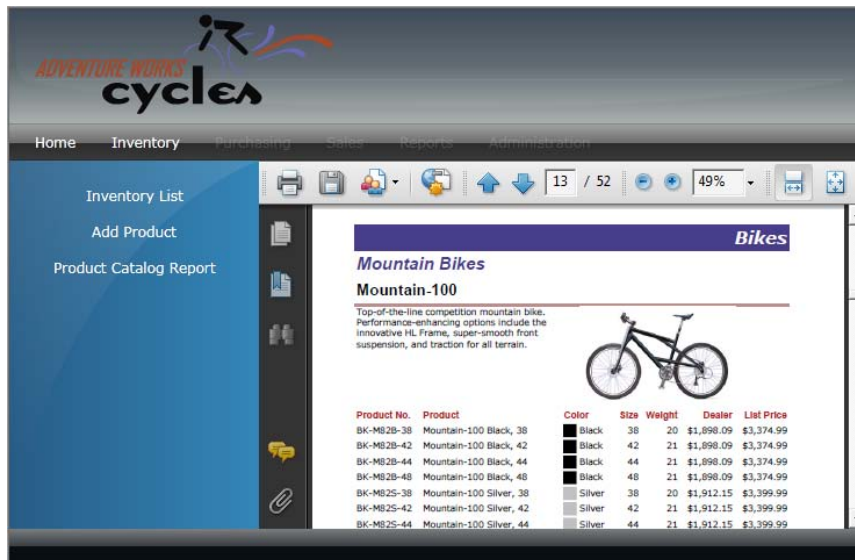


Figure 13-5. The “Product Catalog” report

■ **Note** An issue worth mentioning (because it can create confusion and frustration) is that the Adobe Acrobat Reader plug-in sometimes starts to fail to load, leaving you with just a white screen where the report should be. You can fix this by opening Task Manager and ending the `AcroRd32.exe` process. Then attempt to view the report again, and the plug-in should load correctly.

Summary

You now have implemented a means for printing screens using Silverlight’s printing functionality and displaying PDF reports (generated on the server) “within” the application. These both provide viable solutions for this important business application task without requiring a lot of code to be written or resorting to third-party solutions.

CHAPTER 14



Out of Browser Mode, and Interacting with the Operating System

Although Silverlight started as a sandboxed platform that only ran within a web browser window, it's grown to become a mature platform for building applications— that inevitably required a certain level of integration and interaction with the host operating system.

With the introduction of Silverlight 3, you gained the ability to run Silverlight applications outside the browser in its own window, launching it from an icon on the desktop or the start menu, and using it in much the same way as a standard (native) application. Now with Silverlight 4 you can also request elevated trust permissions to gain direct access to the file system and COM. This permits your applications to have a closer relationship with the host operating system, enabling a whole range of new capabilities and possibilities for your applications. You can take advantage of these new features to build more powerful and functional applications in Silverlight.

Let's take a look at how you can configure your application to run outside the browser, and interact with the host operating system much like standard (native) applications do.

Out of Browser

Initially, Silverlight was designed to run inside a browser window, either as a “widget” on a web page, or filling the whole browser window as an application. However, Silverlight 3 introduced a major new feature that enabled Silverlight applications to be “installed” locally on the user's machine. Installing the application involves placing the .xap file in a permanent location within the user's profile (ie. outside of the user's browser cache), and creating shortcuts on the user's desktop and/or start menu to enable them to run it. When the user clicks/double-clicks on the shortcut, the application will open in its own window as if it were a standard/native application. This is known as Out Of Browser (OOB) mode, and we'll be referring to it throughout this chapter as such.

As the application is being “installed” within the user's profile, there is no need for the user who wants to install it to have administrative privileges, which simplifies the deployment of your application. Additional features (such as “toast” notifications) are enabled by running the application in OOB mode, and when the application is running in OOB mode you can request elevated trust permissions—opening up an even wider range of possibilities and features (discussed later in this chapter).

One of the biggest benefits however is that this feature works cross platform—specifically working on both Windows and OSX, making Silverlight now a great way of building cross-platform applications.

Configuring OOB Mode

By default, your application is not configured for it to be installable and run in OOB mode. You need to explicitly turn this feature on, via the project's properties. Start by selecting the “Enable running application out of the browser” check box (as demonstrated in Figure 14-1).

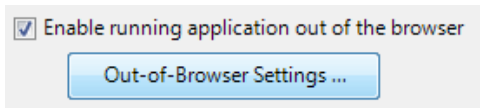


Figure 14-1. The “Enable running application out of the browser” option

You will now be able to click on the Out-of-Browser button below the check box, which will open up the window shown in Figure 14-2.

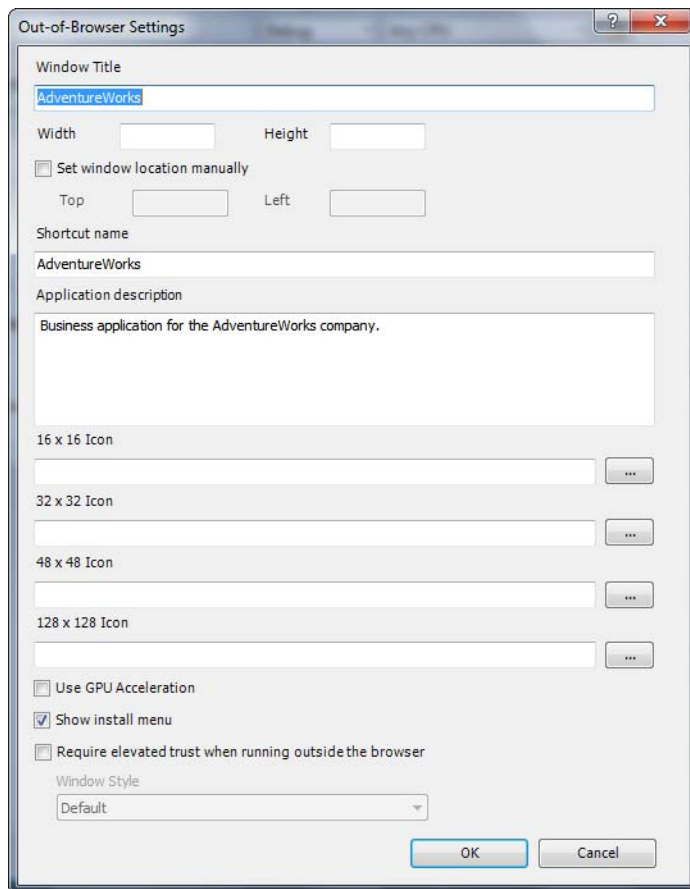


Figure 14-2. The Out-Of-Browser Settings window

This dialog provides you with a variety of options that apply when the application is running in OOB mode. Despite being a mass of options, they can be broken up into three primary areas—window options, install options, and other miscellaneous options.

Window Options

The window options enable you to specify the text that should appear in the title bar of the OOB window, and the window's initial width and height (it will default to 800×600 without these options set, and will be resizable unless you turn off the default window chrome by using custom chrome). By default, the application will center itself in the screen, but by selecting the “Set window location manually” check box you can specify exactly where on the screen you want your application to appear.

Install Options

The install options enable you to specify the name of the shortcut that will be created on the user's desktop and/or start menu, a description of the application (that will appear in the tooltip on the application's shortcuts), and a number of icons of varying sizes. These icons will be used for the application's shortcuts, and will also be displayed in the install dialog, the title bar of the window, and as the taskbar/dockbar icon. If no image files are assigned to these settings, default Silverlight icons will be used instead.

■ **Note** These icons should be assigned image files, not .ico files (as .ico is an image format specific to Windows). The Silverlight runtime will automatically create icons native to the host operating system from the image files as required.

Miscellaneous Options

The various other miscellaneous options include enabling GPU acceleration for cached compositions, whether an install option should appear in the context menu when the user right-clicks on the application, whether the application requires elevated trust when running outside the browser, and whether the application should use custom chrome (via the “Window style” option).

■ **Note** You can read the options that were set in the Out-of-Browser Settings dialog for the application at runtime from the `Deployment.Current.OutOfBrowserSettings` object. However, you cannot modify them at runtime.

Installing the Application to Run Out of Browser

There are two ways that the user can install your application. One is by simply right-clicking anywhere in the application, and selecting the “Install XXX application on this computer...” option from the context

menu (assuming this menu is enabled in the Out-of-Browser Settings dialog, which it is by default when the application is configured to be able to be installed and run outside the browser), as shown in Figure 14-3.

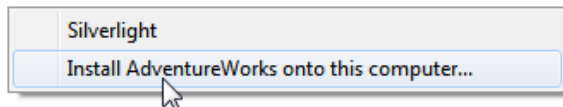


Figure 14-3. The Silverlight context menu item to install the application locally

■ **Note** This context menu will not be displayed for areas in the Silverlight application that handles the `MouseRightButtonDown` event (by setting the `e.Handled` property to true in the event handler), and implements a custom behavior for this action. It will also not be shown on elements that use the `ContextMenuService` (from the Silverlight Toolkit) to display a custom context menu.

Alternatively, you can provide your own friendlier install option within your application. Installing the application is as easy as using the following line of code somewhere in your application:

```
Application.Current.Install();
```

■ **Note** You will need to add a using directive to the `System.Windows` namespace at the top of your file (as the `Application` class exists under that namespace).

As soon as the application is installed, it will automatically open and start running outside the browser.

■ **Note** If the application is already installed and you attempt to call this method, an exception will be raised. Therefore you should always check whether the application is installed first before calling this method. How to do this will be covered in the next section.

One thing you need to note is that for the purposes of security, a call to the `Install` method must be in response to a user initiated event. In other words, you can only call this method in an event raised by some sort of user input (such as the `Click` event of a button for example). Attempting to call the `Install` method in a non-user input initiated event (such as the `Loaded` event of a control), will simply be ignored.

■ **Note** You can also create a desktop installer and install the application that way too. This will be discussed in Chapter 15: Application Deployment.

Once the user has initiated the install action, the install dialog (shown in Figure 14-4) will appear, requiring the user to confirm the installation, and specify what icons they want created.

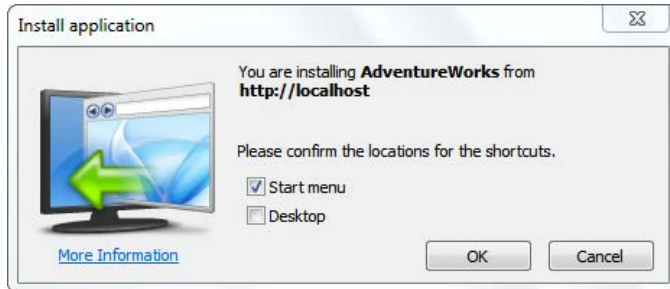


Figure 14-4. The install dialog

■ **Note** If an icon was specified in the Out-of-Browser Settings dialog, it will be displayed in the dialog in place of the generic Silverlight application icon (with the generic icon displayed in Figure 14-4).

A different dialog will be displayed when elevated trust permissions are being requested. This dialog will be discussed later in this chapter.

Determining Installation Status

You can determine whether or not the application is already installed by interrogating the `InstallState` property of the `Application.Current` object. This property returns an enumeration, with the following values:

- `NotInstalled`
- `Installed`
- `Installing`
- `InstallFailed`

For example, the following code can be used to determine whether or not the application is installed:

```
bool isInstalled = (Application.Current.InstallState == InstallState.Installed);
```

You can determine when the installation status of the application changes by handling the `InstallStateChanged` event of the `Application.Current` object. Start by adding an event handler for the event:

```
Application.Current.InstallStateChanged += Application InstallStateChanged;
```

and then implementing the event handler, like so:

```
private void Application InstallStateChanged(object sender, EventArgs e)
{
    // Handle the state change as required here
}
```

Within this event handler you can interrogate the `InstallState` property of the `Application.Current` object in order to determine the new installation status of the application.

Determining Whether Running Inside or Outside the Browser

You can determine whether the application is running inside or outside the browser by interrogating the `IsRunningOutOfBrowser` property of the `Application.Current` object. For example:

```
bool isRunningOutOfBrowser = Application.Current.IsRunningOutOfBrowser;
```

If your application isn't designed to run inside the browser (such as if it requires elevated trust permissions) then you should use this property to detect whether the application is running inside the browser when it starts, and if so, send the user to a view where they can install the application.

■ **Note** Unfortunately there is no way to launch the application in OOB mode from within the browser if it's already installed. Therefore, you will need to ask the user to do so manually if this is necessary.

Where/How is the Application's .xap File Stored and Run?

When a Silverlight application is installed, a folder is created inside the user's profile folders to store it in permanently (outside of the browser's cache). This path will differ between operating systems, but on Windows Vista/7 you can view the installed Silverlight applications by typing the following path into the Windows Explorer address bar: `%userprofile%\AppData\LocalLow\Microsoft\Silverlight\OutOfBrowser`. Each installed application has its own folder under this, containing the `.xap` file for that application, along with a number of associated files—such as a HTML page to host it, an icon, and some metadata files.

The `sllauncher.exe` executable is used to host the Silverlight application when it is running in OOB mode. It is passed the ID of an application to launch, like so:

```
sllauncher.exe appid
```

The application ID will match the name of the folder in the OOB store for that application. For example, the following command line demonstrates an example shortcut path to launch the application:

```
"C:\Program Files\Microsoft Silverlight\sllauncher.exe" 1990526816.localhost
```

Interacting with the OOB Window

When running in OOB mode, you can get a reference to the application's main window like so:

```
Window window = Application.Current.MainWindow;
```

This window object provides you with a number of properties, methods, and events that enable you to interact with and manipulate it.

■ **Note** Many of these properties and methods are needed particularly when you have turned on custom chrome. Custom chrome is discussed in detail later in the chapter.

Closing the Window

Using the reference to the window, you can close it using its Close method:

```
Application.Current.MainWindow.Close();
```

You can detect when the window is being closed, and cancel it by handling its Closing event. Start by adding an event handler to the event:

```
Application.Current.MainWindow.Closing += MainWindow_Closing;
```

And then you can handle it (and cancel the window being closed) like so:

```
private void MainWindow_Closing(object sender, ClosingEventArgs e)
{
    if (e.IsCanceliable)
    {
        MessageBoxResult result = MessageBox.Show("Are you sure you wish to quit?",
                                                    "AdventureWorks", MessageBoxButton.OKCancel);
        e.Cancel = (result != MessageBoxResult.OK);
    }
}
```

Window State, Position and Dimensions

You can use the `WindowState` property of the window to maximize it, minimize it, or return it to normal. It accepts/returns an enumeration of type `WindowState`, which has the following possible values: `Maximized`, `Minimized`, and `Normal`. For example:

```
Application.Current.MainWindow.WindowState = WindowState.Maximized;
```

You can set the size of the window using its `Width` and `Height` properties:

```
Application.Current.MainWindow.Width = 600;  
Application.Current.MainWindow.Height = 800;
```

You can set the position of the window using its `Top` and `Left` properties. However, at runtime you can only set these properties if the “Set window location manually” check box was selected in the Out-Of-Browser Settings window (shown in Figure 14-2), and they must be set before the end of the application’s `Startup` event (after this, any changes to their values will be ignored).

Other

You can set the window to be a `topmost` window (where it always appears on top of other windows regardless of whether or not it has the focus) by setting its `TopMost` property to `true`. For example:

```
Application.Current.MainWindow.TopMost = true;
```

You can determine whether a window is active by interrogating its `IsActive` property, and you can activate it (i.e. bring it to the forefront of the various open windows and give it the focus) using its `Activate` method. For example:

```
Application.Current.MainWindow.Activate();
```

■ **Note** You do not have access to the HTML DOM using the `HtmlPage` object when running outside the browser as you do when running inside the browser.

Checking for Updates

One of the issues with running in OOB mode is that you are no longer requesting the application from the server. Normally, when you browse to a Silverlight application (within a web browser), the browser will check if it has previously downloaded the .xap file (by checking for it in its cache). If it doesn’t find it, then it will automatically download the .xap file from the server. If it does find it, will check with the server as to whether the cached version matches the version on the server (in other words it will check to see if it has the latest version already). If it does, then it will just use the cached version (saving the need to download it again). Otherwise it will download the new version (overwriting the now out of date cached version) before running it.

■ **Note** You can see this behavior using the free Fiddler tool, which you can download from here: www.fiddler2.com.

However, when you are running in OOB mode, the .xap file is simply loaded from its installation location on disk, meaning that it will not be updated as new versions become available on the server. Therefore, a strategy needs to be implemented in order to ensure that the installed application is kept up to date. Luckily, Silverlight has some functionality built-in to help you do that.

You can check whether an update is available using the following line of code:

```
Application.Current.CheckAndDownloadUpdateAsync();
```

With this method call, the application will go back to the URL that it originated from and see whether an update is available. If an update is found, it will be automatically downloaded, and it will be installed just prior to when the application is next run. This action is performed asynchronously, and thus does not have a return value.

You can determine whether or not an update was found by handling the `CheckAndDownloadUpdateCompleted` event. Start by adding an event handler to the event:

```
Application.Current.CheckAndDownloadUpdateCompleted +=  
    Application.CheckAndDownloadUpdateCompleted;
```

And then you can handle the event, like so:

```
private void Application.CheckAndDownloadUpdateCompleted(object sender,  
                                                         CheckAndDownloadUpdateCompletedEventArgs e)  
{  
    if (e.UpdateAvailable)  
    {  
        // Logic can go here  
    }  
}
```

■ **Note** You can check if an error occurred when checking for an update by interrogating the `Error` property of the `CheckAndDownloadUpdateCompletedEventArgs` object passed into the event handler as a parameter.

If an update is found, it won't be installed until the application is next restarted. It's your decision as to whether to force the user to restart the application (perhaps by closing the window, as previously demonstrated), or allow them to continue working with the existing version.

You may choose to automatically perform a check for updates each time the application is started, or alternatively you can simply provide a "feature" that enables the user to check for updates at their leisure.

■ **Note** When running the application in OOB mode *with elevated trust* (as described later in this chapter), the application will only automatically update itself if the application is signed. You won't see this behavior when the source of the update is localhost, as updates are permitted in this scenario for development purposes. However, once you deploy the application to production, it must be signed in order to be able to update itself. Application signing will be covered in Chapter 15: Application Deployment.

Debugging Out of Browser

When you run your Silverlight project in debugging mode in Visual Studio, the application is run within a browser window. However, at times you want to debug your application while it is running in OOB mode. This is especially important when you want to debug features that require elevated trust—something that you can't do when the application is running inside the browser.

You can have the application launch in OOB mode from Visual Studio by:

- Enabling OOB support for your Silverlight application (if it's not already set)
- Setting the Silverlight project as the startup project for the solution (instead of the web project).

The Silverlight application will start in OOB mode with the debugger attached, enabling you to debug the application as required.

■ **Note** Without OOB support configured for the application, simply setting the Silverlight project as the startup project will run it within the browser instead (using a file path). If you find that this is the case, and that the application is still running within the browser, check to ensure that you've configured to application to be able to be run in OOB mode.

You can also attach the Visual Studio debugger to an already running instance of the application running in OOB mode, and debug it in that manner. In Visual Studio, start by going to Tools ► Attach To Process. In the dialog window that appears, scroll down the list of available processes until you find the `sllauncher.exe` process (this is the Silverlight launcher process discussed earlier in this chapter). If there is more than one listed, look for the one with the title that matches that of the Silverlight application that you want to debug. When you click the Attach button, the Visual Studio debugger will attach to the application, and enable you to debug it as required.

Uninstalling the Application

When you no longer want a Silverlight application installed on your machine, you can remove it in one of two ways. The first is by simply right-clicking anywhere within the application's window, and select "Remove this application" from the context menu. The other method is to use the operating system's

uninstall application tool (such as Windows Vista/7's Programs and Features tool). The application will appear as an entry in that tool, and you will be able to uninstall it as you would any other application.

■ **Note** There is no ability to uninstall the application through code.

Toast Notifications

A new feature that came to Silverlight 4 is the ability to display “toast” notifications. These are the little windows that pop up for a couple of seconds (usually in the bottom right-hand corner of your screen) to notify you of something that has happened in the application.

For example, Microsoft Outlook displays a toast notification when a new email is received, and TweetDeck (a popular Twitter client) displays a toast notification when a new tweet appears in your Twitter feed.

With this feature in Silverlight 4, you can implement the same sort of notification in your Silverlight applications. For example, you might like to display a notification when an order is placed in the system, or when a product is getting low in stock. An example of a simple notification window is shown in Figure 14-5.

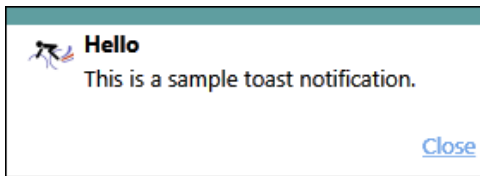


Figure 14-5. A simple toast notification window

■ **Note** This feature only works when the application is running in OOB mode.

To display a toast notification, you create an instance of the `NotificationWindow` class, and assign it some content. Unfortunately, you cannot create a XAML file that inherits from the `NotificationWindow` class as it is sealed. You can however create a user control, and assign an instance of that to the `NotificationWindow` class's `Content` property, which is the strategy we will be demonstrating here. Before we do however, let's start by simply displaying an empty notification. To do so, you simply create a new instance of the `NotificationWindow` class, set the size of the toast notification window (it has a maximum width of 400 pixels, and a maximum height of 100 pixels), and then call its `Show` method (passing it the amount of time in milliseconds to display the notification), like so:

```
NotificationWindow toastWindow = new NotificationWindow();
toastWindow.Width = 280;
toastWindow.Height = 100;
toastWindow.Show(4000);
```

Running this code will result in a small window appearing in the bottom right-hand side of the screen, and automatically disappearing after 4 seconds. Note that these notifications can only be displayed for a maximum of 30 seconds (30000 milliseconds).

■ **Note** Attempting to create an instance of the `NotificationWindow` class when running inside the browser will throw an exception, so always check to ensure that the application is being run in OOB mode before attempting to show a toast notification.

An empty notification window is of little use, so let's now display some content in it, which can be done by assigning a framework element to its `Content` property. The easiest way to implement the content of a toast notification is to create a user control in your project, design the content in its XAML file, and then assign an instance of the user control to the `NotificationWindow`'s `Content` property. For example:

```
toastWindow.Content = new MessageArrivedNotification();
```

If your application is in the background, and is showing the notification window in order to get itself noticed, a good idea is to activate the main window when the notification window is clicked, and bring it to the front. You can implement this by handling the `MouseLeftButtonDown` event of the `LayoutRoot`, and activating the main window in the event handler:

```
private void LayoutRoot MouseLeftButtonDown(object sender, MouseButtonEventArgs e)
{
    Application.Current.MainWindow.Activate();
}
```

It can also be a good idea to provide a way to close the notification window so that the user can dismiss the notification and continue working. The `NotificationWindow` object has a `Close` method, but you need that object to call the method on (which you don't have in the user control unless you want to trawl through the visual tree hierarchy to get it). The solution to this is to simply accept a `NotificationWindow` object in the user control's constructor (it can then stash it away as a member variable for later use), and pass this through when the object is being created.

```
toastWindow.Content = new MessageArrivedNotification(toastWindow);
```

You can then call the `Close` method on this object from within the user control as required.

■ **Note** You may also wish to pass the constructor an object containing data that the notification window can display if required.

Caching Data Locally

As you saw from some of the chapters earlier in this book, RIA Services provides an easy and elegant means for obtaining data from and persisting data to a server. However, what if a connection to the server is suddenly not available? Do you want all the users of your application to be forced to wait until they are (or the server is) back online again? Or do you want them to be able to continue working (as much as possible) in the meantime until a connection to the server is restored?

This type of scenario is known as “occasionally connected applications”, and is generally handled by downloading and caching commonly used data from the server locally that the user may require when they are offline (and keep it updated)—with enough data to enable them to continue working until they get back online again. While the application is offline, it will need to cache the changes, additions, and deletions the user has made to the data, and synchronize these back to the server when the application is next online.

■ **Note** In addition, you may wish to cache data on the client that rarely or never changes in order to reduce the amount of network traffic that passes between the server and the client.

Let’s take a look at some strategies for caching data on the client, and catering for scenarios where the application may be going back and forth between connected and disconnected mode.

Caching to Isolated Storage

As a part of its sandbox, Silverlight doesn’t allow you direct access to the user’s file system (unless you are running with elevated trust—discussed later in this chapter), in order to protect your computer from rogue Silverlight applications. It does however provide you with a virtual file system known as isolated storage that enables you to cache data and files locally. As its name suggests, isolated storage is a virtual file system that is stored on disk, but isolated from the computer’s main file system and that of other applications. Files can be maintained in this store within directories in the same way as you normally organize files in a file system.

■ **Note** An isolated storage store is not specific to the browser that the user is running the application within. The same store is used between browsers and when running outside the browser—however each Windows user on the computer will have their own isolated storage store.

Isolated storage is partitioned into two sections—an application store, and a site store. The application store can be used to store files available to your application only, while the site store can be used to store files available across Silverlight applications originating from the same domain. Each isolated storage store is given a size quota that can be increased (with the permission of the user) if required.

Let’s take a look at the various ways you can persist and retrieve data from an isolated storage store.

Storing Settings in Isolated Storage

The easiest way to persist and retrieve data from an isolated storage store is by using the `IsolatedStorageSettings` class (that can be found under the `System.IO.IsolatedStorage` namespace). This class enables you to easily maintain a collection of settings in isolated storage as key/value pairs that can be maintained as a file in either of the two stores (application and site) discussed earlier. Instead of having to manage serializing and deserializing the settings to and from a file, this class abstracts that away from you so that you only need to maintain the settings as a collection of key/value pairs, and it will handle persisting and retrieving the collection to/from isolated storage for you.

The `IsolatedStorageSetting.ApplicationSettings` property provides a means to maintain settings in the application store, and the `IsolatedStorageSetting.SiteSettings` property provides a means to maintain settings in the site store. Both of these properties return an object of type `IsolatedStorageSettings` that is a wrapper around a `Dictionary` of type `<string, object>`. Both are used in exactly the same manner as a `Dictionary` (and hence each other), so you will focus on demonstrating maintaining application settings here.

■ **Note** Apart from application settings, another perfect use for storing as settings is dictionary data (used to populate drop-down lists, etc). For dictionaries that rarely or never change, caching them in isolated storage will result in less network traffic needing to pass between the server and the client, and a much more responsive application that doesn't need to wait to load these from the server.

Persisting a Value to Isolated Storage

Each setting is a key/value pair, with the key needing to be a string, but the value able to be any type of object. To persist a value to isolated storage, you simply need to assign it to the indexer of this class, using the key as its index:

```
IsolatedStorageSettings.ApplicationSettings["Key"] = "Value";
```

■ **Note** You don't need to use the `Add` method when adding a new setting to the dictionary. Simply assigning the value as demonstrated will take care of this (and save you the trouble of needing to determine whether the key exists in the dictionary and add/update it accordingly).

As mentioned, the `IsolatedStorageSettings` class is a `Dictionary` of type `<string, object>`, meaning that any type of object (not just strings) can be persisted as an isolated storage setting, and that object will be automatically serialized to (and deserialized when retrieving it from) isolated storage.

The changes to the settings are maintained in memory, and will be automatically saved to isolated storage when the application is closed. However, if the application (or the operating system) were to crash before this time, the changes made would be lost. Therefore you should call the `Save` method of the `IsolatedStorageSettings` object to explicitly save the current settings collection to isolated storage:

```
IsolatedStorageSettings.ApplicationSettings.Save();
```

■ **Note** When you first try to access the settings, all the data stored as settings will be loaded from isolated storage into memory and deserialized. The more data that you store in the settings, the longer this will take, and the more memory it will use. Therefore it is recommended that you do not store an excessive amount of data as settings in isolated storage or this will have an adverse effect on the performance of your application (and potentially of the user's computer). An alternative is to serialize large collections/objects, and persist them as individual files within the isolated storage store instead. This will enable them to be loaded only when required, and disposed of (removing them from memory) once you no longer need them. How you do this will be discussed later in this chapter.

Retrieving a Value from Isolated Storage

When you want to retrieve a setting from isolated storage, you will need to check that it exists first. You can use the `Contains` method to do this, and then retrieve its value (if it exists) using its key as the index, like so:

```
if (IsolatedStorageSettings.ApplicationSettings.Contains("Key"))
{
    string value = IsolatedStorageSettings.ApplicationSettings["Value"].ToString();
    // Do something with the value
}
```

Alternatively, you can use the `TryGetValue` method that bundles checking that the setting exists and retrieving its value if it does into a single task. With the out parameter being a generic type, it will also save the need for casting the returned object:

```
string value;
bool settingExists =
    IsolatedStorageSettings.ApplicationSettings.TryGetValue("Key", out value);
```

Binding to Settings

You can bind to isolated storage settings, by binding to them using indexed property binding, as was demonstrated back in Chapter 10: Advanced XAML and Data Binding. The following XAML demonstrates binding to a setting named "Setting1":

```
<TextBox Name="Setting1TextBox" Text="{Binding [Setting1], Mode=TwoWay}" />
```

Due to the non-creatable nature of the `IsolatedStorageSettings` class (as it has a private constructor), you will need to assign the `DataContext` of the control (or a control further up in the object hierarchy) in the code-behind, as follows:

```
Setting1TextBox.DataContext = IsolatedStorageSettings.ApplicationSettings;
```

■ **Note** As a general rule, you are better off wrapping the settings in properties and exposing them from a class instead of attempting to bind to them directly.

Enumerating the Settings

If you want to enumerate through the existing settings, the `Keys` property will expose what keys exist in the store, and you can then get their values as required:

```
foreach (string key in IsolatedStorageSettings.ApplicationSettings.Keys)
{
    object value = IsolatedStorageSettings.ApplicationSettings[key];
    // Do something with the value
}
```

Removing Settings

You can remove a setting using the `Remove` method of the dictionary, passing it the key of the setting to remove:

```
IsolatedStorageSettings.ApplicationSettings.Remove("Key");
```

You can remove all the settings in a dictionary by calling its `Clear` method:

```
IsolatedStorageSettings.ApplicationSettings.Clear();
```

Storing Files in Isolated Storage

Isolated storage is a virtual file system, meaning that you can create directories and files within isolated storage in much the same way as you would with the operating system's file system. For example, you may wish to store images, documents, and even `.xap` files downloaded from the server (providing dynamically loaded content, as will be discussed in Chapter 15: Application Deployment) in isolated storage. Let's look at how to perform some of these tasks in an isolated storage store.

Opening an Isolated Storage Store

The first step is to open the store that you want to work with. To open the application store, you use the `GetUserStoreForApplication` method of the `IsolatedStorageFile` class. Alternatively, you will need to use the `IsolatedStorageFile` class's `GetUserStoreForSite` method to open the site store.

■ **Note** For the purposes of this demonstration you will be working with the application store, but working with the site store is performed in exactly the same manner after opening the site store instead.

The following code demonstrates opening the application store:

```
using (IsolatedStorageFile store = IsolatedStorageFile.GetUserStoreForApplication())
{
    // Work with the files and directories in the store here
}
```

■ **Note** As a general rule, you should use a `using` block as demonstrated previously, in order to ensure that the store instance is disposed of properly once you are finished working with it.

Working with Directories

Once you have opened a store you can create, move, and delete directories using the methods on the object instance you have of the `IsolatedStorageFile` class. For example, you can create a directory named “Images” like so:

```
store.CreateDirectory("Images");
```

If you want to create a subdirectory named “Icons” within the “Images” directory, you will need to include the full path to create, as follows:

```
store.CreateDirectory(@"Images\Icons");
```

You can determine whether a directory exists using the `DirectoryExists` method:

```
bool dirExists = store.DirectoryExists("Images");
```

■ **Note** You should attempt to keep the paths to your files as short as possible. The full path to the file, including the path to the isolated storage store on the disk, must be less than 260 characters. Exceeding this length will result in an exception being thrown. The number of characters available for your directory and file names will depend upon what operating system the user is running, so as a general rule, it’s best to keep the paths of the files within the isolated storage store to a maximum of 70 characters in length.

Enumerating Through Files and Directories

The `IsolatedStorageFile` class provides a method to enable you to enumerate through the files and directories within an isolated storage store. For example, you can obtain a string array containing the names of the directories using the `GetDirectoryNames` method, like so:

```
string[] directories = store.GetDirectoryNames();
```

You will note that only the directories within the root of the store are returned in this example. To get a list of the subdirectories within a given directory, you need to pass in the path to that directory as a search pattern (followed by a wildcard). The following example demonstrates getting the names of the directories within the “Images” folder:

```
string[] directories = store.GetDirectoryNames(@"Images\*");
```

Similarly, you can get a list of the files in the isolated storage store using the `GetFileNames` method:

```
string[] fileNames = store.GetFileNames();
```

As with getting the subdirectories within a directory, to get the files within a directory you will need to pass in the path to that directory as a search pattern, followed by a wildcard:

```
string[] fileNames = store.GetFileNames(@"Images\*");
```

Working with Files

Various methods are provided to work with files, such as creating, copying, moving, deleting, and opening a file. For example, you can determine whether a file exists using the `FileExists` method:

```
bool fileExists = store.FileExists("Test.txt");
```

Writing a Text File

The following code demonstrates writing a string to a text file in isolated storage:

```
using (IsolatedStorageFileStream fileStream =
    store.OpenFile("Test.txt", FileMode.Create))
{
    using (StreamWriter writer = new StreamWriter(fileStream))
    {
        writer.WriteLine("Text written from Silverlight");
        writer.Close();
    }

    fileStream.Close();
}
```

■ **Note** You will need to add a using directive to the `System.IO` namespace in order for this and the following examples to work.

Writing a Binary File

The following code demonstrates writing a byte array (named `fileContents`) to a file in isolated storage:

```
using (IsolatedStorageFileStream fileStream =
    store.OpenFile("Test.jpg", FileMode.Create))
{
    fileStream.Write(fileContents, 0, fileContents.Length);
    fileStream.Close();
}
```

Reading a Text File

The following code demonstrates reading the contents of a text file from isolated storage into a string:

```
string fileContents = null;

using (IsolatedStorageFileStream fileStream =
    store.OpenFile("Test.txt", FileMode.Open))
{
    using (StreamReader reader = new StreamReader(fileStream))
    {
        fileContents = reader.ReadToEnd();
        reader.Close();
    }

    fileStream.Close();
}

// Do something with the file contents
```

Reading a Binary File

The following code demonstrates reading the contents of a binary file from isolated storage into a byte array:

```
byte[] fileContents = null;

using (IsolatedStorageFileStream fileStream =
    store.OpenFile("Test.jpg", FileMode.Open))
{
    fileContents = new byte[fileStream.Length];
    fileStream.Read(fileContents, 0, (int)fileStream.Length);
    fileStream.Close();
}

// Do something with the file contents
```

Serializing an Object to a File

Generally, you will be maintaining data in objects and collections within your application, and you may wish to persist this data to isolated storage. Rather than writing the data to a file using a custom format (and then needing to parse it and rehydrate the corresponding objects/collections when you want to load the data again), you can simply serialize/deserialize the data to/from XML, using the `DataContractSerializer` class to help you.

Let's say you have a collection of `Product` objects that you want to serialize to a file within an isolated storage store. The following code demonstrates serializing and writing a collection of `Product` objects to isolated storage:

```
using (IsolatedStorageFileStream fileStream =
    store.OpenFile("ProductCollection.xml", FileMode.Create))
{
    DataContractSerializer serializer =
```

```

        new DataContractSerializer(typeof(List<Product>));
        serializer.WriteObject(fileStream, productCollection);
    }

```

■ **Note** You will need to add a using directive to the `System.Runtime.Serialization` namespace in order for this and the following example to work.

Deserializing and Rehydrating an Object from File

The following code demonstrates deserializing a collection of `Product` objects read from a file in isolated storage, and rehydrating the collection:

```

using (IsolatedStorageFileStream fileStream =
    store.OpenFile("ProductCollection.xml", FileMode.Open))
{
    DataContractSerializer serializer =
        new DataContractSerializer(typeof(List<Product>));
    List<Product> productCollection =
        serializer.ReadObject(fileStream) as List<Product>;

    // Do something with the collection

    fileStream.Close();
}

```

Disk Space Quota

When a Silverlight application is first run, it is given a quota of 1MB of space to use in its isolated storage store. This quota is assigned on a per-domain basis, meaning that the quota is shared between all the Silverlight applications originating from the same domain (even though each application has its own separate isolated storage store). Attempting to exceed this quota will result in an exception being raised.

■ **Note** Whenever you work with isolated storage, you should place this code within a try/catch block in order to handle this possibility. Even if your application only expects to use a very small amount of isolated storage space, it can still exceed the quota, as the quota is shared between all applications originating from the same domain. Another application from the same domain may have used most of or the entire quota already, which will affect the amount of space your application has left to use.

You can determine how much quota you have free by checking the `AvailableFreeSpace` property of the `IsolatedStorageFile` object. This value returns the number of bytes still available for use in the isolated storage store:


```
long freeSpace = store.AvailableFreeSpace;
```

You can also retrieve the total quota assigned to the domain via the store's `Quota` property, and retrieve the amount of quote already used via the store's `UsedSize` property.

When a Silverlight application is installed to run outside the browser, its quota will be automatically increased from the initial 1MB to 25MB. However, you can request additional space at any time if it is required. This requires that the user provide permission for the domain to be assigned further quota by displaying a dialog (as shown in Figure 14-6).

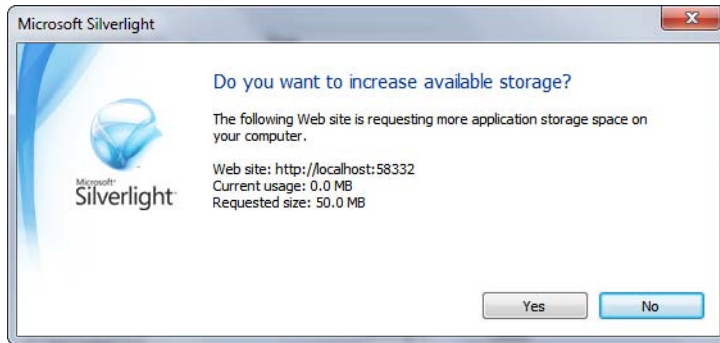


Figure 14-6. The dialog for the application requesting increased isolated storage quota

To request additional quota, you can simply call the `IncreaseQuotaTo` method of the `IsolatedStorageFile` object, passing it the amount of space that you want (in bytes). For example, you can use the following code to request your quota to be increased to 50MB of isolated storage space:

```
bool increasePermitted = store.IncreaseQuotaTo(50 * 1024 * 1024);
```

The method returns a Boolean value indicating whether permission for the additional space has been permitted or denied by the user. Requesting a smaller quota than that already granted will throw an `ArgumentException` exception stating that “The new quota must be larger than the old quota”, so be prepared for this scenario and put the call in a try/catch block (or make sure the value you are setting is greater than that of the store's `Quota` property).

■ **Note** The call to the `IncreaseQuotaTo` method must originate from a user initiated event. In other words, you can only request additional isolated storage quota in an event raised by some sort of user input (such as the `Click` event for example). Attempting to request additional isolated storage quota in a non-user input initiated event (such as the `Loaded` event of a control) will simply be denied. This rule only applies when the application is running in sandboxed mode. When the application is running with elevated trust permissions (as described later in this chapter), this limitation is lifted.

Encrypting Data

Something to be aware of when writing data to isolated storage is that it is easily accessible by burrowing into the user's profile folders. The isolated storage store is maintained in a hidden folder with the exact same file/directory structure as they were created in the store, as demonstrated in Figure 14-7.

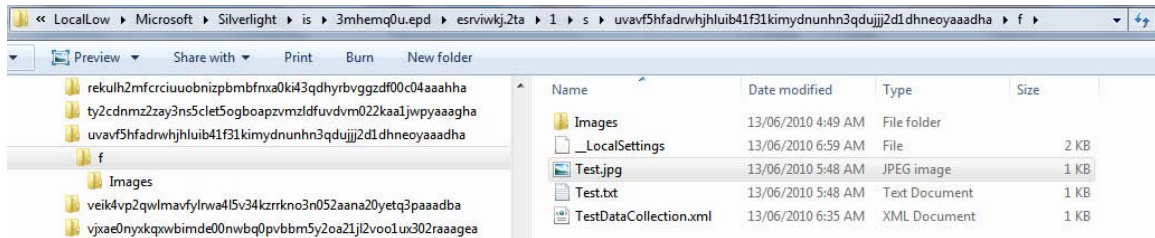


Figure 14-7. An isolated storage folder on disk

The location of the isolated storage stores on disk differs between operating systems, but in Windows Vista/7 you can find the area by typing “%userprofile%\AppData\LocalLow\Microsoft\Silverlight\is” into the Windows Explorer address bar. When you write files to isolated storage, you are simply writing them to a folder somewhere underneath this path.

As you can see, from Figure 14-7, the path to the isolated storage store is somewhat obfuscated, but if you create a file in the isolated storage store with a unique name, you can find it fairly easily (and hence the isolated storage store) using the search functionality in Windows Explorer. You can then open the files and inspect their contents (and modify them) if you wish.

The point you need to note from this exercise is that nothing you write to isolated storage should be considered secure, and thus you should not write sensitive data to isolated storage, or assume that it can't be modified outside your application.

You may wish to consider encrypting your files in order to help prevent these sorts of issues if they are of concern to you. The following MSDN article should point you in the right direction for doing so: [http://msdn.microsoft.com/en-au/library/dd153755\(VS.95\).aspx](http://msdn.microsoft.com/en-au/library/dd153755(VS.95).aspx).

Compressing Data

With a default quota of 1MB, depending on the type and number of files you want to cache in isolated storage, you may wish to consider attempting to make the most efficient use of the available space within isolated storage by compressing the files (saving the need to request additional quota from the user so soon).

Silverlight doesn't have any built-in compression routines, but you can use an open source library that provides this functionality instead. Two of the best options available to you are Silverlight ports of the SharpZipLib and DotNetZip projects. The SharpZipLib has a GPL license, whilst the DotNetZip has a less restrictive Ms-PL license.

You can download the Silverlight port of the SharpZipLib project from <http://slsharpziplib.codeplex.com>, however the DotNetZip project has no official port attempt (the original project can be found here: <http://dotnetzip.codeplex.com>). If you download the Document Toolkit Extensions project however, you will find a Silverlight port of both projects in it that is available here: <http://documenttoolkit.codeplex.com>.

Refer to the documentation of the project that you choose to use for details of how to use it.

The Sync Framework

Not available at the time of this writing, but worth looking into, is the Sync Framework for Silverlight. This is a framework for synchronizing data between a Silverlight application and the server, providing a range of new possibilities for occasionally connected Silverlight applications. Data will be maintained in a local store for use by the Silverlight application while it is offline, and changes in the data during the time offline will be synchronized back with the server once it's back online (and vice versa). The homepage for the Sync Framework on the Microsoft web site can be found at <http://msdn.microsoft.com/en-us/sync/default.aspx>.

Caching to a Client-Side Database

Isolated storage provides a way for caching files on the client, but it is not a suitable means of caching data—it's certainly in no way comparable with the features that a database provides.

You could store a collection of objects in isolated storage as was demonstrated using the `IsolatedStorageSettings` class, but it's not a very efficient way of doing so if the collection is large. When you first attempt to get an instance of the application or site settings object, all of the settings will be loaded from isolated storage into memory. To get a single object would require loading all the settings into memory, and to simply insert/update/delete a setting would require all the settings to be loaded and then persisted back to isolated storage again.

We also discussed and demonstrated serializing objects as files to isolated storage as well, but for large sets of data this can be an inefficient means of maintaining the data also.

Unfortunately, Silverlight has no built-in client-side database engine to solve these problems, which can be an issue when it comes to implementing occasionally connected applications. There are a number of potential solutions however (of differing degrees of suitability), that will be discussed briefly.

Sterling

Sterling is a lightweight object-oriented database that uses an extensible binary serialization engine, written by Jeremy Likness. It maintains keys and table indexes for fast querying, and shows a lot of potential (it's currently in alpha at the time of this writing). You can get it from here:

<http://sterling.codeplex.com>.

Silverlight Database (SilverDB)

The Silverlight Database engine also serializes the tables and their data serialized to a file in isolated storage, but without the key and index support that Sterling provides. It is worth investigating if you want to provide a simple predefined means to persisting and retrieving data in isolated storage. You can get this library here: <http://silverdb.codeplex.com>.

SQLite

SQLite is a popular database engine with a small footprint and an extremely permissive license and has been around for quite some time now. It is written in C, and ported to many environments (original project here: www.sqlite.org).

Noah Hart ported SQLite to C#, named C#-SQLite (found at <http://code.google.com/p/csharp-sqlite>), which runs against the full .NET Framework, but not Silverlight.

This project was ported to Silverlight by Tim Anderson as a proof of concept (not in a production-ready state), and can be downloaded from here: www.itwritings.com/blog/1695-proof-of-concept-c-sqlite-running-in-silverlight.html.

SQLite was a part of (the now deprecated, but still available and widely used) Google Gears, and this provides another possibility for use as a database engine and store. You can create and store data in a SQLite database using the JavaScript API provided by Google Gears. An example of doing so is demonstrated here: www.mojoportal.com/silverlight-google-gears-awesome-at-least-in-firefox.aspx.

■ **Note** Since JavaScript is required to communicate with the database, it can only be used within the browser (in other words, it will not be accessible to applications running outside the browser).

Sharp HSql

Craig Dunn produced a proof-of-concept port of the Sharp HSql database engine to Silverlight (that was itself a port of the hsqdb database engine written for Java), and has made it available here: <http://conceptdev.blogspot.com/2009/07/sql-in-silverlight.html>.

Commercial Options

As you can see from the descriptions of the previous database engines, the Silverlight Database (SilverDB) project doesn't really solve the problem and the Silverlight ports of the SQLite and Sharp HSql database engines are not considered production ready. There are however some commercial offerings that you may wish to consider:

- **Perst:** is an open source database engine released under the GPL license, so it's not a commercial offering per se. However, for use in closed-source applications you will need to buy a license, in which case it becomes a commercial offering. The cost of a license isn't published.

You can purchase/download it from here: www.mcobject.com/perst.

- **Siaqodb:** is a database engine targeting a number of platforms. A single developer license for its Silverlight edition (at the time of writing) is \$US186.

You can purchase/download it from here: <http://siaqodb.com>.

- **EffiProz Embedded C# Database:** is a database engine targeting both Silverlight and Windows Phone 7. A single developer license for its Silverlight edition (at the time of writing) is \$US500.

You can purchase/download it from here: www.ffmpeg.com/product_sl.aspx

Communicating with Local Database Engines

When your application is running in OOB mode and with elevated trust on Windows systems, you will have access to COM. You can take advantage of this by using the ADO COM objects, and interacting with the various database engines installed on the user's machine (such as SQL Server, SQL Compact, Microsoft Access, and so on.). This option is only available in the conditions previously mentioned,

making it less than ideal, but does provide an option. Using the ADO COM objects will be discussed later in this chapter.

Detecting Network Connection Availability

In order to effectively support occasionally connected scenarios, you need to be able to detect whether the application is online or offline and cater for the current state accordingly. You also need to be able to detect when this state changes (from offline to online) and start re-synchronizing cached data back with the server.

Silverlight has the built-in ability to detect network availability and any changes in it, and you can find the classes to help you do so under the `System.Net.NetworkInformation` namespace. You can use the `NetworkInterface` class to help determine whether a network connection is available, and you can use the `NetworkChange` class to notify you when this state changes.

For example, the following code determines if a network connection is available:

```
bool isNetworkAvailable = NetworkInterface.GetIsNetworkAvailable();
```

■ **Note** This method may report that you have a connection available, but there's no guarantee that you actually have a connection to the Internet, or that the server you want to communicate with is accessible. It simply reports whether or not there is a network adapter available that currently has an IP address (it won't if it has no network connection).

You can monitor for changes in network availability by handling the `NetworkAddressChanged` event of the `NetworkChange` class, like so:

```
NetworkChange.NetworkAddressChanged += NetworkChange.NetworkAddressChanged;  
private void NetworkChange.NetworkAddressChanged(object sender, EventArgs e)  
{  
    // Handle network state changed as required  
}
```

■ **Note** This event doesn't tell you whether a network connection is now available or not—merely that the IP address of a network adapter has changed. You will need to use it in conjunction with the `NetworkInterface.GetIsNetworkAvailable` method in order to determine the new network connection state.

File System Open/Save Dialogs

Isolated storage is the means provided to store and retrieve data locally on the user's machine, but sometimes you need to read and write to actual files within the user's file system. For example, you may

want to be able to save a file that was downloaded from the server, or upload a file to the server from the user's machine.

As part of the Silverlight sandbox, you can't read or write files directly from/to disk without user interaction (providing implicit permission to do so) unless you have elevated permissions (as will be discussed later in this chapter). In sandboxed mode however, the Open/Save dialogs provide this user interaction to read/write files by providing the user with a means to select what local files can be read or written to, and provides the permission to do so. Let's take a look at how you can use these dialogs, and the file based operations that they enable.

■ **Note** Attempting to display an open or save file dialog must be the result of a user initiated action. In other words, you can only open an open/save file dialog in an event raised by some sort of user input (such as the Click event, for example). Attempting to display the open/save file dialog in a non-user input initiated event (such as the Loaded event of a control) would raise an exception. This rule only applies when the application is running in sandboxed mode. When the application is running with elevated trust permissions (as described later in this chapter), this limitation is lifted.

The Open File Dialog

You can use the `OpenFileDialog` class (as shown in Figure 14-8) to display a dialog enabling the user to select a file to open. This will provide you with the permission to access the selected file, by returning one or more `FileInfo` objects (from the `System.IO` namespace) that enable you to open the file and read the data from it.

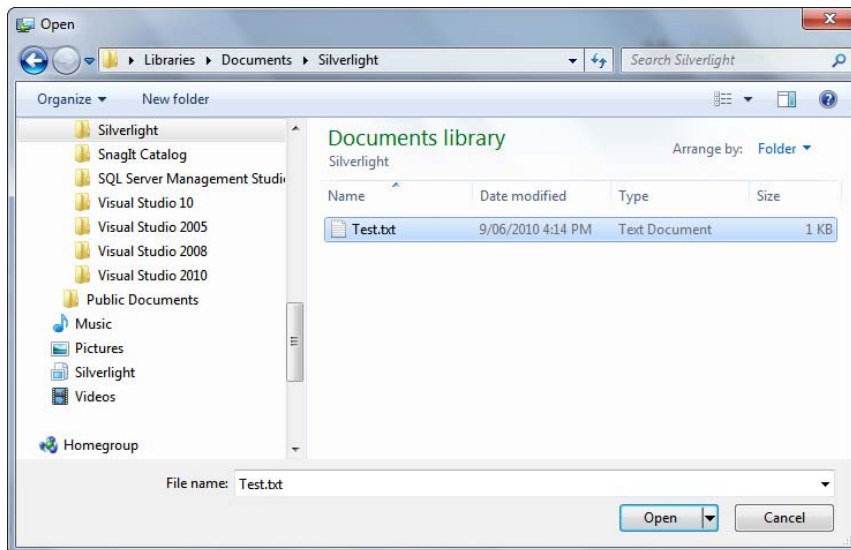


Figure 14-8. The Open File dialog

Displaying the Dialog

The code to display the open file dialog is as follows:

```
OpenFileDialog dialog = new OpenFileDialog();
bool? fileSelected = dialog.ShowDialog();

if (fileSelected.HasValue && fileSelected.Value)
{
    // Code to open and read the file goes here
}
```

The `ShowDialog` method is the code that actually displays the dialog, and returns a nullable Boolean indicating whether the user selected a file or cancelled the operation. This variable will be set to true if the user selected a file and then clicked the Open button.

■ **Note** This method may throw an `InvalidOperationException` or `SecurityException` exception. Ensure that your code catches this exception and handles it accordingly.

Filtering the Files

You can limit the dialog to displaying files of only a given type (or types), by specifying one or more file filters using the `Filter` property of the dialog object. Unfortunately, the way to assign these filters is somewhat archaic, but aligns to what you may be used to when configuring file filters for dialogs in other technologies such as Windows Forms. For example, to limit files to just text files (in other words, files with a `.txt` extension), you can use the following syntax (before opening the dialog):

```
dialog.Filter = "Text Files|*.txt";
```

This string is split into two sections, separated by a pipe character. The first section denotes the text to display representing the filter in the drop-down list, and the second section denotes the actual filter to apply to limit the files displayed.

You can extend this filter to include a number of file types/extensions by separating the file extensions in the second section of the filter string with a semi-colon, as demonstrated below:

```
dialog.Filter = "Documents|*.doc;*.docx;*.rtf";
```

Alternatively, you can have multiple filters that the user can select from in the dialog, like so:

```
dialog.Filter = "Word 2002/2003 Documents|*.doc|Word 2007/2010 Documents|*.docx";
```

As you can see, another pipe character is used to separate each entry. If the first entry is not the one you want to use as the default selection, you can specify a different one using the `FilterIndex` property. This is a one-based index, so you would set this property to 2 in order to select the Word 2007/2010 Documents filter (in other words, the second filter) from the example above.

Opening and Reading the Selected File

Once the user has selected a file, you can retrieve the `FileInfo` object that it has created. When running in sandbox mode, you will only be able to view limited information about the selected file. Attempting to access file properties such as its directory or its attributes will be denied, resulting in a `SecurityException` being thrown. You can get its name from the `FileInfo`'s `Name` property, and its size from its `Length` property.

■ **Note** You can't create an instance of the `FileInfo` class yourself (unless you are running with elevated trust, as discussed later in this chapter). Attempting to do so will result in a `SecurityException`, stating that access to the file has been denied.

The `FileInfo` object is assigned only limited permissions for the actions you can perform on it, but it does allow you to open the file and read its contents. There are two methods on the object that you can use to do so—the `OpenText` method, and the `OpenRead` method.

Opening and Reading Text Files

If you know that (or expect) the selected file to be a text file, you can use the `OpenText` method to read its contents. This method allows you to read the file by retrieving:

- A given number of characters at a time
- The contents line by line
- The entire contents of the file in a single step

The latter two methods are the most common when dealing with text files, and are demonstrated in the following sections.

Reading the File Line by Line

To read the file a line at a time, use the `ReadLine` method:

```
string fileContents = null;

using (StreamReader fileStream = dialog.File.OpenText())
{
    while (!fileStream.EndOfStream)
    {
        fileContents = fileStream.ReadLine();
        // Do something with the line contents
    }

    fileStream.Close();
}
```



```
// Do something with the file contents
```

Reading the Whole File in a Single Step

To read the whole file in a single step, use the `ReadToEnd` method:

```
string fileContents = null;

using (StreamReader fileStream = dialog.File.OpenText())
{
    fileContents = fileStream.ReadToEnd();
    fileStream.Close();
}

// Do something with the file contents
```

Opening and Reading Binary Files

When working with other non-text file types (such as images, documents, etc), you will need to use the `OpenRead` method to read their contents, as a byte array.

Reading the Whole File in a Single Step

The code below demonstrates reading the entire contents of a file into a byte array using the `Read` method:

```
byte[] fileContents = null;

using (FileStream fileStream = dialog.File.OpenRead())
{
    fileContents = new byte[dialog.File.Length];
    fileStream.Read(fileContents, 0, (int)dialog.File.Length);
    fileStream.Close();
}

// Do something with the file contents
```

Reading the File in a Chunks

You can read the file in chunks (also using the `Read` method), like so (especially useful when the file is large):

```
const int chunkSize = 4096; // 4kb blocks

using (FileStream fileStream = dialog.File.OpenRead())
{
    byte[] chunk = new byte[chunkSize];
    int bytesRead = 0;
    int position = 0;

    do
    {
```

```

        bytesRead = fileStream.Read(chunk, 0, chunkSize);

        // Do something with the chunk

        position += bytesRead;
    } while (bytesRead > 0);

    fileStream.Close();
}

```

Reading and Displaying an Image File

If you want to load an image to display in an Image control, the code is even simpler. The following code demonstrates populating a `BitmapImage` object with the contents of a file and assigning it to the `Source` property of an Image control (named `imageControl` in the code):

```

using (FileStream fileStream = dialog.File.OpenRead())
{
    BitmapImage image = new BitmapImage();
    image.SetSource(fileStream);

    // Assign the BitmapImage object to the Source property of an Image control
    imageControl.Source = image;

    fileStream.Close();
}

```

Enabling Multiple File Selection

By default, the user can only select a single file from the open file dialog. However, you can set its `Multiselect` property to `true` in order to enable the user to select multiple files at the same time. The `File` property will only return the first file selected, but you can use the `Files` property instead. That returns an array of `FileInfo` objects that you can loop through and read as required.

The Save File Dialog

You can use the `SaveFileDialog` class (as shown in Figure 14-9) to display a dialog enabling the user to specify the name and location of a file to save, or select a file to overwrite. This will provide you with the permission to write to the selected file, by returning a `FileInfo` object that enables you to create the file and write data to it.

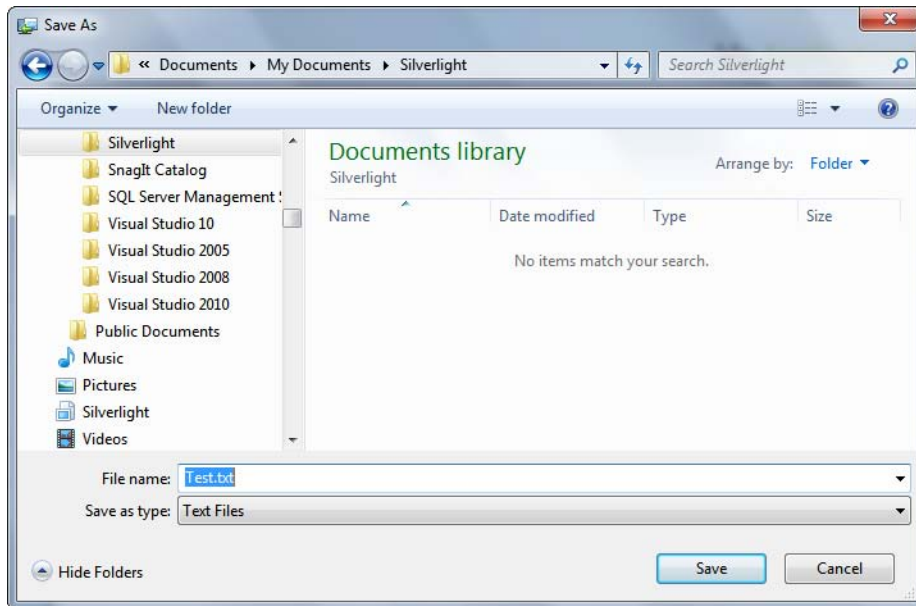


Figure 14-9. The Save File dialog

Displaying the Dialog

The code to display the open file dialog is as follows:

```
SaveFileDialog dialog = new SaveFileDialog();
bool? fileEntered = dialog.ShowDialog();

if (fileEntered.HasValue && fileEntered.Value)
{
    // Code to open and write the file goes here
}
```

You'll note that this is very similar to how the `OpenFileDialog` class is used.

■ **Note** If the user enters invalid characters as the file name, this will raise a message box notifying the user as such, and the dialog simply won't accept a file name with these characters. However, on an Apple Macintosh, the dialog will accept these characters, but will consequently throw an `ArgumentException`. Ensure that your code catches this exception and handles it accordingly.

Setting a Default Extension

Like the `OpenFileDialog` class, you can set various filters to limit the files displayed in the dialog. If you specify one or more filters and the user does not enter an extension with the file name that they enter, then the extension associated with the selected filter will be automatically appended to the file name. If no filter is specified, or it is an All Files (*.*) type filter, then the value assigned to the `DefaultExt` property of the `OpenFileDialog` class will be appended to the file name instead (if provided).

Writing to the File

Create a new (empty) file with the name and path provided by the user by calling the `OpenFile` method that creates and opens the file for writing. You will note that unlike the `OpenFileDialog` class (which has separate methods for reading files as text or binary), the `SaveFileDialog` class has only this single method. We'll discuss how to write text or binary data to the file shortly.

Unfortunately, you can't set a default name for the file in the dialog. You can however get the name of the file (excluding path information) from the dialog class's `SafeFileName` property.

Writing a Text File

To write text to the file, you need to create a `StreamWriter` from the stream returned from the `OpenFile` method. You can then use the `Write` or the `WriteLine` methods of the `StreamWriter` object to write text to the file as required.

```
using (Stream stream = dialog.OpenFile())
{
    using (StreamWriter writer = new StreamWriter(stream))
    {
        writer.WriteLine("Text written from Silverlight");
        writer.Close();
    }

    stream.Close();
}
```

Writing a Binary File

Writing binary data to a file is actually easier than writing text because there's no need for a `StreamWriter` (or similar) to help you in doing so. You can simply use the `Write` method of the stream that's returned when opening the file to write the contents of a byte array to the file. The code below demonstrates writing the contents of a byte array named `fileContents` to the file that the user has specified it should be written to via the save file dialog:

```
using (Stream stream = dialog.OpenFile())
{
    stream.Write(fileContents, 0, fileContents.Length);
    stream.Close();
}
```

Drag Drop Target

New in Silverlight 4 is the ability for your application to act as a drop target for files that have been dragged from the desktop or Windows Explorer. This feature is even available when the application is running within a browser window. This provides another means for enabling you to open a file from the user's file system, without the user having to go via the open file dialog. For example, you might want to create a file uploader, where users can drag files onto your application and have them automatically uploaded to the server. This is just one of the potential uses that this feature can help provide.

The first step when implementing this feature is to decide where you want files to be able to be dropped within your application. For this simple demo, let's enable the user to drop a file on an Image control, and have that Image control display that file (if it is an image file). Unfortunately (despite supporting the required events), the Image control won't raise the drag/drop events when a file is dragged over and dropped on the control, but you can place it inside a Border control, which will raise these events for us instead.

■ **Note** The Border control requires its Background property to be assigned a value, or its drag/drop events won't be raised either.

Therefore, the Border control in your view will be a drop target, and to enable it to accept dropped files, you need to set its AllowDrop property to true, and handle the drag/drop events that you require:

```
<Border Name="imageBorder" Height="300" Width="300" AllowDrop="True"
    BorderThickness="1" BorderBrush="Black" Background="White"
    Drop="imageBorder_Drop">
    <Image Name="droppedImage" Stretch="Uniform" />
</Border>
```

Now, when the user drags a file from the file system over this control, the DragEnter event will be raised when the file is dragged within the bounds of the control, the DragOver event will be raised as the dragged file is moved over the control, and the DragLeave event will be raised when the file is dragged outside of the bounds of the control again. However, the event you are most interested in is the Drop event, which is raised when a file is dropped onto the control.

```
private void imageBorder Drop(object sender, DragEventArgs e)
{
    // Code to open/read the dropped files goes here
}
```

The DragEventArgs object passed through to this event handler enables you to get the details of the files dropped on the control, using the GetData method of the IDataObject object that it exposes:

```
FileInfo[] files = e.Data.GetData(DataFormats.FileDrop) as FileInfo[];
```

Now you can loop through these files, opening them and reading their contents as required (in the same manner as demonstrated when discussing the open file dialog). The code below demonstrates

opening a dropped file, ensuring it is an image file, reads its contents, and displays it in the Image control in your view:

```
if (files.Length != 0)
{
    // Ensure that the file is an image (by validating the extension)
    string[] validExtensions = new string[] { ".jpg", ".png" };

    // Note that you need a using directive to System.Linq for this line.
    // NOTE: Multiple files can be dropped on the application, we are
    // displaying only the first file.
    if (validExtensions.Contains(files[0].Extension.ToLower()))
    {
        try
        {
            // Open the file, read it into a BitmapImage object, and
            // display it in the Image control.
            using (FileStream fileStream = files[0].OpenRead())
            {
                BitmapImage bitmap = new BitmapImage();
                bitmap.SetSource(fileStream);
                fileStream.Close();

                droppedImage.Source = bitmap;
            }
        }
        catch { }
    }
}
```

Note that you will need to add using directives to the following namespaces to the top of your code file:

```
using System.IO;
using System.Windows.Media.Imaging;
```

■ **Note** Unfortunately, dropping files on a Silverlight application when it's running on an Apple Macintosh requires some workarounds in order to support the feature, and will only work in Safari (not Firefox). These workarounds are documented in the following MSDN article: [http://msdn.microsoft.com/en-us/library/ee670998\(VS.95\).aspx](http://msdn.microsoft.com/en-us/library/ee670998(VS.95).aspx)

Clipboard

The user can use the standard Cut/Copy/Paste shortcut keys (Ctrl+X, Ctrl+C, and Ctrl+V) from within controls such as the TextBox and the RichTextBox to cut/copy/paste the selected text in the control to

and from the host operating system's clipboard. However, if you wished to cut/copy/paste text to and from the clipboard programmatically (prior to Silverlight 4), you were out of luck.

Silverlight 4 finally introduced the `Clipboard` class (found in the `System.Windows` namespace) to enable you to do so. This class enables you to get the text currently on the clipboard (using its `GetText` method), set the text on the clipboard (using its `SetText` method), and query whether there is any text on the clipboard (using its `ContainsText` method).

You can get the text currently on the clipboard like so:

```
string clipboardText = Clipboard.GetText();
```

And you can set the text on the clipboard like this:

```
Clipboard.SetText("Some text");
```

To check whether there is any text available on the clipboard:

```
bool clipboardHasText = Clipboard.ContainsText();
```

Note, however, that there are a few limitations and issues that you should be aware of. The first is that a dialog will appear, asking the user for permission to allow the application access to the clipboard (as shown in Figure 14-10). This permission dialog will only appear once, and will not appear again until the next time the application is run (although you can select the check box that says to remember your answer).

■ **Note** If the user refuses the application permission to access the clipboard, an exception will be thrown. Therefore always remember to wrap your calls to the clipboard in a try/catch block.

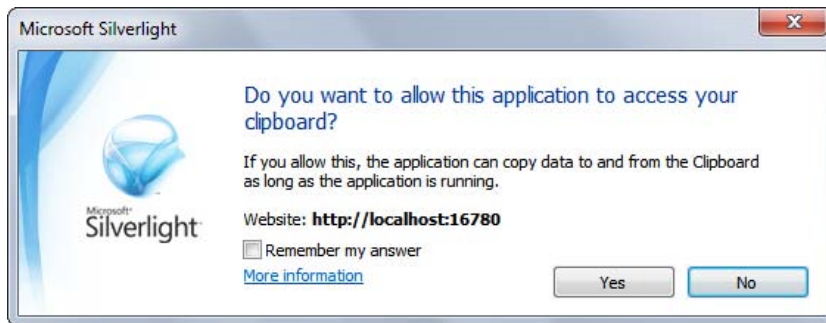


Figure 14-10. The dialog requesting permission to access the clipboard

Another issue to be aware of is that like displaying the open/save dialogs, the clipboard interaction must be user initiated. Attempting to interact with the clipboard in a non-user input initiated event (such as the `Loaded` event of a control) will immediately raise an exception (in other words, the dialog asking for permission will not be displayed—permission will automatically be refused). The only

exception to this rule where this limitation will be lifted is when the application is running with elevated trust permissions (as described later in this chapter).

■ **Note** If your application has elevated trust permissions when running in OOB mode, the Silverlight runtime will assume it has permission to access the clipboard, and skip displaying the permission dialog.

The other major limitation of the `Clipboard` class is that it only supports copying and pasting text to/from the clipboard. Other types of data (such as images) are not supported. The only way to copy an image to the clipboard would be to encode it as Base64 (an encoding commonly used to convert binary data to a text representation). However, no other application is likely to be able to recognize this, and thus doing so has very limited uses and benefits).

■ **Note** Worth noting is that the `DataGrid` now has a built-in copy feature where you can select a number of rows, and press `Ctrl+C` to copy them to the clipboard (in tab delimited form). You can then paste them into another application, such as Microsoft Excel.

Three copy modes are available, and you can select which mode you want to support by setting the `ClipboardCopyMode` property on the `DataGrid` control. The modes are `None` (the user can't copy row data to the clipboard), `ExcludeHeader` (don't include the column headers as the first line of data), and `IncludeHeader` (do include the column headers as the first line of data).

Additionally, you can specify what data is copied for a column (if it should be different to what is displayed) using the `ClipboardContentBinding` property on the `DataGridColumn`, and you can also handle the `CopyingRowClipboardContent` event on the `DataGrid`, which is raised as each row is copied so that you can see what content is being copied to the clipboard (although you can't modify that content, limiting its uses).

Full Screen

You can set your Silverlight application to run in full screen mode if you wish (even when it is running inside the browser). This feature was designed to enable streaming video to be played full screen and for games, but you may find some uses for it in your business applications too. Let's take a look at how to do this.

Initiating Full Screen Mode

The following code sets the application to run in full screen mode:


```
Application.Current.Host.Content.IsFullScreen = true;
```

Once the application is in full screen mode, the user can press the Escape key to go back to running normally. When the application enters full screen mode, it will briefly flash up a message notifying the user of this (as shown in Figure 14-11).

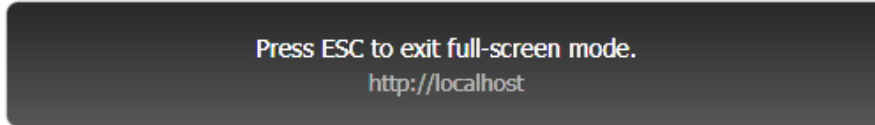


Figure 14-11. The message that briefly appears when entering full-screen mode

Alternatively, you can set the `IsFullScreen` property to false to return to normal mode programmatically. You will need to do this when enabling full screen mode when running with elevated trust, as the Escape key does not automatically exit full screen in that scenario.

As with opening an open/save dialog or interacting with the clipboard, setting the application to display full screen must be a user initiated action (in other words, you can only do so from a Click event or similar, where the event is raised in response to user input). Attempting to do so otherwise (such as in the Loaded event of a page/view) will result in the action simply being ignored.

If the application is running in OOB mode with elevated trust, this limitation is lifted. In this scenario, you can set the application to run in full screen mode from the application's Startup event, or from the Loaded event or constructor of a page/view (for example) if you wish. However, when doing so from the application's Startup event or the constructor of the RootVisual, you must use the `BeginInvoke` command when setting the `IsFullScreen` property to true such that it does so on the user interface thread (otherwise it won't work). The code below demonstrates how to do this:

```
Deployment.Current.Dispatcher.BeginInvoke(() =>
{
    Application.Current.Host.Content.IsFullScreen = true;
});
```

Detecting the Switch to and from Full Screen Mode

You can detect when the application has entered or left full screen mode by handling the `FullScreenChanged` event on the `Application.Current.Host.Content` object, like so:

```
Application.Current.Host.Content.FullScreenChanged += Application.FullScreenChanged;

private void Application.FullScreenChanged(object sender, EventArgs e)
{
    // Full screen changed logic goes here
}
```

Retaining Full Screen Mode When Unfocused

While the application is running in full screen mode, attempting to switch to another application will take it back out to running in normal mode (either in its browser window or out-of-browser window).

You can prevent this behavior (so that it remains in full screen mode, but with the other window in front of it) using the following code:

```
Application.Current.Host.Content.FullScreenOptions =  
    System.Windows.Interop.FullScreenOptions.StaysFullScreenWhenUnfocused;
```

After setting this option, when you next attempt to go into full screen mode, the user will be asked for permission to enable the application to remain in full screen mode regardless of other applications obtaining the focus (as shown in Figure 14-12).

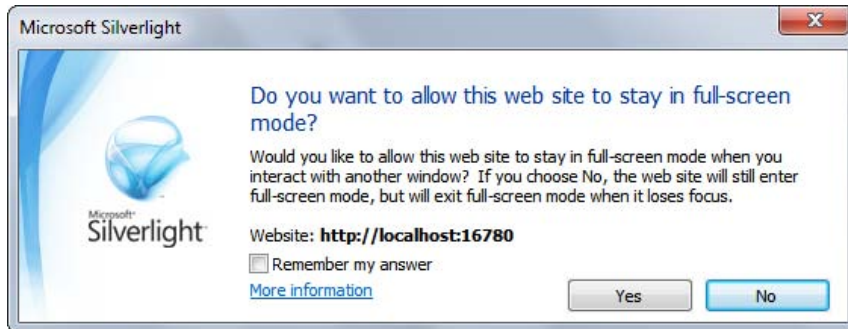


Figure 14-12. The dialog asking whether it should be able to stay in full-screen mode

This message won't be displayed when the application is running in OOB mode with elevated trust.

■ **Note** Unfortunately, attempting to display an open/save dialog box will return the application to normal mode, even if the `FullScreenOptions` property is set to `StaysFullScreenWhenUnfocused`.

Keyboard Access

One of the primary limitations of full screen mode is that the majority of keyboard input is disabled. This behavior is intended to prevent a rogue Silverlight application impersonating the Windows lock screen (for example), where you may unintentionally enter your user credentials and have someone steal them from you. Mouse input will still work as normal in full screen mode, as will the following keyboard keys: the arrow keys (up, down, left, right), Space, Tab, Page Up, Page Down, Home, End, and Enter. As mentioned earlier, the Escape key is captured by the Silverlight runtime and exits the full screen mode.

Running with elevated trust lifts this restriction.

Elevated Trust

Prior to Silverlight 4, Silverlight applications could only run in a sandboxed mode (with partial trust). This is/was a protective measure used to prevent a rogue Silverlight application from damaging the user's machine. This was an essential security measure, but vastly limited the possibilities available

when creating Silverlight applications. By restricting you to the sandbox, you couldn't interact with the user's machine. For example, you couldn't:

- Read or write files from the user's file system (without their interaction via the open/save file dialogs)
- Interact with hardware attached to the user's machine (such as printers, scanners, biometric devices, etc)
- Query a local database engine (such as SQL Server)
- Interact with other applications installed on the user's machine (such as Word and Excel)

For some business applications, these restrictions meant that Silverlight was not a suitable application development platform to meet their requirements.

Silverlight 4 however introduced a newly elevated trust mode that enabled Silverlight applications to break out of the sandbox (partially, at least) to help overcome a number of the prevailing concerns of Silverlight's usefulness as an application development platform. This mode is only enabled when the application is running outside of the browser, but enables your applications a much wider range of possibilities and potential uses by integrating with the host machine and operating system.

Let's go through some features of Silverlight 4 that are only available when running with elevated trust, but first you'll look at how to enable elevated trust and determine whether the application has elevated trust permissions or not.

■ **Note** Despite having elevated trust, the application will not exceed the user's Windows privileges. Therefore, having elevated trust will not enable users to read or write to file paths that they have no access to, nor will they be able to perform actions via COM that their Windows privileges do not permit.

Enabling Elevated Trust

Enabling elevated trust in your application is a very simple process. In the Out-Of-Browser Settings window properties (shown previously in Figure 14-2), which you can get to via the project's properties, select the "Require elevated trust when running outside the browser" check box. Elevated trust is only available when running in OOB mode, so the location of this option makes sense. You'll note that with this option set, the install dialog is quite different from the one displayed without the option set (as was shown in Figure 14-4). Instead, the user will get a more severe looking dialog, as shown in Figure 14-13.

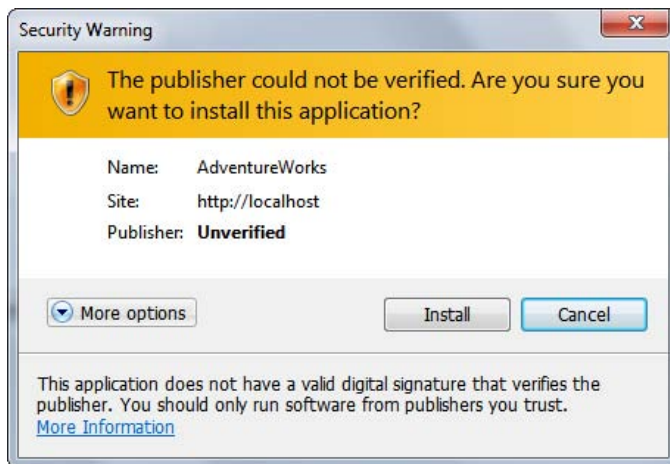


Figure 14-13. The install dialog for unsigned applications requesting elevated trust permissions

This dialog can be made a little friendlier by signing your XAP file. Doing so will change the warning at the top of the dialog to a simple question asking whether you want to install the application, and will also display the application's icon. XAP signing will be discussed in Chapter 15: Application Deployment.

Before invoking any functionality that requires elevated trust, you should ensure that the application is actually running with elevated trust permissions. Despite having set the option that it should do so, while it is running within the browser it won't have these privileges. It's only when the application is installed and actually running in OOB mode that the application will have the elevated trust permissions.

You can then determine whether the application has elevated trust permissions when it's running by checking the `HasElevatedPermissions` property of the current `Application` object, like so:

```
bool hasElevatedTrust = Application.Current.HasElevatedPermissions;
```

Whenever your application attempts to use features that require elevated trust, it should check this property first to ensure that it has permission to use them.

■ **Note** Often, when an application makes extensive use of features that require elevated trust, it will insist that it be installed and run in OOB mode when accessed via a browser by simply displaying an install button. The means for implementing this was discussed earlier in this chapter.

File System Access

Having elevated trust permissions opens up the user's file system to direct reading and writing (without the need to go via the open/save file dialogs), *but only to the folders within the user's profile*. Attempting to open a file (for reading or writing) from outside this location will result in a `SecurityException` being thrown, stating that access to the path is denied.

■ **Note** If you find that you really need to read or write to files outside the user's profile folders (without requiring user interaction via the open/save file dialogs), you can actually bypass Silverlight's built-in file access functionality, and do so via COM instead (discussed later in this chapter). Of course, this will still only be permitted if the user's Windows permissions enable them to read or write to the paths you want them to read from or write to.

The classes that enable you to read from and write to the file system can be found under the `System.IO` namespace, and closely mirror those available in the full .NET Framework. For example, you will find the `Directory`, `DirectoryInfo`, `File`, `FileInfo`, and `Path` classes in this namespace that you can use when running with elevated trust. Despite many of these classes having methods that appear to allow you certain functionality, not all of these methods are actually implemented.

Let's take a look at some of the tasks you can perform on the file system when running with elevated trust permissions.

Getting the Path of a Special Folder

To access a file for reading or writing, you first have to determine the location of the file (a task performed by the user via the open/save file dialogs in the examples detailed earlier in this chapter). You only have access to the folders within the user's profile, but need to find out what the paths to these folders are. To do so, you can use the `GetFolderPath` method of the `Environment` class, passing it a value from the `Environment.SpecialFolder` enumeration that denotes the folder you are after. The code below demonstrates obtaining the path to the user's Documents folder:

```
string docPath = Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments);
```

■ **Note** Not all special folders listed in the `Environment.SpecialFolder` enumeration are accessible. For example, attempting to get the path to the machine's Program Files folder will result in a `SecurityException` being thrown, stating that access to the path is denied.

Enumerating Files and Directories within a Folder

Once you've obtained a folder's path, you can enumerate through the files and directories within the folder.

To enumerate through the files within a folder, you can create a `DirectoryInfo` object (passing in the path to its constructor), and use the methods that it exposes. For example, you can enumerate through the files within the user's Documents folder like so:

```
string docPath = Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments);  
DirectoryInfo docPathInfo = new DirectoryInfo(docPath);
```

```
IEnumerable<FileInfo> docPathFiles = docPathInfo.EnumerateFiles();  
foreach (FileInfo file in docPathFiles)  
{  
    // Work with the FileInfo object here as required  
}
```

■ **Note** When calling the `EnumerateFiles` method, you can also use its overloads to provide a pattern to filter the files that should be returned, and whether to return files from all the subdirectories below the folder as well.

Likewise, you can enumerate through the directories within the folder like so:

```
string docPath = Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments);  
DirectoryInfo docPathInfo = new DirectoryInfo(docPath);  
  
IEnumerable<DirectoryInfo> docPathSubdirectories =  
    docPathInfo.EnumerateDirectories();  
  
foreach (DirectoryInfo directory in docPathSubdirectories)  
{  
    // Work with the DirectoryInfo object here as required  
}
```

■ **Note** Like the `EnumerateFiles` method, the `EnumerateDirectories` method also provides some overloads enabling you to supply a pattern filtering what directories should be returned, and whether to return directories from all the subdirectories below the folder as well.

You can return both files and directories within a folder in a single method call using the `EnumerateFileSystemInfos` method (in the same manner as described for the other two methods), which returns a collection of `FileSystemInfo` objects instead.

■ **Note** You can also access corresponding methods from the `Directory` class (without the need to create a `DirectoryInfo` object first). Both means are completely valid, and the one you use will depend on your personal preference.

Determining Whether a File or Folder Exists

You can determine whether a file exists by using the `Exists` method on the `File` class. For example, the following code determines whether a file named “Test.txt” exists under the user’s Documents folder:

```
string docPath = Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments);  
bool fileExists = File.Exists(Path.Combine(docPath, "Test.txt"));
```

In a similar fashion, you can determine whether a folder exists by using the `Exists` method on the `Directory` class. For example, the following code determines whether a folder named “Test” exists under the user’s Documents folder:

```
string docPath = Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments);  
bool folderExists = Directory.Exists(Path.Combine(docPath, "Test"));
```

■ **Note** Alternatively, you can create a `FileInfo` or `DirectoryInfo` object (passing in the name of the file/directory to its constructor), and check the `Exists` property on that instead.

Reading Data from a File

You can read text or binary data from a file in the same manner as was demonstrated previously when discussing the open file dialog, after obtaining or creating a `FileInfo` object pointing to the file.

Elevating trust permissions opens up a number of other methods of reading data from a file that are somewhat easier to use than the previously described methods. From the `File` class, you now have access to the `ReadAllText`, `ReadLines`, `ReadAllBytes`, `OpenText`, and `OpenRead` methods. Each of these reduces the amount of code required to read data from a file. For example, the following code demonstrates reading the entire contents of a text file into a string:

```
string fileContents = File.ReadAllText(filePath);
```

Saving Data to a File

As with reading data from a file, having elevated trust permissions opens up a number of other easier methods of writing data to a file. From the `File` class, you now have access to the `WriteAllText`, `WriteAllLines`, `WriteAllBytes`, `OpenWrite`, `OpenText`, `CreateText`, and `Create` methods. For example, the following code demonstrates writing “Hello” to a text file:

```
File.WriteAllText(filePath, "Hello");
```

Appending Text to a File

The `File` class also exposes the `AppendText`, `AppendAllLines`, `AppendAllText` methods that enable you to append text to a file. If the given file doesn’t exist, it will create it for you. For example, you may wish to keep a log file containing all of the errors experienced on the client for later analysis. These methods will

enable you to implement that behavior. For example, the following code demonstrates appending “Hello” to a file:

```
File.AppendAllText(filePath, "Hello");
```

Creating a New Folder

You can create a new folder within the user’s profile via the `CreateDirectory` method of the `Directory` class. For example, to create a folder named “Test” under the user’s Documents folder, you would use the following code:

```
string docPath = Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments);  
Directory.CreateDirectory(Path.Combine(docPath, "Test"));
```

■ **Note** Alternatively, you can use the `CreateSubdirectory` method of a `DirectoryInfo` object to achieve the same outcome.

Deleting a File

You can delete a file either via the `Delete` method of a `FileInfo` object, or the `Delete` method of the `File` class. For example:

```
File.Delete(filePath);
```

Copying or Moving a File

You can copy a file to a folder using either the `CopyTo` method of a `FileInfo` object, or the `Copy` method of the `File` class. For example:

```
File.Copy(sourceFilePath, destinationFilePath);
```

Moving a file is a very similar process, using either the `MoveTo` method of a `FileInfo` object, or the `Move` method of the `File` class:

```
File.Move(sourceFilePath, destinationFilePath);
```

■ **Note** The `Copy` method has an overload with an additional `overwrite` parameter that enables you to specify that any existing file in the destination folder with the same name can be overwritten.

Accessing File/Folder Attributes

It's worth noting that once you are running with elevated trust permissions, you are able to access the file attributes (such as creation time, directory, etc) that you were unable to access from the `FileInfo` objects returned from the `OpenFileDialog` class when running in the normal sandboxed mode. However, even with elevated trust, selecting a file located outside the user's profile folders via the `OpenFileDialog` class will still deny access to its file attributes—file attributes will only be accessible if the file resides in the user's profile folders, regardless of trust level.

COM Automation

A somewhat controversial feature added to Silverlight 4 was support enabling COM automation. Most people's memories of dealing with COM are overwhelmingly negative (for the most part), and generally prefer that it remains just that—a memory. A more important argument against support for COM is that it is a Windows only feature—deviating away from Silverlight's cross platform nature.

■ **Note** There has been discussion about implementing similar operating system integration for Macs in future versions of Silverlight, but none is available in Silverlight 4.

However, COM does underpin many of Windows features and that of a number of high profile Windows applications (such as those in the Office suite). Therefore, having COM support in Silverlight opens up a wide range of new possibilities for applications running with elevated trust, some of which will be discussed here.

Determining Whether COM Automation is Available

Before attempting to call COM, you should perform a basic check to ensure that you are actually able to do so. In order to access a COM component, you must be running in OOB mode, with elevated trust, and on a Windows operating system. The `IsAvailable` property of the `AutomationFactory` class (found under the `System.Runtime.InteropServices.Automation` namespace) will return a Boolean value indicating whether these conditions are satisfied, denoting that the application is able to access COM automation.

```
bool canAccessCOM = AutomationFactory.IsAvailable;
```

Accessing COM Components

Once you know that you have access to COM automation, you need to create or obtain an instance of the COM component you wish to automate. You can do so using either the `CreateObject` or the `GetObject` methods of the `AutomationFactory` class. You can use the `CreateObject` method to create a new instance of the COM object, and you can use the `GetObject` to obtain an existing instance of the COM object. For example, if you wanted to interact with a running instance of Word, you could use the `GetObject` method to get that existing instance (returning the most recent instance if there are multiple instances running), or you could use the `CreateObject` method to start a new instance.

■ **Note** This behavior can differ between COM components.

In order to create or obtain a COM object instance, you need to know its programmatic identifier (most commonly known as its progID). Every COM component has a GUID that uniquely identifies that component, known and referred to as a CLSID. This is a unique identifier that the COM component is registered in Windows with. However, this CLSID is not particularly “human friendly”, so each COM component has a corresponding meaningful string that can be used to access it by—its progID. For example, the progID to automate Word is `Word.Application`.

■ **Note** In order to find the progID for a specific COM component, refer to its associated documentation.

Once you have obtained an instance of a COM object, you need to assign it to a variable. However, you have no assembly reference providing the types that you can cast the instance to (since you can't add any sort of reference to those types in Silverlight), creating a problem. Therefore, if you assign the instance to a variable of type `object`, the only way to call methods on the COM object and access properties is via reflection. Help is at hand however, with the dynamic keyword (new in .NET Framework 4 and Silverlight 4).

The dynamic keyword enables you to assign an object instance to a variable and, and call methods/access properties on that object—without needing to cast it to a particular type first. This is particularly useful when interacting with COM components, as you don't know their type, and cannot cast an object instance to one. Essentially, a variable's type is resolved at runtime (as opposed to the `var` keyword that resolves a variable's type at compile time)—a concept known as late binding.

■ **Note** In order to use the dynamic keyword, you need to add a reference to the `Microsoft.CSharp.dll` to your project first.

Putting it all together now, let's create a new instance of a Word application (that has a progID of `Word.Application`):

```
using (dynamic word = AutomationFactory.CreateObject("Word.Application"))
{
    // Put code to interact with Word here
}
```

■ **Note** As a general rule, you should use a `using` block as demonstrated above, in order to ensure that the instance is disposed of properly once you are finished interacting with it.

Obtaining an existing instance of Word is almost identical to the previous example, simply replacing the `CreateObject` method with `GetObject` instead:

```
using (dynamic word = AutomationFactory.GetObject("Word.Application"))
{
    // Put code to interact with Word here
}
```

■ **Note** Attempting to obtain an existing instance of a COM object using the `GetObject` method will raise a generic `Exception` exception if no instance is currently running, stating that it failed to get an object instance for the specified `progID`.

One of the downsides of defining a variable as a dynamic type is that you don't get any help from IntelliSense in the Visual Studio code editor. Therefore, it may be worth writing the code somewhere where you do (such as creating an application using the full .NET Framework and using the Visual Studio Tools for Office to write the initial code, then translating it to run in Silverlight instead)—making the code-writing process much easier and less trial-and-error.

Determining Whether a COM Component is Available

There's no way to tell if a COM component is registered on the system or not, without attempting to create or get an instance of the component. For example, you may want to interact with Word, but the user may not have Word installed on their machine (or may not have the right version of Word).

When attempting to create or get an instance of a COM component, a generic `Exception` exception will be raised, stating that no object was found registered with the specified `progID`. Therefore, always wrap your calls to COM components in a `try/catch` block, as there's no guarantee that the COM component will actually be available on the user's machine.

Using the Windows Script Host Object

The Windows Script Host is a COM component built into Windows that is generally used to script and automate administrative tasks. It contains a number of COM objects that you can use to perform tasks such as running applications, reading from and writing to the registry, and creating shortcuts, to name a few.

For example, you can launch an application using the `Run` method of the `WScript.Shell` COM object. The following code demonstrates launching the Windows Calculator application:

```
using (dynamic shell = AutomationFactory.CreateObject("WScript.Shell"))
{
    shell.Run("calc.exe");
}
```

■ **Note** A full breakdown of the variety of tasks you can perform with the Windows Script Host component can be found in the following MSDN article: [http://msdn.microsoft.com/en-us/library/9y04zt1a\(VS.85\).aspx](http://msdn.microsoft.com/en-us/library/9y04zt1a(VS.85).aspx).

Reading and Writing Files

A complete range of file system access functions are available via the `Scripting.FileSystemObject` COM component built into Windows, without the limitations imposed by Silverlight's built-in functions (that only enable you to access files and folders underneath the user's profile folders).

For example, the following code demonstrates obtaining the path to the user's Program Files folder (which it uses the Windows Script Host described previously to do), creating a sub-folder within that folder, and creating a simple text file in that sub-folder (none of this was possible with Silverlight's built-in file system access classes and methods):

```
using (dynamic shell = AutomationFactory.CreateObject("WScript.Shell"))
{
    // Get the user's Program Files path
    string programFilesPath = shell.ExpandEnvironmentStrings("%ProgramFiles%");

    using (dynamic fso =
        AutomationFactory.CreateObject("Scripting.FileSystemObject"))
    {
        string appPath = Path.Combine(programFilesPath, "MyApp");
        string filePath = Path.Combine(appPath, "Test.txt");

        // Create the folder if it doesn't exist
        if (!fso.FolderExists(appPath))
            fso.CreateFolder(appPath);

        // Create a text file within the folder
        dynamic file = fso.CreateTextFile(filePath, true);
        file.WriteLine("Hello");
        file.Close();
    }
}
```

■ **Note** More information on how to use the `FileSystemObject` component can be found in the following MSDN article: [http://msdn.microsoft.com/en-us/library/6kxy1a51\(VS.85\).aspx](http://msdn.microsoft.com/en-us/library/6kxy1a51(VS.85).aspx)

Automating Office Applications

As discussed earlier, you can automate Word by creating/obtaining an instance of it using its progID of `Word.Application`. The example below demonstrates creating a new instance of Word, adding a document, and inserting some text into it:

```
using (dynamic word = AutomationFactory.CreateObject("Word.Application"))
{
    dynamic document = word.Documents.Add();
    document.Content = "This was inserted by Silverlight";
    word.Visible = true;
}
```

You can automate Excel in a very similar fashion, by creating/obtaining an instance of it using its progID of `Excel.Application`. The example below demonstrates creating a new instance of Excel, adding a worksheet, and inserting some text into its first cell:

```
using (dynamic excel = AutomationFactory.CreateObject("Excel.Application"))
{
    dynamic workbook = excel.Workbooks.Add();
    dynamic cell = workbook.ActiveSheet.Cells[1, 1];
    cell.Value = "This was inserted by Silverlight";
    excel.Visible = true;
}
```

Integration with the Office applications is one of the more common uses that you will have for COM with your business applications in Silverlight. This will allow the user to extract data from the system and manipulate it for their own uses, such as producing reports and letters in Word, and charts in Excel.

■ **Note** When automating the Office applications, always remember to set their `Visible` property to `true`, as they are hidden by default until you do so.

Handling COM Events

You can handle events raised by COM components using the `GetEvent` method of the `AutomationFactory` class. You pass the object and the name of the event to handle to this method, and it will return an `AutomationEvent` object, whose `EventRaised` event will be raised in turn when the COM object's event is raised.

For example, following on from the previous example of opening Excel and creating a new workbook, you could handle the `BeforeClose` event of the `Workbook` object returned from the `excel.Workbooks.Add` method, like so:

```
AutomationEvent beforeClosingEvent =
    AutomationFactory.GetEvent(workbook, "BeforeClose");

beforeClosingEvent.EventRaised += (s1, e1) =>
```

```
{
    System.Diagnostics.Debug.WriteLine("The workbook is closing!");
};
```

This will write a line to the Output window in Visual Studio when the workbook is about to be closed in Excel.

Communicating with Local Databases

As discussed earlier in this chapter, there's no built-in client-side database that can be used by Silverlight to store data locally on the user's machine. However, COM automation enables you to read and write data to an existing database on the user's machine. For example, the following code demonstrates reading data from a local SQL Server database using ADO:

```
using (dynamic connection = AutomationFactory.CreateObject("ADODB.Connection"))
{
    using (dynamic rs = AutomationFactory.CreateObject("ADODB.RecordSet"))
    {
        string connectionString = @"Provider=SQLNCLI10.1;Data Source=.\SQL2008;" +
                                   "Database=AdventureWorks2008;Integrated Security=SSPI";
        connection.Open(connectionString);

        rs.Open("SELECT TOP 5 ProductID, Name FROM Production.Product", connection);

        while (!rs.EOF)
        {
            int productID = (int)rs.Fields.Item("ProductID").Value;
            string name = rs.Fields.Item("Name").Value;

            // Write out the data to Visual Studio's Output window
            System.Diagnostics.Debug.WriteLine("{0} - {1}", productID, name);
            rs.MoveNext();
        }

        rs.Close();
    }

    connection.Close();
}
```

And the following code demonstrates inserting a row into the database:

```
using (dynamic connection = AutomationFactory.CreateObject("ADODB.Connection"))
{
    using (dynamic command = AutomationFactory.CreateObject("ADODB.Command"))
    {
        string connectionString = @"Provider=SQLNCLI10.1;Data Source=.\SQL2008;" +
                                   "Database=AdventureWorks2008;Integrated Security=SSPI";
        connection.Open(connectionString);
```

```

        command.ActiveConnection = connection;
        command.CommandText = "INSERT INTO ErrorLog(ErrorTime, UserName, " +
            "ErrorNumber, ErrorSeverity, ErrorState, ErrorProcedure, ErrorLine, " +
            "ErrorMessage)" +
            "VALUES (GETDATE(), 'UserName', 0, 0, 0, 'None', 0, 'Test Insert!')";
    }
    connection.Close();
}

```

Accessing Local Hardware Devices

With Silverlight 4, you now have built-in access to the webcam, microphone, and printer, but what if you have other hardware devices attached locally that you want to access? If these hardware devices are accessible via a COM component then this is entirely possible. Windows also provides a generic automation layer for some device types that you can take advantage of to gain access to devices. For example, Windows includes an automation layer that enables access to the Windows Image Acquisition (WIA) platform, which provides access to imaging hardware such as cameras and scanners. Features of its associated COM component include enabling you to enumerate the connected cameras and scanners, and take pictures/scan images.

For example, the following code demonstrates displaying the Acquire Image dialog, and saving the acquired image to the user's Pictures folder:

```

using (dynamic acquireDialog = AutomationFactory.CreateObject("WIA.CommonDialog"))
{
    dynamic image = acquireDialog.ShowAcquireImage();

    if (image != null)
    {
        string path =
            Environment.GetFolderPath(Environment.SpecialFolder.MyPictures);

        image.SaveFile(Path.Combine(path, "ScannedImage.jpg"));
    }
}

```

■ **Note** You can get more information about the WIA automation layer from the following MSDN article:

[http://msdn.microsoft.com/en-us/library/ms630827\(VS.85\).aspx](http://msdn.microsoft.com/en-us/library/ms630827(VS.85).aspx)

Creating and Installing your own COM Component

The built-in COM components demonstrated here provide a wide array of additional functionality that you can access from your Silverlight application, but there is also the potential for you to create your own COM component providing further native functionality, and have that downloaded and registered by your application (if the available Windows privileges allow it, or otherwise preinstalled by an administrator).

You can expose classes from projects targeting the full .NET Framework by making their corresponding assemblies COM-Visible. This would enable you to implement features not possible in Silverlight and make them available to Silverlight applications. An excellent article written by Elton Saulsberry on doing so is published here: www.codeproject.com/KB/silverlight/Silverlight4COM.aspx.

The downsides of this particular method is that it requires the user to have the full .NET Framework installed on their machine, in which case the application may be better completely implemented in WPF (for example). Alternatively, you may wish to develop the component using Visual C++ and ATL in order to avoid this requirement. You can find an old (but helpful) article written by C. Lung in 1999 on doing so here: www.codeguru.com/cpp/com-tech/atl/tutorials/article.php/c17.

Custom Chrome

When running your application in OOB mode, it will have the standard window chrome as determined by your operating system's display theme. However, this standard chrome can look quite out of place in comparison to the design of your application's user interface, and it would be nice to be able to implement your own custom chrome that is more integrated with your application's design instead. This wasn't possible in Silverlight 3, but it is now possible in Silverlight 4 when the application has elevated trust permissions.

When your application is configured to run with elevated trust, you can then select the Window Style option that you want to use from the Out-Of-Browser Settings window (shown back in Figure 14-2). The default option is Default (where your application is contained within the standard window chrome), but the other options available include No Border and Borderless Round Corners.

With the default chrome, a very basic Silverlight application running out of browser might look like the example shown in Figure 14-14.

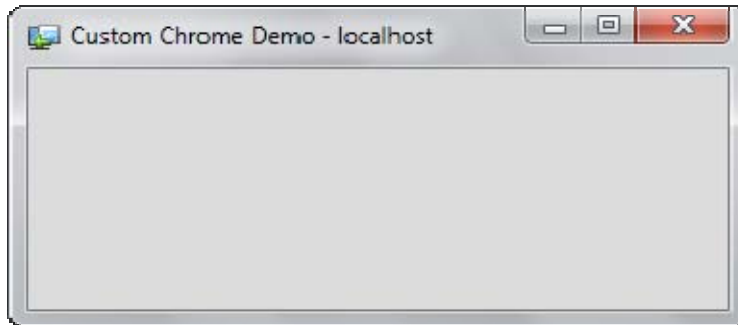


Figure 14-14. An OOB application window with the default window chrome

By setting the Window Style option to No Border, you will get the output shown in Figure 14-15.

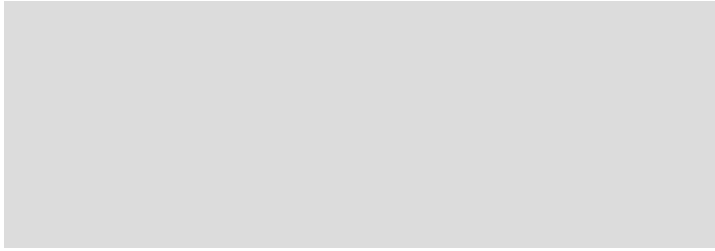


Figure 14-15. An OOB application window with no chrome

This is a very square window shape, whereas most windows these days have rounded corners. You can have a borderless window with rounded corners by setting the Window Style option to Borderless Round Corners. This will give you the output shown in Figure 14-16.

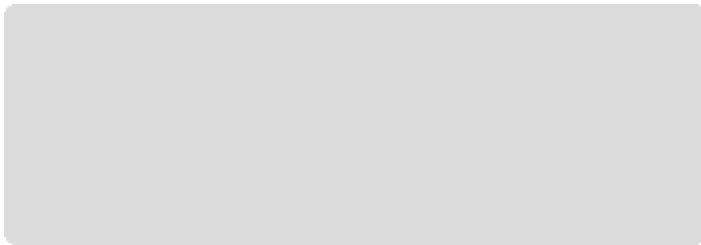


Figure 14-16. An OOB application window with no chrome and rounded corners

One of the issues that you will find when turning on custom chrome is that not only will your application no longer have a border, but also, it will no longer have a title bar either. This means that the user can no longer move the application around the screen, minimize or maximize it, close it, or resize it. You will now have to implement each of these features manually.

Minimizing and maximizing your application's window (ie. setting its state) is easy, as is closing it (each was discussed earlier in this chapter). Similarly, the ability to move and resize the window is also surprisingly easy to implement. Let's implement a custom title bar that enables the user to move the window around the screen, and a resize handle to enable the user to resize the window, as shown in Figure 14-17.



Figure 14-17. An OOB application window with a custom title bar and resize handle

Moving the Window

To enable the window to be moved, let's create a custom title bar using the following XAML:

```
<Rectangle Name="TitleBar" Height="15" Fill="#D8727272" VerticalAlignment="Top"
  MouseLeftButtonDown="TitleBar_MouseLeftButtonDown" />
```

Note that you are handling the `MouseLeftButtonDown` event on the `Rectangle`. You will use this event to initiate the window move, as per the following code:

```
private void TitleBar_MouseLeftButtonDown(object sender, MouseButtonEventArgs e)
{
    Application.Current.MainWindow.DragMove();
}
```

That's all there is to it. The `DragMove` method on the window will tell the window to move based upon the position of the mouse (relative to its current position), and it will automatically continue to do so until the left mouse button is released.

Resizing the Window

To enable the window to be resized, let's create a handle that will appear in the bottom right-hand corner of the window that the user can click and drag to resize it, using the following XAML:

```
<Path Name="ResizeGlyph" Data="M20,0 L20,20 L0,20 z" Fill="#D8727272"
  Grid.Row="2" HorizontalAlignment="Right" VerticalAlignment="Bottom"
  Height="20" Width="20" Cursor="SizeNWSE"
  MouseLeftButtonDown="ResizeGlyph_MouseLeftButtonDown" />
```

Note that you are handling the `MouseLeftButtonDown` event of the `Path`. You will use this event to initiate the window resizing, as per the following code:

```
private void ResizeGlyph_MouseLeftButtonDown(object sender, MouseButtonEventArgs e)
{
    Application.Current.MainWindow.DragResize(WindowResizeEdge.BottomRight);
}
```

This works in exactly the same way as moving the window, where the `DragResize` method on the window will tell the window to resize based upon the position of the mouse (relative to its current position), and it will automatically continue to do so until the left mouse button is released. Note that you need to specify (as the parameter to the `DragResize` method) what edge of the window the user is resizing, such that the resizing can be handled accordingly.

Other Restrictions Lifted By Elevated Trust

Running with elevated trust lifts a number of other restrictions imposed by the sandbox, including:

- When attempting to communicate with a server that isn't the site of origin for the application (ie. communicating cross-domain), you no longer require a cross-

domain policy file to be located in the root of that server in order to do so (as was detailed in Chapter 8: Securing Your Application).

- Attempting to communicate with a server via sockets no longer requires a cross-domain policy file. Also, the application is not limited to only communicating over ports within the range 4502-4534 (as was detailed in Chapter 8: Securing Your Application).
- As described earlier, most keyboard input was disabled while the application was in full screen mode. This will no longer be the case.
- Consent messages will no longer be displayed when the application attempts to access the clipboard, switch to full screen mode, etc. However, the user will have to provide permission for the application to access the video and audio streams.
- Copying to and pasting from the clipboard, displaying open/save file dialogs, and initiating full screen mode no longer need to be user-initiated actions.

Restrictions Imposed by Elevated Trust

Although elevated trust lifts a number of the restrictions placed on Silverlight applications, having elevated trust does impose one restriction on the application that is not placed on standard OOB applications.

Earlier in this chapter we discussed having your OOB Silverlight application check the server for an update, download it, and automatically update it the next time it is started. However, this poses a potential security risk to machines if the application has elevated trust permissions (simply due to the additional access to the user's machine granted to the application). A rogue application update could potentially be put in place designed to harm the user's machine. With the application having elevated trust, it could potentially do so.

Therefore, Silverlight applications having elevated trust will only be able to automatically update if the application is signed (this confirms that the update is sourced from the original application developer, and should therefore be safe to install). When the update has been downloaded, Silverlight will check that the signature of the original application matches that of the update, and will only install the update if they do.

Rather confusingly however, Silverlight applications with elevated trust *will* update if sourced from localhost, even if they aren't signed. This feature was implemented to make testing easier in a development environment, but can lead to frustration when developers deploy their application to a production environment and find out that it won't update (and don't know why).

Therefore, Silverlight applications with elevated trust should always be signed. Application signing is discussed in Chapter 15: Application Deployment.

Disabling Elevated Trust Applications with a Windows Group Policy

System administrators may want to install the Silverlight runtime, but not permit Silverlight applications that request elevated trust permissions to be installed on the machine. This can be implemented via a Windows group policy setting, as detailed here:

www.microsoft.com/GetSilverlight/resources/documentation/grouppolicysettings.aspx#TrustedApplications

Summary

As you can see, Silverlight provides a number of ways to interact with its host's operating system, providing read/write access to isolated storage, files, and the clipboard; installing on the machine and running in a similar fashion to a native application, and even breaking out of its sandbox to run with elevated trust permissions.

By enabling Silverlight applications to break out of the sandbox (and particularly with COM support), a wide range of new possibilities is opened up to applications developed in Silverlight 4. These possibilities narrow the gap between what is and isn't possible to achieve in Silverlight and WPF, and enables Silverlight to be considered for a wider range of development projects.

CHAPTER 15



Application Deployment

Once you have developed your application in Silverlight, you then need to ensure that you deliver it in an effective manner to your users. One of Silverlight's advantages is that it makes it very easy to install and run applications on your end-user's computer. All that the user needs is a browser and a web connection. With just those two things you can have a "clean" PC up and running with a Silverlight application in a matter of minutes, with both the Silverlight runtime and the application deployed to the user's machine automatically (with their permission), and with no machine restarts necessary.

As we all know, first impressions are important. The first impression that the user will have of your application is waiting for it to download, and very possibly requiring the installation of Silverlight itself. Therefore, you need to ensure that they have the best experience possible, as their experience here can greatly shape their opinion and acceptance of the application.

This chapter will take you through how to deploy your Silverlight applications (to both the server and the clients), and we will also take a look at how you can improve on the default installation experience to help entice users to make the leap and install Silverlight (if they don't already have it) in order to run your application.

Deploying the Application to the Server

Depending on the nature of your Silverlight application, it may simply involve a client component (the Silverlight application itself), or it may include both server and client components (both a Silverlight application running on the client, and services running on the server). Both the server and the client components are deployed to the server, and it's the server that then deploys the client component(s) to the clients.

When you deploy a Silverlight application to the server, you are essentially deploying the web project (this consists of both the server and the client components). The process is essentially identical to deploying a traditional ASP.NET web application/web site to a server.

You have three primary options of deploying your application to the server. You can:

- Simply copy the files to the server (and configure the web server manually),
- Use Visual Studio's built-in publishing functionality, or
- Create a setup program.

We'll take a look at these methods shortly, but let's first discuss the requirements for the server.

Server Requirements

Silverlight applications (as a general rule) are downloaded from a server and therefore the server must be able to serve up the application to the client via web server software. In addition (depending on the various needs of your Silverlight application), the application may place additional demands on the server to serve it up data. Let's look at the requirements that your Silverlight application may place on the server, and what it needs installed to meet these needs.

Server Operating System and Web Server

One of the things that many people do not realize is that you *do not* need a Windows-based server (running Internet Information Services—better known as IIS) to host your Silverlight application. Silverlight is a client-based technology, and hence is executed on the client, not the server. Therefore, they simply need to be downloadable from a server, and will be run entirely by the Silverlight runtime on the client, requiring no special handling on the host server. This means that you could potentially host a Silverlight application on a Linux box running an Apache HTTP Server or Apache Tomcat server (for example), or even on cloud-based platforms such as Windows Azure. Essentially, any web server that can serve files over HTTP can be used to deploy Silverlight applications.

That said, as soon as you want to run ASP.NET based services on the server (such as those created using RIA Services) and have your Silverlight application communicate with these, the server needs to be Windows-based (as they will be written to run against the full .NET Framework), and running Internet Information Services (IIS). However, the Silverlight application itself still does not place any requirements on the server.

■ **Note** If you don't have custom web services accompanying your Silverlight business application, but use existing services hosted elsewhere (such as on Windows Azure), then the application does not need to be deployed from a Windows-based server.

RIA Services

As the focus of this book has been around creating end-to-end business applications that use RIA Services, we will assume here that you are also deploying these services as a part of your server deployment. Hence the server must:

- Be a Windows-based server
- Be running Internet Information Services (IIS)
- Have the .NET Framework 4 installed (full profile)

Installing RIA Services Assemblies to the GAC

You can choose to deploy RIA Services to the Global Assembly Cache (GAC) so that it's accessible to all web applications, or you can distribute these as a part of your deployment. By default, the RIA Services

assemblies referenced by your web project are not copied to your Bin folder, and hence are not deployed with your application when you publish your web site to the server.

To deploy RIA Services to the GAC, start by downloading the RIA Services installer from www.silverlight.net/getstarted/riaservices/. If you tried running this installer, you will probably find that it complains about requiring both Visual Studio 2010 and Silverlight 4 to be installed on the machine (neither of which the server will generally have). However, you can type the following line at the command prompt to perform a server install, which will only install the components required by a server:

```
msiexec /i RIAServices.msi SERVER=true
```

Installing RIA Services Assemblies to the Bin Directory

On a hosted server, you most likely will not be able to deploy the RIA Services assemblies to the GAC, but you can deploy the RIA Services assemblies required by your application along with it to its Bin directory. Simply select all the assemblies starting with `System.ServiceModel.DomainServices` in the Solution Explorer window for your web project, and set their “Copy Local” property (in the Properties window) to True. They will be copied into the Bin directory in your project, and deployed when you publish the web project to the server.

■ **Note** If you encounter RIA Services related issues when attempting to run your Silverlight application, Saurabh Pant (a member of the RIA Services team) has a detailed blog post on troubleshooting these, which should be your first port of call: <http://blogs.msdn.com/b/saurabh/archive/2010/03/16/ria-services-application-deployment.aspx>.

Cross-Domain Policy File

If your services are deployed to a different location from where your Silverlight application will be downloaded from (this leads to cross-domain issues, as discussed back in Chapter 8: Securing Your Application), you will need to place the cross-domain policy file in the root of the web site containing the services in order for the Silverlight application to be permitted to access them. See Chapter 8 for more information on the required contents of this file and where it needs to be placed on the server.

Using Xcopy

The most primitive means of deploying your application to a server is by copying the required files to the required location (a process often referred to as xcopy, from the old DOS command), and configuring an Application in IIS to point to the folder that these files were copied to.

Copy all the non-code related files from the web project (the HTML file hosting the Silverlight plug-in, any JavaScript files, the web.config file, the ClientBin folder, the Bin folder, and so on.) to a folder on the server. This may be a new folder, or you can add to/update a folder containing an existing web site.

■ **Note** The quickest way to determine what files are required to be copied to the server is to publish your project to a folder on your local hard drive (using the procedure detailed in the next section), and then copy all the contents of that folder to your server. When doing this, ensure that the “Items to deploy” option in the Package/Publish Web tab in the project properties has been set to “Only files required to run this application”.

If you are creating a new web site, then you need to configure an Application in IIS. How you go about doing so will largely depend on the version of IIS that it will be running under, and the type of user interface available to you (i.e., either the Internet Information Services Manager, or a web-based user interface). Therefore, we will not go through a step-by-step process for this in detail, but it will follow the same process as configuring an ASP.NET 4 web site.

Publishing

Visual Studio 2010 provides a feature enabling you to publish a web site in a myriad of ways to a server—both local and remote. Depending on the settings you are using, publishing a web site using Visual Studio will both copy the required files to a server, and create the Application for you in IIS (if it’s not already configured). This feature makes it easy to deploy your project from right within Visual Studio.

Start by right-clicking on your web project in the Solution Explorer window, and select the “Package/Publish Settings” item from the context menu. This will open the Package/Publish Web tab in the project properties, and enable you to configure what will be deployed to the server as a part of the publishing process, and where on the server it will be published.

Once you have configured the settings appropriately, right-click on the web project in the Solution Explorer window again, and select the Publish item from the context menu. This will display the Publish Web dialog shown in Figure 15-1.

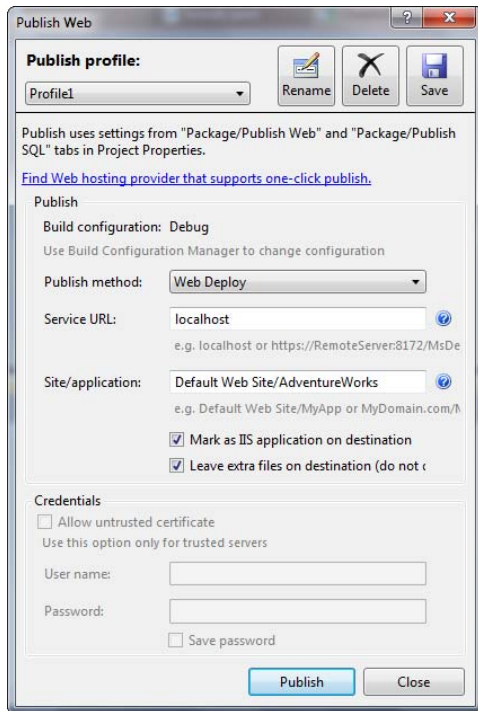


Figure 15-1. The Publish Web dialog

You can then configure how you want to publish your web site in this dialog, and click the Publish button to start the publishing process. It will copy the files used in your web project (that will include your Silverlight application), and will create the Application in IIS (if you have selected the “Mark as IIS application on destination” option in the Publish Web dialog).

■ **Note** You can publish your project to the file system, and then xcopy these files to the server if required.

Creating a Web Package

Instead of publishing the web site directly to the server, you can instead create a web package, which is essentially a .zip file containing your web site and configuration files dictating how the web site should be created under IIS on the server. You can then take this package and import it into IIS (either using the Internet Information Services Manager, or via the command line using the msdeploy tool).

■ **Note** You can get more information about the msdeploy tool from here:

<http://blogs.iis.net/msdeploy/default.aspx>

To create a package, right-click on your web project in the Solution Explorer window, and select the “Build Deployment Package” item from the context menu. This will immediately compile the web site into a package, using the settings configured in the Package/Publish Web tab in the project properties for the web project.

You can then open up the Internet Information Services Manager and import the package. This will install the files on the server, and create the Application in IIS. Alternatively, if you want to import the package via the command line, you can find instructions on doing so in a text file created in the same folder as the package.

Creating a Setup Program

If your application is to be a packaged product sold to companies who then install it on their own server, creating a setup program is the best way to provide it to your customers to install.

Add a new project to your solution, and use the Web Setup Project project template (you will find this under the Other Project Types > Setup and Deployment > Visual Studio Installer category).

In the Solution Explorer window, right-click on the setup project, and select Add ➤ Project Output. In the dialog that appears, select the web project from the Project drop-down, and select both the “Primary Output” and the “Content Files” outputs from the list. Press OK.

Select the setup project in the Solution Explorer window again, then go to the Properties window and assign values to the properties, as you require. Once you are done, you can compile the project.

You can test the setup program locally by compiling your project, then right-clicking on the setup project in the Solution Explorer window and selecting the Install item from the context menu.

Deploying the Application to the Client

One of the great features of Silverlight is that you can go from a “clean PC” (a machine without Silverlight installed or the application downloaded), to being up and running with the application in a matter of minutes. This makes Silverlight a very effective means of deploying applications to client machines.

When the client navigates to a URL hosting a Silverlight application, you can structure the HTML page that hosts the Silverlight plug-in to notify the user if they don’t have Silverlight installed, and point them to a link where they can download the Silverlight runtime and install it. Once the Silverlight runtime is installed, the HTML page will be automatically refreshed and the download of the application will begin, running it once the download is complete.

As you can see, deploying a Silverlight application to the client is a two-step process. The process involves:

- Downloading and installing the Silverlight runtime (if it’s not already installed)
- Downloading the application if it’s not already downloaded (or is not up to date)

Let’s take a closer look at how this process works, how to customize it to create a more streamlined user experience, and other methods you could use to deploy the Silverlight runtime and your application to the users.

The Default Silverlight Installation Process

Let's start by taking a look at what happens when the user attempts to run your Silverlight application, but doesn't have Silverlight installed. Let's assume that no version of Silverlight is currently installed on the user's machine, and that they have navigated to the URL for your application. The user will be presented with a blank page, containing the image shown in Figure 15-2.



Figure 15-2. The default image displayed in the browser when Silverlight is not installed

If you view the source of the HTML page that's used in your application to host the Silverlight plug-in (as was detailed back in Chapter 2: Getting Started with Silverlight), you will find the following lines of HTML in its body:

```
<object data="data:application/x-silverlight-2," type="application/x-silverlight-2"
  width="100%" height="100%">
  <param name="source" value="ClientBin/AdventureWorks.xap" />
  <param name="onError" value="onSilverlightError" />
  <param name="background" value="white" />
  <param name="minRuntimeVersion" value="4.0.50401.0" />
  <param name="autoUpgrade" value="true" />
  <a href="http://go.microsoft.com/fwlink/?LinkId=149156&v=4.0.50401.0"
    style="text-decoration:none">
    
  </a>
</object>
```

If the browser does not know what the object is supposed to be (because the Silverlight plug-in hasn't been installed), it will ignore the object tag and simply display the content between the opening and closing object tags instead. The default HTML between the tags (in bold) simply displays the image you see in Figure 15-2, and when it is clicked on will download the Silverlight plug-in from the Microsoft web site.

■ **Note** We'll look at improving this default Silverlight installation experience to make it somewhat friendlier and more enticing by swapping out this default "install Silverlight" HTML with your own content later in this chapter.

You will also note the following line in the head section of the HTML file:

```
<script type="text/javascript" src="Silverlight.js"></script>
```

This script contains a number of helper functions, such as automatically reloading the page once it recognizes that the Silverlight plug-in is installed, and additional functions to enable you to create a Silverlight object in a web page via JavaScript.

Once the Silverlight runtime has been installed, the page will be automatically refreshed, and the application will begin to download from the server (and be displayed once the download is complete).

■ **Note** If an earlier version of Silverlight has been installed on the user's machine but your application is targeting a more recent version, the new version will be downloaded and installed automatically, but the user will have to restart their browser. The Silverlight plug-in is backward compatible, meaning that any Silverlight application written in a previous version of Silverlight will run without issues in a more recent version of the plug-in. Additional Silverlight versioning information can be found here: [http://msdn.microsoft.com/en-us/library/cc265156\(VS.95\).aspx](http://msdn.microsoft.com/en-us/library/cc265156(VS.95).aspx).

Customizing the Silverlight Installation Experience

You saw in the previous section how easy it was to get set up and running with Silverlight on a clean PC. However, despite the ease that you can deploy your applications using Silverlight, the default experience isn't particularly ideal. Let's take a look at why you should customize the Silverlight installation experience, and how you can do so.

Why You Should Customize the Silverlight Installation Experience

When a user navigates to your application, the last thing they want is something blocking their way (like the installation of a plug-in). Depending on their level of commitment to running your application, it opens up a barrier for the user, providing a decision point that may lead to them deciding to abandon the attempt and turn away. You really want to avoid this possibility at all costs.

■ **Note** This is not such an issue with standard business applications that are running in corporate environments. The IT department will have bought into Silverlight as the technology running one or more of their applications, it is likely to have already been deployed to user's PCs, and the use of the application will have been mandated by the company management. However, there are many other scenarios where this issue should be considered. If you are providing your application as software as a service (SAAS) or a packaged product, then this will be an issue. Alternatively, if you have users working from home, it's important that you provide a low friction install experience for getting them up and running with your application.

Unfortunately, the default “Silverlight requires installation” experience does nothing to help the issue of getting user buy-in. The simple image and link (shown previously in Figure 15-2) fails at the following points:

- It tells the user nothing about:
 - Your application or your business
 - Why your application needs Silverlight
 - Why they should install Silverlight
 - What the value that installing Silverlight provides
 - Instructions for timid users to walk them through how to complete the installation
- It does not:
 - Provide any preview that demonstrates what the user will get on the other side of the installation process
 - Attempt to engage and entice the user, why it’s worth going through the process of installing Silverlight in order to run your application
 - Maintain consistency with the rest of your web site or the Silverlight application. By not providing any visual consistency, its appearance will be jarring to the user, thus leading to a sub-optimal user experience
 - Make any attempt to ease any fears that the user might have
- It does not provide any background information on Silverlight, such as:
 - Link to MS Silverlight site
 - What it will (or will not) do to their computer
 - What Silverlight is
 - Whether Silverlight can be easily uninstalled
 - How many major web sites use Silverlight, and that the user will gain access to these sites as well
 - Why the plug-in can be trusted (apart from the Microsoft logo). For all the user knows, this could be an advertisement designed to trick them into installing something malicious, or that they don’t need/want
 - Why the user may not be able to install Silverlight on their machine. For example, they may be running an unsupported operating system or browser. In addition, that the user will need administrative rights on their machine to be allowed to complete the installation

You should personalize the installation experience, tailoring it to your business and the application, and entice the user to work through the minor impediment of not having Silverlight in order to run your application. Ideally, if you had a designer work on the design of your application, you should also have them design the Silverlight installation and application deployment process too, in order to maintain

consistency in all three phases (Silverlight installation, application deployment, and application execution).

It's important to think like a user, and try to identify with them and their psychology when they first use your application. There are many classes of users, including those who will:

- Click on and install anything that gets them where they want to go
- Be fearful of using computers and what they install on them
- Have short attention spans and give up quickly if they come to any sort of barrier

Take these behaviors into account and design your Silverlight installation experience accordingly.

■ **Note** As the popularity of Silverlight grows, these initial installation issues will become less of a problem. However, you will need to support users with a freshly installed operating system (and therefore will not have Silverlight installed yet). Also, with the regular releases of new Silverlight versions, it's always likely that the user will be running an older version of Silverlight than what you are targeting. As discussed back in Chapter 1: Introduction, you can get a rough idea of the reach of a particular version of Silverlight at a site such as www.riastats.com. At the time of this writing, this site is reporting that a little over 60% of Internet-connected devices have a version of Silverlight installed, with the vast majority of these installs already being Silverlight 4.

Seeing What the User Sees Without Silverlight Installed

The first step in customizing the Silverlight installation experience is to see what the user sees in the browser if they don't have Silverlight installed. You can either:

- Uninstall Silverlight from your machine using the operating system's uninstall application tool (such as Windows Vista/7's Programs and Features tool), or
- Simply disable the Silverlight plug-in in your browser of choice. For example, in Internet Explorer go to Tools ⇨ Manage Add-ons, select the Microsoft Silverlight plug-in, and then click the Disable button. Now when you navigate to the URL hosting your application, it won't be able to load the Silverlight plug-in, and thus will enable you to test what will happen when users don't have Silverlight installed

If you then open the Manage Add-ons dialog again and re-enable the Silverlight plug-in and close the dialog again, you will note that within a second or two the page will be automatically refreshed and your application will be loaded. This is thanks to the `Silverlight.js` file that is referenced in the HTML/ASPX page hosting your application (as was detailed earlier) and distributed with your application. When it detects that the Silverlight plug-in isn't installed, it will continually monitor for it until it is found (when the user has successfully installed the Silverlight plug-in), at this point it will automatically refresh the page in order to load your application. Therefore, all you really need to focus on is the designing an effective landing page.

Implementing a Custom Landing Page

To implement a custom-landing page, you simply need to replace the following lines of HTML between the opening and closing object tags (discussed earlier in this chapter) in the page that is hosting your Silverlight application with your own custom HTML:

```
<a href="http://go.microsoft.com/fwlink/?LinkId=149156&v=4.0.50401.0"
    style="text-decoration:none">
    
</a>
```

Ideally, you would replace it with some HTML that will help address each of the issues listed in the previous section (detailing why you should customize the Silverlight installation experience), but still retain the link to download the Silverlight runtime from the default HTML (not necessarily the image however). Try to retain as much consistency as possible with the rest of your web site structure and style, make sure the user knows what is happening and what they need to do, and brand it with your company and/or application.

Note that if you want to provide the users step-by-step instructions on installing Silverlight, each browser requires slightly different steps to be taken, and each display different dialogs. Ideally, you would recognize the browser type on the server, and provide the corresponding instructions accordingly. For this purpose, you can use the .aspx page (instead of the .html page), determine the browser type and version on the server (using this article as a guide: <http://msdn.microsoft.com/en-us/library/3yekbd5b.aspx>), and output your instructions to the page (between the opening and closing object tags) accordingly.

■ **Note** It can be a good exercise to disable the Silverlight plug-in using the method described earlier, and investigate how other Silverlight applications customize their page notifying the user that they need to install Silverlight. You will note how much of a difference it can make to the Silverlight installation experience just by properly customizing this page. Good examples of custom Silverlight install experiences that are well executed include the www.Silverlight.net web site, and the Netflix Watch Instantly web site.

Further Guidance

A great source of information regarding creating custom Silverlight installation experiences, and a must-read when creating your own, is the Silverlight Installation Experience Guidance whitepaper, which you can find a link to on this site: www.silverlight.net/learn/whitepapers/. This is an extremely detailed document that can be used to guide you in creating a functional and effective custom Silverlight installation experience, and also has a template (in the accompanying download) that you can use.

Pushing the Silverlight Runtime to Users in the Enterprise

One of the issues in deploying the Silverlight runtime (particularly in enterprise environments) is that the user will need to be a local administrator on their machine in order to install the Silverlight plug-in

(as is the case with any plug-in). If the user doesn't already have the Silverlight runtime installed on their machine and does not have local administration rights, they won't be able to install the Silverlight plug-in (and hence run your application).

■ **Note** Worth clarifying, the user *does* need to be a local administrator on their machine in order to install the Silverlight *runtime/plugin*. However, they *don't* need to be a local administrator on their machine in order to install a Silverlight *application* (to run outside the browser).

The Silverlight runtime can however be pushed out to users using the Microsoft Systems Management Server automated installation, or by setting up a group policy. Despite being for Silverlight 2.0, the Silverlight Enterprise Deployment whitepaper available from www.silverlight.net/learn/whitepapers/ is a very thorough and detailed guide for methods of deploying Silverlight in enterprise environments.

Building a Client Desktop Installer

In Chapter 14: Out Of Browser Mode, and Interacting with the Operating System, we discussed how to install your Silverlight application to run outside the browser, in much the same way as a standard (native) desktop application. In some scenarios, you may find that web deployment of your application is not always suitable or even possible. In this case you will need to have a more traditional setup program that can be run on the client and install the Silverlight application as an out-of-browser application. You can do this using the silent install feature of the Silverlight launcher (sllauncher.exe).

■ **Note** Unfortunately the Silverlight runtime has not been packaged into a merge module, so it can't be included as a part of your setup program, and will need to be installed separately. This method therefore assumes the client already has the Silverlight runtime installed. You can download the Silverlight runtime installer from the www.silverlight.net web site.

The sllauncher.exe application accepts various command-line parameters, which can be used to install/uninstall/run a Silverlight application.

If you type the following line on the command line:

```
"%ProgramFiles%\Microsoft Silverlight\sllauncher.exe"
```

It will display the message box shown in Figure 15-3.

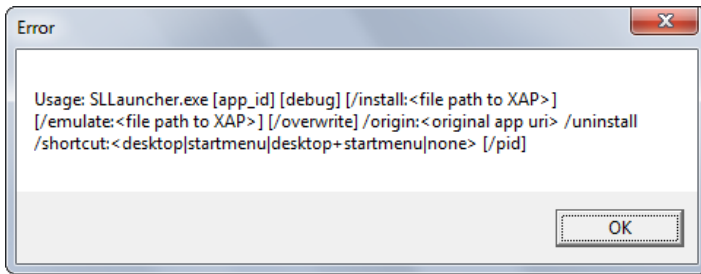


Figure 15-3. The SLLauncher command-line parameters dialog

■ **Note** These examples will only work under Windows, and you will need to change %ProgramFiles% to %ProgramFiles(x86)% when running under the 64-bit edition of Windows in this and the following examples.

This tells you what command-line parameters you can take advantage of to enable us to install the application on the machine. Of particular interest are the following options:

- /install: the path to the XAP file (required)
- /origin: the URI specifying the site-of-origin for the application (ie. where it was downloaded from). Even though the application was not downloaded from the Web, this is a required parameter and must be assigned a URI
- /shortcut: what shortcuts should be created - start menu, desktop, both, or none (optional)
- /overwrite: when used will overwrite any existing version already installed (optional)

An example of installing an application using these options is below:

```
"%ProgramFiles%\Microsoft Silverlight\sllauncher.exe" /install:"AdventureWorks.xap" /origin:"http://localhost/AdventureWorks.xap" /shortcut:desktop+startmenu /overwrite
```

You can take advantage of this silent install feature in your own setup programs by adding the .xap file from your Silverlight project to your setup project, and creating a custom action to extract your .xap file to a temporary location and running sllauncher.exe with the appropriate command-line parameters to install it. A full explanation of this procedure however is beyond the scope of this book. Alternatively, a somewhat simpler option may be to deploy a batch file along with the application, which when executed, will install the application using the method described here.

Improving on the Default Application Loading Screen

After improving the Silverlight installation experience, the next step is to improve the experience for your users while your Silverlight application is being downloaded from the user. You are no doubt

familiar by now with the blue spinning balls animation that is displayed while your application is being downloaded (as shown in Figure 15-4), which provides a generic indicator that something is currently happening. If after half a second the application is still downloading, Silverlight will automatically display this default loading animation until the application has finished downloading.

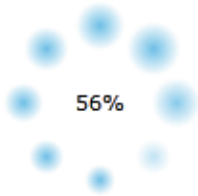


Figure 15-4. The default “Downloading Application” animation

However, as with the default Silverlight installation experience, having no branding and no personalization displayed to the user after five seconds or more of waiting is not an ideal user experience. You can however replace this animation with your own custom one, commonly referred to as either a pre-loader or a splash screen.

An application pre-loader is essentially a XAML page that is requested from the server by the Silverlight plug-in prior to the application itself. Designed to be very small (much smaller than the application), it will be downloaded first and be displayed while the main application continues to download. Whereas the main application will be compiled and compressed into a .xap file, the pre-loader is not a part of the Silverlight project—instead it is a standalone XAML file (added to your web project) and is therefore not compiled into a .xap file. This also means that you can’t write any code-behind against the CLR, however you can make use of the JavaScript API (as was used back in Silverlight 1.0) to control the user interface, which can be written in a .js file and downloaded along with the XAML file.

Let’s take a look at how to create and test an application pre-loader.

Creating the Application Pre-loader Files

To create your pre-loader, start by adding a new item to your *web* project, using the Silverlight 1.0 JScript Page item template (found in the Silverlight category), and name it AppPreloader.xaml. This template will actually create two files—a XAML file (with the name you specified in the Add New Item window), and a JavaScript file (with the same name as the XAML file, but with a .js extension).

■ **Note** Follow through these instructions step by step, without jumping around, trying to compile, and so on. This is very important. Unfortunately, the process isn’t particularly straightforward (until you fully know what you’re doing), and not going through each step one at a time is likely to lead to confusion and frustration.

Once the file has been created and opened, you’ll note that there is no XAML designer support. However, if you close the file and open it again, the designer will appear (but will display an error due to it not supporting the referenced default namespace used by Silverlight 1.0).

Delete the contents of both the XAML file and the JavaScript file, as it's just the files you need—not their contents. You will be using a different default namespace than that used in the XAML file created for us, and trying to use the default namespace from the item template will end up just creating problems and confusion for you. Likewise, none of the contents of the JavaScript file are required—you'll add your own code shortly.

Now add the following XAML to the XAML file:

```
<Grid xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">

</Grid>
```

■ **Note** There is no IntelliSense in the XAML editor as yet. This will come once the next step has been performed.

Click the “Reload the designer” button in the designer. Attempting to compile the solution now will result in the following error: “Project file must include the .NET Framework assembly 'WindowsBase, PresentationCore, PresentationFramework' in the reference list”. With the designer open now, you can have these references automatically added to your project by simply dragging a TextBox from the Toolbox into the designer. Alternatively you can simply type an open angle bracket (<) in the XAML editor (with the designer open), which will have the same effect. You should now be able to successfully compile the project.

Designing the Application Pre-loader

Now that you have the required files properly configured, you can start adding the content you want to display in the pre-loader. One of the most important points when creating a pre-loader is to try and keep it very small. As a general rule, try to keep it less than 5 percent of your application's total size, and no more than 10KB so that it can download and be displayed to the user as quickly as possible, even on slow connections.

Almost all pre-loaders display an indicator to the user showing the current progress of the application download. This enables the user to be aware that something is actually happening, and how much longer it may take. The Silverlight plug-in raises an event at a regular interval (that you can handle in the JavaScript file), reporting the current progress of the application download. The pre-loader should also display some branding for both the company and the application (one of the main reasons for undertaking this exercise).

■ **Note** When implementing this pre-loader you will need to stick to using only the controls in the *core* Silverlight runtime. Essentially you are limited to using Silverlight 1.0 controls only. Controls available for you to use include the Canvas, Grid, StackPanel, InkPresenter, MediaElement, TextBlock, Image, Rectangle, Line, Ellipse, Path, Polygon, and Polyline. However, other controls such as the ProgressBar (which would be very useful in a pre-loader), Button, and so on are unfortunately not available.

Figure 15-5 demonstrates the pre-loader you will be creating.

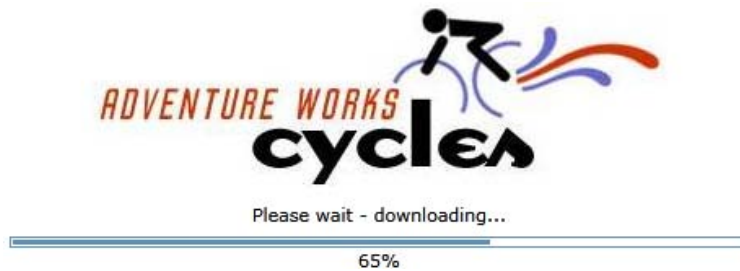


Figure 15-5. The custom pre-loader

As you can see, it displays an image with the company logo, tells the user what it's doing, and displays a progress bar and the progress as a percentage. The XAML required to create this pre-loader is below:

```
<Grid xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
  <StackPanel VerticalAlignment="Center">
    <Image Width="352" Height="111" Source=" ../Logo.png" />
    <TextBlock HorizontalAlignment="Center" Margin="0,10,0,5"
      Text="Please wait - downloading..." />
    <Grid HorizontalAlignment="Center">
      <Rectangle Stroke="#FF5F91B4" Height="7" Width="454" />
      <Rectangle Fill="#FF5F91B4" Margin="2,0,0,0"
        Height="3" Width="450" HorizontalAlignment="Left">
        <Rectangle.RenderTransform>
          <TransformGroup>
            <ScaleTransform x:Name="ProgressBarTransform" ScaleX="1"/>
          </TransformGroup>
        </Rectangle.RenderTransform>
      </Rectangle>
    </Grid>

    <TextBlock Name="ProgressText" Text="0%"
      HorizontalAlignment="Center" Margin="0,0,0,5" />
  </StackPanel>
</Grid>
```

Unfortunately the `ProgressBar` control isn't available for us to at this stage (as it's not in the core Silverlight runtime), so you need to create your own using the primitive controls available to us. You have a rectangle that forms the outer border of the progress bar, and an inner rectangle that is used as the progress indicator. The trick for this is that you create the progress indicator rectangle setting its width as it will be when displaying a value of 100 percent, but apply a scale transform that will be used to scale its width in proportion to the progress value it is indicating. By altering the transform's `ScaleX` property with values between 0 and 1 (in JavaScript code), you will be able to display the progress from 0 percent to 100 percent accordingly—providing an elegant way to implement the progress bar without needing to worry about calculating the required width in pixels in the JavaScript code. You've given the

transform a name so you can refer to it in the code and adjust the scale accordingly (as you will see in the next section).

You will also note the use of the Image control. The `Logo.png` file it refers to is a file in the web project, and you are using a relative path to specify its location. Don't be concerned about the fact that the image doesn't appear in the XAML designer—it may not appear there, but it will appear when the pre-loader is being displayed at runtime.

■ **Note** As with the advice given earlier, you should ensure that any images used are as small as possible, and keep the entire pre-loader and its resources under 10KB (if you can). You may like to try running the image through a tool that shrinks the size of images, as these can result in a drastic size reduction. Some of these tools are online tools (and therefore easy to try), such as www.imageoptimizer.net.

Updating the Application Download Progress

Now that you have the visual aspects of your application pre-loader in place, you need to update the progress being displayed. This involves updating both the progress bar indicator's scale transform, and the text displaying the progress as a percentage.

The Silverlight plug-in has two events it raises that you can handle:

- `OnSourceDownloadProgressChanged`
- `OnSourceDownloadComplete`

The `OnSourceDownloadProgressChanged` event will be raised on a regular basis, and you can use the progress property of the `EventArgs` object passed into the handler as a parameter to determine the progress. This will be a value between 0 and 1, so you can assign this value directly to the `ScaleX` property of the progress bar indicator's scale transform, and multiply it by 100 to display the value as a percentage.

The code below should be added to your JavaScript file in order to update the pre-loader when the `onSourceDownloadProgressChanged` event is raised:

```
function onSourceDownloadProgressChanged(sender, EventArgs) {
    var progress = Math.round((EventArgs.progress * 100));
    var ProgressBarTransform = sender.findName("ProgressBarTransform");
    var ProgressText = sender.findName("ProgressText");

    if (ProgressBarTransform != null)
        ProgressBarTransform.ScaleX = EventArgs.progress;

    if (ProgressText != null)
        ProgressText.Text = progress + "%";
}
```

■ **Note** This event handler will be wired up in the next section.

The `OnSourceDownloadComplete` event is called once the application has completed downloading but is rarely used, as the application will load immediately after it's raised.

Configuring the Application Pre-loader in the HTML File

You now have your application pre-loader's design in place, and the code to update it when the application download progress changes, but now you need to actually wire both of these things up so that the Silverlight plug-in knows about them.

Start by opening up the HTML page that hosts the Silverlight plug-in. In the head tag, add the following reference to the JavaScript file:

```
<script type="text/javascript" src="AppPreloader.js"></script>
```

Now locate the Silverlight plug-in, and add the following two parameters to it:

```
<param name="splashscreen" value="AppPreloader.xaml"/>  
<param name="onSourceDownloadProgressChanged"  
  value="onSourceDownloadProgressChanged" />
```

These two parameters connect everything together, and you're now ready to test it all.

■ **Note** Ensure that the HTML or ASPX page you are modifying is actually the file set as the start page for your web project. Otherwise you may face some confusion wondering why the pre-loader isn't being displayed, when you are simply browsing the wrong file when you run your application.

Testing the Application Pre-loader

Testing the application pre-loader may initially seem like a simple enough task, however it's a deceptively complex process. The issue is that when running your application locally, the download process will generally be extremely fast, and the pre-loader will not have time to actually be displayed (or will be displayed only momentarily). This obviously makes it difficult to test and debug. However, there are some tricks you can use to slow its display down. Let's take a look at some of these.

■ **Note** You may often get errors such as "Failed to download the splash screen" or "Could not download the Silverlight application. Check web server settings" when debugging your application locally. Alternatively you may get a blank screen, with neither the pre-loader nor the application itself loading. This appears to be an issue with the application download in this scenario being faster than it takes to download and display the pre-loader. This should not be a problem in production environments however.

Adding a Large File to the Silverlight Project

The easiest thing you can do to try and slow the downloading process and give your application pre-loader some time to be displayed is to add one or more large files to your Silverlight project in order to increase its size. The larger the project, the longer it will take to download. Simply add a large file to your project and set its Build Action property to Content (select the file in Solution Explorer, and then set the property value in the Properties window). This is a simple way of slowing the download process, but not particularly the most effective.

Not Assigning the Application's RootVisual

The application pre-loader is displayed until the RootVisual property of the Application object is assigned. Therefore, if you don't assign a value to this property, the pre-loader will continue to be displayed indefinitely.

In your Silverlight project, open up the code-behind for the App.xaml file, find the Application Startup event handler method, and comment out the following line of code:

```
this.InitializeRootVisual();
```

Now when you run your project, the pre-loader will continue to be displayed, without displaying your main application. This trick won't help you in testing your progress bar, but will help in testing the look and feel of your pre-loader.

■ **Note** Another option is to sleep the thread (using the `System.Threading.Thread.Sleep` method) for a fixed period immediately prior to initializing the RootVisual.

Using Fiddler

Fiddler is a web debugging proxy that intercepts all incoming and outgoing HTTP/HTTPS connections, enabling you to log and modify the traffic. You can use Fiddler to slow down the downloading of the application, giving us time to view and test the pre-loader.

■ **Note** You can download Fiddler for free from www.fiddler2.com

When you open Fiddler, it will immediately start capturing the traffic. Select both the Simulate Modem Speeds and the Disable Caching options from the Rules ► Performance menu. This will slow down the traffic to simulate 56k modems speeds, and also disables caching such that when you refresh the page the application will need to be downloaded from the server again (enabling you to test your pre-loader again without the application being immediately loaded from the browser cache).

Unfortunately, simply running your application is not enough for this all to work. Fiddler bypasses capturing local traffic, but there is a trick that you can use to ensure that it intercepts your application.

Change the use of “localhost” in the URL in your browser’s address bar to “127.0.0.1.” (the localhost loopback IP address, including the final dot). This will trick Fiddler into capturing the traffic, and you will find that the application now downloads slowly enough for you to adequately test your pre-loader. Once the application has loaded, you can simply refresh the page to have it downloaded again (as it will not be cached).

Partitioning Your Application

The next issue that you should target is the size of your application. As soon as you start referencing various assemblies and adding assets to your application, it can rapidly increase in size. It will depend on the nature of your application as to whether this is an issue you should be concerned about or not. In an internal corporate environment where the application will be downloaded from a server located on the local network, this is generally not a particularly big issue. However, if your application is being deployed over the Web, then it becomes something that you should pay closer attention to. The larger the application, the longer it will take for the user to download it and get it up and running, and the more bandwidth it will require (for both you and the user). It provides more of a barrier for the user to use your application, and gives them another reason to abandon the attempt. Let’s take a look at some strategies you can implement to reduce the size of your application, or partition it into modules that can be downloaded on demand.

Initial Steps

The first step you should take is to look at the size of your .xap file and decide on how acceptable its size is. You may want to use the feature of Fiddler, discussed previously, to simulate modem speeds in order to determine how acceptable the time it takes to download may be to various users.

The next step is to ensure you aren’t referencing any assemblies or assets not actually being used by your application. If your .xap file is still large then it may be worth analyzing its contents (by changing its extension to .zip and unzipping it, as discussed previously in Chapter 2: Getting Started with Silverlight). Look for the largest files and determine if you can remove them or replace them with a suitable alternative.

Assembly Caching

One thing that you can do to partition your application into smaller pieces is to turn on assembly caching for your application. You will find this in the “Reduce XAP size by using application library caching” option in the project properties for your Silverlight project. What this will do is instead of creating a single large .xap file containing all the assemblies required by your application (that aren’t a part of the core Silverlight runtime), it will move all the assemblies (that have an accompanying extension mapping configuration file, discussed shortly) out of this file and into separate .zip files, as you can see in Figure 15-6.

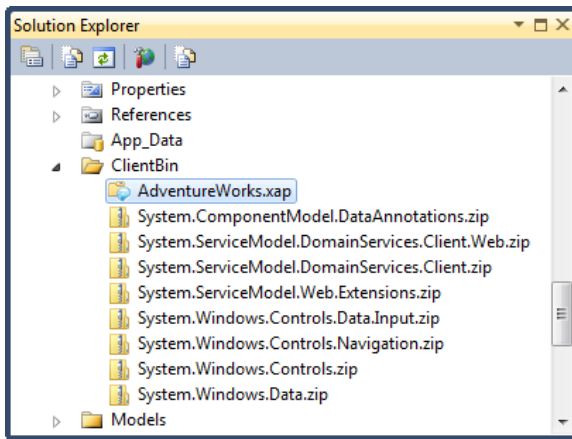


Figure 15-6. The ClientBin folder when using assembly caching

Why Use Assembly Caching

This will result in the .xap file becoming much smaller. It won't actually reduce the initial size of your application, but it provides a big advantage when it comes to deploying updates to your application. Without assembly caching turned on, each time you deploy a new version of your application, the user needs to download the entire application from the server again (including the assemblies referenced by your application that haven't changed).

By turning assembly caching on, when you deploy an update to the application to the users, only the .xap file containing the application itself will need to be downloaded again. The assemblies referenced by your application that remained unchanged will not need to be downloaded again because they will remain cached in the user's browser cache (unless they have been updated on the server too).

Some of the assemblies (such as that containing the DataGrid control) are quite sizable, so having these only downloaded when the application is initially deployed will result in much smaller updates.

■ **Note** One of the big downsides of turning this option on is that it can't be used with applications configured to be installable and run outside the browser (that may or may not be a concern to you). Out of browser applications require all of their assemblies to be located in the .xap file (excluding those assemblies downloaded on demand, as described in the next section). If your application is configured to run outside the browser, trying to turn this option on will result in a dialog appearing, requiring you to decide on one option or the other. However, if your application is designed to run solely within the browser, then you should take advantage of this option.

How Assembly Caching Works

The way that assembly caching works is that for each assembly referenced by the project that must be packaged with your application (for example, assemblies included in the core Silverlight runtime

excluded), it will look for a corresponding external part manifest file in the same path from where it is referenced. These files have the same name as the assembly, but with a `.extmap.xml` extension instead of `.dll`. For example, you will find that the `System.Windows.Data.dll` assembly (that needs to be packaged with your application) has a corresponding external part manifest file named `System.Windows.Data.extmap.xml`. If you open this file, you will find the following contents:

```
<?xml version="1.0"?>
<manifest xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <assembly>
    <name>System.Windows.Data</name>
    <version>2.0.5.0</version>
    <publickeytoken>31bf3856ad364e35</publickeytoken>
    <relpath>System.Windows.Data.dll</relpath>
    <extension downloadUri="System.Windows.Data.zip" />
  </assembly>
</manifest>
```

This essentially defines the path where the assembly can be found, and where it can download the package containing it. When an external part manifest file is found for an assembly, it will be excluded from the `.xap` file, and packaged into its own `.zip` file (with the name specified as the download URI).

■ **Note** If the external part manifest file specifies an absolute URI, then the assembly won't be packaged into a `.zip` file. Instead, it will be downloaded from the specified URI instead of from the site from where the application is being downloaded. Note, however, that this is a cross-domain operation, and hence requires a cross-domain policy file put in place in the root of the site from where the package is to be downloaded. See Chapter 8: Securing Your Application for more information on implementing cross-domain policy files.

The `AppManifest.xaml` file in the `.xap` file will then reference it in its `Deployment.ExternalParts` section, so that the Silverlight plug-in knows what additional packages it needs to download before it can run the application.

Enabling Your Own Assemblies to be Cached

If you inspect your `.xap` file's contents (by changing its extension to `.zip` and unzipping it, as discussed previously in Chapter 2: Getting Started with Silverlight), you may find that not all the assemblies have been packaged as external parts (excluding the core assembly for the application), and are still packaged in your `.xap` file—particularly assemblies that you created yourself. In order for an assembly to be cached, it must be signed with a strong name, and have an external part manifest file (as described in the previous section).

To sign your assembly with a strong name, open up the project properties for the project, and open the Signing tab. Select the "Sign the assembly" check box (as seen in Figure 15-7), select `<New...>` from the drop-down list, and enter a name and password for the strong name key file in the dialog that appears. The next time your application is compiled it will then be signed.

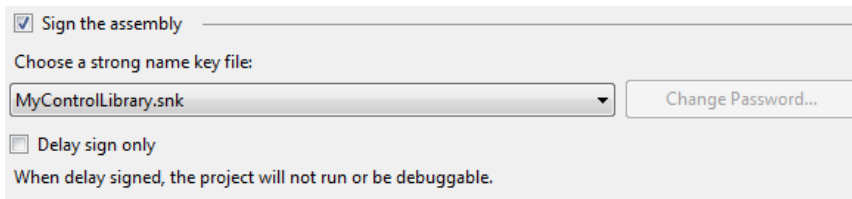


Figure 15-7. Configuring the strong name for the assembly

The next step is to create the external part manifest file. The easiest way to create this is to use the DevCorner ExtMap Maker Utility tool created by Alex Golesh, which you can download from http://blogs.microsoft.co.il/blogs/alex_golesh/archive/2009/07/16/small-utility-extmap-maker.aspx. This is a command-line utility that you can point to your assembly (passing the path to the assembly as its command-line parameter), and it will create a corresponding external part manifest file for you in the same path as the assembly is located. For example:

```
emm MyControlLibrary.dll
```

Put this in the bin\Debug and bin\Release folders of your assembly before adding the reference to it to your Silverlight application, and then recompile the Silverlight application. You will find that the assembly is now created as a separate package for download.

Downloading Modules on Demand

Another way you can partition your application is by breaking it up into modules, each in their own assembly. These can be excluded from the main .xap package (minimizing the size of your main application), and downloaded only when (and if) they are required. For example, you may like to have separate modules for each functional area in your application, where users may only have permission to access one or two of those functional areas. Likewise, you may like to split the administrative functionality into another module, as only a subset of users will be permitted access to it.

There are a number of ways to modularize your application, and the method you decide to use will often depend on if you want to use a framework or not. Let's start by discussing some of the options available to you, and then look at implementing a simple means of modularizing your application.

You need to start by making a decision on whether you are prepared to use an existing framework in order to modularize your application, or create your own custom solution. Incorporating a framework can provide many benefits (in addition to the modularization), however it may also impose on the structure and design of your application—which, depending on the stage you are at in your development process, may or may not be an issue. They can also introduce additional complexity into your application and incur a significant learning curve. It will depend on the nature of both your application and your team as to whether you are prepared to undertake this process.

The most popular frameworks for implementing modularization in Silverlight applications include Prism and the Managed Extensibility Framework (MEF).

Downloading Modules on Demand Using Prism

Prism is both a framework for creating modular applications, and corresponding guidance dictating patterns and practices for doing so. The application is partitioned into loosely coupled modules, which

are run within regions in an application shell. This provides a set pattern for modularizing your application where the modules can then be deployed to the client as required.

You can find out more about Prism and download the framework from <http://compositewpf.codeplex.com>. You can read a great article about on-demand loading in Prism and Silverlight written by Nigel Ferrissey here: www.codeproject.com/KB/silverlight/sl3prismnav.aspx.

Downloading Modules on Demand Using MEF

MEF is another popular choice, but has a different goal and focus to that of Prism. MEF is a framework for creating extensible applications and could essentially be considered a dependency injection framework (although that point is debated). Technically, MEF could be used in conjunction with Prism, replacing Unity that Prism uses as its dependency injection framework. Prism has a broader focus than MEF, with a focus on user interface composition that MEF does not have. MEF instead focuses on component discovery and composition in order to create extensible applications. You can take advantage of MEF's built-in, on-demand deployment capabilities using its `DeploymentCatalog` catalog.

You can find out more about MEF and download the framework from <http://mef.codeplex.com>, and find out more about using the `DeploymentCatalog` catalog from <http://mef.codeplex.com/wikipage?title=DeploymentCatalog>.

Downloading Modules on Demand Using the Navigation Framework

Since we haven't delved into these frameworks in this book, you may want to consider a simpler alternative that doesn't rely on a complex framework such as Prism or MEF, but simply works with the navigation framework as discussed back in Chapter 4: The Navigation Framework.

Using the navigation framework, you ideally need to be able to split your application up with views and their corresponding logic in separate assemblies—when the user is navigating to a view in another assembly, to check whether that assembly is in the cache, and if not then download it. Once it is downloaded, the requested view within that assembly should then be displayed within the frame.

David Poll has an excellent method for doing this that he has documented on his blog here: www.davidpoll.com/2010/02/01/on-demand-loading-of-assemblies-with-silverlight-navigation-revisited-for-silverlight-4-beta/. He provides an easy way of implementing this behavior, which you can take advantage of yourself by referencing some assemblies he's created in your Silverlight project, and following the instructions he has provided in his blog post.

■ **Note** The downloaded XAPs will be maintained in the browser cache using David's method, but you may want to consider adding them to the application's isolated storage store in order to maintain them permanently.

Digitally Signing Your Application

As briefly discussed back in Chapter 14: Out Of Browser Mode, and Interacting with the Operating System, when the user tries to install your application to run outside the browser with elevated privileges, it will display a rather stern warning in the install dialog (shown in Figure 15-8), stating that the publisher could not be verified. When you see this warning, it means that the application has not been signed.

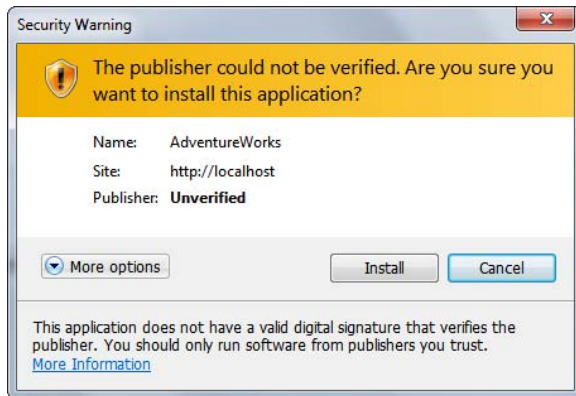


Figure 15-8. The installation dialog for unsigned applications that request elevated trust

Signing a XAP file involves obtaining a verified Authenticode code-signing certificate from a trusted certificate authority (CA). The process of obtaining a certificate is beyond the scope of this book, but you can find a thorough walkthrough written by Jeff Wilcox on his blog, available here: www.jeff.wilcox.name/2010/02/codesigning101/.

■ **Note** A code-signing certificate is not the same as a SSL certificate.

In summary, you purchase a certificate from a CA, they will verify your identity, and (generally) within a couple of days they will issue you with a valid certificate. You can then export the certificate from your certificate store as a .pfx file, which you can then use to sign your XAP file.

■ **Note** You only need to consider signing your application if it can be installed to run outside the browser and if it is requesting elevated trust permissions. Otherwise, signing your application will have no impact on its execution.

To sign your XAP file, open the Project Properties for your Silverlight application, and open the Signing tab, as shown in Figure 15-9.

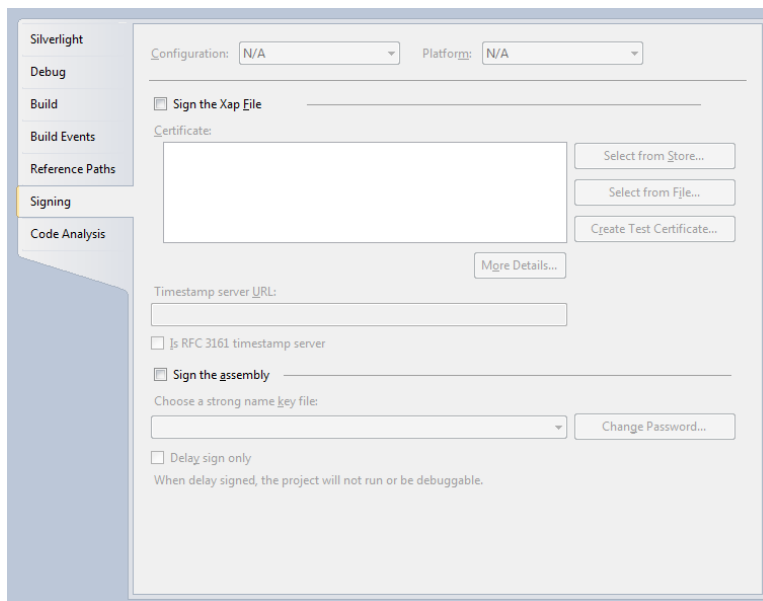


Figure 15-9. The Signing tab in the Project Properties

Select the “Sign the Xap File” check box (enabling that section of the tab), and click the Select from File button. Find your .pfx file and select it. Its details will appear in the Certificate box.

■ **Note** Alternatively, if you don’t have a certificate from a CA, you can create your own temporary certificate for use while you are testing the application. Click the Create Test Certificate button and enter a password in the two text boxes in the dialog that appears. This simple process will create a .pfx file for you and add it to your project for use during the compilation process.

The final step is to install the certificate as a Trusted Root certificate on your machine. Click on the More Details button, which will display the dialog shown in Figure 15-10.

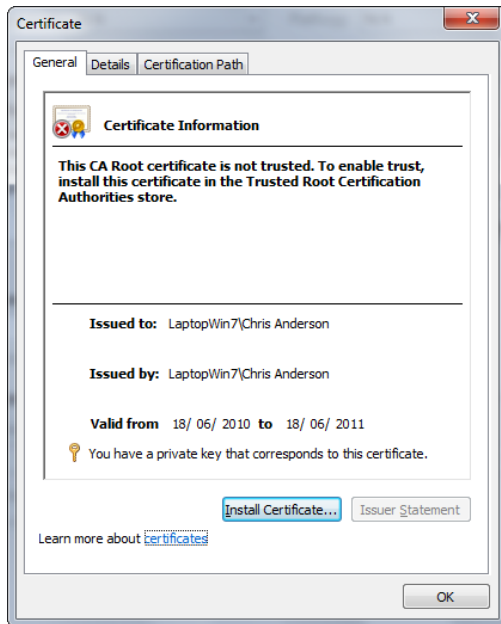


Figure 15-10. The Certificate Information dialog

If this dialog says that this CA root certificate is not trusted, then you will need to install it. Click the Install Certificate button, and click through the wizard to the screen shown in Figure 15-11.

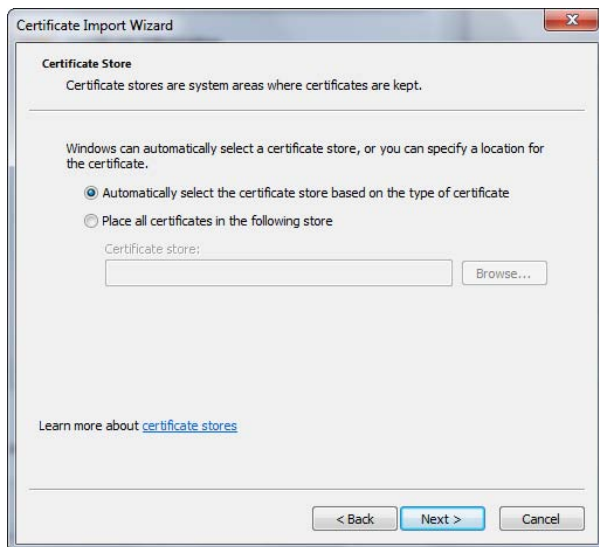


Figure 15-11. The Certificate Import Wizard dialog

Select the “Place all certificates in the following store” option, and click the Browse button. This will show the dialog shown in Figure 15-12.

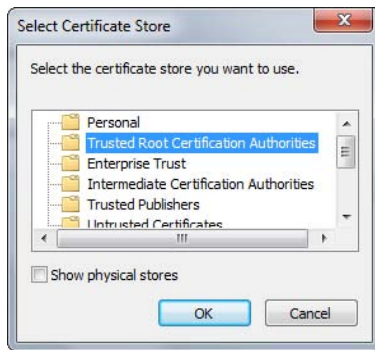


Figure 15-12. *Selecting a certificate store*

Select the Trusted Root Certification Authorities store, and press OK. Finish the wizard, and the certificate will be successfully installed.

Now when you compile and run your application, you will get the much friendlier install dialog as displayed in Figure 15-13.

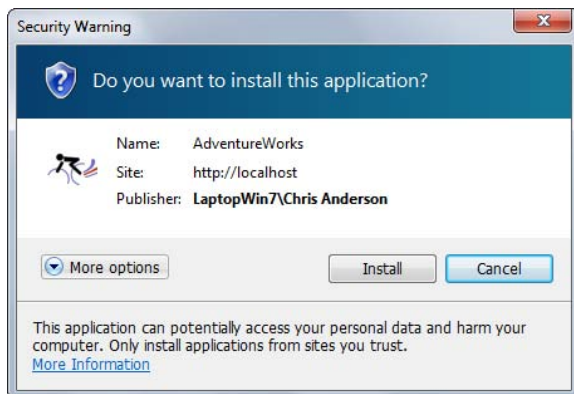


Figure 15-13. *The installation dialog for signed applications that request elevated trust*

■ **Note** One of the other key benefits of signing your application is that (as discussed back in Chapter 14: Out Of Browser Mode, and Interacting with the Operating System) in order for applications running in OOB mode to be updatable, the XAP file must be signed. The updates will work when the site of origin for the application is localhost, but they will fail once the application goes into production if it isn't signed. In addition, when the application is being updated, the certificate of the existing application must match that of the update (and the certificate must still not have expired) or the update will also fail.

Summary

In this chapter you've seen how to deploy your application to both the server and the client, and personalize the Silverlight install and application download experiences for the user to your company/application. Despite being fairly easy, adding personalization and branding to these aspects of the application is often overlooked by Silverlight developers. However, doing so can provide a more positive experience for the user and avoid confusion. Finally, partitioning your application effectively can be used to streamline the entire application deployment experience to the user, reducing both waiting times and user frustration.

Index

■ A

- About button, 20
- About.xaml, 28, 74
- AboutView.xaml, 84–85
- AcceptsReturn property, 202
- AcroRd32.exe, 426
- actions
 - configuring triggers and actions using Expression Blend, 291
 - creating custom actions, 295
 - how triggers and actions interact, 291
 - implementing triggers and actions by hand, 292
 - Microsoft.Expression.Interactions.dll assembly, 291
 - SubmitChanges method, 291
 - System.Windows.Interactivity.dll assembly, 290
 - TargetedTriggerAction, 295
 - TriggerAction, 295
 - using predefined triggers and actions, 290
 - See also* triggers
- Activate method, 434
- Active state, 355
- Add method, 212
- AddNewItem method, 200
- Adobe Flash, 4, 11
- Adobe Reader plug-in, 404
- Adventure Works Cycles, 101
- AdventureWorks2008 database, 101, 157, 249
- AdventureWorksModel.edmx, 102
- AllowDrop property, 460
- Anderson, Chris, 3
- animations
 - applying animation effects to user interfaces, 284
 - transition animations, 284
 - using Expression Blend for, 284
- Apache servers, 484
- App class, 28–29
- App.xaml, 28–29, 258, 261, 274
- App.xaml.cs, 311
- App_Data folder, 21
- AppendAllLines method, 470
- AppendAllText method, 470
- AppendText method, 470
- application dashboards, 404
- application deployment
 - adding display content to the pre-loader, 497
 - Application_Startup event handler method, 501
 - AppManifest.xaml, 504
 - AppPreloader.xaml, 496
 - Build Action property, 501
 - building a client desktop installer, 494
 - configuring the application pre-loader in the HTML file, 500
 - creating a setup program, 488
 - creating a web package, 487
 - creating the application pre-loader, 496
 - creating the application pre-loader, XAML code listing, 498
 - customizing and personalizing the Silverlight installation experience, 490
 - default Silverlight installation process, 489
 - deploying an application to a client, 488
 - deploying the server and client components to the server, 483
 - DevCorner ExtMap Maker Utility tool, 505
 - digitally signing your application, 506
 - downloading application modules on demand, 505
 - enabling your own assemblies to be cached, 504
 - Experience Guidance whitepaper, downloading, 493

application deployment (*cont.*)

- Fiddler, 501–502
- giving the application pre-loader more time to be displayed, 501
- hosting Silverlight applications on an Internet Information Services (IIS) server, 484
- Image control, 499
- implementing a custom-landing page, 493
- installing a valid certificate as a Trusted Root certificate, 508
- installing RIA Services assemblies to the Bin directory, 485
- installing RIA Services assemblies to the GAC, 484
- Manage Add-ons dialog, 492
- Managed Extensibility Framework (MEF), 505–506
- msdeploy tool, 487
- obtaining a verified Authenticode code-signing certificate, 507
- OnSourceDownloadComplete event, 500
- OnSourceDownloadProgressChanged event, 499
- packaging of external-part manifest files, 504
- partitioning your application into smaller modules, 502
- Prism, 505
- ProgressBar control, 498
- providing a low-friction install experience, 490
- Publish Web dialog, 486
- pushing the Silverlight runtime to enterprise users, 493
- reducing the size of your application, 502
- replacing the default application-loading animation, 495
- RootVisual property, 501
- server operating system and web server, requirements for, 484
- Silverlight 1.0 JScript Page item template, 496
- Silverlight Enterprise Deployment
 - whitepaper, downloading, 494
- Silverlight.js, 492
- SLLauncher command-line parameters
 - dialog, 494
- sllauncher.exe, 494–495

- slowing the application-downloading process, 500
- testing the application pre-loader, 500
- turning on assembly caching for an application, 502
- updating the progress bar indicator, 499
- using the Navigation Framework to download application modules on demand, 506
- using Visual Studio 2010 to publish a web site to a server, 486
- using Xcopy to deploy an application to a server, 485
- Web Setup Project template, 488
- See also* Silverlight 4; Visual Studio 2010

- application store, 439
- Application.Current object, 431–432
- Application_Startup event handler, 28, 501
- Application_UnhandledException event handler method, 185
- ApplicationNameTestPage.aspx, 26, 31
- ApplicationNameTestPage.html, 26, 31
- ApplicationSettings property, 440
- ApplicationStrings.resx, 32
- AppManifest.xaml, 504
- AppPreloader.xaml, 496
- AreRowDetailsFrozen property, 184
- ArgumentException exception, 448, 458
- ASP.NET, 1, 7, 10
 - ASP.NET_SessionId, 136
 - Dynamic Data scaffolding framework, 113
 - Membership API, 246
 - Report Viewer control, 411
 - session state, 136
 - Web Forms, 74
- ASPNETDB.MDF, 247
- Aspose.Words for .NET, 406
- assembly caching
 - explanation of, 503
 - reasons for using, 503
 - turning on, 502
- AssemblyInfo.cs, 302
- Assets folder, 29
- Associate method, 122
- Association attribute, 114–115
- AsyncCallback object, 133
- asynchronous server communication, 133
- attached properties, 44, 369
- AttachEvent method, 423

- attribute syntax, 41, 289
- attributes, 37
- Authentication property, 258
- AuthenticationBase class, 249, 253–254
- AuthenticationContext object, 258
- AuthenticationService class, 27, 249, 253, 258
- AuthenticationService domain service, 99
- AuthorizationAttribute class, 251
- AuthorizationContext object, 252
- AuthorizationContext property, 252
- authorize changeset phase, 103
- AutoCommit property, 199, 222
- AutoCompleteBox control, 209
- AutoEdit property, 199
- AutoGenerateColumns property, 154
- AutoGenerateField property, 195
- AutoGeneratingField event, 195
- AutoLoad property, 148
- automatic sizing, 56
- AutomationEvent object, 476
- AutomationFactory class, 472, 476
- AvailableFreeSpace property, 447

■ B

- Back button, 20
- Background property, 273, 277
- base control classes, 51
- BasedOn property, 276
- BaseReportHandler class, 413
- BaseReportRenderer class, 414
- basicHttpBinding, 134
- BeforeClose event, 476
- BeginEdit method, 200, 220, 222
- BeginInit method, 345
- BeginLoadProduct method, 397
- BeginPrint event, 408
- BeginSaveProduct method, 396
- behaviors
 - Behavior class, 297
 - creating and using behaviors, 297
 - definition of, 297
 - SubmitChanges method, 297
 - System.Windows.Interactivity
 - namespace, 297
 - when to use a behavior over a trigger/action combination, 299
- Binding markup extension, 47, 68
- BindingExpression class, 323

- bindings
 - basicHttpBinding, 134
 - configuring, 134
 - customBinding, 134
 - definition of, 134
 - netTcpBinding, 134
 - pollingDuplexHttpBinding, 134
 - wsHttpBinding, 135
- BindingValidationError event, 229, 232–233
- BindsDirectlyToSource property, 317
- BitmapImage, 157
- BitmapImage object, 457
- Blacklight, 15
- BlackoutDates property, 205
- blendability, 47
 - CreateList property, 300
 - d:DesignData markup extension, 301
 - d:DesignInstance markup extension, 300
 - defining the shape of the data, 300
 - design-time data mechanisms, 299
 - IsDesignTimeCreatable property, 300
 - using sample data at design time, 301
- BlurEffect, 284
- Build Action property, 501
- business applications
 - common features of, 11
 - designing and developing in Silverlight 4, 1–2
- BusyContent property, 151
- BusyIndicator control, 21, 29, 150
- Button control, 42, 56, 365
- ButtonBase class, 389
- ButtonClickTrigger class, 294

■ C

- C#, assigning event handlers in code, 59
- caching data
 - ApplicationSettings property, 440
 - ArgumentException exception, 448
 - AvailableFreeSpace property, 447
 - binding to isolated storage settings, 441
 - caching to a client-side database, 450
 - caching to isolated storage, 439
 - Clear method, 442
 - compressing data files, 449
 - Contains method, 441
 - creating, moving, and deleting
 - directories, 443
 - DataContractSerializer class, 446

caching data (*cont.*)

- deserializing and rehydrating an object
 - from file, 447
- detecting network availability, 452
- determining how much quota is free, 447
- DotNetZip, 449
- EffiProz Embedded C# Database, 451
- encrypting data files, 449
- enumerating through files and
 - directories, 443
- enumerating through the existing
 - settings, 442
- FileExists method, 445
- GetDirectoryNames method, 443
- GetIsNetworkAvailable method, 452
- GetUserStoreForApplication method, 442
- GetUserStoreForSite method, 442
- Google Gears, 451
- handling occasionally connected
 - applications, 439
- IncreaseQuotaTo method, 448
- IsolatedStorageFile class, 442, 447
- IsolatedStorageSettings class, 440–441, 450
- Keys property, 442
- NetworkAddressChanged event, 452
- NetworkChange class, 452
- NetworkInterface class, 452
- opening an isolated storage store, 442
- persisting a value to isolated storage, 440
- Perst, 451
- Quota property, 448
- reading the contents of a binary file from
 - isolated storage, 446
- reading the contents of a text file from
 - isolated storage, 446
- Remove method, 442
- removing the settings in a dictionary, 442
- requesting additional quota, 448
- retrieving a value from isolated storage, 441
- Save method, 440
- serializing or deserializing data to/from
 - XML, 446
- Sharp HSql, 451
- SharpZipLib, 449
- Siaqodb, 451
- Silverlight Database (SilverDB), 450
- SiteSettings property, 440
- SQLite, 450

- Sterling, 450
- storing files in isolated storage, 442
- storing settings in isolated storage, 440
- Sync Framework for Silverlight, 450
- System.IO namespace, 445
- System.IO.IsolatedStorage namespace, 440
- System.Net.NetworkInformation
 - namespace, 452
- System.Runtime.Serialization
 - namespace, 447
- TryGetValue method, 441
- writing a byte array to a file in isolated
 - storage, 445
- writing a string to a text file in isolated
 - storage, 445
- See also* isolated storage
- Calendar control, 205
- CallMethodAction action, 387, 391
- Cancel button, automatic creation of, 199
- CancelButtonContent property, 199
- CancelButtonStyle property, 199
- CancelEdit method, 200, 220, 222, 226
- CanConvertFrom method, 345
- CanConvertTo method, 345
- CanExecute method, 391
- CanExecute property, 388
- CanExecuteChanged event, 388
- CanUserReorderColumns property, 159
- CanUserResizeColumns property, 159
- Canvas control, 53
- CellEditingTemplate property, 155
- CellStyle property, 158
- Changed method, 223
- changeset, 98
 - authorize changeset phase, 103
 - ChangeSet object, 130
 - ChangeSetEntry object, 130
 - execute changeset phase, 103
 - validate changeset phase, 103
- CheckAndDownloadUpdateCompleted event, 435
- CheckBox control, 203
 - list of states, 352
- Children property, 368–369
- ChildWindow class, 29, 180, 183
- Chinnathambi, Kirupa, 292
- Class property, 116
- Clear method, 442
- ClearValue method, 341
- Click event, 58–59, 61

- clientaccesspolicy.xml, 263–264
- ClientBin folder, 24
- clipboard
 - Clipboard class, 462–463
 - ClipboardContentBinding property, 463
 - ClipboardCopyMode property, 463
 - ContainsText method, 462
 - CopyingRowClipboardContent event, 463
 - GetText method, 462
 - SetText method, 462
 - System.Windows namespace, 462
- Close method, 433, 438
- CodePlex, 51
 - Document Toolkit Extensions project, 406
 - ImageTools for Silverlight, 157
- collection syntax, 43
- CollectionChanged event, 144, 172, 368
- collections
 - advantages of using data source controls, 149
 - AutoLoad property, 148
 - BusyContent property, 151
 - CollectionChanged event, 144
 - CollectionViewSource, 145
 - Completed event, 149, 151
 - consuming data from the server, 146
 - Content property, 152
 - Data property, 146
 - Data Sources window, 146–147
 - DataContext property, 146, 148
 - DisplayAfter property, 152
 - DomainClient class, 146
 - DomainContext property, 148
 - DomainDataSource control, 145–146
 - DomainDataSourceView, 145
 - Entities property, 149
 - EntityQuery object, 146, 149
 - implementing a code-based approach to populating the user interface, 148
 - implementing a pulling strategy to populating the user interface, 146
 - INotifyPropertyChanged interface, 144
 - IsBusy property, 151
 - ItemsSource property, 146, 148–149
 - Load method, 149
 - LoadedData event, 148
 - LoadOperation object, 149
 - ObservableCollection, 144, 149
 - OverlayStyle property, 152
 - PagedCollectionView, 145
 - ProductSummary object, 147
 - ProgressBarStyle property, 151
 - QueryName property, 148
 - Silverlight's support for generic collection types, 144
 - specifying a domain service host, 150
 - understanding the domain context, 146
 - using the BusyIndicator control, 150
 - views and view manipulations, 144
 - Visibility property, 152
- CollectionViewSource, 145, 310, 312, 316–317
- ColorAnimationUsingKeyFrames object, 355
- Column property, 56, 156
- ColumnDefinitions property, 54
- COM automation
 - accessing COM components, 472
 - accessing local hardware devices, 478
 - automating Microsoft Office applications, 476
 - AutomationEvent object, 476
 - AutomationFactory class, 472, 476
 - BeforeClose event, 476
 - CreateObject method, 472, 474
 - creating a new instance of a Word application, 473
 - creating and installing a COM component, 478
 - creating or obtaining a COM object instance, 473
 - determining whether a COM component is registered on the system, 474
 - determining whether COM automation is available, 472
 - dynamic keyword, 473
 - EventRaised event, 476
 - Excel.Application, 476
 - Exception exception, 474
 - GetObject method, 472, 474
 - handling events raised by COM components, 476
 - having COM support in Silverlight 4, 472
 - IsAvailable property, 472
 - programmatic identifier (progID), 473
 - reading and writing data to local databases, 477
 - Scripting.FileSystemObject COM component, 475

- COM automation (*cont.*)
 - System.Runtime.InteropServices.Automation namespace, 472
 - Windows Script Host, 474
 - Word.Application, 473, 476
 - Workbook object, 476
 - WScript.Shell COM object, 474
- ComboBox control, 43, 70, 204
- CommandButtonsVisibility property, 200, 289
- CommandParameter property, 391
- commands
 - ButtonBase class, 389
 - CallMethodAction action, 391
 - CanExecute method, 391
 - CanExecute property, 388
 - CanExecuteChanged event, 388
 - CommandParameter property, 391
 - creating, 388
 - definition of, 387
 - DelegateCommand command, 390
 - Execute method, 388, 391
 - ICommand interface, 388–389
 - InvokeCommandAction action, 390
 - invoking, 389
 - passing a parameter to a command, 391
 - RelayCommand command, 391
- CommitButtonContent property, 200
- CommitButtonStyle property, 200
- CommitEdit method, 200
- Common Language Runtime (CLR), 5
- CommonStates state group, 354, 360
- CommunicationException exception, 264
- Completed event, 149, 151, 185, 239–240
- Component Object Model (COM), 6
- Composition attribute, 115
- Concurrency Mode property, 129
- concurrency violations
 - concurrency checks involving the three entity versions, 128
 - Concurrency Mode property, 129
 - ConcurrencyCheck attribute, 128
 - configuring the entity model to check for concurrency violations, 128
 - handling, 127
 - methods of identifying, 127
 - ModifiedDate property, 129
 - RoundtripOriginal attribute, 128
 - strategies for identifying concurrency violations, 128
 - TimeStamp attribute, 128
- Conditional attribute, 218
- ConflictMembers property, 130
- connectionStringName property, 248
- consolidated namespaces, 40, 301
- Contains method, 441
- Contains operator, 169
- ContainsText method, 462
- content controls
 - attached properties, 369
 - Button control, 365
 - ChildrenArea property, 368–369
 - CollectionChanged event, 368
 - containing a single control as content, 365
 - containing multiple controls as content, 367
 - Content property, 365–368
 - ContentArea property, 369
 - ContentControl class, 367
 - ContentPresenter, 365–366
 - ContentProperty attribute, 367
 - ContentTemplate property, 366
 - DefaultStyleKey property, 366
 - GetContentArea method, 369
 - Grid control, 365
 - HeaderText property, 366
 - propa snippet, 370
 - prodp snippet, 370
 - Register method, 370
 - RegisterAttached method, 370
 - SetContentArea method, 369
 - TemplateBinding markup extension, 366
 - See also* controls; custom controls; user controls
- content element syntax, 41, 289
- Content property, 152, 203, 210, 365–366–368, 437
- ContentArea property, 369
- ContentControl class, 41, 52, 367
- ContentFrameStyle style, 90
- ContentLoaded event, 211
- ContentPresenter, 365–366
- ContentProperty attribute, 41, 367
- ContentTemplate property, 366
- contract, 134
- Control class, 52, 361
- control templates, 65
 - applying an alternate control template, 278
 - default control templates, 278
 - defining visual states, 279

- Generic.xaml, 278
- how control templates are structured, 279
- in-state animation, 280
- Opacity property, 280
- parts and states model, 279
- PressedBorder element, 280
- state transitions, 281
- Style property, 279
- template parts, 282
- Template property, 279
- templating a control, 278
- Themes folder, 278
- triggering a state change on a control, 280
- using Expression Blend to copy a control's default template, 278
- using visual state groups, 281
- controls
 - assigning controls to grid cells, 56
 - assigning event handlers, 58
 - automatic sizing, 56
 - base control classes, 51
 - Button control, 56
 - Canvas control, 53
 - Click event, 58–59, 61
 - Column property, 56
 - ColumnDefinitions property, 54
 - ContentControl class, 52
 - Control class, 52
 - DependencyObject class, 52
 - direct events, 59
 - DockPanel control, 57
 - fixed-width sizing, 55
 - FrameworkElement class, 52
 - Grid control, 54, 56
 - layout controls, 53
 - layout controls in the Silverlight Toolkit, 57
 - obtaining missing controls, 50
 - Orientation property, 57
 - removing a control's margin, 57
 - routed events, 59
 - Row property, 56
 - RowDefinitions property, 54
 - StackPanel control, 57
 - UIElement class, 52
 - weighted proportion sizing, 55
 - Windows Forms and, 57
 - WrapPanel control, 57
 - See also* content controls; custom controls; user controls
- Controls folder, 29
- Convert method, 320
- ConvertBack method, 320
- ConverterCulture property, 322
- ConverterParameter property, 321–322
- ConvertFrom method, 345
- ConvertTo method, 345
- Copy method, 471
- CopyTo method, 471
- Create method, 470
- CreateDirectory method, 471
- CreateList property, 300
- CreateObject method, 472, 474
- CreateSubdirectory method, 471
- CreateText method, 470
- creating Silverlight applications
 - About button, 20
 - App_Data folder, 21
 - Back button, 20
 - BusyIndicator control, 21
 - DataForm control, 20
 - deep linking, 20
 - Email field, 20
 - Error window, 20
 - friendly name, 22
 - Login button, 20, 22
 - Logout button, 22
 - New Project window, Visual Studio 2010, 16
 - Password field, 20
 - Register Now button, 20
 - Silverlight Business Application template, 17–19
 - Silverlight Class Library project, 18
 - Silverlight Navigation Application template, 17–18
 - Silverlight Unit Test Application template, 19
 - WCF RIA Services Class Library template, 19
 - See also* Silverlight 4; Visual Studio 2010
- Creating the View Model, 382
- crossdomain.xml, 263
- Crystal Reports, 404, 410
- Current property, 259
- CurrentIndex property, 200, 209
- CurrentItem property, 194, 212, 315
- CurrentItemChanged event, 211

custom controls

- Active state, 355
- adding a Storyboard to the
 - VisualTransition, 357
- adding state-transition animations, 356
- advantages of, 347
- binding to properties in the code, 358
- CheckBox control, list of states, 352
- ColorAnimationUsingKeyFrames object, 355
- CommonStates state group, 354, 360
- compiling the MyControlLibrary project, 364
- connecting the code and the default control template, 361
- Control class, 361
- creating a control template in Expression Blend, 351
- creating a custom control, 348
- creating a new Silverlight Class Library project, 348
- DefaultStyleKey property, 349–350
- defining a contract, 360
- defining template parts, 357
- defining the base state for a control, 351
- defining the control's behavior, 359
- defining the control's default template, 350
- Duration property, 355
- Fill property, 352, 355
- Generic.xaml, 348–349, 351
- GetTemplateChild method, 361–362
- grouping states into sets, 353
- handling events for elements defined in the control template, 362
- HeaderText property, 358
- identifying what states need to be supported, 352
- implementing the behavior for the
 - WaitIndicator control, 363
- implementing the states, 354
- Inactive state, 354
- IsBusy property, 365
- LayoutRoot object, 355
- MyControlLibrary assembly, 359
- OnApplyTemplate method, 356, 360–363
- one-way binding with the
 - TemplateBinding markup extension, 358
- Opacity property, 352, 354–355, 357
- RepeatBehavior property, 355

- SetVisualState method, 363–364
- splitting Generic.xaml into smaller pieces, 359
- Static state, 354
- structure of a control template, 350
- TargetType property, 350
- TemplateBinding markup extension, 358–359
- TemplatePart attribute, 360
- TemplateVisualState attribute, 360
- testing the control, 364
- Text property, 358, 360
- Themes folder, 348, 350, 359
- transitioning from one state to another in a control, 363
- two-way binding using the
 - RelativeSource markup extension, 359
- Visibility property, 352
- visual state groups, 353
- VisualStateManager, 350, 356–357, 363
- VisualTransition object, 356
- WaitIndicator control, 353, 357
- WaitIndicator.cs, 348–349
- WaitIndicator.xaml, 359
- See also* content controls; controls; user controls

- custom operations, 111

- customBinding, 134

- CustomDataForm class, 29

- customErrors element, 256

- CustomValidation attribute, 116

■ D

- d:DesignData markup extension, 301

- d:DesignInstance markup extension, 300

- data binding

- assigning the source of a binding, 304

- binding modes, 68

- binding the property of a control, 307

- binding the property of a control in the code-behind, 323

- binding to a collection, 69

- binding to a property in the view's code-behind, 308

- binding to a resource, 308

- binding to an object resource, 310

- binding to collection views, 312

- binding to indexed properties, 312

- binding to nested properties, 311
- binding to objects in a
 - CollectionViewSource, 316
- binding to objects in a
 - PagedCollectionView, 312
- BindingExpression class, 323
- BindsDirectlyToSource property, 317
- code example of, 66
- CollectionViewSource, 312, 317
- Convert method, 320
- ConvertBack method, 320
- ConverterCulture property, 322
- ConverterParameter property, 321–322
- creating a class as a binding source, 309
- creating a simple value converter, 320
- creating an object collection, 313
- creating and binding a resource
 - automatically using the Data Sources window, 310
- CurrentItem property, 315
- data binding expression builder in Visual Studio 2010, 324
- DataContext property, 65, 67, 304, 308, 311
- DataItem property, 324
- defining resources in the code-behind, 311
- DomainDataSourceView, 312–313
- enhancing data binding, 317
- FirstName property, 311
- GetBindingExpression method, 319, 323
- getting and setting a control's attached property values, 324
- GetValue method, 324
- IDataErrorInfo interface, 304
- IEditableObject interface, 319
- INotifyCollectionChanged interface, 304
- INotifyDataErrorInfo interface, 304
- INotifyPropertyChanged interface, 304
- instantiating a class as a resource and binding to it, 314
- instantiating a class as a resource in XAML, 309
- IsBusy property, 306
- IsSynchronizedWithCurrentItem property, 315
- ItemsSource property, 305–306
- IValueConverter interface, 319
- Mode property, 68
- MultiBinding, 329

- NotOperatorValueConverter value converter, 320
- OneTime mode, 68
- OneWay mode, 68
- PagedCollectionView, 314, 316
- PagedEntityCollectionView, 312–313, 326
- ParentBinding property, 323
- passing a culture to a value converter, 322
- passing a parameter to a value converter, 321
- passing an entire object to a value converter, 322
- RelativeSource binding, 308
- RelativeSource property, 306
- SetBinding method, 323
- setting the Source property of a binding, 305
- Silverlight's data binding model, 65
- source, definition of, 65
- specifying an alternate value instead of the TargetNullValue property, 318
- specifying bindings using property element syntax, 329
- StaticResource markup extension, 305, 310
- StringFormat property, 317
- System.Windows.Data namespace, 323
- target, definition of, 65
- TemplatedParent mode, 307
- tips for debugging data-binding problems, 328
- TwoWay mode, 68, 306
- UpdateSource method, 319
- using a value converter, 320
- using the ElementName property on a binding, 305
- using the FallbackValue property, 318
- using the UpdateSourceTrigger property, 319
- value converter, definition of, 319
- ViewModel class, 313
- data entry forms
 - AcceptsReturn property, 202
 - Add method, 212
 - adding a delete button to each row, 216
 - adding or deleting rows, 212
 - AddNewItem method, 200
 - assigning property values using reflection, 221
 - AutoCommit property, 199, 222

data entry forms (*cont.*)

- AutoCompleteBox control, 209
- AutoEdit property, 199
- AutoGenerateField property, 195
- AutoGeneratingField event, 195
- BeginEdit method, 200, 220, 222
- BindingValidationError event, 229, 232–233
- BlackoutDates property, 205
- Calendar control, 205
- calling the IEditableObject methods, 222
- Cancel button, automatic creation of, 199
- CancelButtonContent property, 199
- CancelButtonStyle property, 199
- CancelEdit method, 200, 220, 222, 226
- Changed method, 223
- CheckBox control, 203
- checking whether changed forms have been saved, 211
- choosing a method for exposing data-validation errors, 234
- ComboBox control, 204
- CommandButtonsVisibility property, 200
- CommitButtonContent property, 200
- CommitButtonStyle property, 200
- CommitEdit method, 200
- Completed event, 239–240
- Conditional attribute, 218
- configuring the form labels, 197
- Content property, 203, 210
- ContentLoaded event, 211
- creating a DataForm control with automatically generated fields, 193
- creating a DataForm control with explicitly defined fields, 196
- creating a flexible layout using the Grid control, 189
- creating the data-entry user interface, 189
- CurrentIndex property, 200, 209
- CurrentItem property, 194, 212
- CurrentItemChanged event, 211
- customizing the header, 200
- data input controls available in Silverlight and the Silverlight Toolkit, 201
- DataContext property, 191, 214, 216, 224, 235

- DataForm control and its ValidationSummary control, 225
- DataFormCommandButtonsVisibility enumeration, 200
- DataGrid control and its ValidationSummary control, 226
- DataGridCheckBoxColumn, 212
- DataGridTemplateColumn, 212
- DataGridTextColumn, 212
- DatePicker control, 206
- DecimalPlaces property, 208
- defining the property-level and object-level validation rules, 227
- DeleteItem method, 200
- deleting the selected row(s), 215
- Description property, 195, 198
- Display attribute, 195–196
- DisplayDateEnd property, 205
- DisplayDateStart property, 205
- displaying a calculated value in a user interface, 222
- DisplayMemberPath property, 204
- DisplayMode property, 206
- DomainDataSource control, 191
- DomainDataSourceView class, 212–213
- domain-level validation, 227
- DomainOperationException exception, 241–242
- DomainUpDown control, 208
- EditTemplate, code listing, 197
- enabling change-tracking, 238
- EndEdit method, 220, 222, 234
- entering multiple rows in a spreadsheet-like fashion, 214
- EntitiesInError property, 242
- Entity class, 228, 235
- EntityChangeSet object, 239
- EntityConflict object, 242
- Error property, 240
- ErrorMessage property, 238
- Errors property, 225
- ErrorsChanged event, 232, 235
- exposing validation errors to the user interface, 228
- FallBackItem property, 208
- Filter property, 209
- FindNameInContent method, 211
- FirstDayOfWeek property, 206
- FocusManager object, 211

- GetErrors method, 232
- Grid control, 191
- GroupName property, 203
- handling concurrency violations, 242
- handling data-submission errors, 240
- HasChanges property, 212
- HasError property, 240
- HasErrors method, 232
- HeaderTemplate property, 200
- HeaderVisibility property, 200
- HorizontalScrollBarVisibility property, 202
- how data entry controls display validation errors, 224
- IDataErrorInfo interface, 228
- IEditableObject interface, 199, 226
- implementing data validation via validation attributes, 223
- implementing the IDataErrorInfo interface, code listing, 230
- implementing the IEditableObject interface, 220
- implementing the
 - INotifyPropertyChanged interface with magic strings, 217
- Increment property, 208
- INotifyDataErrorInfo interface, 228, 232–233
- INotifyPropertyChanged interface, 222, 232
- Insert method, 212
- InvalidInputAction property, 208
- IsChecked property, 203
- IsCyclic property, 207, 209
- IsEditable property, 207, 209
- IsItemChanged property, 212
- IsReadOnly property, 202, 212
- IsTabStop property, 210
- IsTextCompletionEnabled property, 210
- IsThreeState property, 203
- IsTodayHighlighted property, 206
- ItemsSource property, 194, 204, 208–209, 212
- ItemTemplate property, 205, 209
- Label control, 210
- Label property, 198
- LabelPosition property, 198
- laying out the data entry form, 189
- LineTotal property, 222
- List control, 205
- maintaining an empty row for the entry of new records, 214
- manually assigning property values, 221
- MarkErrorAsHandled method, 240
- MaxLength property, 202
- MemberwiseClone method, 220
- MinimumPopulateDelay property, 210
- MinimumPrefixLength property, 210
- NewItemTemplate, 197
- notifying the user interface of object-level validation errors, 234
- notifying the user interface of property-level validation errors, 229
- NotifyOnValidationError property, 229, 233
- NumericUpDown control, 208
- object-level validation, 226
- OnPropertyChanged method, 217–219
- OnUnitPriceChanged method, 223
- OnUnitPriceChanging method, 223
- OperationErrorStatus enumeration, 241
- originalState object, 221
- ProductNumber property, 195
- PropertyChanged event, 216–217, 222, 232, 235
- PropertyChangedEventArgs object, 217
- property-level validation, 226
- PropertyNames property, 242
- RadioButton control, 203
- ReadOnlyTemplate, 196
- Required attribute, 198, 237
- ResourceName property, 238
- ResourceType property, 195, 238
- RowEditEnded event, 214
- SelectedDate property, 205
- SelectedDateFormat property, 206
- SelectedDates property, 205
- SelectedItem property, 204
- SelectedItems property, 215
- SelectedValue property, 204
- SelectedValuePath property, 204
- selecting entities in the Data Sources window, 189
- SelectionMode property, 205, 215
- setting the initial focus to a given data input control, 211
- setting the tab order for each data input control, 210
- StringFormat property, 201
- structuring objects for use by data entry forms, 216

data entry forms (*cont.*)

- SubmitChanges method, 239
- SubmitOperation object, 239–240
- SubmittedChanges event, 239–240
- SubmittedChangesEventArgs object, 240
- submitting data changes to the server, 238
- submitting data changes via a domain context, 239
- submitting data changes via the DomainDataSource control, 239
- SubmittingChanges event, 239
- System.ComponentModel namespace, 216
- System.ComponentModel.DataAnnotations namespace, 227, 229, 235
- System.Diagnostics namespace, 219
- System.Linq namespace, 215
- System.Reflection namespace, 218–219
- System.Windows.Input namespace, 211
- TabIndex property, 210
- TabNavigation property, 210
- Target property, 210
- TextAlignment property, 202
- TextBox control, 201
- TextWrapping property, 202
- three types of data validation, 226
- TimePicker control, 207
- TimeUpDown control, 206
- toolkit namespace, 194
- ToString method, 201, 208
- TryValidateObject method, 237
- TryValidateProperty method, 236
- turning off the automatic edit and automatic commit features, 199
- UnhandledException event, 240
- using lambda expressions instead of magic strings, 219
- using reflection instead of magic strings, 218
- using reflection to validate magic strings in debug mode, 218
- using the DataGrid control for data entry, 212
- ValidateObject method, 237
- ValidateProperty method, 236
- ValidatesOnDataErrors property, 232–233
- ValidatesOnExceptions property, 229, 233

- ValidatesOnNotifyDataErrors property, 233

- ValidationAttribute class, 227

- ValidationContext object, 236

- ValidationException exception, 229–230, 236

- ValidationSummary control, 224

- Validator object, 235–236

- ValueMemberBinding property, 208

- ValueMemberPath property, 208

- VerticalScrollBarVisibility property, 202

- XAML Power Toys, 191–193

- See also* data validation

- Data property, 146

- Data Sources window, 146–147, 156

- data templates, 65

- data transfer object (DTO) classes, 27

- data validation

- BindingValidationError event, 229, 232–233

- choosing a method for exposing data-validation errors, 234

- defining the property-level and object-level validation rules, 227

- domain-level validation, 227

- ErrorMessage property, 238

- exposing validation errors to the user interface, 228

- GetErrors method, 232

- HasErrors method, 232

- IDataErrorInfo interface, 228

- implementing the IDataErrorInfo interface, code listing, 230

- INotifyDataErrorInfo interface, 228, 232–233

- notifying the user interface of object-level validation errors, 234

- notifying the user interface of property-level validation errors, 229

- NotifyOnValidationError property, 229, 233

- object-level validation, 226

- property-level validation, 226

- ResourceName property, 238

- ResourceType property, 238

- System.ComponentModel.DataAnnotations namespace, 227, 229, 235

- three types of data validation, 226

- TryValidateObject method, 237

- TryValidateProperty method, 236

- ValidateObject method, 237

- ValidateProperty method, 236
- ValidatesOnDataErrors property, 232–233
- ValidatesOnExceptions property, 229, 233
- ValidatesOnNotifyDataErrors property, 233
- ValidationAttribute class, 227
- ValidationContext object, 236
- ValidationException exception, 229–230, 236
- Validator object, 235–236
 - See also* data entry forms
- databases
 - EffiProz Embedded C# Database, 451
 - Google Gears, 451
 - Perst, 451
 - Sharp HSql, 451
 - Siaqodb, 451
 - Silverlight Database (SilverDB), 450
 - SQLite, 450
 - Sterling, 450
- DataContext property, 65, 67, 146, 148, 153, 156, 191, 214, 216, 224, 235, 304, 308, 311, 334, 384
- DataContractSerializer class, 446
- DataForm control, 20, 193, 196, 225
- DataFormCommandButtonsVisibility
 - enumeration, 200
- DataGrid control, 71, 226
 - built-in features and behaviors, list of, 158
- DataGridCheckBoxColumn, 155, 212
- DataGridTemplateColumn, 155, 212
- DataGridTextColumn, 155, 212
- DataItem property, 324
- DataPager control, 175
- DataSets, 120
- DatePicker control, 206
- DecimalPlaces property, 208
- deep linking, 20, 80, 83
- DefaultExt property, 459
- defaultProvider property, 247
- DefaultStyleKey property, 332, 349–350, 366
- DelegateCommand command, 390
- Delete method, 471
- DeleteItem method, 200
- dependency properties
 - ClearValue method, 341
 - creating a standard CLR property wrapper, 337
 - declaring a dependency property on a class, 336
 - dependency property identifier, 336
 - dependency property metadata, 338
 - DependencyObject class, 336–337
 - DependencyProperty class, 336
 - DependencyPropertyChangedEventArgs object, 339
 - determining whether a dependency property has a local value assigned, 341
 - development of, 335
 - GetMetadata method, 340
 - getting and setting a dependency property value, 337
 - GetValue method, 337
 - INotifyPropertyChanged interface, 338
 - making dependency properties easier to create, 340
 - never adding any logic to the getters/setters of dependency properties, 338
 - overview of, 341
 - propdp snippet, 340
 - property snippets, 340
 - PropertyChanged event, 338
 - PropertyMetadata's constructor, 339
 - ReadLocalValue method, 341
 - Register method, 336
 - registering a dependency property, 336
 - resetting a dependency property's value, 341
 - SetValue method, 337
 - sldp snippet, 340
 - sldpc snippet, 340
 - value precedence, ordering of, 340
 - ValueProperty dependency property identifier, 336
 - why dependency properties are required, 335
- DependencyObject class, 52, 336–337
- DependencyProperty class, 336
- DependencyPropertyChangedEventArgs object, 339
- Description property, 195, 198
- design patterns
 - Gang of Four, 1
 - Model-View-ViewModel (MVVM), 2
- DesignerProperties class, 347
- DesignHeight property, 47
- designing user experiences, 70

- design-time data mechanisms, 299
 - DesignWidth property, 47
 - DevCorner ExtMap Maker Utility tool, 505
 - DevExpress, 406
 - direct events, 59, 346
 - Directory class, 469–471
 - DirectoryInfo object, 468, 470–471
 - Display attribute, 118, 195–196
 - DisplayAfter property, 152
 - DisplayDateEnd property, 205
 - DisplayDateStart property, 205
 - DisplayMemberPath property, 204
 - DisplayMode property, 175, 206
 - Dispose method, 424
 - DockPanel control, 57
 - Document Toolkit, 406
 - domain operations, 97
 - domain services, 97
 - calling the insert/update/delete operations for each entity in the changeset, 109
 - ChangeSet object, 130
 - ChangeSetEntry object, 130
 - ConflictMembers property, 130
 - creating a domain service, 104
 - creating the ProductService domain service, procedure for, 107
 - custom operation, code example, 112
 - delete operation method names, 110
 - delete operation, code example, 110
 - domain operations, definition of, 107
 - Domain Service Class Wizard, 106, 113
 - domain service life cycle, 103
 - Enable Client Access check box, 106
 - EnableClientAccess attribute, 106
 - Expose OData Endpoint check box, 106
 - Generate Associated Classes for Metadata check box, 106
 - GetOriginal method, 130
 - GetProduct domain operation, 108
 - GetProducts operation, 108
 - insert operation method names, 109
 - insert operation, code example, 110
 - invoke operation, code example, 111
 - IsDeleteConflict property, 130
 - LinqToEntitiesDomainService class, 130
 - metadata classes, 106
 - naming convention for, 125
 - OnError method, 131
 - OptimisticConcurrencyException exception, 131
 - OutputCache attribute, 109
 - ProductService class, 107
 - ProductService domain service, 115
 - ProductService.cs, 104
 - Query attribute, 109
 - query operations, 107
 - ResolveConflicts method, 130
 - resolving conflicts in the domain service, 130
 - returning a collection of entities, 107
 - returning a single entity, 108
 - selecting the entities to expose, 105
 - StoreEntity property, 130
 - update operation method names, 109
 - update operation, code example, 110
 - DomainClient class, 146
 - DomainContext property, 148
 - DomainDataSource control, 145–146, 159, 166, 184, 191, 239, 295, 298, 384
 - DomainDataSource method, 169, 171, 173
 - DomainDataSourceView, 145, 178, 184, 212–213, 312–313
 - domain-level validation, 227
 - DomainOperationException exception, 186, 241–242, 250
 - DomainUpDown control, 208
 - DotNetZip, 449
 - drag drop target
 - AllowDrop property, 460
 - DragEnter event, 460
 - DragEventArgs object, 460
 - DragLeave event, 460
 - DragOver event, 460
 - enabling an application to act as a drag drop target for files, 460
 - GetData method, 460
 - IDataObject object, 460
 - DragMove method, 481
 - DragOver event, 460
 - DropShadowEffect, 284
 - Dunn, Craig, 451
 - Duration property, 355
 - dynamic keyword, 473
- **E**
- Editable attribute, 118
 - EditTemplate, code listing, 197
 - EffiProz Embedded C# Database, 451
 - Eini, Oren, 219

- ElementName property, 305
- elements, 37
- elevated trust mode
 - accessing file and folder attributes, 472
 - AppendAllLines method, 470
 - AppendAllText method, 470
 - appending text to a file, 470
 - AppendText method, 470
 - Copy method, 471
 - copying or moving a file, 471
 - CopyTo method, 471
 - Create method, 470
 - CreateDirectory method, 471
 - CreateSubdirectory method, 471
 - CreateText method, 470
 - creating a new folder, 471
 - Delete method, 471
 - deleting a file, 471
 - determining the path of a special folder, 468
 - determining whether a file or folder exists, 470
 - Directory class, 469–471
 - DirectoryInfo object, 468, 470–471
 - disabling elevated trust applications with a Windows group policy setting, 482
 - enabling for an application, 466
 - EnumerateDirectories method, 469
 - EnumerateFiles method, 469
 - EnumerateFileSystemInfos method, 469
 - enumerating files and directories within a folder, 468
 - Environment class, 468
 - Exists method, 470
 - File class, 470–471
 - FileInfo object, 470–471
 - FileSystemInfo object, 469
 - GetFolderPath method, 468
 - HasElevatedPermissions property, 467
 - install dialog for unsigned applications, 466
 - introduction of, in Silverlight 4, 466
 - Move method, 471
 - MoveTo method, 471
 - OOB mode and, 466
 - OpenFileDialog class, 472
 - opening the user's file system to direct reading and writing, 467
 - OpenRead method, 470
 - OpenText method, 470
 - OpenWrite method, 470
 - ReadAllBytes method, 470
 - ReadAllText method, 470
 - reading text or binary data from a file, 470
 - ReadLines method, 470
 - restrictions imposed by elevated trust, 482
 - restrictions lifted by elevated trust, 481
 - SecurityException exception, 467
 - System.IO namespace, 468
 - WriteAllBytes method, 470
 - WriteAllLines method, 470
 - WriteAllText method, 470
 - writing data to a file, 470
 - See also* security (client-side); security (server-side)
- Email field, 20
- Enable Client Access check box, 106
- EnableClientAccess attribute, 106, 262
- EndEdit method, 220, 222, 234
- EndInit method, 345
- endpoint, 134
- EndPrint event, 408–409
- Entities property, 149
- EntitiesInError property, 242
- Entity class, 228, 235
- Entity Framework, 94
 - AdventureWorksModel.edmx, 102
 - creating a data access layer, 101
 - creating an entity model, 102
 - Entity Data Model Wizard, 102
 - LINQ to Entities, 101
 - object-relational mapper (ORM), 101
- EntityChangeSet object, 239
- EntityConflict object, 242
- EntityConflict property, 242
- EntityQuery object, 146, 149, 167, 172, 177
- EntityQueryable object, list of extension methods, 166
- EnumerateDirectories method, 469
- EnumerateFiles method, 469
- EnumerateFileSystemInfos method, 469
- Environment class, 468
- Error property, 240
- Error window, 20
- errorInfo object, 127
- ErrorMessage property, 238
- Errors property, 225
- ErrorsChanged event, 232, 235, 386
- ErrorWindow.xaml, 29

- Escape key, 464–465
- EventRaised event, 476
- EventTrigger trigger, 290, 293, 295, 392
- Excel.Application, 476
- Exception exception, 474
- Exception type, 186
- Exclude attribute, 114
- execute changeset phase, 103
- Execute method, 388, 391
- Exists method, 470
- Experience Guidance whitepaper,
 - downloading, 493
- Expose OData Endpoint check box, 106
- Expression Blend 4
 - assigning event handlers in XAML, 58
 - code-related features, 270
 - creating a control template in, 351
 - designing user interfaces, 14
 - Expression Blend Interactivity library, 387
 - Expression Studio, 14
 - features for designing XAML-based user interfaces, 269
 - SketchFlow and user experience designers, 270
 - System.Windows.Interactivity.dll assembly, 290
 - working with vector graphics, 270
- Expression Design, 269
- Extensible Application Markup Language (XAML), 3, 5–6
 - About.xaml, 28
 - adding your own namespace declarations, 40
 - advantages of learning XAML syntax, 36
 - App.xaml, 28–29
 - assigning property values to controls, 37, 41
 - attached properties, 44
 - attribute syntax, 41
 - attributes, 37
 - base control classes, 51
 - Binding markup extension, 47
 - blendability, 47, 299
 - Button control, 42
 - ButtonClickTrigger class, 294
 - Canvas control, 53
 - challenges in learning XAML, 35
 - code-behind files (.xaml.cs), 37
 - collection syntax, 43
 - ColumnDefinitions property, 54
 - ComboBox control, 43
 - CommandButtonsVisibility property, 289
 - configuring triggers and actions using Expression Blend, 291
 - consolidated namespaces, 40
 - content element syntax, 41
 - ContentControl class, 41, 52
 - ContentProperty attribute, 41
 - Control class, 52
 - controlling the appearance of controls through XAML, 269
 - controls, 50
 - core syntax, 37
 - creating a simple user interface layout, code listing, 60
 - creating a simple XAML file, code listing, 38
 - creating custom actions, 295
 - creating custom triggers, 294
 - defining a hierarchy of objects, 38
 - defining strings and other types as constants, 288
 - defining the default namespace for a file, 40
 - definition of, 35
 - DependencyObject class, 52
 - DesignHeight property, 47
 - DesignWidth property, 47
 - DomainDataSource control, 295
 - elements, 37
 - ErrorWindow.xaml, 29
 - EventTrigger, 290, 293, 295
 - FindName method, 50
 - FrameworkElement class, 52
 - Grid control, 39, 54
 - Home.xaml, 28
 - how triggers and actions interact, 291
 - implementing triggers and actions by hand, 292
 - inserting comments, 287
 - Invoke method, 296
 - InvokeActions method, 294, 296
 - Items property, 43
 - layout controls, 53
 - Loaded event, 290
 - MainPage.xaml, 28
 - Markup Compatibility namespace, 47
 - markup extensions, 47
 - mc:Ignorable property, 47

- Microsoft.Expression.Interactions.dll
 - assembly, 291–292
- MoXAML Power Toys, 303
- namespaces, 49
- namespace prefix declarations, 39
- namespace properties, 45
- object graph (object tree), 38
- opening and closing tags, 37
- property element syntax, 43
- referencing controls, 37
- RelativeSource markup extension, 48
- Resources property, 43
- root visual, definition of, 38
- root-level node, 38
- RootVisual property, 38
- RowDefinitions property, 54
- SDK consolidated namespace, 40
- Silverlight User Control item template, 38
- StackPanel control, 42
- StaticResource markup extension, 47
- Styles.xaml, 29
- SubmitChanges event, 295
- SubmitChanges method, 291, 296
- SubmitChangesAction class, 295
- System namespace, 288
- System.Windows.Interactivity.dll
 - assembly, 290, 292
- System.Windows.Media namespace, 39
- TargetedTriggerAction, 295
- TemplateBinding markup extension, 48
- TextBlock control, 42
- ToolTipService control, 44
- trigger, definition of, 290
- TriggerAction, 295
- TypeConstraint attribute, 295
- UIElement class, 52
- UserControl control, 39
- using OR to combine enumerated values, 289
- using predefined triggers and actions, 290
- VisualTreeHelper class, 39
- x:Class, 45
- x:ClassModifier, 46
- x:FieldModifier, 46
- x:Key, 46
- x:Name, 45
- x:Null markup extension, 48

- .xaml extension, 37
- XML, 35
- XmlnsDefinition attribute, 40
- See also* navigation framework
- ExternalReference attribute, 115

■ F

- FallBackItem property, 208
- FallbackValue property, 318
- Ferriesey, Nigel, 506
- Fiddler, 262, 501–502
- File class, 470–471
- file system
 - AllowDrop property, 460
 - ArgumentException exception, 458
 - BitmapImage object, 457
 - creating a StreamWriter object, 459
 - creating and opening a file for writing, 459
 - DefaultExt property, 459
 - DragEnter event, 460
 - EventArgs object, 460
 - DragLeave event, 460
 - DragOver event, 460
 - enabling an application to act as a drag drop target for files, 460
 - enabling multiple file selection, 457
 - FileInfo object, 453, 455, 457
 - Files property, 457
 - Filter property, 454
 - FilterIndex property, 454
 - GetData method, 460
 - IDataObject object, 460
 - InvalidOperationException exception, 454
 - Multiselect property, 457
 - Open File dialog, 453
 - OpenFile method, 459
 - OpenFileDialog class, 453, 458
 - opening and reading binary files, 456
 - opening and reading text files, 455
 - opening and reading the selected file, 455
 - OpenRead method, 456
 - Read method, 456
 - reading a file in chunks, 456
 - reading a whole file in a single step, 456
 - reading and displaying an image file, 457

file system (*cont.*)
 ReadLine method, 455
 ReadToEnd method, 456
 SafeFileName property, 459
 Save File dialog, 457
 SaveFileDialog class, 457
 SecurityException exception, 454
 ShowDialog method, 454
 specifying one or more file filters, 454
 System.IO namespace, 453
 Write method, 459
 WriteLine method, 459
 writing binary data to a file, 459
 writing text to a file, 459
 FileExists method, 445
 FileInfo object, 453, 455, 457, 470–471
 Files property, 457
 FileSystemInfo object, 469
 Fill property, 352, 355
 Filter property, 209, 454
 FilterDescriptor object, 169
 FilterDescriptors property, 169
 FilterIndex property, 454
 FindName method, 50
 FindNameInContent method, 211
 First Floor Software, 266, 406
 FirstDayOfWeek property, 206
 FirstName property, 311
 fixed-width sizing, 55
 FloatableWindow control, 183
 FocusManager object, 211
 FormField.xaml, 333
 Forms Authentication provider, 246
 FormsAuthentication element, 258
 FragmentNavigation event, 87
 Frame control
 functions of, 75
 list of events, 87
 navigation methods, 78
 FrameworkElement class, 52
 friendly name, 22
 FriendlyName property, 254, 260
 full screen mode
 detecting, 464
 disabling of keyboard input, 465
 Escape key, 464–465
 FullScreenChanged event, 464
 FullScreenOptions property, 465
 IsFullScreen property, 464
 Loaded event, 464

retaining full screen mode when
 unfocused, 464
 setting an application to run in full
 screen mode, 463
 Startup event, 464

■ G

Gang of Four, 1
 Generate Associated Classes for Metadata
 check box, 106
 Generated_Code folder, 25, 95
 Generic.xaml, 275, 278, 332, 348–349, 351, 359
 GetBindingExpression method, 319, 323
 GetContentArea method, 369
 GetData method, 460
 GetDirectoryName method, 443
 GetErrors method, 232
 GetFolderPath method, 468
 GetIsNetworkAvailable method, 452
 GetMetadata method, 340
 GetObject method, 472, 474
 GetOriginal method, 130
 GetProduct domain operation, 108
 GetProducts operation, 108
 GetReport function, 416
 GetTemplateChild method, 361–362
 GetText method, 462
 GetUserStoreForApplication method, 442
 GetUserStoreForSite method, 442
 GetValue method, 324, 337
 Global Assembly Cache (GAC), 484
 Global.asax, 136
 GlobalSuppressions.cs, 27
 Golesh, Alex, 505
 Google Gears, 451
 Grid control, 39, 54, 56, 191, 365
 GroupDescriptions collection property, 175
 GroupDescriptor object, 173
 GroupDescriptors property, 173
 GroupName property, 203

■ H

HasChanges property, 212
 HasElevatedPermissions property, 467
 HasError property, 240
 HasErrors method, 232
 HasMorePages property, 408
 HeaderTemplate property, 200

HeaderText property, 358, 366
 HeaderVisibility property, 200
 Helpers folder, 29
 Heuer, Tim, 183
 High-Level Shader Language (HLSL), 285
 Home.xaml, 28, 74, 77
 HorizontalScrollBarVisibility property, 163, 202
 HTML Bridge, 419
 HtmlPage class, 419
 HtmlViewer control, 421
 HTTP requests

- choosing a networking stack, 139
- HttpWebRequest class, 139
- making HTTP requests in Silverlight, 138
- methods for downloading and uploading data, 138
- RegisterPrefix method, 139
- WebClient class, 138
- WebRequest class, 138–139

HyperlinkButton, 179

HyperText Markup Language (HTML), 35

■ I

ICommand interface, 388–389

icons

- buying icons formatted as XAML, 284
- purchasing a stock icon library, 283
- vector icons, 284

IDataErrorInfo interface, 228, 230, 304, 382

IDataObject object, 460

IDisposable interface, 424

IEditableObject interface, 199, 220, 226, 319

IEnumerable collection, 98

IFrame

- AttachEvent method, 423
- creating, 420
- destroying, 424
- determining when content is loaded, 423
- Dispose method, 424
- ensuring that the Windowless property is set, 421
- HtmlViewer control, 421
- IDisposable interface, 424
- Invoke method, 424
- OpenReportInNewWindow method, 425
- overlying on an application, 420
- positioning and sizing, 421–422
- Report.ashx, 425
- setting windowless mode, 420

- using the HtmlViewer Control, 425
- using the WebBrowser control, 424
- Windowless property, 420

Image control, 499

images, buying in XAML format, 284

ImageTools for Silverlight, 157

implicit styling, 271

Inactive state, 354

Include attribute, 114

Include method, 114

IncludeTotalCount property, 177

IncreaseQuotaTo method, 448

Increment property, 208

Ingebrigtsen, Einar, 219

Initialize method, 252

InitializeRootVisual method, 28

INotifyCollectionChanged interface, 304

INotifyDataErrorInfo interface, 228, 232–233, 304, 382, 386

INotifyPropertyChanged interface, 69, 144, 217, 222, 232, 304, 334–335, 338, 382–383, 386

Insert method, 212

Install method, 430

InstallState property, 431–432

InstallStateChanged event, 432

Instance property, 252

in-state animation, 280

IntelliSense, 41

Internet Information Services (IIS), 22

- hosting Silverlight applications, 484
- Internet Information Services Manager, 486–487

InvalidInputAction property, 208

InvalidOperationException exception, 135, 454

Invoke method, 296, 424

invoke operations, 111

InvokeActions method, 294, 296

InvokeCommandAction action, 390

IPagedCollectionView interface, 178

IQueryable expression, 121, 143, 166

IsActive property, 434

IsAuthenticated property, 259

IsAuthorized method, 251–252

IsAvailable property, 472

IsBusy property, 151, 306, 365

IsChecked property, 203

IsCyclic property, 207, 209

IsDeleteConflict property, 130

IsDesignTimeCreatable property, 300

IsEditable property, 207, 209

- IsFullScreen property, 464
 - IsInDesignTool property, 347
 - IsItemChanged property, 212
 - isolated storage
 - application store, 439
 - ApplicationSettings property, 440
 - ArgumentException exception, 448
 - AvailableFreeSpace property, 447
 - binding to isolated storage settings, 441
 - caching data to, 439
 - Clear method, 442
 - compressing data files, 449
 - Contains method, 441
 - creating, moving, and deleting directories, 443
 - DataContractSerializer class, 446
 - deserializing and rehydrating an object from file, 447
 - determining how much quota is free, 447
 - DotNetZip, 449
 - encrypting data files, 449
 - enumerating through files and directories, 443
 - enumerating through the existing settings, 442
 - FileExists method, 445
 - GetDirectoryNames method, 443
 - GetUserStoreForApplication method, 442
 - GetUserStoreForSite method, 442
 - IncreaseQuotaTo method, 448
 - IsolatedStorageFile class, 442, 447
 - IsolatedStorageSettings class, 440–441
 - Keys property, 442
 - opening an isolated storage store, 442
 - persisting a value to, 440
 - Quota property, 448
 - reading the contents of a binary file from isolated storage, 446
 - reading the contents of a text file from isolated storage, 446
 - Remove method, 442
 - removing the settings in a dictionary, 442
 - requesting additional quota, 448
 - retrieving a value from, 441
 - Save method, 440
 - serializing or deserializing data to/from XML, 446
 - SharpZipLib, 449
 - site store, 439
 - SiteSettings property, 440
 - storing files in, 442
 - storing settings in, 440
 - Sync Framework for Silverlight, 450
 - System.IO namespace, 445
 - System.IO.IsolatedStorage namespace, 440
 - System.Runtime.Serialization namespace, 447
 - TryGetValue method, 441
 - writing a byte array to a file in isolated storage, 445
 - writing a string to a text file in isolated storage, 445
 - See also* caching data
 - IsolatedStorageFile class, 442, 447
 - IsolatedStorageSettings class, 440–441, 450
 - IsReadOnly property, 202, 212
 - IsRunningOutOfBrowser property, 86, 432
 - IsSynchronizedWithCurrentItem property, 315
 - IsTabStop property, 210
 - IsTextCompletionEnabled property, 210
 - IsThreeState property, 203
 - IsTodayHighlighted property, 206
 - IsTotalItemCountFixed property, 175
 - ISupportInitialize interface, 345
 - IsUserAccessAuthorised property, 417
 - IsValid method, 116
 - Items property, 43, 163
 - ItemsPanel property, 161
 - ItemsPanelTemplate property, 161
 - ItemsSource property, 146, 148–149, 153–154, 163, 178, 194, 204, 208–209, 212, 305–306
 - ItemTemplate property, 160–161, 205, 209
 - IValueConverter interface, 157, 319
- ## ■ J
- JavaScript, 6
 - JavaScript Beautifier, 28
 - Jongerius, Emiel, 219
 - JournalOwnership property, 82, 86
- ## ■ K
- Key attribute, 118
 - KeyDown event, 180
 - Keys property, 442
 - Klementiev, Sergey, 243

■ **L**

- Label control, 210
- Label property, 198
- LabelPosition property, 198
- layout controls
 - Canvas control, 53
 - definition of, 53
 - Grid control, 54
- LayoutChanged event, 421
- LayoutRoot object, 355
- Libs folder, 30
- Likeness, Jeremy, 219, 450
- LineTotal property, 222
- LINQ, 215
- LINQ to Entities, 101
- LINQ to SQL, 94
- LinqToEntitiesDomainService class, 121, 130
- Linux, 5
- List control, 205
- ListBox control, configuring and customizing, 160
- Load method, 149, 167
- Loaded event, 290, 464
- LoadedData event, 148, 185
- LoadOperation object, 149, 177, 185
- LoadProduct command, 396
- LoadSize property, 177
- Local Report Engine, 410–411
- LocalReport object, 414
- LoggedOut event, 258
- Login button, 20, 22
- Login dialog, 257
- Login folder, 29
- Login method, 259, 387
- LoginError event, 392
- LoginFailed event, 392
- LoginSuccessful event, 392
- Logout button, 22

■ **M**

- machine.config, 247, 249
- MainPage class, 76–77, 88
- MainPage.xaml, 28, 74, 84, 333, 335
- Manage Add-ons dialog, 492
- Managed Extensibility Framework (MEF), 505–506
- MappedUri property, 84
- MarkErrorAsHandled method, 185, 240

- Markup Compatibility namespace, 47
- markup extensions
 - Binding, 47
 - list of, 47
 - RelativeSource, 48
 - StaticResource, 47
 - TemplateBinding, 48
 - x:Null, 48
- master pages, 75
- MaxLength property, 202
- mc:Ignorable property, 47
- MemberwiseClone method, 220
- MergedDictionaries property, 63
- metadata classes, 106
 - .metadata.cs files, 106
 - MetadataType attribute, 113
 - ProductMetadata class, 113
- Microsoft BizSpark program, obtaining an MSDN subscription, 14
- Microsoft Expression Gallery, 202
- Microsoft Web Platform Installer 2.0, downloading, 13
- Microsoft XPS Document Writer, 406
- Microsoft.Expression.Interactions.dll assembly, 291–292
- MinimumPopulateDelay property, 210
- MinimumPrefixLength property, 210
- minRequiredNonalphanumericCharacters property, 257
- minRequiredPasswordLength property, 257
- Mode property, 68
- Models, 376–378
- Models folder, 27, 30, 99, 142
- Model-View-ViewModel (MVVM) design
 - pattern, 2, 71, 76, 270
 - additional configurations, 379
 - BeginLoadProduct method, 397
 - BeginSaveProduct method, 396
 - benefits of implementing MVVM, 375
 - ButtonBase class, 389
 - CallMethodAction action, 387, 391
 - CanExecute method, 391
 - CanExecute property, 388
 - CanExecuteChanged event, 388
 - class diagram for the LoginViewModel and ViewModelBase classes, 382
 - CommandParameter property, 391
 - commands, definition of, 387
 - connecting the Models to the ViewModels, 384

- Model-View-ViewModel (MVVM) design pattern (*cont.*)
 - connecting the ViewModel to the View, 384
 - considering the ViewModel as a Model of the View, 378
 - core principles of, 376
 - creating a command, 388
 - creating a simple View, code listing, 380
 - creating the ViewModel, 382
 - DataContext property, 384
 - DelegateCommand command, 390
 - dependency properties versus standard properties, 395
 - describing the View and ViewModel, 374
 - designing the View or ViewModel first, 394
 - DomainDataSource control, 384
 - each layer not knowing about the layer above, 377
 - enabling the View to invoke an operation on the ViewModel, 386
 - enabling the View to respond to an event in the ViewModel, 391
 - ErrorsChanged event, 386
 - EventTrigger trigger, 392
 - Execute method, 388, 391
 - exposing Model objects directly to the View, 393
 - Expression Blend Interactivity library, 387
 - goal of zero code-behind, 377
 - ICommand interface, 388–389
 - IDataErrorInfo interface, 382
 - implementation pros and cons, 393–395
 - implementing an example login scenario, 380
 - INotifyDataErrorInfo interface, 382, 386
 - INotifyPropertyChanged interface, 382–383, 386
 - instantiating a ViewModel in XAML, code listing, 384
 - interacting with the server via RIA Services, 395
 - interactions between the MVVM layers, overview of, 392
 - investigating other popular frameworks, 394
 - InvokeCommandAction action, 390
 - invoking a command, 389
 - lack of standardization of the pattern, 373
 - list of popular frameworks that support MVVM, 402
 - LoadProduct command, 396
 - Login method, 387
 - LoginError event, 392
 - LoginFailed event, 392
 - LoginSuccessful event, 392
 - maintaining a separation between look and behavior, 376
 - Models, 376, 378
 - NavigateAction action, 392
 - ObservableCollection collection, 386
 - one View interacting with multiple ViewModels, 379
 - one ViewModel serving multiple Views, 379
 - Page class, 377
 - passing a parameter to a command, 391
 - Password property, 385
 - PasswordTextBox text box, 385
 - primary purpose of, 374
 - Product entity, 395
 - Product property, 396
 - ProductDetailsViewModel class, code listing, 397
 - PropertyChanged event, 383, 386
 - pulling data into the View from the ViewModel, 385
 - RelayCommand command, 391
 - Text property, 385
 - three component layers, 376
 - untestability of a View's logic and behavior when written in its code-behind, 375
 - UserName property, 385
 - UserNameTextBox text box, 385
 - Validator class, 383
 - View to edit the details of a product, XAML code listing, 400
 - ViewModelBase class, 382
 - ViewModels, 376, 378
 - Views, 376–377
 - whether to use actions/behaviors or commands, 391
 - wrapping Model objects in ViewModels, 393
- ModifiedDate property, 129
- Moonlight project, 5

MouseLeftButtonDown event, 438, 481
 MouseRightButtonDown event, 430
 Move method, 471
 MoveCurrentToNext method, 184
 MoveCurrentToPosition method, 184
 MoveTo method, 471
 MoXAML Power Toys, 333
 features of, 303
 msdeploy tool, 487
 MultiBinding, 329
 Multiselect property, 457
 MyControlLibrary assembly, 359

■ N

namespaces, 49
 namespaces, 39
 adding your own namespace
 declarations, 40
 consolidated namespaces, 40, 301
 declaring namespace prefixes, 39
 Markup Compatibility namespace, 47
 namespace properties, 45
 SDK consolidated namespace, 40
 System namespace, 288
 System.ComponentModel, 216, 345, 347
 System.ComponentModel.DataAnnotations, 113, 115, 121, 227, 229, 235
 System.Diagnostics, 219
 System.IO, 445, 453, 468
 System.IO.IsolatedStorage, 440
 System.Linq, 215
 System.Net.NetworkInformation, 452
 System.Net.Sockets, 139
 System.Reflection, 218–219
 System.Runtime.InteropServices.Automation, 472
 System.Runtime.Serialization, 447
 System.ServiceModel.DomainServices.Server, 114
 System.Transactions, 132
 System.Windows, 430, 462
 System.Windows.Browser, 419
 System.Windows.Controls.Primitives, 332
 System.Windows.Data, 323
 System.Windows.Input, 211
 System.Windows.Interactivity, 297
 System.Windows.Markup, 302
 toolkit, 194
 XmlnsDefinition attribute, 40
 XmlnsPrefix attribute, 302
 Navigate method, 78
 NavigateAction action, 392
 Navigated event, 87–88
 NavigateUri property, 179
 Navigating event, 87
 navigation framework
 About.xaml, 74
 AboutView.xaml, 84–85
 alternative user interface frameworks,
 list of, 91
 caching a view and maintaining its state,
 89
 components of, 75
 ContentFrameStyle style, 90
 creating additional views, 76
 deep linking, 80, 83
 FragmentNavigation event, 87
 Frame control, 75
 Frame control events, list of, 87
 handling navigation events, 87
 history of, 73
 Home.xaml, 74, 77
 implementing visual transition effects,
 90
 initiating navigation operations, 77
 integrating views with the browser
 history, 85
 IsRunningOutOfBrowser property, 86
 JournalOwnership property, 82, 86
 MainPage class, 76–77, 88
 MainPage.xaml, 74, 84
 MappedUri property, 84
 master pages, 75
 Navigate method, 78
 Navigated event, 87–88
 Navigating event, 87
 navigation methods on the Frame
 control, 78
 navigation methods on the
 NavigationService object, 78
 NavigationCacheMode property, 89
 NavigationContext object, 77, 81
 NavigationFailed event, 85, 87–88
 NavigationService object, 76, 87–88
 NavigationStopped event, 87
 OnNavigatedFrom method, 88
 OnNavigatingFrom method, 88
 Page class, 76–77

navigation framework (*cont.*)
 ParamProductID property, 81
 parsing query string parameters, 81
 passing complex data types between views, 82
 passing data between views, 80
 passing data to a view in the URI, 77
 passing data using query string parameters, 80
 ProductDetails view, 82
 ProductID parameter, 81
 QueryString property, 81
 reading query string parameters, 81
 responding to a navigation operation within a view, 88
 Silverlight Business Application template, 73
 single document interface (SDI) applications, 73
 Styles.xaml, 90
 TargetName property, 79
 Title property, 76
 TransitioningContentControl, 90
 Uri class, 78
 URI mapper, 83
 Uri property, 84
 using a HyperlinkButton control, 79
 using a URI to specify a view's location, 77
 using the Source property of the Frame control, 79
 ValueConverter, 79
 views, 73
 Views folder, 74
 See also Extensible Application Markup Language (XAML)
 NavigationCacheMode property, list of values, 89
 NavigationContext object, 77, 81
 NavigationFailed event, 85, 87–88
 NavigationService object, 76, 78, 87–88
 NavigationStopped event, 87
 .NET Reflector, downloading, 16
 .NET, 5
 netTcpBinding, 134
 NetworkAddressChanged event, 452
 NetworkChange class, 452
 NetworkInterface class, 452
 NewItemTemplate, 197
 nHibernate, 102
 NotificationWindow class, 437

NotifyOnValidationError property, 229, 233
 NotOperatorValueConverter value converter, 320
 NullReferenceException, 88
 NumericUpDown control, 155, 208

■ O

object graph (object tree), 38
 object-level validation, 226
 object-oriented design (OOD), 1
 object-relational mapper (ORM), 101
 ObservableCollection collection, 144, 149, 386
 Office Compatibility Pack, 405
 OnApplyTemplate method, 356, 360–363
 OnError method, 126, 131
 OneTime mode, 68
 OneWay mode, 68
 OnNavigatedFrom method, 88
 OnNavigatingFrom method, 88
 OnPropertyChanged method, 217–219
 onSilverlightError function, modifying, 31
 OnSourceDownloadComplete event, 500
 OnSourceDownloadProgressChanged event, 499
 OnUnitPriceChanged method, 223
 OnUnitPriceChanging method, 223
 OOB mode
 Activate method, 434
 application ID, 433
 Application.Current object, 431–432
 benefits of, 427
 CheckAndDownloadUpdateCompleted event, 435
 checking whether an application update is available, 434
 Close method, 433, 438
 closing the application's main window, 433
 configuring an application to run outside the browser, 428
 Content property, 437
 creating a custom title bar, 481
 debugging an application while it is running in OOB mode, 436
 determining an application's installation status, 431
 determining whether a window is active, 434

- determining whether an application is running inside or outside the browser, 432
- displaying toast notifications, 437
- DragMove method, 481
- enabling an application's window to be moved, 481
- enabling an application's window to be resized, 481
- finding the path to the application, 432
- implementing custom chrome, 479
- Install method, 430
- installing an application to run out of browser, 429
- InstallState property, 431–432
- InstallStateChanged event, 432
- IsActive property, 434
- IsRunningOutOfBrowser property, 432
- launching an application in OOB mode from Visual Studio, 436
- MouseLeftButtonDown event, 438, 481
- MouseRightButtonDown event, 430
- NotificationWindow class, 437
- Out-of-Browser Settings dialog, 429–430
- performing an automatic check for application updates, 435
- referencing the application's main window, 433
- running an application in elevated trust mode, 466
- running applications in, 6
- setting the position of the application's main window, 434
- Silverlight 3 and, 427
- sllauncher.exe, 432, 436
- specifying install options, 429
- specifying miscellaneous options, 429
- specifying window options, 429
- System.Windows namespace, 430
- TopMost property, 434
- uninstalling an application, 436
- Window Style option, 479–480
- WindowState property, 434
- Opacity property, 280, 352, 354–355, 357
- Open File dialog
 - BitmapImage object, 457
 - displaying, 454
 - enabling multiple file selection, 457
 - FileInfo object, 453, 455
 - Files property, 457
 - Filter property, 454

- FilterIndex property, 454
- Multiselect property, 457
- OpenFileDialog class, 453
- opening and reading binary files, 456
- opening and reading text files, 455
- opening and reading the selected file, 455
- OpenRead method, 456
- Read method, 456
- reading a file in chunks, 456
- reading a whole file in a single step, 456
- reading and displaying an image file, 457
- ReadLine method, 455
- ReadToEnd method, 456
- ShowDialog method, 454
- specifying one or more file filters, 454
- See also* Save File dialog
- Open XML, 405
- OpenFile method, 459
- OpenFileDialog class, 458, 472
- OpenRead method, 456, 470
- OpenReportInNewWindow method, 425
- OpenText method, 470
- OpenWrite method, 470
- OperationErrorStatus enumeration, 186, 241
- OptimisticConcurrencyException exception, 131
- OrderByDescending clause, 171
- Orientation property, 57
- originalState object, 221
- OutputCache attribute, 109
- OverlayStyle property, 152

■ P

- Page class, 76–77, 377
- PagedCollectionView, 145, 165, 170, 172, 174, 312, 314, 316
- PagedEntityCollectionView, 312–313, 326
- PageIndex property, 178
- PageSize property, 177–178
- PageVisual property, 408
- Pant, Saurabh, 485
- ParamProductID property, 81
- ParentBinding property, 323
- parts and states model, 279
- Password field, 20
- Password property, 385
- PasswordTextBox text box, 385
- Path property, 68

- Perpetuum Software, 405
- persist changeset phase, 103
- Perst, 451
- pixel shaders
 - BlurEffect, 284
 - creating, 285
 - DropShadowEffect, 284
 - High-Level Shader Language (HLSL), 285
- Poll, David, 506
- pollingDuplexHttpBinding, 134, 139
- PopulateReportDataSources function, 416
- presentation model types
 - Associate method, 122
 - creating a presentation model class, 120
 - implementing custom objects rather than entities, 119
 - IQueryable expression, 121
 - populating and exposing, 121
 - ProductInventoryPM object, 123
 - ProductPM object, 120, 123
 - ProductPMService, 121
 - System.ComponentModel.DataAnnotations namespace, 121
 - updating, 122
- PressedBorder element, 280
- Print method, 408
- PrintDocument class, 407, 409
- PrintPage event, 408
- PrintPageEventArgs object, 408
- Prism, 505
- Product entity, 114
- Product property, 396
- ProductClassValidation attribute, 117
- ProductDetails view, 82
- ProductDetailsReportData class, code listing, 417
- ProductDetailsViewModel class, code listing, 397
- ProductID parameter, 81
- ProductInventory entity, 114–115
- ProductInventory field, 114
- ProductInventoryPM object, 123
- ProductMetadata class, 113
- ProductNumber property, 195
- ProductPM object, 120, 123
- ProductPMService, 121
- ProductService class, 107
- ProductService domain service, 115
- ProductService.cs, 104
- ProductSummary object, 142, 147, 179
- programmatic identifier (progID), 473
- ProgressBar control, 498
- ProgressBarStyle property, 151
- project structure
 - About.xaml, 28
 - App class, 28–29
 - App.xaml, 28–29
 - Application_Startup event handler, 28
 - Application_UnhandledExceptions event handler, 29
 - ApplicationStrings.resx, 32
 - Assets folder, 29
 - BusyIndicator control, 29
 - ChildWindow class, 29
 - ClientBin folder, 24
 - Controls folder, 29
 - CustomDataForm class, 29
 - ErrorWindow.xaml, 29
 - Extensible Application Markup Language (XAML), 28
 - Generated_Code folder, 25
 - Helpers folder, 29
 - Home.xaml, 28
 - InitializeRootVisual method, 28
 - Libs folder, 30
 - Login folder, 29
 - MainPage.xaml, 28
 - Models folder, 30
 - onSilverlightError function, modifying, 31
 - project links, 24
 - recommended project template modifications, 31
 - resource files, 26
 - Resources folder, 29
 - Silverlight tab, 25
 - Startup event, 28
 - Styles.xaml, 29
 - Views folder, 28
 - web application projects, 24
 - Web folder, 30
 - XAP files, 32
 - See also* Extensible Application Markup Language (XAML)
- projection, definition of, 95
- propa snippet, 370
- propdp snippet, 340, 370
- Properties window, 41
- property element syntax, 43
- property snippets, 340
- PropertyChanged event, 69, 216–217, 222, 232, 235, 334, 338, 383, 386

PropertyChangedEventArgs object, 217
 PropertyGroupDescription object, 174
 property-level validation, 226
 PropertyMetadata object, 339
 PropertyNames property, 242
 Publish Web dialog, 486

■ Q

quantityAvailableColumn column, 155
 Query attribute, 109
 query operations, 107
 QueryName property, 148
 QueryString property, 81
 Quota property, 448

■ R

RadioButton control, 203
 Range attribute, 116
 Read method, 456
 ReadAllBytes method, 470
 ReadAllText method, 470
 ReadLine method, 455
 ReadLines method, 470
 ReadLocalValue method, 341
 ReadOnlyTemplate, 196
 ReadToEnd method, 456
 Rectangle control, 183
 Register method, 336, 370
 Register Now button, 20
 RegisterAttached method, 370
 RegisterPrefix method, 139
 RegisterRenderers method, 413
 RegistrationContext object, 258
 RegistrationData class, 99, 255
 RegistrationDataResources.resx, 27
 RegistrationDatas property, 258
 RegistrationForm view, 259
 RegularExpression attribute, 116
 RelativeSource binding, 308
 RelativeSource markup extension, 48, 359
 RelativeSource property, 306
 RelayCommand command, 391
 RemoteOnly mode, 256
 Remove method, 442
 RepeatBehavior property, 355
 Report Sharp-Shooter for Silverlight, 406
 report viewers, 409
 Report.ashx, 425

reporting

Adobe Reader plug-in, 404
 application dashboards, 404
 Aspose.Words for .NET, 406
 BaseReportHandler class, 413
 BaseReportRenderer class, 414
 choosing a reporting strategy, 407
 choosing a reporting tool, 409
 choosing an output format, 411
 creating a report renderer, 414
 creating the HtmlViewer control, 418
 creating the HTTP handler, 413
 Crystal Reports, 404
 Crystal Reports for Visual Studio, 410
 deploying the report viewer, 418
 displaying the PDF report generated on the server, 418
 Document Toolkit, 406
 examples of simple and complex reports, 404
 generating a report as a Word or Excel document, 405
 generating a report as HTML, 405
 generating and displaying a PDF report, 404, 409
 GetReport function, 416
 HTML Bridge, 419
 HtmlPage class, 419
 implementing reporting functionality in an application, 403
 IsUserAccessAuthorised property, 417
 Local Report Engine, 410–411
 LocalReport object, 414
 Microsoft XPS Document Writer, 406
 overview of serving reports to a Silverlight application, 412
 PopulateReportDataSources function, 416
 PrintDocument class, 407
 printing functionality in Silverlight 4, 405, 407
 ProductDetailsReportData class, code listing, 417
 ProductDetailsReportRenderer class, code listing, 415
 RegisterRenderers method, 413
 report designers, 409
 Report Sharp-Shooter for Silverlight, 406
 report viewers, 409
 reporting engines, 409
 ReportPath property, 416

Reporting (*cont.*)

- returning data to populate a report, 416
- securing access to a report, 417
- Silverlight Viewer for Reporting Services, 405
- SQL Server Reporting Services, 405
- Telerik Reporting, 406
- Visual Studio Report Designer, 410
- XML Paper Specification (XPS), 406
- XtraReports Suite, 406
- reporting engines, 409
- ReportPath property, 416
- Required attribute, 115, 198, 237
- RequiresAuthentication attribute, 250
- requiresQuestionAndAnswer property, 255
- RequiresRole attribute, 251–252
- RequiresSecureEndpoint property, 262
- ResolveConflicts method, 130
- resource files, 26
- ResourceName property, 238
- resources
 - advantages of, 62
 - defining, 62
 - MergedDictionaries property, 63
 - redefining, 63
 - resource dictionaries, 63
 - Resources property, 62
 - SolidColorBrush resource, 62
 - Styles.xaml, 63
- Resources folder, 27, 29
- Resources property, 43, 288
- ResourceType property, 195, 238
- RIA Services. *See* WCF RIA Services
- rich Internet applications (RIAs)
 - Adobe Flash, 4
 - building, 1, 4
- root visual, definition of, 38
- root-level node, 38
- RootVisual property, 38, 501
- RoundtripOriginal attribute, 119, 128
- routed events, 59
- Row property, 56
- RowDefinitions property, 54
- RowDetailsTemplate property, 159, 183
- RowDetailsVisibilityChanged event, 184
- RowDetailsVisibilityMode property, 159, 184
- RowEditEnded event, 214

■ S

- SafeFileName property, 459
- Save File dialog
 - ArgumentException exception, 458
 - creating a StreamWriter object, 459
 - creating and opening a file for writing, 459
 - DefaultExt property, 459
 - displaying, 458
 - FileInfo object, 457
 - OpenFile method, 459
 - OpenFileDialog class, 458
 - SafeFileName property, 459
 - SaveFileDialog class, 457
 - Write method, 459
 - WriteLine method, 459
 - writing binary data to a file, 459
 - writing text to a file, 459
 - See also* Open File dialog
- Save method, 440
- Scalable Vector Graphics (SVG), 35
- Scripting.FileSystemObject COM component, 475
- SDK consolidated namespace, 40
- security (client-side)
 - accessing and updating user information, 259
 - App.xaml, 258, 261
 - authenticating and registering users, 256
 - Authentication property, 258
 - AuthenticationContext object, 258
 - AuthenticationService class, 258
 - communicating over HTTPS using SSL, 262
 - Current property, 259
 - EnableClientAccess attribute, 262
 - encrypting sensitive data passed between the server and client, 262
 - Fiddler, 262
 - FormsAuthentication element, 258
 - FriendlyName property, 260
 - implementing, 256
 - IsAuthenticated property, 259
 - LoggedOut event, 258
 - Login dialog, 257
 - Login method, 259
 - minRequiredNonalphanumericCharacters property, 257

- minRequiredPasswordLength property, 257
- RegistrationContext object, 258
- RegistrationDatas property, 258
- RegistrationForm view, 259
- RequiresSecureEndpoint property, 262
- storing data locally, 262
- strategies for implementing client-side restrictions, 260
- User object, 260
- UserRegistrationService class, 258
- Visibility property, 261
- WebContext object, 258–259, 261
- WindowsAuthentication element, 258
- See also* security (server-side)
- security (server-side)
 - AdventureWorks2008 database, 249
 - ASP.NET Membership API, 246
 - ASPNETDB.MDF, 247
 - AuthenticationBase class, 249, 253–254
 - AuthenticationService class, 249, 253
 - AuthorizationAttribute class, 251
 - AuthorizationContext object, 252
 - AuthorizationContext property, 252
 - clientaccesspolicy.xml, 263–264
 - communicating over HTTPS using SSL, 262
 - CommunicationException exception, 264
 - configuring a database for Forms Authentication, 247
 - connectionStringName property, 248
 - cross-domain access policies, 263
 - crossdomain.xml, 263
 - customErrors element, 256
 - defaultProvider property, 247
 - DomainOperationException exception, 250
 - EnableClientAccess attribute, 262
 - encrypting sensitive data passed
 - between the server and client, 262
 - exposing custom user profile data, 253
 - Fiddler, 262
 - Forms Authentication provider, 246
 - FriendlyName property, 254
 - implementing cross-domain policies for
 - HTTP-based communication, 264
 - implementing cross-domain policies for
 - socket-based communication, 265
 - implementing cross-scheme access
 - policies, 265
 - implementing server-side security using
 - RIA Services, 245
 - Initialize method, 252
 - Instance property, 252
 - IsAuthorized method, 251–252
 - machine.config, 247, 249
 - preventing SQL injection attacks, 255
 - RegistrationData class, 255
 - RemoteOnly mode, 256
 - RequiresAuthentication attribute, 250
 - requiresQuestionAndAnswer property, 255
 - RequiresRole attribute, 251–252
 - RequiresSecureEndpoint property, 262
 - requiring authentication, 250
 - role-based operation access restrictions, 251
 - sanitizing error messages sent to the
 - client, 256
 - SecurityException exception, 263
 - ServiceContext object, 253
 - Silverlight Spy, 266
 - specifying an authentication provider, 246
 - SqlMembershipProvider, 247
 - SqlProfileProvider, 247
 - SqlRoleProvider, 247
 - strategies for restricting application
 - access, 266
 - user authentication, 245
 - User object, 254
 - user registration, 254
 - UserRegistrationService class, 254
 - web.config, 247, 250, 254, 267
 - Windows Authentication provider, 246
 - See also* security (client-side)
- SecurityException exception, 263, 454, 467
- SelectedDate property, 205
- SelectedDateFormat property, 206
- SelectedDates property, 205
- SelectedItem property, 204
- SelectedItems property, 215
- SelectedValue property, 204
- SelectedValuePath property, 70, 204
- SelectionMode property, 205, 215
- Services folder, 27, 99
- SetBinding method, 323
- SetContentArea method, 369
- Setter element, 272
- SetText method, 462
- SetValue method, 337

- SetVisualState method, 363–364
- .shared.cs extension, 96
- Sharp HSql, 451
- SharpZipLib, 449
- Shifflet, Karl, 191
- ShowDialog method, 454
- Siaqodb, 451
- Silverlight 1.0 JScript Page item template, 496
- Silverlight 4
 - adding a new Silverlight User Control item to a project, 332
 - Adobe Flash, 4, 11
 - advantages of, 7–8
 - ASP.NET, 10
 - Aspose.Words for .NET, 406
 - BaseReportHandler class, 413
 - BaseReportRenderer class, 414
 - BeginPrint event, 408
 - Blacklight, 15
 - BlurEffect, 284
 - building rich Internet applications (RIAs), 1, 4
 - choosing a reporting tool, 409
 - choosing an output format, 411
 - community support for, 1
 - comparing to HTML-based applications, 7
 - comparing to Microsoft's other core platforms, 9
 - compatibility with major browsers and operating systems, 5
 - Component Object Model (COM), 6
 - creating a report renderer, 414
 - creating reusable user interface components, 331
 - creating the HtmlViewer control, 418
 - creating the HTTP handler, 413
 - Crystal Reports for Visual Studio, 410
 - data input controls available in Silverlight and the Silverlight Toolkit, 201
 - DataPager control, 175
 - default Silverlight installation process, 489
 - deploying the report viewer, 418
 - detecting network availability, 452
 - developing business applications in, 1
 - displaying the PDF report generated on the server, 418
 - Document Toolkit, 406
 - DropShadowEffect, 284
 - elevated trust mode, 466
 - EndPrint event, 408–409
 - Expression Blend 4, 14
 - Extensible Application Markup Language (XAML), 3, 5–6
 - generating and displaying a PDF report, 409
 - GetIsNetworkAvailable method, 452
 - GetReport function, 416
 - HasMorePages property, 408
 - history of, 6–7
 - HTML Bridge, 419
 - HtmlPage class, 419
 - implicit styling, 271
 - installing applications with elevated trust permissions, 6
 - IsUserAccessAuthorised property, 417
 - lack of support for mobile operating systems, 5
 - Local Report Engine, 410–411
 - LocalReport object, 414
 - market penetration of, 4
 - Microsoft Web Platform Installer 2.0, downloading, 13
 - Microsoft XPS Document Writer, 406
 - Moonlight project, 5
 - MoXAML Power Toys, 303
 - .NET, 5
 - .NET Reflector, 16
 - NetworkAddressChanged event, 452
 - NetworkChange class, 452
 - NetworkInterface class, 452
 - official web site for, 13
 - overlying an IFrame on an application, 420
 - overview of, 4
 - overview of serving reports to a Silverlight application, 412
 - PageVisual property, 408
 - PopulateReportDataSources function, 416
 - Print method, 408
 - PrintDocument class, 407, 409
 - printing functionality in, 405, 407
 - PrintPage event, 408
 - PrintPageEventArgs object, 408
 - ProductDetailsReportData class, code listing, 417
 - ProductDetailsReportRenderer class, code listing, 415

- recommended tools for setting up a development environment, 13
- RegisterRenderers method, 413
- report designers, 409
- Report Sharp-Shooter for Silverlight, 406
- report viewers, 409
- reporting engines, 409
- ReportPath property, 416
- returning data to populate a report, 416
- RIA Services, 3, 7
- running applications in OOB mode, 6
- running Silverlight applications on
 - Linux, 5
- securing access to a report, 417
- securing data and operations
 - appropriately, 245
- Silverlight 4 Tools, installing, 15
- Silverlight runtime, 4–5
- Silverlight Spy, 16, 266
- Silverlight Toolkit, 15
- Silverlight Viewer for Reporting Services, 405
- SilverlightShow, 3
- SketchFlow, 14
- SQL Server 2008 Express Edition, 15
- SQL Server Reporting Services, 405
- style resources in, 271
- System.Net.NetworkInformation
 - namespace, 452
- System.Windows.Browser namespace, 419
- Telerik Reporting, 406
- Visual Studio 2008 and, 14
- Visual Studio 2010 and, 13
- Visual Studio Report Designer, 410
- WCF RIA Services, 15
- WebBrowser control, 418
- when to use, 8–9
- Windows Forms, 9
- Windows Phone 7 and Silverlight 3, 5
- Windows Presentation Framework (WPF), 6, 9
- working with collections, 144
- XAP files, 5
- XML Paper Specification (XPS), 406
- XtraReports Suite, 406
- See also* Silverlight Toolkit; Visual Studio 2010
- Silverlight Business Application template, 17–18, 95, 274
 - default user interface, 73
 - Solution Explorer window, 23
 - user interface created by, 19
 - Silverlight Class Library project, 18
 - Silverlight Database (SilverDB), 450
 - Silverlight Enterprise Deployment whitepaper, downloading, 494
 - Silverlight Navigation Application template, 17–18
 - Silverlight Spy, downloading, 16
 - Silverlight tab, 25
 - Silverlight Toolkit, 82
 - BusyIndicator control, 150
 - ChildWindow class, 183
 - data input controls available in Silverlight and the Silverlight Toolkit, 201
 - DataForm control, 193, 196
 - DateTimeTypeConverter, 345
 - layout controls, 57
 - predefined themes in, 283
 - TimeTypeConverter, 345
 - TransitioningContentControl, 90
 - WeakEventListener class, 60
 - Silverlight Unit Test Application template, 19
 - Silverlight User Control item template, 38
 - Silverlight Viewer for Reporting Services, 405
 - Silverlight.js, 27, 492
 - single document interface (SDI) applications, 73
 - site store, 439
 - SiteSettings property, 440
 - SizeChanged event, 347, 421
 - SketchFlow, 14, 71, 270
 - skinning, definition of, 282
 - sldp snippet, 340
 - sldpc snippet, 340
 - sllauncher.exe, 432, 436, 494–495
 - sockets
 - pollingDuplexHttpBinding, 139
 - Silverlight’s support for socket communication, 139
 - System.Net.Sockets namespace, 139
 - SolidColorBrush resource, 62
 - Solution Explorer window, 23, 485–486, 488
 - SortDescriptions collection property, 172
 - SortDescriptor object, 171
 - SortDescriptors property, 171
 - source, 65
 - Source property, 175
 - SourceObject property, 293
 - SQL Server 2008, 101

- SQL Server 2008 Express Edition
 - downloading, 15
 - Runtime with Management Tools
 - option, 15
- SQL Server Management Studio, 101
- SQL Server Reporting Services, 405
- SQLite, 450
- SqlMembershipProvider, 247
- SqlProfileProvider, 247
- SqlRoleProvider, 247
- StackPanel control, 42, 57
- standard properties, 334
- StartsWith operator, 169
- Startup event, 28, 464
- state transitions, 281
- Static state, 354
- StaticResource markup extension, 47, 62, 64, 67, 85, 276, 288, 305, 310
- Sterling, 450
- StoreEntity property, 130
- StreamWriter object, 459
- StringFormat property, 201, 317
- StringLength attribute, 116
- Style property, 272, 279, 286
- style resources
 - App.xaml, 274
 - Background property, 273, 277
 - BasedOn property, 276
 - defining and using, 271
 - defining as a resource dictionary, 274
 - defining as a resource dictionary in a referenced assembly, 275
 - defining at the application level, 274
 - defining at the view level, 273
 - defining constants in XAML, 285
 - defining explicit styles, 272
 - defining implicit styles, 273
 - defining in Generic.xaml, 275
 - inheriting style resources, 276
 - locations for defining style resources in a project, 273
 - merging multiple resource dictionaries into a single Resources property, 274
 - naming style resources based on their purpose, 275
 - restoring a control's default style, 286
 - Setter element, 272
 - Silverlight Business Application
 - template, 274
 - StaticResource markup extension, 276
 - storing complex values in styles, 273
 - Style property, 272, 286
 - Styles.xaml, 29, 63, 90, 274
 - styling constraints, 277
 - TargetType property, 272
 - ThemeStyles.xaml, 275
 - UserFieldsStyle style, 275
 - x:Key, 272
- styles
 - definition of, 64
 - explicit and implicit styles, 64
 - Style property, 64
- Submit method, 103–104, 131
- SubmitChanges event, 295
- SubmitChanges method, 98, 103, 239, 291, 293, 296–297
- SubmitChangesAction class, 295
- SubmitOperation object, 239–240
- SubmittedChanges event, 239–240
- SubmittedChangesEventArgs object, 240
- SubmittingChanges event, 239
- summary lists
 - AdventureWorks2008 database, 157
 - Application_UnhandledException event
 - handler method, 185
 - AreRowDetailsFrozen property, 184
 - AutoGenerateColumns property, 154
 - BitmapImage, 157
 - browsing and manipulating summary lists, 165
 - CanUserReorderColumns property, 159
 - CanUserResizeColumns property, 159
 - CellEditingTemplate property, 155
 - CellStyle property, 158
 - ChildWindow class, 180, 183
 - choosing between the DataGrid and
 - ListBox controls, 164–165
 - CollectionChanged event, 172
 - Column property, 156
 - Completed event, 185
 - configuring and customizing a DataGrid
 - control, 154
 - configuring and customizing a ListBox
 - control, 160
 - configuring DataGrid columns, 154
 - Contains operator, 169
 - creating a domain operation, 143
 - creating multiple levels of groupings, 173
 - customizing the layout of ListBox items, 161
 - Data Sources window, 156

- DataContext property, 153, 156
- DataGrid control, list of built-in features and behaviors, 158
- DataGrid's row details feature, 183
- DataGridCheckBoxColumn, 155
- DataGridTemplateColumn, 155
- DataGridTextColumn, 155
- DataPager control, 175
- defining a data template, code listing, 160
- displaying additional row details, 159
- displaying an image in a column, 156
- displaying grouped data, 159
- DisplayMode property, 175
- domain context method, 169
- DomainDataSource control, 159, 166, 184
- DomainDataSource method, 169, 171, 173
- DomainDataSourceView, 178, 184
- DomainOperationException exception, 186
- editing data in a DataGrid control, 158
- EntityQuery object, 167, 172, 177
- EntityQueryable object, list of extension methods, 166
- Exception type, 186
- FilterDescriptor object, 169
- FilterDescriptors property, 169
- filtering a summary list, 168
- finding and drilling down on a record, 178
- FloatableWindow control, 183
- GroupDescriptions collection property, 175
- GroupDescriptor object, 173
- GroupDescriptors property, 173
- grouping summary list data, 172
- grouping the items in a
 - PagedCollectionView, 174
- handling errors, 184
- HorizontalScrollBarVisibility property, 163
- HyperlinkButton, 179
- implementing a calculated column, 158
- implementing a master/details view, 184
- IncludeTotalCount property, 177
- integrating the DomainDataSource with the DataPager control, 176
- IPagedCollectionView interface, 178
- IQueryable expression, 143, 166
- IsTotalItemCountFixed property, 175
- Items property, 163
- ItemsPanel property, 161
- ItemsPanelTemplate property, 161
- ItemsSource property, 153–154, 163, 178
- ItemTemplate property, 160–161
- IValueConverter interface, 157
- KeyDown event, 180
- ListBox control, limitations of, 163
- Load method, 167
- LoadedData event, 185
- LoadOperation object, 177, 185
- LoadSize property, 177
- manually configuring the columns of a DataGrid control, 156
- MarkErrorAsHandled method, 185
- Models folder, 142
- MoveCurrentToNext method, 184
- MoveCurrentToPosition method, 184
- NavigateUri property, 179
- NumericUpDown control, 155
- opening details in a new view, 179
- opening details in a pop-up window, 180
- OperationErrorStatus enumeration, 186
- OrderByDescending clause, 171
- overview of, 141
- PagedCollectionView, 165, 170
- PageIndex property, 178
- PageSize property, 177–178
- paging a summary list, 175
- paging using the domain context method, 177
- presenting a summary list to the user, 152
- ProductSummary object, 142, 179
- PropertyGroupDescription object, 174
- quantityAvailableColumn column, 155
- querying data exposed by a domain operation, 165
- Rectangle control, 183
- reordering columns, 159
- resizing columns, 159
- retrieving data from the server, 153
- returning summary list data as a collection of presentation model objects, 142
- returning the results from a domain operation, 143
- RowDetailsTemplate property, 159, 183
- RowDetailsVisibilityChanged event, 184

summary lists (*cont.*)

- RowDetailsVisibilityMode property, 159, 184
- SortDescriptions collection property, 172
- SortDescriptor object, 171
- SortDescriptors property, 171
- sorting a DataGrid by its values, 158
- sorting a summary list, 170
- sorting the items in a
 - PagedCollectionView, 172
- sorting using the domain context
 - method, 171
- Source property, 175
- StartsWith operator, 169
- System.ComponentModel.DataAnnotations, 143
- Tag property, 180
- TextChanged event, 169
- throwOnError parameter, 186
- using third-party DataGrids, 154
- XAML-based approach versus code-based, 153
- Sync Framework for Silverlight, 450
- System namespace, 288
- System.ComponentModel namespace, 216, 345, 347
- System.ComponentModel.DataAnnotations, 143
- System.ComponentModel.DataAnnotations namespace, 113, 115, 121, 227, 229, 235
- System.Diagnostics namespace, 219
- System.IO namespace, 445, 453, 468
- System.IO.IsolatedStorage namespace, 440
- System.Linq namespace, 215
- System.Net.NetworkInformation namespace, 452
- System.Net.Sockets namespace, 139
- System.Reflection namespace, 218–219
- System.Runtime.InteropServices.Automation namespace, 472
- System.Runtime.Serialization namespace, 447
- System.ServiceModel.DomainServices.Server namespace, 114
- System.Transactions namespace, 132
- System.Windows namespace, 430, 462
- System.Windows.Browser namespace, 419
- System.Windows.Controls.Primitives namespace, 332
- System.Windows.Data namespace, 323
- System.Windows.Input namespace, 211
- System.Windows.Interactivity namespace, 297

- System.Windows.Interactivity.dll assembly, 290, 292
- System.Windows.Markup namespace, 302
- System.Windows.Media namespace, 39

■ T

- TabIndex property, 210
- TabNavigation property, 210
- Tag property, 180
- tags in XAML, 37
- target, 65
- Target property, 210
- TargetedTriggerAction, 295
- TargetName property, 79
- TargetNullValue property, 318
- TargetObject property, 293
- TargetType property, 272, 350
- Telerik Reporting, 406
- template parts, 282
- Template property, 279
- TemplateBinding markup extension, 48, 358–359, 366
- TemplatedParent mode, 307
- TemplatePart attribute, 360
- templates
 - control templates, 65
 - data templates, 65
 - definition of, 65
- TemplateVisualState attribute, 360
- Text property, 358, 360, 385
- TextAlignment property, 202
- TextBlock control, 42
- TextBox control, 201
- TextChanged event, 169
- TextWrapping property, 202
- Themes folder, 278, 348, 350, 359
- ThemeStyles.xaml, 275
- theming
 - creating a custom theme, 282
 - definition of, 282
 - predefined themes in the Silverlight Toolkit, 283
 - project templates for installed themes, 283
 - skinning, 282
- throwOnError parameter, 186
- TimePicker control, 207
- TimeStamp attribute, 119, 128
- TimeUpDown control, 206

Title property, 76
 toast notifications
 displaying, 437
 Silverlight 4 and, 437
 toolkit namespace, 194
 ToolTipService control, 44
 TopMost property, 434
 ToString method, 201, 208
 Tozon, Andrej, 292
 transactions
 encapsulating database changes within a transaction, 131
 Submit method, 131
 System.Transactions namespace, 132
 TransitioningContentControl, 90
 TriggerAction, 295
 triggers
 configuring triggers and actions using Expression Blend, 291
 creating custom triggers, 294
 definition of, 290
 EventTrigger, 290, 293, 295
 how triggers and actions interact, 291
 implementing triggers and actions by hand, 292
 Microsoft.Expression.Interactions.dll assembly, 291
 System.Windows.Interactivity.dll assembly, 290
 using predefined triggers and actions, 290
 See also actions
 TryGetValue method, 441
 TryValidateObject method, 237
 TryValidateProperty method, 236
 TwoWay mode, 68, 306
 type converters
 creating and implementing, 343
 DateTimeTypeConverter, 345
 TimeTypeConverter, 345
 TypeConverter attribute, 345
 TypeConverter class, 344
 TypeConstraint attribute, 295

■ U

UIElement class, 52
 UnhandledException event, 240
 UpdateSource method, 319
 UpdateSourceTrigger property, 319
 Uri class, 78
 URI mapper, 83
 Uri property, 84
 user authentication, 245
 User class, 99
 user controls
 adding a new Silverlight User Control item to a project, 332
 BeginInit method, 345
 CanConvertFrom method, 345
 CanConvertTo method, 345
 configuring the user control's DataContext property, 343
 constraining the user control's size, 347
 consuming the property values in a control, 342
 ConvertFrom method, 345
 ConvertTo method, 345
 creating a CustomTextBox class, 332
 creating a simple FormField user control, 333
 creating a ValueChanged event, 346
 DataContext property, 334
 DefaultStyleKey property, 332
 dependency properties, 335
 DesignerProperties class, 347
 determining whether a control is executing in a designer or at runtime, 347
 direct events, defining, 346
 EndInit method, 345
 exposing events, 346
 exposing methods, 346
 FormField.xaml, 333
 Generic.xaml, 332
 INotifyPropertyChanged interface, 334–335
 IsInDesignTool property, 347
 ISupportInitialize interface, 345
 MainPage.xaml, 333, 335
 MoXAML Power Toys, 333
 options for adding behavior to an existing control, 331
 PropertyChanged event, 334
 SizeChanged event, 347
 standard properties, 334
 System.ComponentModel namespace, 345, 347
 System.Windows.Controls.Primitives namespace, 332

- user controls (*cont.*)
 - type converters, creating and implementing, 343
 - TypeConverter attribute, 345
 - TypeConverter class, 344
 - ways to expose a property from a user control, 334
 - See also* content controls; controls; custom controls
- user interfaces
 - alternative user interface frameworks, list of, 91
 - applying animation effects, 284
 - avoiding unsightly user interfaces, 269
 - creating a simple user interface layout, code listing, 60
 - developing, 71
 - Expression Blend and, 14, 269
 - Visual Studio 2010, 14
- User object, 254, 260
- User.shared.cs, 27, 99
- UserControl control, 39
- UserFieldsStyle style, 275
- UserName property, 385
- UserNameTextBox text box, 385
- UserRegistrationService class, 27, 254, 258
- UserRegistrationService domain service, 99

■ V

- validate changeset phase, 103
- ValidateObject method, 237
- ValidateProperty method, 236
- ValidatesOnDataErrors property, 232–233
- ValidatesOnExceptions property, 229, 233
- ValidatesOnNotifyDataErrors property, 233
- validation
 - Class property, 116
 - custom class validation attributes, 117
 - custom property validation attributes, 116
 - CustomValidation attribute, 116
 - defining validation logic for entities on the server, 115
 - IsValid method, 116
 - predefined validation attributes, 115
 - ProductClassValidation attribute, 117
 - Range attribute, 116
 - RegularExpression attribute, 116
 - Required attribute, 115
 - StringLength attribute, 116
 - System.ComponentModel.DataAnnotations namespace, 115
 - ValidationAttribute class, 116
 - ValidationContext object, 236
 - ValidationErrorResources.resx, 27
 - ValidationException exception, 229–230, 236
 - ValidationResult class, 116
 - ValidationSummary control, 224–226
- Validator object, 235–236, 383
- value converters
 - Convert method, 320
 - ConvertBack method, 320
 - ConverterParameter property, 321–322
 - creating a simple value converter, 320
 - definition of, 319
 - NotOperatorValueConverter value converter, 320
 - passing a culture to a value converter, 322
 - passing a parameter to a value converter, 321
 - passing an entire object to a value converter, 322
 - using a value converter, 320
 - ValueConverter, 79
- value precedence, ordering of, 340
- ValueChanged event, creating, 346
- ValueMemberBinding property, 208
- ValueMemberPath property, 208
- ValueProperty dependency property identifier, 336
- vector icons, 284
- VerticalScrollBarVisibility property, 202
- ViewModel class, 313
- ViewModelBase class, 382
- ViewModels, 376, 378
 - considering the ViewModel as a Model of the View, 378
 - testability of, 379
- views, 73
 - CollectionViewSource, 145
 - creating additional views, 76
 - DomainDataSourceView, 145
 - implementing a master/details view, 184
 - PagedCollectionView, 145
 - view manipulations, 144
 - Views folder, 28, 74
- Visibility property, 152, 261, 352

Visual Studio 2010

- assigning event handlers in XAML, 58
 - Code Analysis feature, 27
 - data binding expression builder, 324
 - designing user interfaces, 14
 - Extension Manager, 283
 - IntelliSense, 41
 - launching an application in OOB mode, 436
 - New Project window, 16
 - Professional edition, 13
 - prodp snippet, 340
 - Properties window, 41
 - publishing a web site to a server, 486
 - using as an IDE, 13
 - Visual Studio 2008 and Silverlight 4, 14
 - Visual Studio Development Server, 22
 - Visual Studio Online Gallery, 283
 - Visual Studio Report Designer, 410
 - Visual Web Developer 2010 Express, 13
 - XAML Power Toys, 191–193
 - See also* Extensible Application Markup Language (XAML); navigation framework; Silverlight 4
- VisualStateManager, 350, 356–357, 363
- VisualTransition object, 356
- VisualTreeHelper class, 39

■ W

- WaitIndicator class, 349
- WaitIndicator control, 353, 357
- WaitIndicator.cs, 348–349
- WaitIndicator.xaml, 359
- WCF RIA Services, 15
- Adventure Works Cycles, 101
 - AdventureWorks2008 database, 101
 - AdventureWorksModel.edmx, 102
 - Associate method, 122
 - Association attribute, 114–115
 - AuthenticationService domain service, 99
 - authorize changeset phase, 103
 - calling the insert/update/delete operations for each entity in the changeset, 109
 - changeset, 98, 103
 - ChangeSet object, 130
 - ChangeSetEntry object, 130
 - Class property, 116
 - code-generation process, 95
 - Composition attribute, 115
 - concurrency checks involving the three entity versions, 128
 - Concurrency Mode property, 129
 - ConcurrencyCheck attribute, 119, 128
 - configuring the entity model to check for concurrency violations, 128
 - ConflictMembers property, 130
 - consuming the domain services, 98
 - controlling client entity generation, 113
 - creating a data access layer using the Entity Framework, 101
 - creating a domain service, 104
 - creating a presentation model class, 120
 - creating an entity model, 102
 - creating domain operations, 97
 - creating one or more domain services, 97
 - creating the ProductService domain service, procedure for, 107
 - custom class validation attributes, 117
 - custom operation, code example, 112
 - custom operations, 111
 - custom property validation attributes, 116
 - CustomValidation attribute, 116
 - decorating entities with RIA Services attributes and data annotations, 112
 - defining validation logic for entities on the server, 115
 - delete operation method names, 110
 - delete operation, code example, 110
 - Display attribute, list of properties, 118
 - domain operations, definition of, 107
 - Domain Service Class Wizard, 106, 113
 - domain service life cycle, 103
 - Editable attribute, 118
 - Enable Client Access check box, 106
 - EnableClientAccess attribute, 106
 - encapsulating database changes within a transaction, 131
 - encapsulating logic in a separate class library, 125
 - Entity Data Model Wizard, 102
 - Entity Framework, 94
 - errorInfo object, 127
 - Exclude attribute, 114
 - execute changeset phase, 103
 - Expose OData Endpoint check box, 106
 - exposing a Product entity, 104

WCF RIA Services (*cont.*)

- ExternalReference attribute, 115
- Generate Associated Classes for
 - Metadata check box, 106
- Generated_Code folder, 25, 95
- GetOriginal method, 130
- GetProduct domain operation, 108
- GetProducts operation, 108
- handling data concurrency violations, 127
- handling server errors, 126
- IEnumerable collection, 98
- implementing custom objects rather than entities, 119
- Include attribute, 114
- Include method, 114
- including or excluding associated entities, 114
- including or excluding properties, 114
- insert operation method names, 109
- insert operation, code example, 110
- inspecting the generated code in the Silverlight project, 124
- installing RIA Services assemblies to the Bin directory, 485
- installing RIA Services assemblies to the GAC, 484
- invoke operation, code example, 111
- invoke operations, 111
- IQueryable expression, 121
- IsDeleteConflict property, 130
- IsValid method, 116
- Key attribute, 118
- linking Silverlight and web projects, 97
- LINQ to Entities, 101
- LINQ to SQL, 94
- LinqToEntitiesDomainService class, 130
- making associated entities editable, 115
- metadata classes, 106, 113
- methods of identifying concurrency violations, 127
- Models folder, 99
- ModifiedDate property, 129
- naming convention for domain services, 125
- object-relational mapper (ORM), 101
- OnError method, 126, 131
- OptimisticConcurrencyException
 - exception, 131
- OutputCache attribute, 109
- overview of, 94–95
- persist changeset phase, 103
- populating and exposing the
 - presentation model types, 121
- predefined validation attributes, 115
- Product entity, 114
- ProductClassValidation attribute, 117
- ProductInventory entity, 114–115
- ProductInventory field, 114
- ProductInventoryPM object, 123
- ProductMetadata class, 113
- ProductPM object, 120, 123
- ProductPMSERVICE, 121
- ProductService class, 107
- ProductService domain service, 115
- ProductService.cs, 104
- projection, definition of, 95
- Query attribute, 109
- query operations, 107
- Range attribute, 116
- RegistrationData class, 99
- RegularExpression attribute, 116
- Required attribute, 115
- ResolveConflicts method, 130
- resolving conflicts in the domain service, 130
- returning a collection of entities, 107
- returning a single entity, 108
- returning data from web services in
 - DataSets, 120
- RoundtripOriginal attribute, 119, 128
- selecting the entities to expose, 105
- separating the presentation, middle, and data tiers, 94
- Services folder, 99
- .shared.cs extension, 96
- sharing custom code between the
 - middle and presentation tiers, 123
- Silverlight Business Application
 - template, 95
- simplifying access to data in Silverlight
 - applications, 93
- SQL Server 2008, 101
- StoreEntity property, 130
- strategies for identifying concurrency
 - violations, 128
- StringLength attribute, 116
- Submit method, 103–104, 131
- SubmitChanges method, 98, 103
- System.ComponentModel.DataAnnotations
 - namespace, 113, 115, 121

- System.ServiceModel.DomainServices.Server namespace, 114
 - System.Transactions namespace, 132
 - TimeStamp attribute, 119, 128
 - update operation method names, 109
 - update operation, code example, 110
 - updating the presentation model types, 122
 - User class, 99
 - User.shared.cs, 99
 - UserRegistrationService domain service, 99
 - validate changeset phase, 103
 - ValidationAttribute class, 116
 - ValidationResult class, 116
 - WCF RIA Services Class Library project template, 125
 - WCF RIA Services Toolkit, features of, 132
 - WCF RIA Services Class Library template, 19
 - Web Application project, 98, 102
 - WebContext class, 126
 - WCF Services
 - ASP.NET session state, 136
 - ASP.NET_SessionId, 136
 - basicHttpBinding, 134
 - bindings, configuring, 134
 - bindings, definition of, 134
 - communicating with the service in a RESTful manner, 137
 - configuring and using, 137
 - consuming a WCF Service, 135
 - contract, 134
 - customBinding, 134
 - endpoint, 134
 - Global.asax, 136
 - InvalidOperationException exception, 135
 - maximum message size returned as a response, 135
 - netTcpBinding, 134
 - OData protocol, 137
 - pollingDuplexHttpBinding, 134
 - sharing code, business objects, and business logic between the server and client, 136
 - similarities between Data Services and RIA Services, 137
 - wsHttpBinding, 135
 - WeakEventListener class, 60
 - Web Application project, 102
 - structure of, 98
 - web application projects, 24
 - ApplicationNameTestPage.aspx, 26, 31
 - ApplicationNameTestPage.html, 26, 31
 - AuthenticationService class, 27
 - data transfer object (DTO) classes, 27
 - GlobalSuppressions.cs, 27
 - Models folder, 27
 - RegistrationDataResources.resx, 27
 - Resources folder, 27
 - Services folder, 27
 - Silverlight.js, 27
 - structure of, 26
 - User.shared.cs, 27
 - UserRegistrationService class, 27
 - ValidationErrorResources.resx, 27
 - Web folder, 30
 - Web Setup Project template, 488
 - web.config, 246–247, 250, 254, 267, 414
 - WebBrowser control, 418
 - WebClient class, 138
 - WebContext object, 126, 258–259, 261
 - WebRequest class, 138–139
 - weighted proportion sizing, 55
 - Wilcox, Jeff, 507
 - Window Style option, 479–480
 - Windowless property, 420
 - Windows Authentication provider, 246
 - Windows Azure, 484
 - Windows Forms, 1, 9, 57
 - Windows Phone 7, 5
 - Windows Presentation Framework (WPF), 6, 9
 - Windows Script Host, 474
 - WindowsAuthentication element, 258
 - WindowState property, 434
 - Word.Application, 473, 476
 - Workbook object, 476
 - WrapPanel control, 57
 - Write method, 459
 - WriteAllBytes method, 470
 - WriteAllLines method, 470
 - WriteAllText method, 470
 - WriteLine method, 459
 - WScript.Shell COM object, 474
 - wsHttpBinding, 135
- **X**
- x:Class, 45

- x:ClassModifier, 46
- x:FieldModifier, 46
- x:Key, 46, 272
- x:Name, 45
- x:Null markup extension, 48
- XAML. *See* Extensible Application Markup Language (XAML)
- XAML Power Toys, 191–193
- XAP files, 5, 32
- XML Paper Specification (XPS), 35, 406

- XmlnsDefininition attribute, 301
- XmlnsDefinition attribute, 40
- XmlnsPrefix attribute, 302
- XtraReports Suite, 406

■ Z

- zero code-behind, 377
- Zwikstra, Koen, 16

